ИСП
Институт Системного Программирования
им. В.П. Иванникова
Российской Академии наук

ISSN 2079-8156 (Print)
ISSN 2220-6426 (Online)

Труды Института Системного Программирования РАН

Proceedings of the Institute for System Programming of the RAS

Том 29, выпуск 6

Volume 29, issue 6

Москва 2017

ISSN 2079-8156 (Print), ISSN 2220-6426 (Online)

Труды Института системного программирования РАН

Proceedings of the Institute for System Programming of the RAS

Труды ИСП РАН – это издание с двойной анонимной системой рецензирования, публикующее научные статьи, относящиеся ко всем областях системного программирования, технологий программирования и вычислительной техники. Целью издания является формирование научно-информационной среды в этих областях путем публикации высококачественных статей в открытом доступе.

Издание предназначено для исследователей, студентов и аспирантов, а также практиков. Оно охватывает широкий спектр тем, включая, в частности, следующие:

- операционные системы;
- компиляторные технологии;
- базы данных и информационные системы;
- параллельные и распределенные системы;
- автоматизированная разработка программ;
- верификация, валидация и тестирование;
- статический и динамический анализ;
- защита и обеспечение безопасности ПО;
- компьютерные алгоритмы;
- искусственный интеллект.

Журнал издается по одному тому в год, шесть выпусков в каждом томе.

Поддерживается открытый доступ к содержанию издания, обеспечивая доступность результатов исследований для общественности и поддерживая глобальный обмен знаниями.

Труды ИСП РАН реферируются и/или индексируются в:

Proceedings of ISP RAS are a double-blind peer-reviewed journal publishing scientific articles in the areas of system programming, software engineering, and computer science. The journal's goal is to develop a respected network of knowledge in the mentioned above areas by publishing high quality articles on open access.

The journal is intended for researchers, students, and practitioners. It covers a wide variety of topics including (but not limited to):

- Operating Systems.
- Compiler Technology.
- Databases and Information Systems.
- Parallel and Distributed Systems.
- Software Engineering.
- Software Modeling and Design Tools.
- Verification, Validation, and Testing.
- Static and Dynamic Analysis.
- Software Safety and Security.
- Computer Algorithms.
- Artificial Intelligence.

The journal is published one volume per year, six issues in each volume.

Open access to the journal content allows to provide public access to the research results and to support global exchange of knowledge. **Proceedings of ISP RAS** is abstracted and/or indexed in:

УДК004.45

Адрес: 109004, г. Москва, ул. А. Солженицына, дом 25.
Телефон: +7(495) 912-44-25
E-mail: info-isp@ispras.ru
Сайт: http://www.ispras.ru/proceedings/

Address: 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
Tel: +7(495) 912-44-25
E-mail: info-isp@ispras.ru
Web: http://www.ispras.ru/en/proceedings/

Содержание

Table of Contents

Подход к реализации системы верифицированного исполнения программного кода

А.В. Козачок <a.kazachok@academ.mks.rsnet.ru>
Е.В. Кочетков <e.kochetkov@academ.msk.rsnet.ru>
Академия Федеральной службы охраны Российской федерации,
302034, Россия, г. Орёл, ул. Приборостроительная, д. 35

Аннотация. В настоящей статье представлено описание технической реализации системы верифицированного исполнения программного кода. Функциональным предназначением данной системы является проведение исследования произвольных исполняемых файлов операционной системы в условиях отсутствия исходных кодов с целью обеспечения возможности контроля исполнения программного кода в рамках заданных функциональных требований. Описаны предпосылки создания такой системы, дан порядок действий пользователя по двум типовым сценариям использования. Представлено общее описание архитектуры построения системы и используемые для ее реализации программные средства, а также механизм взаимодействия элементов системы. Рассмотрен модельный пример реализации такой системы, демонстрирующий возможность гибкого задания комплекса функциональных ограничений, применение которых основывается на атрибуте времени совершения операции. В завершении статьи приведено краткое сравнение с наиболее близкими аналогами.

Ключевые слова: формальная верификация, автомат безопасности, контролируемое выполнение, вредоносное программное обеспечение

DOI: 10.15514/ISPRAS-2017-29(6)-1

Для цитирования: Козачок А.В., Кочетков Е.В. Подход к реализации системы верифицированного исполнения программного кода. Труды ИСП РАН, том 29, вып. 6, 2017 г., стр. 7-24. DOI: 10.15514/ISPRAS-2017-29(6)-1

1. Введение

Развитие общества и государства влечет повсеместное внедрение информационных технологий во все сферы государственного управления и жизнеобеспечения общества, в том числе и при построении автоматизированных систем управления технологическими процессами (АСУ ТП), функционирующими на объектах критической информационной

инфраструктуры (КИИ). Защита таких объектов относится к стратегическим целям государства в области обеспечения информационной безопасности [1].

Достижение этой цели невозможно без рассмотрения вопросов информационной безопасности, связанное с риском появления ранее неизвестных угроз. К ним относится оказание деструктивного программного воздействия на элементы сетевой инфраструктуры объекта КИИ, что может привести к нарушению штатного режима функционирования АСУ ТП.

Вопросы обеспечения информационной безопасности информационных систем органов государственной власти в значительной степени проработаны, так как они непосредственно или косвенно касаются защиты конфиденциальной информации. Внимание к вопросам обеспечения информационной безопасности объектов КИИ, включающих информационные системы, вычислительные сети, АСУ ТП, функционирующих в оборонной, топливной, атомной и других областях, существенно возросло за последние годы, что обусловлено рядом инцидентов, связанных с нарушением штатного режима функционирования таких объектов [2].

Характерной особенностью современных компьютерных атак на объекты КИИ является их долговременность, целенаправленность и комплексность – APT-атаки (Advanced Persistent Thread) [3], суть которых заключается в проведении злоумышленниками ряда подготовительных мероприятий по комплексной оценке защищенности информации, циркулирующей в КИИ, включающих идентификацию используемых программно-аппаратных средств защиты информации, маршрутов ее движения, порядка обработки и хранения, а также проводимых организационных мероприятий по ее защите от несанкционированного доступа. Такой подход злоумышленников к организации атак позволяет обнаруживать в системе защиты информации наиболее уязвимые элементы и использовать их для проникновения в сетевую инфраструктуру объекта КИИ.

Очевидно, что для оказания деструктивных программных воздействий на критические процессы, протекающие в объектах КИИ, без учета внутреннего нарушителя или с его частичным привлечением, применяется специализированное вредоносное программное обеспечение (ВПО) в совокупности с элементами социальной инженерии [4].

Основным средством противодействия ВПО в сетях объектов КИИ, совместно с другими механизмами защиты, являются средства антивирусной защиты. Согласно экспериментальной оценке современных средств антивирусной защиты вероятности ошибок первого и второго рода в применяемых в них механизмов обнаружения ВПО составляют порядка 10^{-3}–10^{-4}. Однако даже при этом существенное количество образцов ВПО остаются необнаруженными [5].

По мнению авторов, наиболее перспективным является разработка механизмов, базирующихся на «запрещающей» стратегии разделения полномочий и исполняемых файлов [6], ввиду того, что реализуемые в ее

рамках решения позволяют существенно повысить уровень доверия к контенту, приходящему из внешней среды.

2. Предшествующие работы

Для решения данной задачи авторами предлагается реализация системы верифицированного исполнения программного кода (СВИПК), базирующиеся на применении метода формальной верификации «Model checking» для проверки соответствия исполняемого файла заданной спецификации безопасности как на статическом, так и на динамическом этапе проверки [6,7]. Ее применение позволяет допускать к использованию только такие программы, уровень доверия к которым подтверждается формальным математическим доказательством и непрерывным контролем за их выполнением. Работа СВИПК представляет собой последовательное выполнение ряда этапов, реализованных в виде отдельных блоков и представленных на рис. 1.

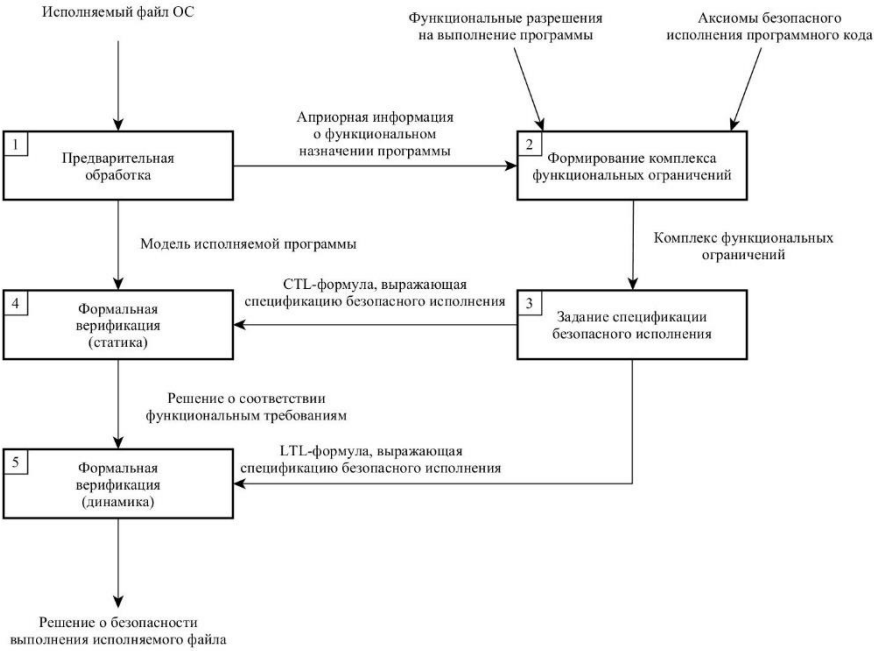


Рис. 1. Функциональная схема системы верифицированного исполнения программного кода
Fig. 1. Functional diagram of the system of verified program code execution

На вход блока 1 подается исследуемый исполняемый файл, безопасность использования которого необходимо установить. В данном блоке происходит

проверка на наличие в коде программы конструкций самомодификации (включая упаковщики) и механизмов защиты кода программы от анализа. Далее происходит преобразование исполняемого файла из бинарного представления в модель Крипке. Кроме того, на этом этапе осуществляется сбор и систематизация априорных сведений о функциональном предназначении программы, которые подаются на вход блока 2. В блоке «Формирование комплекса функциональных ограничений» на основе априорных сведений о функциональном предназначении программы формируется перечень ограничений на ее функциональные возможности, выполнение которых необходимо для ее безопасного использования. На 3 этапе осуществляется процесс преобразования комплекса функциональных ограничений в формулы темпоральной логики, выражающие требуемое свойство безопасного исполнения. В блоке 4 происходит формальная верификации модели исполняемого файла, построенной в блоке 1, на соответствие функциональным требованиям, сформулированным в блоке 2 методом формальной верификации «Model checking». В блоке 5 производится верификация исполнения программы на каждом шаге путем проверки трассы на соответствие спецификации безопасного исполнения, выраженной LTL-формулой. Выходом данного блока является решение о безопасном выполнении исполняемого файла на динамическом этапе проверки.

Для обеспечения масштабируемости при описании поведения программы авторами разработана модель функционирования процесса в операционной системе (ОС), основанная на применении субъектно-объектного подхода [8]. Ее особенностью является высокоуровневая абстракция взаимодействия между процессом и ресурсами ОС, что позволяет применить полученные на ее основе результаты к широкому классу аналогичных систем. Применение данной модели необходимо для совершения перехода от объекта реального мира (процесса) к формальной модели, позволяющей произвести процедуру верификации.

Для задания функциональных ограничений на выполнение программой некоторых действий с возможностью однозначного перехода к формулам темпоральной логики авторами предлагается формальный логический язык описания таких требований на основе модели функционирования процесса в ОС [9].

Основная идея формальной верификации применительно к обнаружению вредоносного кода состоит в построении формальной (математической) модели исследуемой программы, которая отражает (моделирует) ее возможное поведение в операционной системе. Требования корректности безопасного поведения при этом описываются в виде спецификации, которая отражает рамки разрешенного поведения программы. На основе спецификаций и модели исполняемого файла, используя метод «Model checking», осуществляется проверка, согласуется ли возможное поведение с разрешенным. Поскольку происходит именно математическая верификация,

решение о согласованности, то есть соответствии возможного поведения требуемому, является корректным.

В данном контексте под возможным поведением программы понимается модель программы, построенная на основе многократного запуска исполняемого файла с использованием интеллектуального фаззера [10]. Требуемое поведение представляет собой систему ограничений на функциональные возможности программы, то есть разрешенное поведение.

3. Архитектура системы верифицированного исполнения программного кода

Система верифицированного исполнения программного кода (СВИПК) представляет собой комплекс программных средств, обеспечивающих контролируемое безопасное исполнение пользовательских приложений в среде эмулятора. Архитектура СВИПК представлена на рис. 2. Она состоит из следующих компонентов:

- базовая ОС;
- программное средство управления СВИПК;
- полносистемный эмулятор ОС.

Выбор базовой ОС определяется удобством конфигурации политик безопасности и стабильностью работы. Она является необходимой платформой для функционирования программного средства управления СВИПК и полносистемного эмулятора. Базовая ОС должна быть сконфигурирована таким образом, чтобы пользователь имел возможность работы только с программным средством управления СВИПК и не имел возможности вносить изменения в конфигурацию базовой ОС, состав входящих в нее программных средств и т. д.

Программное средство управления СВИПК представляет собой оконное кроссплатформенное приложение, которое является связующим звеном между пользователем и эмулятором гостевой ОС. Назначением данного программного средства является выполнение следующих функций:

- запуск полносистемного эмулятора по требованию пользователя;
- создание, изменение, удаление профилей программ;
- построение модели исследуемой программы на основе трасс исполнения, полученных интеллектуальным фаззером;
- задание функциональных ограничений на работу программ;
- поддержание постоянного сетевого соединения динамически подключаемым модулем, находящимся в адресном пространстве отслеживаемых приложений для осуществления непрерывного взаимодействия.

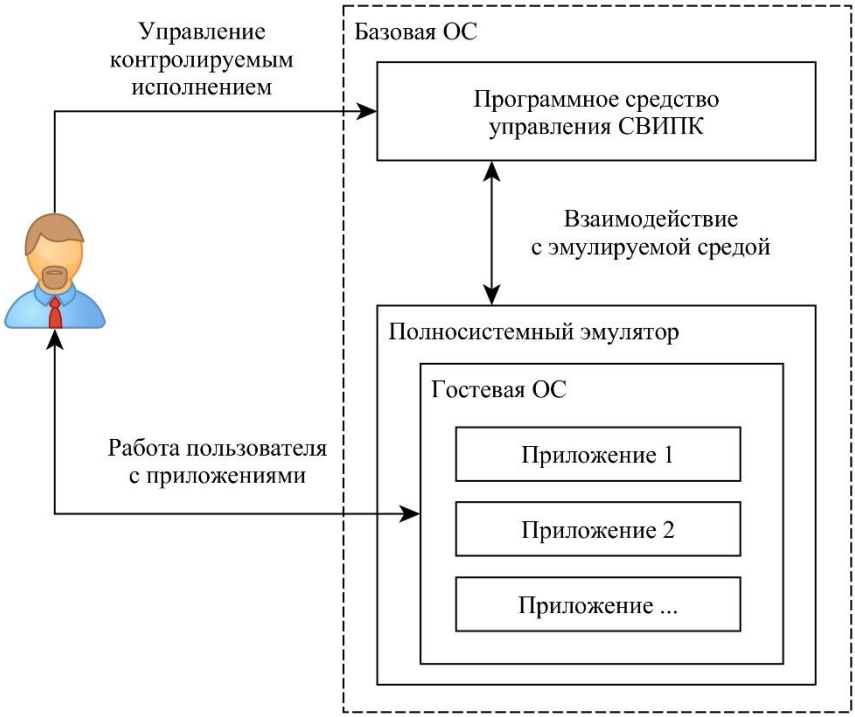


Рис. 2. Архитектура системы верифицированного исполнения программного кода
Fig. 2. Architecture of the system of verified program code execution

Полносистемный эмулятор ОС должен позволять полностью имитировать работу реального оборудования. Он запускает на исполнение гостевую ОС и создает для нее виртуальное окружение. На этом этапе производится эмуляция работы процессора, оперативной памяти и другого аппаратного обеспечения различных архитектур и платформ. Работа исследуемых приложений происходит через исполнение инструкций процессора гостевой ОС. Выбор полносистемного эмулятора для реализации СВИПК определяется следующими факторами:

- широкий диапазон поддерживаемого оборудования;
- возможность детальной конфигурации состава и параметров периферийного оборудования;
- наличие протокола, позволяющего в реальном времени управлять работой полносистемного эмулятора из внешнего приложения;
- открытый исходный код, позволяющий произвести соответствующие проверки на отсутствие недекларированных возможностей.

Ключевым фактором является поддержка внешних плагинов, позволяющих разрабатывать дополнительные модули и интегрировать их динамически в рабочую среду полносистемного эмулятора.

Все исполняемые файлы в эмулируемой ОС условно разделим на две категории:

- априорно безопасные;
- априорно небезопасные.

К первой категории исполняемых файлов относятся такие файлы, которые входят в штатный дистрибутив ОС, то есть заложены разработчиками и априорно не должны нанести вред. Ко второй категории относятся исполняемые файлы, попавшие в файловое пространство ОС из внешней среды, созданные сторонними разработчиками, достоверной информации об использовании которых нет.

Для каждой используемой программы создается «Профиль», включающий следующие сведения:

- контрольная сумма исполняемого файла;
- модель безопасного исполнения.

Профиль программы необходим для исключения этапов предварительной обработки и статической верификации исполнения при повторном использовании программы.

На рис. 3 представлена обобщенная архитектура СВИПК.

Рис. 3. Обобщенная архитектура системы верифицированного исполнения программного кода
Fig. 3. Generalized architecture of the system of verified program code execution

Использование СВИПК для запуска приложения возможно по следующим двум сценариям в зависимости от того, было ли проведено соответствующее исследование безопасности ранее:

- «первоначальная проверка»;
- «регулярное использование».

Этапы работы в рамках первого сценария производятся в отношении исполняемых файлов программ, впервые запускаемых пользователем. Данный сценарий работы предполагает осуществление следующих действий:

1. Пользователь осуществляет запуск исполняемого файла. Модуль «Предварительной обработки» осуществляет проверку на соответствие минимальным требованиям безопасности. С помощью интерфейса модуля «Задания спецификации» и с учетом требований принятой политики безопасности пользователь формирует спецификацию, ограничивающую функциональные возможности исследуемой программы.
2. В рамках подсистемы «Статической верификации безопасности исполнения» производится построение модели программы и ее формальная верификации методом «Model checking» на предмет соответствия заданной спецификации.
3. «Подсистемой динамической верификации безопасности исполнения» осуществляется перехват системных вызовов, совершаемых исследуемой программой и построение трассы исполнения.
4. Модулем «Формальной верификации трассы исполнения» реализуется процедура верификации трассы исполнения программы на предмет соответствия комплексу функциональных ограничений на динамическом этапе проверки.
5. Пользователь осуществляет непосредственную работу с программой.

Сценарий «регулярного использования» отличается тем, что работа пользователя с заданной программой начинается с этапа 4.

4. Прототип системы верифицированного исполнения программного кода

Для технической реализации прототипа СВИПК были выбраны программные средства, удовлетворяющие обозначенному выше описанию.

В качестве базовой ОС предлагается использование ОС семейства Linux, а именно «Ubuntu 16.04», ее выбор определяется удобством конфигурации политик безопасности. В качестве «гостевой» ОС предполагается использование «Windows XP SP3» (для демонстрации работы прототипа).

В качестве полносистемного эмулятора предполагается использование «Qemu», ключевой особенностью которого является поддержка системы плагинов, разработанная «Институтом системного программирования им. В.П. Иванникова РАН» [11]. Разработчиками создан механизм «signal-subscriber», позволяющий реагировать на события, генерируемые ядром «Qemu». Взаимодействие программного средства управления СВИПК с полносистемным эмулятором производится посредством протокола QMP (Qemu machine protocol) [12]. Данный протокол позволяет внешним приложениям управлять экземпляром виртуальной машины «Qemu» путем

обмена сообщениями в формате JSON. На рис. 4 представлена структурная схема прототипа СВИПК.

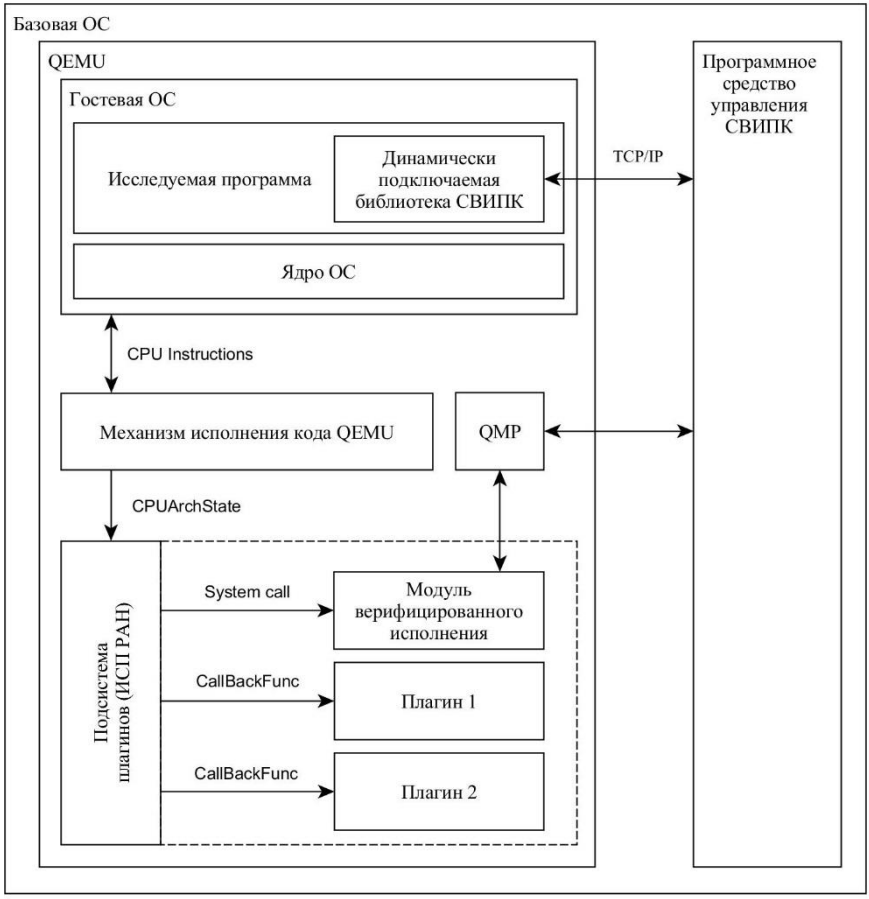


Рис. 4. Структурная схема прототипа системы верифицированного исполнения программного кода
Fig. 4. Structural scheme of the verified program code execution system prototype

При запуске любой исполняемой программы в ее адресное пространство проецируется специальная динамически подключаемая библиотека. В момент вызова функции точки входа происходит сбор информации о параметрах запуска и самом исполняемом файле. Затем собранная информация посредством сетевой подсистемы гостевой ОС передается программе управления СВИПК. Если контрольная сумма исполняемого файла существует в базе профилей множества «Априорно безопасных» программ, то

функционирование гостевой ОС продолжается в штатном режиме. Таким образом, программы, входящие в стандартный дистрибутив ОС, не контролируются СВИПК.

Если контрольная сумма отсутствует в обозначенном выше множестве, то СВИПК переходит в режим контролируемого исполнения. В случае существования профиля исполняемого файла по заданной контрольной сумме, происходит его запуск в контролируемой автоматом безопасности среде. Если такого профиля нет, то происходит работа по описанному выше сценарию «первоначальной проверки».

Автомат безопасного исполнения представляет собой плагин, обрабатывающий каждый системный вызов, сгенерированный приложением. В случае выхода контролируемого приложения за рамки спецификации безопасности, работа эмулятора приостанавливается, а пользователю выводится сообщение, содержащее причину блокировки, при этом приложение пользователя завершается.

База спецификаций хранится в виде набора файлов в файловой системе базовой ОС и доступна для программы управления СВИПК. Для каждого запускаемого в эмуляторе приложения происходит загрузка соответствующей спецификации в автомат безопасности, интегрированный в виде плагина в экземпляр «Qemu», через протокол QMP.

Предложенная СВИПК предоставляет следующие возможности:

- задание функциональных ограничений на выполнение программы в рамках эмулируемой ОС;
- формальная верификация функциональных требований до запуска исполняемого файла в рабочей среде пользователя;
- контролируемое выполнение пользовательских приложений в эмулируемой среде под непрерывным контролем автомата безопасного исполнения.

6. *Модельный пример*

Согласно «Разрешающей» стратегии построения модели исполняемой программой, все совершаемые ею действия изначально запрещены. Для недопущения совершения процессом действий, которые могут привести к потенциально неблагоприятным последствиям каждый шаг исполнения программы должен быть верифицирован на соответствию базису аксиом «Безопасного исполнения программного кода» и функциональных разрешений. Для описания выполнения этого требования вводится предикат:

$$isDynSecure(trace^*, FA) = \begin{cases} \forall s \in trace^* \vDash AX_{sec} \vee FA, & true \\ \text{иначе}, & false \end{cases},$$

где: $trace^*$ – расширенная трасса исполнения; AX_{sec} – аксиомы безопасного исполнения; FA – множество функциональных разрешений.

Дополненная трасса исполнения описывается следующим выражением:

$$trace^* = trace \parallel \omega_{curr},$$

где: $trace$ – последовательность действий процесса, совершенных на настоящий момент; ω_{curr} – действие процесса, совершаемое на следующем шаге.

Базис аксиом «Безопасного исполнения программного кода» представлен в табл. 1.

Табл. 1. Базис аксиом «Безопасного исполнения программного кода»
Table 1. The basis of axioms for secure execution program code

Формула	**Интерпретация**
$p^* \xrightarrow{c,r,w,d} m^3$	Выполнение операций над памяти только в своем адресном пространстве
$p^* \xrightarrow{c,r,w,d} e^5$	Выполнение операций над файловыми объектами только в своем домашнем каталоге
$p^* \xrightarrow{r} e^2$	Чтение системных файловых объектов и конфигурации
$p^* \xrightarrow{r,w} e^3$	Чтение и запись параметров пользовательского окружения
$p_i^* \xrightarrow{d} p_i^*$	Процесс может завершать только свою работу

С целью предотвращения передачи конфиденциальной информации через сетевые соединения и сохранения при этом возможности выхода программы в сеть, например для получения обновления, вводится следующе функциональное разрешение: «Разрешается чтения каталога или файловых объектов пользователя отличного от данного пользовательским процессом, только если в будущем не будут созданы любые сетевые соединения». Данное функциональное разрешение выражается следующим образом:

$$p^3 \xrightarrow{r} e^4 \wedge \neg F(p^3 \xrightarrow{c} n^1) \qquad (1)$$

Табл. 2. Таблица истинности функционального разрешения
Table 2. Truth table of the functional authorization

$p^3 \xrightarrow{r} e^4$	$F(p^3 \xrightarrow{c} n^1)$	$\neg F(p^3 \xrightarrow{c} n^1)$	$p^3 \xrightarrow{r} e^4 \wedge \neg F(p^3 \xrightarrow{c} n^1)$
0	0	1	**0**
0	1	0	**0**
1	0	1	**1**
1	1	0	**0**

Формула функционального разрешения принимает истинное значение только в том случае, если левая и правая ее части будут истинны одновременно. В соответствии с таблицей истинности (табл. 2), это достижимо только если

процесс совершит действие $p^3 \xrightarrow{r} e^4$ и при этом относительно этого действия в будущем не совершится действие $F(p^3 \xrightarrow{c} n^1)$, во всех остальных случаях результат вычисления функционального разрешения является ложным.

Таким образом итоговая формула для проверки безопасности исполнения программы в динамике примет следующий вид:

$$G(AX_{sec} \vee [p^3 \xrightarrow{r} e^4 \wedge \neg F(p^3 \xrightarrow{c} n^1)]) \qquad \text{а}$$

На рис. 5 представлена модель исполняемой программы, построенная в блоке 1 «Предварительная обработка» с использованием интеллектуального фазера.

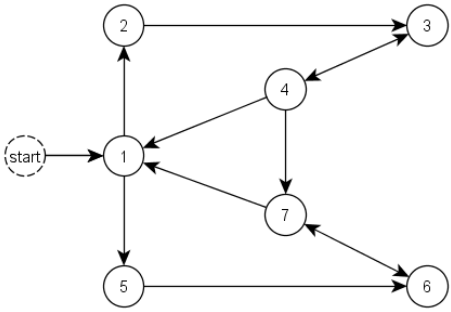


Рис. 5. Модель программы
Fig. 5. The program model

Вершины представляют собой действие процесса, ребра отображают возможность программы совершить действие процесса из соответствующего состояния. Соответствие номера вершины и действия процесса представлены в табл. 3.

Табл. 3. Соответствие номеров вершин и действий процесса
Table 3. States and process actions

№	Действие процесса	№	Действие процесса
1	$p^* \xrightarrow{c} m^3$	5	$p^* \xrightarrow{c} n^*$
2	$p^* \xrightarrow{c} e^*$	6	$p^* \xrightarrow{w} n^*$
3	$p^* \xrightarrow{r} e^*$	7	$p^* \xrightarrow{r} n^*$
4	$p^* \xrightarrow{w} e^*$		

Верифицированное исполнение программы, не нарушающее комплекс функциональных ограничений, пошагово представлено в табл. 3.

Сценарий работы программы: после запуска программа выделяет в своем адресном пространстве некоторую область памяти, далее создает некоторый файловый объект в своем домашнем каталоге, производит чтение информации

из каталога другого пользователя, производит запись в своих файловые объекты, после этого завершает работу.

Табл. 4. Верифицируемое исполнение (Легитимно)
Table 4. Verified execution (Legitimate)

№	ω_{curr}	$trace^*$	Шаг трассы выполнения	AX_{sec}	FA	$isDynSecure$
1	$p^3 \xrightarrow{c} m^3$	(1)	1	**1**	**0**	**1**
2	$p^3 \xrightarrow{c} e^5$	1-(2)	1	**1**	**0**	**1**
			2	**1**	**0**	
3	$p^3 \xrightarrow{r} e^4$	1-2-(3)	1	**1**	**0**	**1**
			2	**1**	**0**	
			3	**0**	**1**	
4	$p^3 \xrightarrow{w} e^5$	1-2-3-(4)	1	**1**	**0**	**1**
			2	**1**	**0**	
			3	**0**	**1**	
			4	**1**	**0**	

Верификатор «Model checking» проверят для каждого шага выполнимость логической формулы стоящей в предикате G, то есть выполнение свойства на каждом шаге. Интерпретация, данного выражения с точки зрения безопасности выполнения программного кода заключается в том, что на каждом шаге выполнения программы должно быть правило, которое данную операцию разрешает, если такого правила нет, то весь предикат принимает отрицательное значение, что свидетельствует о попытке совершения действия, явным образом не разрешенного, а значит запрещенного.

Первым выполняемым действием процесса является $p^3 \xrightarrow{c} m^3$ – выделение памяти в своем адресном пространстве для дальнейшего использования. На данном шаге на вход верификатору поступает трасса исполнения, состоящая из одного действия, поэтому она проводится только один раз для одного состояния. Базис аксиом «Безопасного исполнения программного кода» при этом не нарушается, так как действия, совершаемое процессом, входит в него явно, что отражено в табл. 1, при этом функциональные разрешения FA не выполняются, так как ω_{curr} не входит в формулу FA. В результате верификации первого действия программы проверена его безопасность, так как для заданной трассы исполнения выполняется предикат $isDynSecure$.

Шаг два проверяется аналогично первому. На третьем шаге работы программы при верификации 3 шага исполнения, действие процесса $p^3 \xrightarrow{r} e^4$ не входит в базис аксиом «Безопасного исполнения программного кода» ввиду этого базис не выполняется, но в дополнение к нему существует функциональное разрешение (выр. 1). Функциональное разрешение в данном случае принимает истинное значение в соответствии с таблицей истинности

(табл. 2). Так как для вычисления предиката $isDynSecure$ необходимо выполнение базиса аксиом или функциональных разрешений, то итоговый результат для проверки данного шага принимает истинное значение. По итогам проверки всех этапов работы принимается решение для всей трассы целиком. Проверка на шаге четыре проводится аналогична предыдущим шагам.

Пошаговое выполнение программы, нарушающей в процессе работы, комплекс функциональных ограничений представлено в табл. 5.

Сценарий работы программы: после запуска программа выделяет в своем адресном пространстве некоторую область памяти, далее создает некоторый файловый объект в своем домашнем каталоге, производит чтение информации из каталога другого пользователя, создает несанкционированное подключение к узлу глобальной сети.

Табл. 5. Верифицируемое исполнение (Нарушение)
Table 5. Verified execution (Violation)

№	ω_{curr}	$trace^*$	Шаг трассы выполнения	AX_{sec}	FA	$isDynSecure$
1	$p^3 \xrightarrow{c} m^3$	(1)	1	**1**	**0**	**1**
2	$p^3 \xrightarrow{c} e^5$	1-(2)	1	**1**	**0**	**1**
			2	**1**	**0**	
3	$p^3 \xrightarrow{r} e^4$	1-2-(3)	1	**1**	**0**	**1**
			2	**1**	**0**	
			3	**0**	**1**	
4	$p^3 \xrightarrow{c} n^1$	1-2-3-(4)	1	**1**	**0**	**0**
			2	**1**	**0**	
			3	**0**	**0**	
			4	**1**	**0**	

Первые три шага выполнения программы происходят как и в случае с легитимной программой. Выполнение четвертом шаге при верификации третьего состояния не выполняется базис аксиом «Безопасного исполнения программного кода» и функциональное разрешение, так как выражение $\neg F(p^3 \xrightarrow{c} n^1)$ принимает ложное значение. В итоге проверки по четвертому шагу предикат $isDynSecure$ принимает ложное значение, что свидетельствует о нарушении модели безопасного использования программного кода. В этом случае выполнение программы прерывается.

Рассмотренный модельный пример демонстрирует принципиальную возможности применения методов формальной верификации для обеспечения верифицированного исполнения программного кода в соответствии с функциональными разрешениями и базисом аксиом «Безопасного исполнения программного кода.

7. Сравнение с аналогами

Для предложенного решения, реализующего СВИПК найти прямые аналоги не удалось. Проведенный анализ существующих средств виртуализации исполнения программного кода позволил разделить программные средства создания изолированной программной среды на два класса: «Песочницы, предназначенные выявления признаков вредоносного функционала путем анализа их поведения в изолированной программной среде» и «Песочницы, предназначенные для рядового использования» с целью предотвращения нежелательных последствий от запуска программ из недоверенных источников.

Предложенное в настоящей статье решение по предназначению относится ко второму классу, но по архитектуре реализации оно является полносистемным эмулятором.

Наиболее близкие к предлагаемому решения аналоги представлены в таблице 6.

Табл. 6. Программные средства для создания изолированной программной среды
Table 6. The sandbox software

Аналог	**Платформа**	**Задание ограничений пользователем**	**Применение ограничений имеет временной атрибут**
Sandboxie	Windows	из перечня	–
Time Freeze	Windows	–	–
Shade Sandbox	Windows	из перечня	–
Mbox	Linux	–	–
FireJail	Linux	+	–
Sandbox	Linux (Fedora)	–	–

Ключевым отличием предлагаемого решения от аналогов, представленных в табл. 6, является возможность гибкого разграничения доступа к ресурсам ОС на основе атрибута времени совершения процессом действия.

Применение гибкой системы функциональных ограничений с использованием атрибута времени совершения операции позволяет ограничить лишь потенциально опасные трассы исполнения программы, существенно меньше при этом ограничивая функциональные возможности исследуемой программы в целом.

Первые эксперименты по верифицируемому исполнению программного кода показали линейный рост времени, требуемого для проведения верификации трассы исполнения программы. Также для снижение временных затрат на проведения верификации трассы исполнения комплекс функциональных

ограничений может быть разделен на два подмножества формул по признаку наличия операторов темпоральной логики.

8. Заключение

Предложенная СВИПК существенно повысит уровень доверия к исполняемым файлам, так как позволяет выявить потенциально опасный функционал исполняемой программы еще до ее запуска. В случае нарушения заданных функциональных ограничений пользователь может отказаться от ее допуска в сетевую инфраструктуру объекта КИИ или использовать верифицируемое исполнение предварительно задав функциональные разрешения, которое предотвратит действия, потенциально ведущие к нежелательным последствиям. Использование СВИПК возможно в рамках сетевой инфраструктуры объекта КИИ как в качестве активного контроля исполнения программ, так и в качестве средства сертификация программ перед допуском их исполнения. Ее использование позволяет повысить защищенность информации, циркулирующей в сетях КИИ, за счет использования программ, уровень доверия к которым достаточен для их безопасного использования.

Список литературы

[1]. Указ Президента Российской Федерации от 05.12.2016 № 646 «Об утверждении Доктрины информационной безопасности Российской Федерации».

[2]. Гарбук С. В., Комаров А. А., Салов Е. И. Обзор инцидентов информационной безопасности АСУТП зарубежных государств: Аналитический отчет. ЗАЩИТА ИНФОРМАЦИИ. ИНСАЙД, вып. 6, 2010 г., стр. 50-58.

[3]. Яремчук С. APT: реальность или паранойя? Системный администратор, вып. 7-8, 2012 г., стр. 52-56.

[4]. Довголенко А.А. Социальная инженерия в сети интернет. Информационная безопасность и вопросы профилактики киберэкстремизма среди молодежи Материалы внутривузовской конференции. Под редакцией Г.Н. Чусавитиной, Е.В. Черновой, О.Л. Колобовой, Сборник материалов, 2015 г., стр. 183-191, Магнитогорский государственный технический университет им. Г.И. Носова (Магнитогорск)

[5]. Козачок А.В. Распознавание вредоносного программного обеспечения на основе скрытых Марковских моделей. Диссертация на соискание ученой степени кандидата технических наук. Воронежский государственный технический университет. Орел, 2012, 209 стр.

[6]. Козачок А.В., Кочетков Е.В. Обоснование возможности применения верификации программ для обнаружения вредоносного кода. Вопросы кибербезопасности, вып. 3, 2016 г., стр. 25-32.

[7]. Cimitile A. et al. Model checking for mobile Android malware evolution. Formal Methods in Software Engineering (FormaliSE), 2017 IEEE. ACM 5th International FME Workshop on., 2017, pp. 24-30, IEEE

[8]. Козачок А.В., Кочетков Е.В. Формальная модель функционирования процесса в операционной системе. Труды СПИИРАН, вып. 2, 2017 г., стр. 78-96.

[9]. Kozachok A., Bochkov M., Lai Minh T., Kochetkov E. First order logic for program code functional requirements description. Вопросы кибербезопасности, вып. 3, 2017 г., стр. 2-7.
[10]. Jesse Hertz. Project Triforce: Run AFL on Everything! (online) Доступно по ссылке: https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/projecttriforce-run-afl-on-everything/, 01.11.17
[11]. Институт системного программирования им. В.П. Иванникова РАН. Репозиторий исходного кода эмулятора «Qemu». (online). Доступно по ссылке: https://github.com/ispras/qemu, 09.10.2017
[12]. Documentation/QMP – QEMU. QEMU Machine Protocol. Доступно по ссылке: https://wiki.qemu.org/Documentation/QMP, 01.11.17

Verified program code execution system prototype

A.V. Kozachok <a.kozachok@academ.msk.rsnet.ru>
E.V. Kochetkov <e.kochetkov@academ.msk.rsnet.ru >
Academy of Federal Guard Service,
35, Priborostroitelnaya st., Oryol, 302034, Russia

Abstract. The article represented the technical implementation of the system of verified program code execution. The functional purpose of this system is to investigate arbitrary executable files of the operating system in the absence of source codes in order to provide the ability to control the execution of the program code within the specified functional requirements. The prerequisites for the creation of such a system are described, the user's operating procedure is given according to two typical usage scenarios. A general description of the architecture of the system and the software used for its implementation, the mechanism of interaction of the elements of the system are presented. The model example of implementation this system is presented. Demonstrating the flexible set of functional constraints, based on temporal attribute process action. At the end of the article given a brief comparison with the closest analogues.

Keywords: formal verification, security automata, controlled execution, malware

DOI: 10.15514/ISPRAS-2017-29(6)-1

For citation: Kozachok A.V., Kochetkov E.V. Verified program code execution system prototype. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 6, 2017. pp. 7-24 (in Russian). DOI: 10.15514/ISPRAS-2017-29(6)-1

References

[1]. Ukaz Prezidenta Rossijskoj Federatsii ot 05.12.2016 № 646 «Ob utverzhdenii Doktriny informatsionnoj bezopasnosti Rossijskoj Federatsii» [Decree of the President of the Russian Federation of 05.12.2016 No. 646 «On Approving the Doctrine of Information Security of the Russian Federation»] (in Russian).

[2]. Garbuk S.V., Komarov A.A., Salov E.I. Overview of incidents of information security of SCADA of foreign countries: Analytical report. Zashhita informatsii. Insajd [Data protection. Inside], no. 6, 2010, pp. 50-58 (in Russian).
[3]. Yaremchuk S. APT: Reality or Paranoia. Sistemnyj administrator [System Administrator], no. 7-8, pp. 52-56 (in Russian).
[4]. Dovgolenko A.A. Social engineering in the Internet. Informatsionnaya bezopasnost' i voprosy profilaktiki kiberehkstremizma sredi molodezhi Materialy vnutrivuzovskoj konferentsii. Pod redaktsiej G.N. CHusavitinoj, E.V. CHernovoj, O.L. Kolobovoj [Proc. of Information security and issues of the prevention of cyber extremism among young people Materials of the intra-university conference]. 2015. pp. 183-191 (in Russian)
[5]. Kozachok A.V. Detection of malicious software based on hidden Markov models: PhD thesis. Oryol, 2012, 209 p. (in Russian)
[6]. Kozachok A.V., Kochetkov E.V. Using Program Verification for Detecting Malware. Cybersecurity issues, vol. 16, no. 3, 2016, pp. 25–32 (in Russian)
[7]. Cimitile A. et al. Model checking for mobile Android malware evolution. Formal Methods in Software Engineering (FormaliSE), 2017 IEEE. ACM 5th International FME Workshop on., 2017, pp. 24-30, IEEE
[8]. Kozachok A.V., Kochetkov E.V. Formal model of functioning process in the operating system. SPIIRAS Proceedings, vol. 51, issue 2, 2017, pp. 78-96 (in Russian).
[9]. Kozachok A., Bochkov M., Lai Minh T., Kochetkov E. First order logic for program code functional requirements description. Cybersecurity issues, vol. 3, issue 21, 2017, pp 2-7.
[10]. Jesse Hertz. Project Triforce: Run AFL on Everything! (online) Available at: https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/projecttriforce-run-afl-on-everything/, accessed 01.11.17
[11]. Ivannikov Institute for System Programming of the RAS. Source Repository of the «Qemu». (online) Available at: https://github.com/ispras/qemu, accessed 05.09.17
[12]. Documentation/QMP – QEMU. QEMU Machine Protocol. (online) Available at: https://wiki.qemu.org/Documentation/QMP, accessed 15.10.17

Инкрементальное построение спецификаций моделей окружения и требований для подсистем монолитного ядра операционных систем[1]

И.С. Захаров <ilja.zakharov@ispras.ru>
Е.М. Новиков <novikov@ispras.ru>
Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. Методы и инструменты автоматической статической верификации позволяют выявить все ошибки искомых видов в целевых программах при выполнении определенных предположений даже в условиях отсутствия полных моделей и формальных спецификаций. Эта возможность является основой предлагаемого в работе метода инкрементального построения спецификаций моделей окружения и требований для подсистем монолитного ядра операционных систем. Данный метод был реализован в системе статической верификации Klever и применен для проверки подсистемы поддержки терминальных устройств ядра ОС Linux.

Ключевые слова: операционная система; монолитное ядро; качество программной системы; статическая верификация; формальная спецификация; декомпозиция программной системы; модель окружения.

DOI: 10.15514/ISPRAS-2017-29(6)-2

Для цитирования: Захаров И.С., Новиков Е.М. Инкрементальное построение спецификаций моделей окружения и требований для подсистем монолитного ядра операционных систем. Труды ИСП РАН, том 29, вып. 6, 2017 г., стр. 25-48. DOI: 10.15514/ISPRAS-2017-29(6)-2

1. Введение

Проведенное ранее исследование показало, что одними из наиболее перспективных методов и инструментов для обнаружения всех ошибок искомого вида в монолитном ядре операционных систем (далее — ОС)

[1] Исследование выполнено при финансовой поддержке РФФИ, проект «Инкрементальная статическая верификация подсистем монолитного ядра операционных систем» № 16-31-60097.

являются методы и инструменты автоматической статической верификации [1]. Применение данных методов и инструментов на практике затруднено, поскольку они способны верифицировать программы размером несколько тысяч или десятков тысяч строк кода в зависимости от того, насколько целевой код релевантен проверяемым требованиям, от доступных вычислительных ресурсов и от многочисленных настроек, например, от выбора модели памяти и решателя. Размер монолитного ядра ОС без различных расширений, таких как драйверы устройств, может составлять несколько миллионов строк кода [2], поэтому в качестве первого шага в направлении использования методов и инструментов автоматической статической верификации для проверки качества монолитного ядра ОС были предложены различные подходы к декомпозиции ядра на подсистемы [1].

В данной работе предлагается использовать тот подход к декомпозиции монолитного ядра ОС на подсистемы, который основан на принципе разделения четко выраженной функциональности. Преимуществами данного подхода являются:

- возможность выделять достаточно компактные по размеру подсистемы. Если некоторая подсистема окажется слишком большой или сложной для инструментов автоматической статической верификации, то всегда можно продолжить декомпозицию, основываясь на том же принципе;
- отсутствие необходимости разрабатывать модель функций целевой подсистемы для ее проверки, поскольку инструменты автоматической статической верификации сами строят модель всех функций целевой подсистемы;
- достаточно простая реализация, поскольку для задания подсистем нужно перечислить названия соответствующих директорий и/или файлов с их исходным кодом.

Опыт предыдущего применения инструментов автоматической статической верификации на практике продемонстрировал необходимость моделировать окружение для целевых программ и задавать требования к ним достаточно точным образом [3-6]. При проверке таких сценариев взаимодействия целевой подсистемы монолитного ядра ОС с ее окружением[2], которые никогда не осуществляются на практике в силу тех или иных причин, могут быть выданы ложные сообщения об ошибках. Также инструменты автоматической статической верификации могут пропустить ошибки, если не моделируются какие-то пути выполнения, которые возможны при работе подсистемы в реальном окружении. Аналогично, если требования сформулированы

[2] Окружением для подсистемы монолитного ядра ОС являются другие подсистемы монолитного ядра ОС, различные расширения, такие как драйверы устройств, а также, возможно, косвенным образом аппаратура и приложения из пользовательского пространства.

недостаточно точным образом, то могут быть как выданы ложные сообщения об ошибках, так и пропущены ошибки соответствующих видов.

Примечательным является то, что в отличии от методов и инструментов формальной (дедуктивной) статической верификации, методы и инструменты автоматической статической верификации не предполагают разработку полной модели и формальных спецификаций, которые покрывают все функциональные требования и некоторые дополнительные свойства. Обнаруживать ошибки определенных видов, а также доказывать их отсутствие при выполнении определенных предположений возможно даже при наличии неточных моделей и спецификаций. По мнению авторов работы это обусловлено совокупностью следующих факторов.

- При использовании методов и инструментов автоматической статической верификации традиционно не делается попытка доказать полную формальную корректность целевой программы в смысле соответствия реализации ее модели и спецификациям, а ищутся нарушения достаточно общих требований. К числу таких требований относится, например, выполнение правил безопасного программирования (отсутствие разыменований нулевого указателя, выходов за границы буферов и т.п.) и правил корректного использования программного интерфейса, например, корректное использование специфичного механизма синхронизации для многопоточных программ.
- Проверяется корректность не отдельных функций, а всей программы в целом. При этом для всех функций, входящих в целевую программу, автоматически строится их модель достаточно точная для проверки заданных требований.
- Разработчики постоянно предлагают новые и совершенствуют существующие методы статической верификации, которые уже способны автоматически строить достаточно точные модели и автоматически доказывать выполнимость заданных требований для средних по размеру программ с разумными ограничениями на используемые вычислительные ресурсы и общее время проверки.
- При автоматической статической верификации делаются определенные предположения. Например, инструменты игнорируют код на языке ассемблера. Как правило, это не приводит к проблемам, поскольку такого кода в монолитном ядре ОС не так много [2]. При необходимости можно разработать соответствующую модель на языке программирования Си.

В следующем разделе предлагается метод инкрементального построения спецификаций моделей окружения и требований для подсистем монолитного ядра ОС, который во многом опирается на вышеупомянутую возможность методов и инструментов автоматической статической верификации. Раздел 3

посвящен реализации предложенного метода. В разделе 4 описан процесс инкрементального построения спецификаций моделей окружения и требований для подсистемы поддержки терминальных устройств ядра ОС Linux. Также в данном разделе представлены результаты верификации, полученные на различных этапах данного процесса. В заключении делаются выводы по результатам проведенного исследования и намечаются дальнейшие возможные шаги.

2. Метод инкрементального построения спецификаций моделей окружения и требований для подсистем монолитного ядра ОС

Универсального метода построения моделей и спецификаций для статической верификации программ не существует. Для каждого конкретного проекта подходящий метод выбирается исходя из множества различных ограничений. Наиболее важными по мнению авторов работы являются следующие.

- Возможности и особенности языков моделирования и спецификации, а также непосредственно самих инструментов статической верификации. Например, существует подход, который предлагает разрабатывать программы на специализированном языке программирования таким образом, чтобы спецификации задавались одновременно с реализацией [7]. Такой подход не применим для статической верификации существующих программ. Иногда различные пути, удобные при различных вариантах использования, существуют даже в рамках одного подхода [8]. Часто возможностей существующих инструментов статической верификации оказывается недостаточно: некоторые конструкции языков программирования и их расширений не поддерживаются или для проведения проверки требуется чрезвычайно большое количество вычислительных ресурсов [9-10].
- Квалификация самих верификаторов[3]. Для построения качественных моделей и спецификаций требуется очень хорошее понимание целевой программы, используемых языков моделирования и спецификации, а также возможностей и особенностей инструментов статической верификации. Требования к целевым программам в неформальном виде могут быть не полны или даже противоречивы. Документация, описывающая языки моделирования и спецификации, а также инструменты статической верификации, может быть не достаточно качественной. Как правило, необходимые знания верификаторы приобретают по мере непосредственного выполнения

[3] В данной статье под верификаторами понимаются люди, которые занимаются разработкой моделей, спецификаций, проведением статической верификации и анализом результатов верификации.

конкретных проектов, а не на каких-либо обучающих курсах.

- Различные экономические, организационные и политические ограничения. Известно, что трудоемкость формальной (дедуктивной) верификации чрезвычайно большая и может превосходить трудоемкость разработки программы на порядок [11]. В связи с этим полные модели и формальные спецификации строятся только для наиболее критичных программ или их компонентов общим размером несколько тысяч или десятков тысяч строк кода. В других случаях используют альтернативные методы обеспечения качества программного обеспечения, например, тестирование или статический анализ.

Предлагаемый метод инкрементального построения спецификаций моделей окружения и требований для подсистем монолитного ядра ОС учитывает, с одной стороны, необходимость статической верификации большого, сложного и не всегда хорошо документированного кода, а с другой стороны, возможность методов и инструментов автоматической статической верификации обнаруживать ошибки определенных видов, а также доказывать их отсутствие при выполнении определенных предположений даже при наличии не достаточно точных моделей и спецификаций. Метод состоит из трех шагов, которые рассмотрены далее в соответствующих подразделах.

2.1 Покрытие кода целевых подсистем монолитного ядра ОС

Во введении было отмечено, что инструменты автоматической статической верификации могут пропустить ошибки, если не проверяются какие-то пути выполнения, которые возможны при работе в реальном окружении. Типичным примером служит код, который недостижим из указанной точки входа программы[4] ни на одном из возможных путей выполнения. Для того, чтобы увеличить покрытие кода, необходимо построить соответствующие модели окружения. Например, для программ с функцией *main* в модели окружения требуется вызвать данную функцию со всеми возможными и допустимыми значениями ее аргументов или хотя бы некоторым их подмножеством[5]. Аналогичным образом необходимо вызывать библиотечные функции с различными значениями их аргументов, в правильном порядке и в правильном контексте. При проверке отдельных компонентов или подсистем задача моделирования окружения становится еще более важной, поскольку при анализе вдобавок может отсутствовать как тот код программы, который

[4] Инструменты автоматической статической верификации анализируют целевую программу, начиная с некоторой определенной точки входа. Например, такой точкой входа может быть функция *main*.

[5] Инструменты автоматической статической верификации предлагают специальный механизм, позволяющий задавать сразу все возможные значения определенного базового типа или некоторые диапазоны таких значений.

использует программный интерфейс целевого компонента/подсистемы, так и тот код программы, который используется ими.

В условиях большого количества, сложности и размера подсистем монолитного ядра ОС[6] и недостаточно точного неформального описания требований к их программному интерфейсу, в данной работе для моделирования всевозможных сценариев взаимодействия целевой подсистемы с ее окружением предлагается следующее.

- Анализировать подсистему совместно с теми расширениями монолитного ядра ОС, которые ее используют и для которых разрабатывать модели окружения может быть проще по тем или иным причинам[7]. Например, в случае ядра ОС Linux — вместе с загружаемыми модулями, для которых уже разработаны некоторые достаточно точные спецификации моделей окружения [6].
- Разрабатывать дополнительные спецификации моделей окружения в случае необходимости. Данный шаг является опциональным, однако, как будет показано в следующем разделе, без него не всегда удается увеличить покрытие кода целевой подсистемы монолитного ядра ОС. Если покрыть дополнительный код можно за счет совместного анализа подсистемы с некоторым расширением монолитного ядра ОС, то предлагается воспользоваться данной возможностью в первую очередь, а не разрабатывать дополнительную спецификацию модели окружения.

В обоих случаях в первую очередь необходимо обращать внимание на покрытие наиболее важных и больших участков кода целевой подсистемы монолитного ядра ОС. Например, перед использованием некоторого программного интерфейса обязательно нужно выполнить инициализацию используемых им структур данных. Если одно расширение монолитного ядра ОС позволяет покрыть больше кода целевой подсистемы, чем другое, то предпочтение следует отдать первому.

Стоит отметить, что при таком подходе к покрытию кода целевых подсистем монолитного ядра ОС могут покрываться не все возможные пути выполнения, которые возможны при работе в реальном окружении. Например, некоторый программный интерфейс подсистемы может предоставлять возможности, которые не используются ни одним из выбранных расширений монолитного ядра ОС, а имеющиеся спецификации моделей окружения могут не описывать работу с данным программным интерфейсом или делать это не достаточно

[6] Например, для ядра ОС Linux предполагается порядка 100 подсистем размером порядка 10 тысяч строк кода [1].

[7] Вообще говоря, при таком подходе предполагается, что расширения монолитного ядра ОС корректным образом используют программный интерфейс его подсистем. Выявлять ошибки, которые возникают в противном случае, предполагается другими средствами [5-6, 12-14].

точным образом. Тем не менее это соответствует общей идее метода, поскольку предполагается разрабатывать спецификации моделей окружения инкрементальным образом. При необходимости и возможности спецификации можно доработать, благодаря чему покрыть дополнительный код.

2.2 Уточнение спецификаций моделей окружения

Пути выполнения, которые покрываются в рамках выполнения предыдущего шага метода, могут никогда не осуществляться на практике. Например, из-за неточности спецификаций моделей окружения функции могут быть вызваны с недопустимыми значениями аргументов, в неправильном порядке или в неправильном контексте. Кроме того, некоторые пути выполнения могут быть покрыты, поскольку инструменты автоматической статической верификации делают предположения. Например, предполагается, что функции, для которых при анализе нет ни определений, ни моделей, не имеют побочных эффектов и возвращают произвольное значение допустимое типом. Если в реальности некоторая функция всегда возвращает 0 в случае успеха и некоторое отрицательное целое число, обозначающее код ошибки, то предположение, что функция может также вернуть положительное целое число может привести к анализу невыполнимых путей. В конечном итоге инструменты автоматической статической верификации могут выдать ложные сообщения об ошибках, которые существенно затрудняют анализ результатов верификации.

На втором шаге метода предлагается постепенно уточнять спецификации моделей окружения в той степени, насколько это необходимо для проверки выполнения заданных требований для целевых подсистем монолитного ядра ОС с приемлемым количеством ложных сообщений об ошибках. Например, большинство подсистем обращаются к подсистеме управления памятью, поэтому с большой вероятностью потребуется разработать ее модель[8]. Моделировать те подсистемы, которые не используются целевыми, не потребуется.

2.3 Разработка и уточнение спецификаций требований

В данной работе предлагается проверять в первую очередь те же требования, которые традиционно проверяются при статической верификации расширений монолитного ядра ОС [5-6, 12-13]. К ним относятся правила безопасного программирования, которым должны следовать все программы на языке программирования Си, и правила корректного использования программного интерфейса монолитного ядра ОС. Кроме того, необходимо разрабатывать

[8] В случае с монолитным ядром ОС Linux такая модель уже отчасти разработана, поскольку загружаемые модули также используют подсистему управления памятью [5-6, 12-14].

дополнительные спецификации требований в тех случаях, когда требуется проверять специфичные правила корректности, которым должны следовать только подсистемы монолитного ядра ОС.

Несмотря на сходство требований, проверяемых при статической верификации подсистем монолитного ядра ОС, с требованиями к его расширениям, для проверки подсистем может потребоваться дополнительные уточнения спецификаций. Например, в отличии от расширений монолитного ядра ОС, которые могут загружаться и выгружаться динамически и на момент завершения работы должны освобождать все выделенные для них ресурсы тем или иным способом, подсистемы монолитного ядра ОС не всегда обязаны это делать, поскольку нет смысла освобождать выделенные ресурсы, когда ОС завершает свою работу.

3. Реализация предложенного метода

Предложенный метод был реализован для подсистем ядра ОС Linux в системе статической верификации Klever [12-14]. При этом были задействованы все те возможности, которые уже были реализованы в Klever для поддержки статической верификации загружаемых модулей ядра ОС Linux. Дополнительно были реализованы следующие возможности.

- Для реализации шага метода, представленного в подразделе 2.1:
  - Проверка файлов и директорий с исходным кодом целевой подсистемы ядра ОС Linux совместно с исходным кодом одного из загружаемых модулей.
  - Возможность напрямую вызывать функции программного интерфейса из моделей окружения.
- Для реализации шага метода, представленного в подразделе 2.2:
  - Поддержка нескольких программных интерфейсов, имеющих близкую синтаксическую структуру. Например, могут регистрироваться несколько наборов функций-обработчиков, сохраняемых в переменных с одним и тем же структурным типом.
  - Запрет использования программного интерфейса в моделях окружения в тех случаях, когда он еще не зарегистрирован.
- Для решения проблемы, которая была упомянута в подразделе 2.3:
  - Возможность отключать проверку финального состояния на момент завершения работы подсистемы.

Для загружаемых модулей ядра ОС Linux большинство данных возможностей также необходимы, но они не являются столь существенными. Без отсутствия соответствующей поддержки для подсистем ядра ОС Linux либо не удавалось

достичь приемлемого покрытия кода, либо выдавалось чрезвычайно много ложных сообщений об ошибках.

Помимо доработки компонентов системы статической верификации Klever было исправлено несколько ошибок и сделано несколько дополнительных оптимизаций в инструменте автоматической статической верификации CPAchecker [15].

4. Процесс инкрементального построения спецификаций моделей окружения и требований для подсистемы поддержки терминальных устройств ядра ОС Linux

Реализация предложенного метода инкрементального построения спецификаций моделей окружения и требований для подсистем монолитного ядра ОС была использована для статической верификации подсистемы поддержки терминальных устройств ядра ОС Linux версии 3.14 (далее — TTY-подсистемы) в конфигурации *allmodconfig* на архитектуре *x86_64*. В состав TTY-подсистемы входят 11 файлов общим размером около 12 тысяч строк кода.

В данном разделе описан процесс инкрементального построения спецификаций моделей окружения и требований для данной подсистемы. Для каждого этапа этого процесса представлены соответствующие результаты верификации и сделаны выводы о возможных направлениях для их улучшения.

Все эксперименты проводились на виртуальных машинах OpenStack с 4 виртуальными ядрами процессора модели Intel Xeon E312xx (Sandy Bridge) и 32 ГБ оперативной памяти[9]. После всех доработок версия системы статической верификации Klever была 89d823d[10], а версии инструмента автоматической статической верификации CPAchecker для проверки соответствующих требований следующие:

- Правила корректного использования программного интерфейса ядра ОС Linux и оценка покрытия кода — *ldv-bam:25793*[11].
- Безопасная работа с памятью — *smg_witness_for_ldv:26912*.
- Поиск состояний гонок — *CPALockator:26692*.

Для генерации верификационных задач использовался тот набор спецификаций моделей окружений и требований, а также те параметры, которые используются в указанной версии системы статической верификации Klever по умолчанию. Кроме того, были уточнены некоторые существующие спецификации и разработаны новые (см. подразделы данного раздела). На решение каждой верификационной задачи отводилось максимум 15 минут процессорного времени и 10 ГБ оперативной памяти.

[9] http://www.bigdataopenlab.ru/about.html.

[10] Хэш коммита в Git-репозитории https://forge.ispras.ru/projects/klever/repository.

[11] Здесь и далее в данном списке название ветки и номер ревизии в SVN-репозитории https://svn.sosy-lab.org/software/cpachecker/.

4.1 Покрытие функций TTY-подсистемы при ее проверке с одним загружаемым модулем

В соответствии с предложенным методом TTY-подсистема проверялась вместе с загружаемыми модулями, которые ее использовали. В первую очередь была проведена оценка покрытия ее функций при использовании одного загружаемого модуля *drivers/tty/serial/jsm/jsm.ko*. На данном этапе не проводилась проверка выполнимости какого-либо конкретного требования, так как целью запуска инструмента автоматической статической верификации была оценка покрытия кода сверху. Результаты покрытия функций TTY-подсистемы для данного эксперимента приведены в табл. 1.

Табл. 1. Покрытие функций TTY-подсистемы при ее проверке с загружаемым модулем drivers/tty/serial/jsm/jsm.ko.

Table 1. Function coverage of the TTY subsystem checked together with loadable module drivers/tty/serial/jsm/jsm.ko.

Файл TTY-подсистемы	Процент покрытых функций	Количество покрытых функций	Общее количество функций
drivers/tty/n_tty.c	0%	0	70
drivers/tty/pty.c	10%	3	30
drivers/tty/sysrq.c	10%	5	50
drivers/tty/tty_audit.c	0%	0	12
drivers/tty/tty_buffer.c	0%	0	20
drivers/tty/tty_io.c	9%	11	116
drivers/tty/tty_ioctl.c	0%	0	28
drivers/tty/tty_ldisc.c	0%	0	36
drivers/tty/tty_ldsem.c	0%	0	20
drivers/tty/tty_mutex.c	0%	0	5
drivers/tty/tty_port.c	0%	0	25
Всего	**5%**	**19**	**412**

Детальный анализ покрытия функций TTY-подсистемы показал, что покрытие обеспечено за счет инициализации TTY-подсистемы, которая отчасти схожа с инициализацией загружаемых модулей ядра ОС Linux. Вывод: необходимо анализировать TTY-подсистемы с другими загружаемыми модулями помимо *drivers/tty/serial/jsm/jsm.ko*.

4.2 Покрытие функций TTY-подсистемы при ее проверке с загружаемыми модулями терминальных устройств из директории *drivers/tty*

По аналогии с предыдущем подразделом был проведен эксперимент, в котором TTY-подсистема проверялась вместе со всеми входящими в состав ядра ОС Linux загружаемыми модулями терминальных устройств из директории *drivers/tty*. Всего таких модулей оказалось 38: *drivers/tty/{cyclades.ko, ipwireless/ipwireless.ko, isicom.ko, moxa.ko, mxser.ko, n_gsm.ko, n_hdlc.ko, n_r3964.ko, n_tracerouter.ko, n_tracesink.ko, nozomi.ko, rocket.ko, serial/{8250/{8250.ko, 8250_dw.ko, 8250_pci.ko, serial_cs.ko}, altera_jtaguart.ko, altera_uart.ko, arc_uart.ko, clps711x.ko, fsl_lpuart.ko, ifx6x60.ko, jsm/jsm.ko, kgdboc.ko, max3100.ko, mfd.ko, mrst_max3110.ko, pch_uart.ko, rp2.ko, sccnxp.ko, serial_core.ko, sh-sci.ko, st-asc.ko, timbuart.ko, uartlite.ko}, synclink.ko, synclink_gt.ko, synclinkmp.ko}*. TTY-подсистема проверялась с каждым из этих модулей по-отдельности. Статическая верификация TTY-подсистемы одновременно со всеми этими модулями не рассматривалась ввиду слишком высокой сложности и большого объема исходного кода такого объединения. Результаты покрытия функций TTY-подсистемы приведены в табл. 2.

Табл. 2. Покрытие функций TTY-подсистемы при ее проверке с загружаемыми модулями терминальных устройств из директории drivers/tty.

Table 2. Function coverage of the TTY subsystem checked together with TTY loadable modules from directory drivers/tty.

Файл TTY-подсистемы	Процент покрытых функций	Количество покрытых функций	Общее количество функций
drivers/tty/n_tty.c	0%	0	70
drivers/tty/pty.c	10%	3	30
drivers/tty/sysrq.c	10%	5	50
drivers/tty/tty_audit.c	0%	0	12
drivers/tty/tty_buffer.c	70%	14	20
drivers/tty/tty_io.c	18%	21	116
drivers/tty/tty_ioctl.c	10%	3	28
drivers/tty/tty_ldisc.c	11%	4	36
drivers/tty/tty_ldsem.c	40%	8	20
drivers/tty/tty_mutex.c	0%	0	5
drivers/tty/tty_port.c	28%	7	25
Всего	**16%**	**65**	**412**

Детальный анализ непокрытых функций файла TTY-подсистемы *drivers/tty/tty_port.c* показал следующее[12]:

- Исправление ошибок в компоненте генерации моделей окружения системы статической верификации Klever, которые перечислены в разделе 3, и уточнение спецификаций моделей окружения *tty_operations* (соответствующий заголовочный файл — *include/linux/tty_driver.h*) и *tty_port_operations* (соответствующий заголовочный файл — *include/linux/tty.h*) поможет покрыть 17 из 18 непокрытых функций[13]. Более того, более углубленный анализ показал, что перечисленные выше проблемы могут помешать повысить покрытие функций для всех остальных улучшений, которые представлены далее в этом списке.
- Проверка TTY-подсистемы с другими загружаемыми модулями помимо загружаемых модулей терминальных устройств из директории *drivers/tty* поможет покрыть 16 из 18 непокрытых функций. Однако расширение списка загружаемых модулей может потребовать существенно больше времени на проведение статической верификации, особенно при проверке выполнения большого количества требований.
- Статическая верификация с использованием альтернативной конфигурации, отличной от *allmodconfig*, поможет покрыть 3 из 18 непокрытых функций.
- Статическая верификация загружаемых модулей и подсистем ядра ОС Linux, которые собираются для других архитектур, может помочь покрыть 17 из 18 непокрытых функций. Однако данная возможность не поддерживается системой статической верификации Klever.
- Анализ дополнительных подсистем ядра ОС Linux совместно с TTY-подсистемой мог бы помочь покрыть 13 из 18 непокрытых функций, но данный подход представляется нецелесообразным, поскольку это может достаточно сильно усложнить верификационные задачи из-за увеличения объема анализируемого кода.
- Для покрытия одной функции *tty_port_register_device_attr* требуется использовать многомодульный анализ, то есть анализировать одновременно несколько загружаемых модулей, что ограничено поддерживается системой статической верификации Klever и достаточно сильно усложняет верификационные задачи.

[12] Подробные результаты анализа доступны по следующей ссылке https://getbox.ispras.ru/index.php/s/qvMVWq8HbUUR0v1.

[13] Здесь и далее дается приближенная оценка, поскольку покрытие функций может не всегда увеличиться на указанное значение в силу наличия второстепенных проблем, которые не учитываются при оценке.

Последние три возможности из списка выше в рамках данной работы более не будут рассматриваться, поскольку либо отсутствует необходимая поддержка в системе статической верификации Klever, либо ожидается существенное усложнение задач для инструментов автоматической статической верификации.

Выводы:

- Проверка TTY-подсистемы со всеми загружаемыми модулями терминальных устройств из директории *drivers/tty* увеличила покрытие функций всего на 10%.
- В первую очередь необходимо исправить ошибки в генераторе моделей окружения и уточнить спецификации моделей окружения *tty_operations* и *tty_port_operations*. По результатам этого будет определено, нужно ли анализировать TTY-подсистему с другими загружаемыми модулями помимо загружаемых модулей терминальных устройств из директории *drivers/tty* или в другой конфигурации, отличной от *allmodconfig*.

4.3 Покрытие функций TTY-подсистемы после исправления ошибок в генераторе моделей окружения и спецификаций моделей окружения *tty_operations* и *tty_port_operations*

В рамках данного и всех последующих подразделов TTY-подсистема проверялась вместе со всеми загружаемыми модулями терминальных устройств, которые участвовали в предыдущем эксперименте. Результаты покрытия функций TTY-подсистемы после намеченных в предыдущем подразделе исправлений приведены в табл. 3.

Табл. 3. Покрытие функций TTY-подсистемы после исправления ошибок в генераторе моделей окружения и уточнения спецификаций моделей окружения tty_operations и tty_port_operations.

Table 3. Function coverage of the TTY subsystem obtained with the fixed environment models generator and improved tty_operations and tty_port_operations environment model specifications.

Файл TTY-подсистемы	Процент покрытых функций	Количество покрытых функций	Общее количество функций
drivers/tty/n_tty.c	1%	1	70
drivers/tty/pty.c	96%	29	30
drivers/tty/sysrq.c	20%	10	50
drivers/tty/tty_audit.c	41%	5	12
drivers/tty/tty_buffer.c	80%	16	20
drivers/tty/tty_io.c	83%	97	116

drivers/tty/tty_ioctl.c	75%	21	28
drivers/tty/tty_ldisc.c	97%	35	36
drivers/tty/tty_ldsem.c	85%	17	20
drivers/tty/tty_mutex.c	100%	5	5
drivers/tty/tty_port.c	96%	24	25
Всего	**63%**	**260**	**412**

Детальный анализ непокрытых функций файлов TTY-подсистемы *drivers/tty/{pty.c, tty_audit.c, tty_buffer.c, tty_io.c, tty_ioctl.c, tty_ldisc.c, tty_ldsem.c, tty_port.c}* показал следующее[14].

- Разработка спецификаций моделей окружения *tty_ldisc_ops* (соответствующий заголовочный файл — *include/linux/tty_ldisc.h*), очередей работ (соответствующий заголовочный файл — *include/linux/workqueue.h*) и *class* (соответствующий заголовочный файл — *include/linux/device.h*) поможет покрыть 12, 5 и 1 из 43 непокрытых функций соответственно. При этом отсутствие данных спецификаций моделей окружения может помешать повысить покрытие функций для улучшений, которые представлены далее в этом списке.
- Проверка TTY-подсистемы с другими загружаемыми модулями помимо загружаемых модулей терминальных устройств из директории *drivers/tty* поможет покрыть 6 из 43 непокрытых функций.
- Из-за использования приближений реальное покрытие функций может быть меньше. Например, некоторый конкретный программный интерфейс может считаться покрытым даже несмотря на то, что он не был зарегистрирован, а был зарегистрирован другой программный интерфейс с похожей синтаксической структурой. Подобная неточность может стоить особенно дорого при проверке требований, поскольку при точном анализе будет выяснено, что код функций первого программного интерфейса недостижим, и, соответственно, возможные ошибки в этом коде не будут обнаружены. Для корректного анализа программного интерфейса, который должен покрываться благодаря существующей спецификации моделей окружения *file_operations* (соответствующий заголовочный файл — *include/linux/fs.h*) обязательно необходимо вызывать функцию инициализации TTY-подсистемы *tty_init*. Для спецификации модели окружения *tty_ldisc_ops* — *console_init*.
- Три функции *tty_pair_get_pty*, *tty_throttle* и *ldsem_down_write_trylock* не используются ни в одной подсистеме или загружаемом модуле

[14] Подробные результаты анализа доступны по следующей ссылке https://getbox.ispras.ru/index.php/s/q8VYCYdH8lGon3a.

ядра ОС Linux. Соответственно, покрыть данные функции возможно только за счет их явного вызова из модели окружения. Однако это не кажется целесообразным ввиду отсутствия пользователей данных функций.

- Покрыть функцию *ptmx_open* не удается, поскольку компонент системы статической верификации Klever, который генерирует модели окружения, не поддерживает используемый способ задания программного интерфейса.

Выводы:

- Исправление ошибок в генераторе моделей окружения и уточнение спецификаций моделей окружения *tty_operations* и *tty_port_operations* помогло увеличить покрытие функций TTY-подсистемы почти на 50%.
- В первую очередь необходимо разработать спецификацию модели окружения *tty_ldisc_ops*. Кроме того, модель окружения обязательно должна вызывать функции инициализации TTY-подсистемы *tty_init* и *console_init*.

4.4 Покрытие функций TTY-подсистемы после разработки спецификации модели окружения *tty_ldisc_ops* и вызова функций *tty_init* и *console_init* из модели окружения

С целью дальнейшего повышения покрытия кода, а также для достижимости кода при выполнении более точного анализа, используемого для проверки выполнимости требований, была разработана спецификация модели окружения *tty_ldisc_ops* и из модели окружения были вызваны функции *tty_init* и *console_init*. Получившееся в результате покрытие функций TTY-подсистемы приведено в табл. 4.

Табл. 4. Покрытие функций TTY-подсистемы после разработки спецификации модели окружения tty_ldisc_ops и вызова функций tty_init и console_init из модели окружения.
Table 4. Function coverage of the TTY subsystem obtained with implemented tty_ldisc_ops environment model specification and invocations of tty_init and console_init functions from the environment model.

Файл TTY-подсистемы	Процент покрытых функций	Количество покрытых функций	Общее количество функций
drivers/tty/n_tty.c	98%	69	70
drivers/tty/pty.c	96%	29	30
drivers/tty/sysrq.c	20%	10	50
drivers/tty/tty_audit.c	75%	9	12
drivers/tty/tty_buffer.c	85%	17	20

drivers/tty/tty_io.c	87%	102	116
drivers/tty/tty_ioctl.c	92%	26	28
drivers/tty/tty_ldisc.c	94%	34	36
drivers/tty/tty_ldsem.c	85%	17	20
drivers/tty/tty_mutex.c	100%	5	5
drivers/tty/tty_port.c	92%	23	25
Всего	**83%**	**341**	**412**

В дополнение к результатам предыдущего подраздела детальный анализ непокрытых функций файлов TTY-подсистемы *drivers/tty/{n_tty.c, pty.c, tty_audit.c, tty_buffer.c, tty_io.c, tty_ioctl.c, tty_ldisc.c, tty_ldsem.c, tty_port.c}* показал следующее[15]:

- Разработка спецификаций моделей окружения очередей работ, *class* и *device_attribute* (соответствующий заголовочный файл — *include/linux/device.h*) поможет покрыть 5, 1 и 1 из 31 непокрытых функций соответственно.
- Проверка TTY-подсистемы с другими загружаемыми модулями помимо загружаемых модулей терминальных устройств из директории *drivers/tty* поможет покрыть 3 из 31 непокрытых функций.
- Покрыть 5 из 31 непокрытых функций возможно за счет анализа реализации функций регистрации программных интерфейсов при наличии их моделей.

Выводы:

- Разработка спецификации модели окружения *tty_ldisc_ops* и вызов функций *tty_init* и *console_init* из модели окружения помогли увеличить покрытие функций TTY-подсистемы на 20%.
- Дальнейшие улучшения либо затруднены, либо позволят увеличить покрытие функций несущественным образом (см. дополнительные рассуждения в подразделах 4.2 и 4.3).

4.5 Проверка выполнения требований для TTY-подсистемы

В данной работе для TTY-подсистемы проверялись требования 16 спецификаций: *generic:memory*, *linux:{alloc:{irq, spinlock, usb lock}*, *arch:io*, *drivers:base:{class, dma-mapping}*, *fs:sysfs*, *kernel:{locking:{mutex, rwlock, spinlock}*, *module*, *rcu:update:lock}*, *net:register*, *usb:register}*, *sync:race*. Отсутствие нарушений требований для всех загружаемых модулей терминальных устройств из директории *drivers/tty*, которые проверялись совместно с TTY-подсистемой, было доказано для 4 спецификаций:

[15] Подробные результаты анализа доступны по следующей ссылке https://getbox.ispras.ru/index.php/s/HQLsKieKfpZSIGE.

linux:{alloc:spinlock, fs:sysfs, kernel:{locking:rwlock, rcu:update:lock}}. В дальнейших экспериментах данные спецификации больше не участвовали.

В 4 спецификациях требований с идентификаторами *linux:{alloc:{irq, usb lock}, net:register, usb:register}* были обнаружены ошибки, которые не позволяли генерировать верификационные задачи. После исправления данных ошибок повторилась ситуация, как и для предыдущих 4 спецификаций требований.

Только для одной спецификации требований, *linux:kernel:locking:mutex*, удалось обнаружить 9 нарушений. Однако все они оказались ложными сообщениями об ошибках по причине неточности спецификации модели окружения *tty_operations*, а именно, перед/после вызова функции-обработчика для открытия терминального устройства необходимо захватывать/освобождать мьютекс *legacy_mutex*. Аналогично нужно поступать для некоторых других функций-обработчиков *tty_operations*.

Для 7 спецификаций требований на выполнение проверки не хватило отведенного процессорного времени, причем для 3 из них (*linux:{drivers:base:class, kernel:module}, sync:race*) — для всех загружаемых модулей терминальных устройств из директории *drivers/tty*, которые проверялись совместно с TTY-подсистемой.

Для спецификации требований *generic:memory* для успешного построения модели инструментом автоматической статической верификации потребовалось следующее.

- Для всех загружаемых модулей терминальных устройств из директории *drivers/tty*, которые проверялись совместно с TTY-подсистемой, был включен дополнительный заголовочный файл *linux/user_namespace.h*, который содержит определение типа *struct user_namespace*.
- Для загружаемых модулей *drivers/tty/{cyclades.ko, isicom.ko, moxa.ko, rocket.ko}* в состав верификационных задач был добавлен дополнительный файл ядра *kernel/timer.c* с определением типа *struct tvec_base*, используемого при определение таймеров.

После этого при проверке данной спецификации для 37 из 38 загружаемых модулей терминальных устройств из директории *drivers/tty*, которые проверялись совместно с TTY-подсистемой, были обнаружены утечки памяти. Все выданные нарушения оказались ложными сообщениями об ошибках, поскольку, как было отмечено в подразделе 2.3, для подсистем монолитного ядра ОС не требуется освобождать выделенные ресурсы. Отключение проверки соответствующего требования для *generic:memory* привело к тому, что на проверку перестало хватать процессорного времени для всех загружаемых модулей терминальных устройств из директории *drivers/tty*, которые проверялись совместно с TTY-подсистемой.

4.6 Упрощение моделей окружений для проверки выполнения требований для TTY-подсистемы в случаях нехватки процессорного времени

В предыдущем подразделе было показано, что несмотря на достаточно небольшой размер TTY-подсистемы для проверки выполнения спецификаций некоторых требований не хватает процессорного времени, причем для 4 — для всех загружаемых модулей терминальных устройств из директории *drivers/tty*, которые проверялись совместно с TTY-подсистемой. В рамках данного подраздела исследуются причины данной проблемы, а также рассматриваются некоторые возможности ее преодоления.

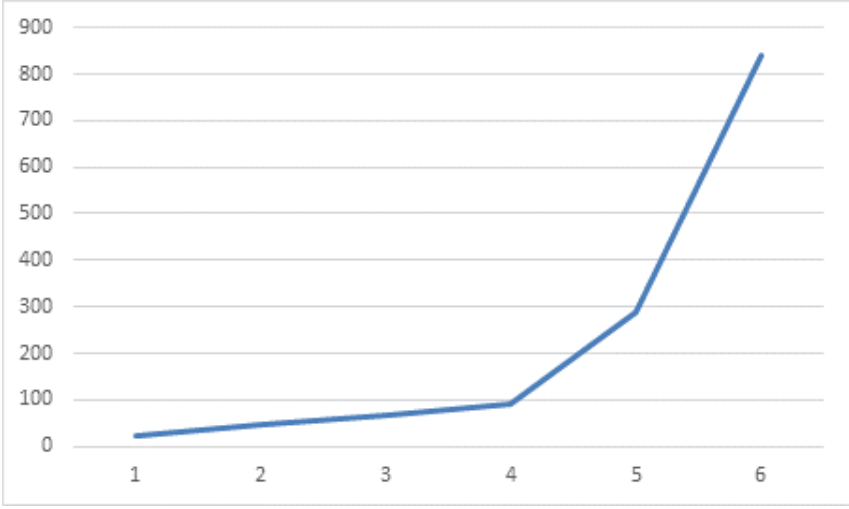


Рис. 1. Зависимость потребления процессорного времени (в секундах) от количества экземпляров программных интерфейсов.
Fig. 1. Usage of CPU time (in seconds) depending on the number of API instances.

Оказалось, что в TTY-подсистеме есть достаточно большое количество экземпляров программных интерфейсов, таких как *tty_operations* и *file_operations*. Для одного из самых требовательных к вычислительным ресурсам требований (отсутствие состояний гонок — *sync:race*) было измерено потребление процессорного времени для различного числа экземпляров программных интерфейсов — рис. 1. Видно, что потребление процессорного времени увеличивается практически экспоненциально, что делает невозможной проверку подсистем ядра ОС Linux и загружаемых модулей с большим количеством экземпляров программных интерфейсов. Для других требований картина может отличаться, однако такое поведение вполне ожидаемо.

Чтобы обойти данную проблему, авторы данного исследования изучили несколько возможных упрощений модели окружения. Во-первых, для

спецификации требований *sync:race*, вместо точной модели окружения, когда для каждого экземпляра программного интерфейса создавался отдельный поток, была использована модель окружения с двумя одинаковыми потоками, которые начинаются непосредственно от начала инициализации TTY-подсистемы и загружаемых модулей ядра ОС Linux. В результате было выдано почти 2,5 тысячи предупреждений, анализ части которых явным образом продемонстрировал, что необходимо использовать точную модель окружения.

Была сделана попытка увеличить максимальный объем вычислительных ресурсов, которые можно использовать на решение каждой верификационной задачи до 150 минут процессорного времени (больше исходного ограничения в 10 раз) и 25 ГБ оперативной памяти (больше исходного ограничения в 2,5 раза). Соответствующий эксперимент для *linux:{drivers:base:class, kernel:{locking:mutex, module}}* позволил обнаружить одно нарушение требования спецификации *linux:drivers:base:class*, которое оказалось ложным сообщением об ошибке (причина и способ его устранения рассмотрены далее). Однако для всех остальных верификационных задач по-прежнему не хватало процессорного времени.

Применительно ко всем спецификациям требований была сделана попытка ограничить количество экземпляров программных интерфейсов. Во-первых, были определены и запрещены те экземпляры программных интерфейсов, для которых регистрация не поддерживалась в спецификациях моделей окружения или была недостижима. Для TTY-подсистемы таких экземпляров оказалось 3, для всех загружаемых модулей терминальных устройств из директории *drivers/tty*, которые проверялись совместно с TTY-подсистемой, — 8. Проверка спецификации требований *generic:memory* после этого показала, что этого упрощения модели окружения не достаточно, поскольку снова не хватило процессорного времени на решение всех верификационных задач.

Радикально сократить время статической верификации помог перебор связанных по регистрации экземпляров программных интерфейсов (то есть рассматривались одновременно только те экземпляры, один из которых регистрируется при инициализации, а остальные регистрируются друг в друге). При этом количество верификационных задач увеличилось с 38 до 55 и получились следующие результаты верификации.

- Для *generic:memory* удалось обнаружить одну ошибку в загружаемом модуле *drivers/tty/serial/mfd.ko* (независимо для 2 верификационных задач). Также было выдано 2 ложных сообщения об ошибках из-за неточности анализа. Отсутствие ошибок удалось доказать для 14 верификационных задач, а для решения оставшихся 37 верификационных задач по-прежнему не хватило процессорного времени. Такого результата следовало ожидать, поскольку для проверки данной спецификации требований используется наиболее тяжеловесный анализ среди всех рассматриваемых спецификаций

требований. Увеличение ограничения на максимальное процессорное время с 15 до 60 минут позволило доказать корректность еще для 5 из 37 верификационных задач.

- Для *sync:race* удалось обнаружить две ошибки в загружаемых модулях *drivers/tty/serial/{st-asc.ko, pch_uart.ko}* (2 верификационные задачи). Для 8 верификационных задач, включая одну из предыдущих, были выданы ложные сообщения об ошибках из-за неточности анализа (7) и модели окружения (1). Отсутствие ошибок удалось доказать для 28 верификационных задач. Решение 4 верификационных задач завершилось неуспешно по разным причинам. Для решения 14 верификационных задач по-прежнему не хватило процессорного времени.
- При проверке спецификации требований *linux:kernel:locking:mutex* было выдано 2 ложных сообщения об ошибках с той же причиной, которая была указана в предыдущем подразделе. Для 46 верификационных задач была доказана корректность, для 7 — не хватило процессорного времени на проверку.
- Для спецификации требований *linux:kernel:module* было выдано 1 ложное сообщение об ошибке из-за неточности модели окружения. Для 49 верификационных задач была доказана корректность, для 4 — не хватило процессорного времени на проверку, для 1 — возникла проблема при обработке трассы ошибки.
- Для *linux:drivers:base:class* было выдано большое количество ложных сообщений об ошибках из-за проблемы, которая указана в подразделе 2.3, поэтому потребовалось запретить проверку финального состояния. После этого для всех верификационных задач было доказано отсутствие нарушений соответствующих требований.
- Для *linux:arch:io* было обнаружено 4 ошибки в загружаемых модулях *drivers/tty/{cyclades.ko, serial/{altera_jtaguart.ko, arc_uart.ko, mfd.ko}}* (для 5 верификационных задач). Для 3 верификационных задач были выданы ложные сообщения об ошибках из-за неточности анализа (2) и модели окружения (1). Для 38 — доказана корректность. Для 9 — не хватило процессорного времени на проверку.
- Для всех загружаемых модулей, при статической верификации которых для *linux:drivers:base:dma-mapping* и *linux:kernel:locking:spinlock* не хватало процессорного времени, перебор связанных по регистрации экземпляров программных интерфейсов помог доказать корректность.

Таким образом, сделанные упрощения моделей окружений позволили существенным образом улучшить результаты верификации, в том числе было обнаружено 7 ошибок в загружаемых модулях терминальных устройств из директории *drivers/tty*, которые проверялись совместно с TTY-подсистемой.

Из-за недостаточной точности анализа инструмента автоматической статической верификации были выданы ложные сообщения об ошибках для 9 верификационных задач, из-за неточности моделей окружения — для 4 верификационных задач.

Обнаружить нарушения проверяемых требований в самой TTY-подсистеме во время проведения экспериментов не удалось. Это демонстрирует высокое качество подсистем ядра ОС Linux. Кроме того, размер исходного кода рассматриваемых загружаемых модулей на порядок превышает размер целевой подсистемы. Ошибки в TTY-подсистеме могут быть найдены, если продолжить улучшать спецификации моделей окружения и требований, а также предлагать новые подходы для сокращения сложности верификационных задач.

5. Заключение

В данной работе был предложен метод инкрементального построения спецификаций моделей окружения и требований для подсистем монолитного ядра ОС, который позволяет использовать методы и инструменты автоматической статической верификации для поиска ошибок в этих подсистемах.

При проведении данного исследования были доработаны компоненты системы статической верификации Klever, а также уточнены существующие и разработаны новые спецификации моделей окружения и требований. Практически все сделанные улучшения в большей или меньшей степени полезны для статической верификации загружаемых модулей ядра ОС Linux, поэтому с течением времени планируется обобщить их и поддерживать для различных вариантов использования.

На различных этапах работы были выявлены и сообщены разработчикам новые проблемы в инструменте автоматической статической верификации CPAchecker. Все эти проблемы были устранены.

Реализация предложенного метода инкрементального построения спецификаций моделей окружения и требований для подсистем монолитного ядра ОС была использована для статической верификации подсистемы поддержки терминальных устройств ядра ОС Linux версии 3.14, которая проверялась совместно с загружаемыми модулями терминальных устройств из директории *drivers/tty*. Благодаря построенным спецификациями удалось повысить покрытие функций целевой подсистемы с 5% до 83% и обнаружить 7 ошибок в рассматриваемых загружаемых модулях. О наличии данных ошибок планируется сообщить разработчикам ядра ОС Linux, если они еще не были исправлены в последних версиях. Ошибки в целевой подсистеме ядра ОС Linux выявлены не были.

Авторы работы планируют продолжить исследование совместно с разработчиками инструментов автоматической статической верификации. В первую очередь необходимо исправить известные проблемы, которые затрудняют или делают невозможным процесс инкрементального построения

спецификаций моделей окружения и требований для подсистем ядра ОС Linux, например:

- поддержать генерацию обращений к программному интерфейсу даже при наличии определения функции, ответственной за его регистрацию;
- поддержать возможность захвата/освобождения блокировок (вызова соответствующих функций) перед/после вызова функций-обработчиков из моделей окружения.

Кроме того, целесообразно расширить область применения предложенного метода и разработанного инструментария на несколько других подсистем ядра ОС Linux.

6. Благодарность

Авторы данной работы выражают благодарность Павлу Андрианову, Антону Васильеву, Владимиру Гратинскому и Алексею Полушкину из команды Linux Driver Verification[16] за участие в обсуждении проблем инкрементальной статической верификации подсистем монолитного ядра операционных систем, за помощь при проведении анализа результатов верификации, а также за поддержку в доработке компонентов системы статической верификации Klever и инструмента автоматической статической верификации CPAchecker.

Список литературы

[1]. Е.М. Новиков. Возможности статической верификации монолитного ядра операционных систем. Труды ИСП РАН, том 29, вып. 2, 2017 г., стр. 97-116. DOI: 10.15514/ISPRAS-2017-29(2)-4

[2]. Е.М. Новиков. Развитие ядра операционной системы Linux. Труды ИСП РАН, том 29, вып. 2, 2017 г., стр. 77-96. DOI: 10.15514/ISPRAS-2017-29(2)-3

[3]. D. Engler, M. Musuvathi. Static analysis versus model checking for bug finding. In Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'04), LNCS, volume 2937, pp. 191-210, 2004.

[4]. T. Ball, S.K. Rajamani. SLIC: A specification language for interface checking of C. Technical Report MSR-TR-2001-21, Microsoft Research, 2001.

[5]. Е.М. Новиков. Развитие метода контрактных спецификаций для верификации модулей ядра операционной системы Linux. Диссертация на соискание ученой степени кандидата физико-математических наук, Институт системного программирования РАН, 2013.

[6]. A. Khoroshilov, V. Mutilin, E. Novikov, I. Zakharov. Modeling Environment for Static Verification of Linux Kernel Modules. In Proceedings of the 9th International Ershov Informatics Conference (PSI'14), LNCS, vol. 8974, pp. 400-414, 2014.

[7]. K.R.M. Leino. Developing verified programs with Dafny. In Proceedings of the 2013 International Conference on Software Engineering (ICSE '13), pp. 1488-1490, 2013.

[8]. П.Н. Девянин, В.В. Кулямин, А.К. Петренко, А.В. Хорошилов, И.В. Щепетков. Сравнение способов декомпозиции спецификаций на Event-B. Программирование, т. 42, № 4, стр. 17-26, 2016.

[16] http://linuxtesting.ru/ldv.

[9]. М.У. Мандрыкин, В.С. Мутилин. Обзор подходов к моделированию памяти в инструментах статической верификации. Труды ИСП РАН, том 29, вып. 1, 2017 г., стр. 195-230. DOI: 10.15514/ISPRAS-2017-29(1)-12

[10]. М.У. Мандрыкин, А.В. Хорошилов. О дедуктивной верификации Си программ, работающих с разделяемыми данными. Труды ИСП РАН, том 27, выпуск 4, стр, 49-68, 2015. DOI: 10.15514/ISPRAS-2015-27(4)-4

[11]. G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, G. Heiser. Comprehensive formal verification of an OS microkernel. ACM Transactions on Computer Systems (TOCS), vol. 32, issue 1, 70 p., 2014.

[12]. Д. Бейер, А.К. Петренко. Верификация драйверов операционной системы Linux. Труды ИСП РАН, том 23, стр. 405-412, 2012. DOI: 10.15514/ISPRAS-2012-23-23

[13]. И.С. Захаров, М.У. Мандрыкин, В.С. Мутилин, Е.М. Новиков, А.К. Петренко, А.В. Хорошилов. Конфигурируемая система статической верификации модулей ядра операционных систем. Программирование, том 41, № 1, стр. 44-67, 2015.

[14]. E. Novikov, I. Zakharov. Towards Automated Static Verification of GNU C Programs. In Proceedings of the 11th International Ershov Informatics Conference (PSI'17), LNCS, volume 10742, 2018 (в печати).

[15]. D. Beyer, M.E. Keremoglu. CPAchecker: A tool for configurable software verification. In: Proceedings of the 23rd International Conference on Computer Aided Verification, Berlin, Heidelberg, Springer, pp. 184-190, 2011.

Incremental development of environment model and requirement specifications for subsystems of operating system monolithic kernels

I.S. Zakharov <ilja.zakharov@ispras.ru>
E.M. Novikov <novikov@ispras.ru>
Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. Methods and tools for automated static verification aim at detecting all violations of checked requirements in target programs under certain assumptions even without complete models and formal specifications. The given feature form a basis of the suggested method for incremental development of environment model and requirement specifications for subsystems of operating system monolithic kernels. This method was implemented on top of static verification framework Klever. It was evaluated by checking the Linux kernel TTY subsystem. During this study some Klever components were improved. Besides, we fixed some existing and developed new environment model and requirement specifications. Almost all made changes also helps at static verification of loadable modules of the Linux kernel. Developers of automated static verification tool CPAchecker fixed several issues that we revealed and reported during the research. Overall developed specifications allowed to increase function coverage of the TTY subsystem from 5% to 83%. Moreover, we revealed 7 bugs in loadable modules verified together with the TTY subsystem.

Keywords: operating system; monolithic kernel; software quality; static verification; formal specification; program decomposition; environment model.

DOI: 10.15514/ISPRAS-2017-29(6)-2

For citation: Zakharov I.S., Novikov E.M. Incremental development of environment model and requirement specifications for subsystems of operating system monolithic kernels. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 6, 2017, pp. 25-48 (in Russian). DOI: 10.15514/ISPRAS-2017-29(6)-2

References

[1]. E.M. Novikov. Static verification of operating system monolithic kernels. Trudy ISP RAN/Proc. ISP RAS, vol, 29, issue 2, pp. 97-116, 2017 (in Russian). DOI: 10.15514/ISPRAS-2017-29(2)-4

[2]. E.M. Novikov. Evolution of the Linux kernel. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 2, pp. 77-96, 2017 (in Russian). DOI: 10.15514/ISPRAS-2017-29(2)-3

[3]. D. Engler, M. Musuvathi. Static analysis versus model checking for bug finding. In Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'04), LNCS, volume 2937, pp. 191-210, 2004.

[4]. T. Ball, S.K. Rajamani. SLIC: A specification language for interface checking of C. Technical Report MSR-TR-2001-21, Microsoft Research, 2001.

[5]. E.M. Novikov. Development of Contract Specifications Method for Verification of Linux Kernel Modules. PhD thesis, Institute for System Programming of the Russian Academy of Sciences, 2013 (in Russian).

[6]. A. Khoroshilov, V. Mutilin, E. Novikov, I. Zakharov. Modeling Environment for Static Verification of Linux Kernel Modules. In Proceedings of the 9th International Ershov Informatics Conference (PSI'14), LNCS, vol. 8974, pp. 400-414, 2014.

[7]. K.R.M. Leino. Developing verified programs with Dafny. In Proceedings of the 2013 International Conference on Software Engineering (ICSE '13), pp. 1488-1490, 2013.

[8]. P.N. Devyanin, V.V. Kulyamin, A.K. Petrenko, A.V. Khoroshilov, I.V. Shchepetkov. Comparison of Specification Decomposition Methods in Event-B. Programming and Computer Software, vol. 42, issue 4, pp. 17-26, 2016.

[9]. M.U. Mandrykin, V.S. Mutilin. Survey of memory modeling methods in static verification tools. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 1, pp. 195-230, 2017 (in Russian). DOI: 10.15514/ISPRAS-2017-29(1)-12

[10]. M.U. Mandrykin, A.V. Khoroshilov. Towards Deductive Verification of C Programs with Shared Data. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 4, pp. 49-68, 2015 (in Russian). DOI: 10.15514/ISPRAS-2015-27(4)-4

[11]. G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, G. Heiser. Comprehensive formal verification of an OS microkernel. ACM Transactions on Computer Systems (TOCS), vol. 32, issue 1, 70 p., 2014.

[12]. D. Bejer, A.K. Petrenko. Linux Driver Verification. Trudy ISP RAN/Proc. ISP RAS, vol. 23, pp. 405-412, 2012 (in Russian). DOI: 10.15514/ISPRAS-2012-23-23

[13]. I.S. Zakharov, M.U. Mandrykin, V.S. Mutilin, E.M. Novikov, A.K. Petrenko, A.V. Khoroshilov. Configurable toolset for static verification of operating systems kernel modules. Programming and Computer Software, vol. 41, issue 1, pp. 49-64, 2015. DOI: 10.1134/S0361768815010065

[14]. E. Novikov, I. Zakharov. Towards Automated Static Verification of GNU C Programs. In Proceedings of the 11th International Ershov Informatics Conference (PSI'17), LNCS, volume 10742, 2018 (to appear).

[15]. D. Beyer, M.E. Keremoglu. CPAchecker: A tool for configurable software verification. In: Proceedings of the 23rd International Conference on Computer Aided Verification, Berlin, Heidelberg, Springer, pp. 184-190, 2011.

Формальная верификация библиотечных функций ядра Linux[1]

[1]Д.В. Ефремов <defremov@hse.ru>
[2]М.У. Мандрыкин <mandrykin@ispras.ru>
[1]НИУ Высшая школа экономики,
101000, Россия, г. Москва, ул. Мясницкая, д. 20
[2]Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. В статье авторами рассматриваются результаты дедуктивной верификации набора из 26 библиотечных функций ядра ОС Linux с помощью стека инструментов AstraVer. В набор включены преимущественно функции, работающие с данными строкового типа. Целью верификации является доказательство свойств функциональной корректности. В статье рассматриваются аналогичные работы по верификации, сравниваются полученные результаты, рассматривается ряд проблем, с которыми сталкивались авторы предыдущих работ, в том числе проблемы, с которыми удалось справится в рамках данной работы и те, которые все ещё препятствуют успешной верификации. Также предлагается методология разработки спецификаций, примененная для рассматриваемого набора функций, которая включает некоторые шаблонные приёмы разработки спецификаций. Авторам удалось доказать полную корректность двадцати пяти функций. В статье приведены результаты доказательства полученных условий верификации каждой функции с помощью нескольких современных SMT-солверов.

Ключевые слова: статический анализ; формальная верификация; дедуктивная верификация; функции стандартной библиотеки.

DOI: 10.15514/ISPRAS-2017-29(6)-3

Для цитирования: Ефремов Д.В, Мандрыкин М.У. Формальная верификация библиотечных функций ядра Linux. Труды ИСП РАН, том 29, вып. 6, 2017 г., стр. 49-76. DOI: 10.15514/ISPRAS-2017-29(6)-3

1. Введение

Ошибки и дефекты в критически важных компонентах операционной системы могут привести к полной её компрометации. Одним из средств повышения

[1] Эта работа поддержана грантом РФФИ 15-01-03024.

уровня доверия к системе является дедуктивная верификация. Она позволяет для отдельных типов дефектов доказать их отсутствие в исследуемом исходном коде.

Код ядра Linux написан на языке Си. В коде ядра широко используется перетипирование указателей, указатели на функции, битовые операции, нестандартные расширения и другие возможности языка Си, осложняющие дедуктивный анализ соответствующих фрагментов Си-кода. Инструменты дедуктивной верификации, как правило, либо не поддерживают такие возможности языка Си, либо не позволяют доказывать для использующих их участков кода соответствие нетривиальным спецификациям функциональной корректности. Большинство инструментов дедуктивной верификации Си-кода налагают явные или неявные ограничения на синтаксис языка и общий стиль кодирования [1].

При верификации кода ядра Linux приходится сталкиваться со многими ограничениями инструментов дедуктивной верификации, их недостаточной зрелостью для полноценной поддержки подобного кода. Большинство ограничений может быть снято в том случае, если разрешена модификация кода. В таком случае возможно заменить неподдерживаемые конструкции или же участки кода, плохо поддающиеся дедуктивной верификации, семантически эквивалентным кодом, который легко обрабатывается инструментами.

В проектах по верификации это считается приемлемым и желательным, так как стоимость доработки инструментов многократно превышает стоимость переписывания части кода проекта.

Однако при верификации, например, модуля ядра Linux невозможно снять все подобные ограничения, так как модуль, являясь частью самого ядра, основывается на его структурах данных, интерфейсе и наследует общий стиль кодирования ядра.

По этой причине в данной работе авторы уделяют особое внимание тому, чтобы доказывать код в максимально неизменном виде.

Одной из основных целей проекта AstraVer [2] является доработка инструментов дедуктивной верификации для работы с кодом ядра Linux. Несмотря на существенные продвижения [3, 4] в разработке инструментов, на момент начала данной работы не существовало цельного и репрезентативного набора тестовых примеров, наглядно отражающего текущее состояние прогресса в направлении достижения целей этого проекта. Данная работа призвана восполнить этот пробел.

Хотя выбранные для анализа библиотечные функции не полностью покрывают сложные для анализа конструкции, используемые в коде ядра Linux, рассматриваемый в работе набор достаточно репрезентативен и позволяет как производить сравнение с аналогичными работами [5, 6], так и наглядно показывать основные сложности моделирования Си-кода с помощью инструментов дедуктивной верификации. Данный набор позволил выявить целый ряд ошибок и недостатков в инструментах AstraVer, которые на

текущий момент уже были исправлены. Также были выявлены некоторые фундаментальные ограничения, присущие используемым в инструментах моделям сущностей языка Си, таких как значения целочисленных типов данных, указатели, адреса, выделенные блоки памяти и др. Эти ограничения не позволяют доказать некоторые свойства реализаций алгоритмов и подробно рассматриваются в данной статье.

Основные результаты данной работы состоят в следующем:

- Были выявлены и подробно рассмотрены основные ограничения методов моделирования целых чисел и указателей в инструментах дедуктивной верификации AstraVer.
- Был разработан и применён на практике ряд приёмов спецификации и доказательства корректности, облегчающий применение инструментов дедуктивной верификации к коду ядра ОС Linux. В том числе использование дублирующих спецификаций, формализующих поведение целевых реализационных функций как с помощью постусловий, так и с помощью модельных функций, а также разработка спецификаций функций на основе их реализаций и контекстов использования.
- Была успешно доказана корректность 26 функций ядра Linux, 23 из которых были доказаны без какого-либо изменения их исходного кода.

Также в рамках работы была предложена, реализована и применена на практике новая композитная модель целых чисел для модуля дедуктивной верификации AstraVer, облегчающая моделирование операций с беззнаковыми целыми и преобразований между различными целочисленными типами данных.

Насколько известно авторам данной статьи, библиотечные функции ядра Linux ранее не анализировались методом дедуктивной верификации систематически. Основной практический результат данной работы — разработанные спецификации библиотечных функций ядра Linux и доказательство соответствия реализаций функций разработанным спецификациям с помощью инструментов дедуктивной верификации AstraVer.

Код функций, спецификации и протоколы доказательств выложены в открытом доступе вместе с инструкциями по воспроизведению результата [7]. Спецификации с исходным кодом могут в дальнейшем служить тестовым набором для инструментов дедуктивной верификации и солверов.

В статье приводится сравнение результатов данной работы с работами по дедуктивной верификации стандартных библиотечных функций klibc [5] и OpenBSD [6].

Статья построена следующим образом: в секции 2 даётся описание применяемых в работе инструментов верификации; в секции 3

рассматриваются аналогичные проекты по разработке спецификаций; в секции 4 кратко описан использованный язык спецификаций ACSL; в секции 5 описываются изменения, которые были внесены в инструменты верификации в ходе данной работы; в секции 6 описываются подходы к разработке спецификаций, которые были выработаны и применены в процессе работы; в секции 7 описываются проблемы, которые не удалось решить; в секции 8 представлены результаты работы.

2. Инструменты верификации

Существуют различные инструменты для дедуктивной верификации программ, в том числе и для верификации кода на языке Си. Инструмент FRAMA-C [8] является фреймворком, позволяющим реализовывать и комбинировать разные виды статического анализа. Для аннотирования кода FRAMA-C использует специальный язык ANSI/ISO C Specification Language [9]. Это язык описания поведения. Он поддерживает написание контрактов функций (на уровне пред- и постусловий), а также инвариантов циклов и аксиоматических теорий. FRAMA-C интегрирует спецификации и код в единое синтаксическое дерево, с которым работают плагины. Для FRAMA-C существует несколько плагинов для дедуктивной верификации кода: WP [8], JESSIE [10] и ASTRAVER [2]. Последний плагин является форком JESSIE. Одна из основных целей, которая заявлена его авторами — это доработка инструмента для верификации кода ядра Linux. Для доказательства функций авторы использовали плагин ASTRAVER.

Плагин дедуктивной верификации ASTRAVER (как и JESSIE) транслирует внутреннее представление FRAMA-C в модель программы на языке WhyML [11], на основе реализованных в нем моделей памяти и операций с числами.

Инструмент WHY3 генерирует условия верификации для программы на языке Why3ML и преобразует их во входные задания для солверов. Среди поддерживаемых WHY3 — такие солверы, как ALT-ERGO, CVC3, CVC4, Z3, SPASS, EPROVER, SIMPLIFY и большое количество других. WHY3 также поддерживает ряд трансформаций для условий верификации, например, подразбиение на отдельные условия верификации по конъюнкциям.

3. Аналогичные работы

Так как инструменты дедуктивной верификации WP и JESSIE являются достаточно зрелыми, ранее они уже успешно применялись для верификации реального кода. Так, в работе [6] доказывается корректность 12 стандартных функций строкового типа, реализованных в OpenBSD. Авторами используется JESSIE в качестве плагина дедуктивной верификации. Полностью корректность (валидность всех условий верификации) удалось доказать для 7 функций, для остальных 5 функций несколько условий верификации остались

недоказанными. Особенностью работы является то, что автор для каждой функции делал три итерации по разработке спецификационного контракта: первую — на основе стандарта и опыта самого автора, вторую — на основе документации (`man`), третью - на основе документации и кода функции. Последняя версия спецификации практически в каждом случае сильно отличалась от первых двух. Это показывает, что разработать формальную спецификацию на существующую реализацию, без доступа к самой реализации, достаточно сложно. Однако подобный итерационный подход позволил автору найти неточности в документации к нескольким функциям, а также отсутствие полноты описания поведения функции в ряде случаев.

Для некоторых функций автору потребовалось внести изменения в их код. Это было связано с двумя конкретными ситуациями. В первой указатели на тип `char` в функциях `strcmp` и `strncmp` приводились к указателям на `unsigned char`. Во второй в функции `strlcat` итератор цикла переполнялся на последнем шаге итерации за счет постфиксного декремента, что приводило к невозможности доказать условие верификации на целочисленное переполнение, хотя и не вело к ошибке при выполнении кода функции. Для доказательства условий верификации использовались солверы Alt-Ergo (0.7.3), Simplify (1.5.4) и Z3 (2.0).

В статье [5] авторы используют Frama-C с плагином дедуктивной верификации WP для верификации кода функций библиотеки klibc. Авторам удалось полностью доказать корректность 14 функций, работающих с данными строкового типа, для 12 функций не удалось доказать часть условий верификации, ещё на 4 функциях проявили себя ошибки в инструментах, воспрепятствовавшие их полноценному запуску. Помимо функций для строкового типа (из `string.h`) в работе также анализировались функции из `stdio.h`. Как отмечается самими авторами, практически все функции из данного заголовочного файла используют системные вызовы, что в конечном итоге ведёт к тому, что спецификации для них получаются слабыми (weak). При работе с кодом функций авторы вносили в него изменения, которые позволяли обходить ограничения применяемых инструментов или упрощать результирующие условия верификации.

Так, авторы заранее изучили проблемы с моделированием перетипирования указателей (type casts), например, `unsigned char*` в `char*`, и старались изменить код, для того чтобы исключить подобные операции.

Помимо этого, проблемы вызвал повторяющийся шаблон кода, где в цикле `while` осуществляется постфиксное уменьшение значения переменной беззнакового типа. Выход из цикла в таком случае происходит, когда переменная равна нулю, но после сравнения всё равно осуществляется уменьшение значения переменной на единицу. В случае беззнакового типа это приводит целочисленному переполнению. В реальности, однако, целочисленное переполнение не ведёт к ошибке в коде функции, так как после

выхода из цикла переменная нигде не используется. Но в такой ситуации не удаётся доказать условие верификации, требующее отсутствия переполнения.

Для доказательства условий верификации использовались солверы ALT-ERGO (0.95.1), CVC3 (2.4.1), Z3 (4.3.1).

Самым развёрнутым документом по разработке спецификаций на языке ACSL является ACSL by Example [12]. В нем доказывается корректность функций из стандартной библиотеки C++. Функции перед разработкой спецификаций переписываются с обобщённой реализации на шаблонах в функции языка Си, работающие на массивах элементов типа `int`. Разработчики регулярно обновляют отчёт добавлением новых функций со спецификациями к ним, исправляют ошибки в прошлых спецификациях и перерабатывают их. Проект длится с 2009 года. Документ содержит большое количество полностью доказанных функций. В качестве солверов используются ALT-ERGO, CVC3, CVC4, Z3, EPROVER. Используется плагин дедуктивной верификации WP.

Отчёт компании GrammaTech [13] содержит описание типовых проблем, с которыми столкнулись авторы при разработке спецификаций к реализации стандартной библиотеки GT libc. Они использовали FRAMA-C с плагином дедуктивной верификации WP. Среди прочего авторами описываются проблемы моделей памяти, возникающие при верификации кода с перетипированием указателей, а также со сравнением указателей.

4. Язык спецификаций ACSL

ACSL является языком спецификации поведения интерфейсов (BISL, Behavioral Interface Specification Language) [14], реализованным во FRAMA-C. ACSL разработан специально для спецификации свойств Си-программ и подходит для написания контрактных спецификаций (пред- и постусловий), формализации свойств безопасности (предикатов на достижимые состояния программы), а также для задания дополнительных спецификаций, необходимых инструменту верификации для проверки контрактных спецификаций и свойств безопасности. ACSL включает средства выражения специфичных для языка Си аспектов управления памятью, таких как адреса и длины выделяемых блоков памяти, преобразования типов указателей, доступность областей памяти для чтения/записи и др. Многие же высокоуровневые логические средства спецификации состояния памяти и поведения программ, такие как сепарационная логика, разрешения (permissions) или поддержка наследования поведения (например, уточнения контрактов функций), непосредственно в язык ACSL не включены. Некоторые из них иногда могут быть выражены с использованием возможностей самого ACSL. При этом уровень поддержки языка ACSL со стороны инструментов верификации, реализованных на основе платформы FRAMA-C может различаться.

Рассмотрим пример контрактной спецификации на языке ACSL. В листинге 1 приведён код функции `strnchr` из ядра Linux. Функция `strnchr`

осуществляет поиск символа c в строке `s`, которая ограничена длиной `cnt`. В спецификации функции требуется, чтобы указатель на строку `s` адресовал валидный участок памяти, размером `min(strlen(s), cnt) + 1`. Это условие является предусловием функции и указано в первой строке спецификации. Функция `strnchr` является чистой, то есть не имеет побочных эффектов, что описывается второй строкой спецификации.

```
1  /*@ requires valid_strn(s, count);
2      assigns \nothing;
3      behavior exists:
4        assumes ∃char *p;
5          s ≤ p < s + strnlen(s, count) ∧ *p ≡ (char %) c;
6        ensures s ≤ \result ≤ s + strnlen(s, count);
7        ensures *\result ≡ (char %) c;
8        ensures ∀ char *p; s ≤ p < \result ⇒ *p ≢ (char %) c;
9      behavior not_exists:
10       assumes ∀ char *p;
11         s ≤ p < s + strnlen(s, count) ⇒ *p ≢ (char %) c;
12       ensures \result ≡ \null;
13     complete behaviors;
14     disjoint behaviors;*/
15 char *strnchr(const char *s, size_t count, int c) {
16  //@ ghost char *os = s;
17  //@ ghost size_t ocount = count;
18  /*@ loop invariant 0 ≤ count ≤ ocount;
19      loop invariant os ≤ s ≤ os + strnlen(os, ocount);
20      loop invariant s - os ≡ ocount - count;
21      loop invariant valid_strn(s, count);
22      loop invariant strnlen(os, ocount) ≡ s - os + strnlen(s, count);
23      loop invariant ∀ char *p; os ≤ p < s ⇒ *p ≢ (char %) c;
24      loop variant count;
25   */
26  for (; count-- /*@%*/ && *s != '\0'; ++s)
27   if (*s == (char)/*@%*/c)
28    return (char *)s;
29  return NULL;
30 }
```

Листинг 1. Функция `strnchr`. Ядро Linux 4.12, файл lib/string.c
Listing 1. Linux kernel 4.12, file файл lib/string.c

Далее спецификация подразбивается на два случая. Первый — когда в строке существует искомый символ, второй — когда он отсутствует. Для разбиения спецификаций поведения функции на несколько различных случаев в языке ACSL есть соответствующее средство — поведения (behaviors). В отличие от некоторых языков спецификаций, где поведения вводятся поверх основного языка спецификаций как синтаксическое расширение (например, в JML), в ACSL поведения являются базовой сущностью языка спецификаций и

практически все спецификации, как в контракте, так и в теле функции могут быть отнесены к одному или нескольким ее поведениям. Для доказательства постусловий с помощью инструмента дедуктивной верификации сформулированы инварианты (loop invariants) на внутренний цикл функции и оценочная функция (loop variant) для него.

Реализация функции `strnchr` содержит в себе в явном виде приведение типа с потерей старшей части значения в строке 27, а также случай с целочисленным переполнением в итераторе цикла из-за постфиксного декремента (строка 26).

Табл. 1. Использование спецификационных конструкций
Table 1. Use of specification structures

Функция	Wrap-around +	Wrap-around Cast
_parse_integer.		
check_bytes8		
kstrtobool		
memchr	✓	✓
memcmp		
memcpy	✓	
memmove	✓	
memscan		
memset	✓	✓
skip_spaces		
strcasecmp		
strcat		
strchr	✓	✓
strchrnul	✓	✓
strcmp		✓
strncmp	✓	
strcpy		
strcspn		
strlcpy		
strlen		
strnchr	✓	✓
strnlen	✓	
strpbrk		
strrchr		✓
strsep		
strspn		

Чтобы указать инструментам верификации, что приведение переменной с типа `int` к типу `char` с потерей части значения является намеренным поведением, используется специальная конструкция `/*@%*/`. Ровно такая же спецификация

используется для обозначения переполнения переменной `cnt`. Семантика этих спецификаций обсуждается в следующем разделе.

В таблице 1 отмечено, в каких функциях из числа верифицированных в рамках данной работы, встречались подобные ситуации с переполнением беззнакового итератора цикла (Wrap-around +) и явным приведением типа с потерей части значения (Wrap-around Cast).

Рассмотрим теперь некоторые проблемы, связанные с поддержкой семантики некоторых конструкций языка ACSL в инструментах дедуктивной верификации на примере подходов к моделированию указателей и машинных целых в инструменте верификации JESSIE, которые были унаследованы инструментом ASTRAVER.

5.Проблемы инструментов верификации

5.1 Блочно-байтовая модель памяти JESSIE

Существует по крайней мере несколько способов логического представления указателей и выделенных блоков памяти в генерируемых условиях верификации. В JESSIE реализована так называемая *блочно-байтовая модель памяти* (*byte-level block memory model*), в которой указатели представлены логически как пары вида (l, o), а блоки памяти — как тройки вида (l, a, s). Здесь метка l уникально идентифицирует блок памяти, o обозначает смещение указателя относительно начального адреса a блока l, а s соответствует размеру блока. Использование уникальных меток блоков позволяет проверять, что доступ к памяти за пределами выделенного блока не происходит даже в том случае, если адресуемая соответствующим указателем область памяти также является выделенной. Несмотря на то, что такой доступ не нарушает сегментирование памяти (и, соответственно, не приводит к ошибкам времени выполнения), соответствующее поведение не допускается стандартом языка Си [15]. (секция 6.5.6, абзац 8 классифицирует создание указателей за пределы выделенных блоков как неопределенное поведение, кроме указателей на область памяти, непосредственно следующую за последним элементом массива). Как объяснено в кандидатской работе [10], описывающей теоретическую основу и архитектуру инструмента JESSIE, блочно-байтовая модель памяти в принципе позволяет выражать семантику часто используемых на практике приемов программирования на языке Си, выходящих за рамки стандарта, таких как реализация функции `memmove`, сохраняя при этом возможность обнаруживать ошибки управления памятью, такие как доступ после освобождения (use-after-free), а также возможные переполнения указателей.

Реализация модели памяти в инструменте JESSIE, однако, имеет ряд отличий от соответствующего относительно простого теоретического описания и налагает дополнительные ограничения на поддерживаемое подмножество языка Си.

Во-первых, указатели реализованы в соответствующей теории JESSIE (на языке WhyML) как значения абстрактного типа $pointer$ с четырьмя соответствующими абстрактными операциями:

$$sub\_pointer : pointer \times pointer \to int,$$
$$shift : pointer \times int \to pointer,$$
$$same\_block : pointer \times pointer \to bool \text{ и}$$
$$address : pointer \to int.$$

Размеры выделенных блоков памяти представлены в теории JESSIE неявно с помощью так называемых *таблиц аллокации* (*allocation tables*), индексированных состоянием программы (то есть изменяемых) значений абстрактного типа $alloc\_table$ с двумя аксиоматически заданными функциями:

$$offset\_min : alloc\_table \times pointer \to int \text{ и}$$
$$offset\_max : alloc\_table \times pointer \to int.$$

Эти функции выражают минимально и максимально допустимое смещение указателя, не выводящее его за пределы соответствующего выделенного блока памяти. Для уникальных меток выделенных блоков и их начальных адресов явное представление в теории JESSIE также отсутствует. Условия верификации, генерируемые инструментом для операций динамического выделения и освобождения памяти (вызовы функций `kmalloc` и `kfree` обрабатываются в JESSIE специальным образом), упоминают только таблицы аллокации и функции $sub\_pointer$, $shift$ и $same\_block$. Это делает соответствующую аксиоматизацию заведомо неполной. В частности, функция $address$ не только никак не упоминается в текущей аксиоматизации теории JESSIE, но и в принципе не может иметь в текущей реализации инструмента достаточно полную аксиоматизацию. В частности, рассмотрим следующее свойство этой функции: "*два валидных указателя, адресующие объекты из различных выделенных блоков памяти не могут иметь одинаковый адрес*". Это свойство не может быть выражено в виде логического утверждения в текущей теории JESSIE, потому что его формализация требует использования условия существования элемента во множестве всех *достижимых* состояний соответствующей таблицы аллокации:

$$\forall p_1, p_2. \big(\exists s_a \in Reachable(s_a). offset\_min(s_a, p_1) \le 0 \land offset\_max(s_a, p_1) \ge 0 \land offset\_min(s_a, p_2) \le 0 \land offset\_max(s_a, p_2) \ge 0 \land \neg same\_block(p_1, p_2)\big) \Rightarrow address(p_1) \neq address(p_2).$$

Так как задача получения явного представления предиката $Reachable(s_a)$ в общем случае является алгоритмически неразрешимой, инструмент может использовать неявное представление, например с помощью выражения соответствующих свойств адреса в каждой точке выделения памяти:

$$\forall p. offset\_min(s_a^*, p) \le 0 \land offset\_max(s_a^*, p) \ge 0 \Rightarrow address(p) \neq address(p^*).$$

Здесь p^* — адрес начала выделяемого блока памяти, s_a^* — состояние таблицы аллокации в точке выделения памяти. Невозможность точной формализации функции $address$ не позволяет генерировать соответствующие условия верификации для обнаружения возможных переполнений указателей и использовать более гибкую формализацию операций сравнения и вычитания указателей (для верификации таких функций, как `memmove`).

Кроме этого, в теории JESSIE смещение и разность указателей, используемые функциями $shift$ и $sub\_pointer$, измеряются в единицах, равных размеру типов адресуемых указателями значений (в соответствии с семантикой адресной арифметики в языке Си), а не в байтах или машинных словах. В частности, выражение `p + 1`, где `p` имеет тип `int *`, транслируется как $shift(p, 1)$, а не как $shift(p, s_{int})$, где s_{int} — константа, равна размеру типа `int` (обычно, 4 байта). Такая трансляция изначально не позволяет выражать многие широко распространенные сочетания перетипирования указателей с адресной арифметикой, включая использования макроса `container_of` (из ядра ОС Linux). Это нетрудно увидеть, рассмотрев два указателя: `p + 1` и `((char *) p) + 1`, где `p` имеет тип `int *` и указывает на начало некоторого выделенного блока памяти. В блочно-байтовой модели памяти со смещениями, измеряемыми в размерах типов, эти указатели имеют одинаковое представление $(l, 1)$, хотя их реальные адреса не могут быть равны (они должны различаться хотя бы на 1, обычно на 3). Это противоречит функциональной консистентности функции $address$. В том числе для того, чтобы не допустить возникновение подобного противоречия (но в основном, по другим причинам, см. [15, 16]) в текущей реализации JESSIE используется два отдельных приема. Во-первых, вводятся специальные дополнительные (логические) *таблицы тегов*, содержащие точные динамические типы объектов в выделенной памяти. Эти таблицы позволяют включать в условия верификации необходимые проверки, ограничивающие использование адресной арифметики (более подробно об этом в [17, 3]). Во-вторых, применяются несколько *нормализующих* преобразований кода, которые трансформируют вложенные структуры и адресуемые поля простых типов в указатели на отдельно выделенные структуры или значения соответствующих типов (эти трансформации описаны в [16]). Это позволяет адресовать вложенные объекты в модели памяти JESSIE. Однако сочетание двух этих приемов приводит к возникновению ряда других существенных ограничений. В частности, объединения, содержащие вложенные структуры в качестве своих полей, не могут быть представлены в модели памяти инструмента. Это связано с невозможностью точного статического выделения среди указателей, получаемых, например, как параметры функции, указателей на структуры, вложенные в объединения. Запись в поля таких структур в соответствии с моделью JESSIE должна транслироваться в *сильные обновления* (*strong coercions*) [17] соответствующих объемлющих объединений с возможным обновлением таблиц тегов и логического представления других

интерпретаций (перетипирований) соответствующей памяти (соответствующей другим полям объединения).

Для преодоления этих и других ограничений текущей модели памяти JESSIE в [4] была предложена новая модель памяти. Эта модель, однако, предполагает использование простой байтовой модели указателей (без привязки их к соответствующим выделенным блокам). Но в силу предполагаемой обычно произвольности стратегии выделения памяти такое моделирование на практике не должно приводить к пропуску случаев нарушения ограничений, налагаемых стандартом языка Си, в случаях попыток разыменования валидной памяти из других выделенных блоков памяти. В таких случаях обычно по крайней мере один из возможных вариантов выделения памяти приводит к разыменованию невалидного указателя и, таким образом, не остается возможности доказать корректность соответствующего разыменования в общем случае. Модель памяти, предложенная в [4], однако, еще не была реализована в инструменте верификации. Поэтому в данной работе была использована существующая реализация модели памяти JESSIE.

Единственное существенное изменение, сделанное в ходе работы в инструменте верификации, было связано с трансляцией неравенств указателей. Так как существующая реализация модели памяти не обеспечивает достаточную выразительность для представления семантики произвольных операций сравнения указателей, мы ограничили поддержку сравнения указателей, разрешив сравнение лишь между указателями на объекты одного выделенного блока памяти, добавив генерацию соответствующих условий верификации и изменив трансляцию соответствующих предикатов вида $p_1 \diamond p_2$ на $sub\_pointer(p_1, p_2) \diamond 0 \wedge same\_block(p_1, p_2)$. Это позволило сократить многие спецификации, так как часто встречающееся дополнительное условие $same\_block(p_1, p_2)$ стало задаваться неявно для всех операций сравнения указателей.

5.2 Модели целых чисел, комбинированная модель целых чисел и аннотации для модульной арифметики

Изначально в JESSIE были реализованы три логические модели машинных целых различного размера и знаковости. Наиболее простая модель, называющаяся `math` (или моделью *неограниченных* целых), предполагает представление всех машинных целых как математических целых. Эта модель не поддерживает проверку возможных арифметических переполнений и не моделирует модульную арифметику ни для каких типов машинных целых (по стандарту языка Си модульная арифметика определена только для беззнаковых целых). В этой модели в принципе возможно моделирование некоторых побитовых операций над неограниченными целыми с помощью соответствующей аксиоматизации, но на практике такое моделирование обычно не эффективно. Другая, наиболее часто используемая модель целых

называется `defensive` (или моделью *ограниченных* целых) и имеет два отличия от модели `math`:

- для целочисленных операций в коде в этой модели генерируются соответствующие условия верификации, предотвращающие возможные арифметические переполнения;
- ограниченные целые в спецификациях моделируются абстрактными типами со специальными функциями отображения (такими как $int32\_of\_integer$ и $integer\_of\_int32$), аксиоматизация которых определяет результат преобразования в ограниченное целое только значений, попадающих в соответствующий диапазон.

Модель `defensive` проста и эффективна и подходит для большинства случаев, за исключением тех, когда требуется точное моделирование машинной арифметики или побитовых операций. Для этих целей в JESSIE реализована модель целых `modulo`, которая предполагает точное моделирование значений машинных целых как битовых векторов.

К сожалению, в JESSIE модель целых чисел может быть выбрана лишь один раз для всей верифицируемой программы с использованием соответствующей прагмы. На практике, однако, желательно иметь возможность выбирать подходящий вариант моделирования семантики операций над целыми для каждого отдельного участка кода, вплоть до отдельно взятых операций. Рассмотрим следующий пример (см. Листинг 2):

```
1  int strncasecmp(const char *s1,
2     const char *s2, size_t len) {
3   unsigned char c1, c2;
4   if (!len) return 0;
5   do {
6     c1 = *s1++;
7     c2 = *s2++;
8     if (!c1 || !c2) break;
9     if (c1 == c2) continue;
10    c1 = tolower(c1);
11    c2 = tolower(c2);
     12    if (c1 != c2) break;
13  } while (--len);
14  return (int)c1 - (int)c2;
15 }
```

Листинг 2. Функция strncasecmp. Ядро Linux 4.12, файл lib/string.c
Listing 2. Function strncasecmp. Linux kernel 4.12, file lib/string.c

Здесь в строках 6 и 7 уместно применение модели машинных целых modulo, так как приведение типа `char` к типу `unsigned char` может привести к переполнению, и в данном случае такое поведение соответствует намерениям программиста. Однако желательно также, чтобы возможное переполнение было обнаружено в случае замены типа возвращаемого значения функции на

`char`. Поэтому для моделирования вычитания в строке 13 уместно применение модели `defensive`.

Для поддержки такой точной спецификации модели целых на уровне отдельных арифметических операций мы реализовали в инструменте поддержку расширения языка ACSL *аннотациями для модульной арифметики*. Были введены следующие новые виды аннотаций:

- для арифметических операций: `+/*@%*/`, `-/*@%*/`, `*/*@%*/`, ...
- для составных присваиваний (compound assignments): `+=/*@%*/`, `-=/*@%*/`, `/=/*@%*/` ...
- для префиксных и постфиксных операторов: `++/*@%*/`, `--/*@%*/`;
- для явных приведений типа: `(unsigned char)/*@%*/`, ...
- для модульной арифметики в спецификациях: `+%`, `-%`, `*%`, ...

Модель машинных целых, используемая для моделирования как обычных арифметических операций (без аннотаций), так и аннотированных операций модульной арифметики является комбинированной, в которой машинные целые моделируются как битовые векторы с аксиоматически заданными функциями отображения в/из математических целых. Использование такого комбинированного моделирования целых и аннотаций для модульной арифметики позволило значительно упростить спецификацию и верификацию многих функций в рамках данной работы.

6. Формальные спецификации

При разработке спецификаций мы руководствовались несколькими приёмами: использованием избыточных спецификаций (явные спецификации и спецификации, устанавливающие соответствие логической функции), разработкой спецификаций на основе кода реализации и контекста

```
/*@ requires cnt ≥ 0;
requires \valid(s+(0..cnt-1));
assigns \nothing;
behavior bigger:
 assumes ∀ ℤ i; 0 ≤ i < cnt ⇒ s[i] ≢ 0;
 ensures \result ≡ cnt;
behavior smaller:
 assumes ∃ ℤ i; 0 ≤ i < cnt ∧ s[i] ≡ 0;
 ensures \result ≤ cnt;
complete behaviors;
disjoint behaviors;*/
size_t strnlen(const char *s, size_t cnt);
```

использования функции.

Листинг 3. Контракт для функции `strnlen`. Проект по верификации klibc
Listing 3. Contract for function `strnlen`. Verification project klibc

Результаты, описанные в работе [6], показывают, что разработать полный контракт функции, отталкиваясь от документации крайне затруднительно: почти всегда, при доказательстве приходится переписывать спецификацию «от реализации». Подобный подход также объясняется тем, что в данной работе спецификации разрабатываются к готовому коду, а не код пишется в соответствии с некоторым набором данных спецификаций. Также в ядре для многих функций отсутствует специальная документация в коде. Мы намеренно не стали руководствоваться стандартной документацией к подобным функциям, так как реализация их в ядре может отличаться от других (например, от реализации в стандартной библиотеке), а документация быть не полной и содержать неточности [6].

```
/*@ predicate valid_strn(char *s, size_t cnt) =
 (∃ size_t n;
 (n < cnt) ∧ s[n] ≡ '\0' ∧ \valid(s+(0..n))) ∨
 \valid(s+(0..cnt));

requires valid_strn(s, cnt);
assigns \nothing;
ensures \result ≡ strnlen(s, cnt);
behavior null_byte:
 assumes ∃ ℤ i; 0 ≤ i ≤ cnt ∧ s[i] ≡ '\0';
 ensures s[\result] ≡ '\0';
 ensures ∀ ℤ i; 0 ≤ i < \result ⇒ s[i] ≢ '\0';
behavior cnt_len:
 assumes ∀ ℤ i; 0 ≤ i ≤ cnt ⇒ s[i] ≢ '\0';
 ensures \result ≡ cnt;
complete behaviors;
disjoint behaviors;*/
size_t strnlen(const char *s, size_t cnt);
```

Листинг 4. Контракт для функции `strnlen`*. Проект по верификации библиотечных функций ядра Linux*

Listing 4. Conntract for function `strnlen`*. The Linux library functions verification project*

Как следствие подобного подхода, спецификации к ряду функций обладают несколько более детализированным видом: так для функции вида `strn*` (см. Листинги 4, 6) на уровне спецификаций мы не требуем обязательного наличия маркера конца строки. В случае `strnlen` (см. Листинг 4) в предусловии предполагается что строка должна быть валидной до минимума из длины строки (если существует маркер её конца) и второго аргумента функции `strnlen`, а возвращаемый результат в постусловии задается явным образом. В случае `strncmp` (см. Листинг 6) также не вводятся ограничения на то, что входные строки должны содержать нулевой байт. Это приводит к тому, что на уровне спецификаций приходится явно описывать поведение функции, когда входные строки, имеющие конец, различаются по длине. Мы старались максимально ослабить предусловия и усилить постусловия для того, чтобы

протестировать инструменты дедуктивной верификации, выразительность языка ACSL, а также возможности солверов.

```
/*@ predicate valid_string{L}(char *s) =
 0 ≤ strlen(s) ∧ \valid_range(s,0,strlen(s));

requires valid_string(s1);
requires valid_string(s2);
requires n < INT_MAX;
assigns \nothing;
ensures n ≡ 0 ⇒ \result ≡ 0;
ensures (n > 0 ⇒ (∀ ℤ i;
 0 ≤ i ≤ minimum(n-1, minimum(strlen(s1), strlen(s2))) ∧
 s1[i] ≡ s2[i])) ⇒ \result ≡ 0;
ensures \result < 0 ⇒ (n > 0 ∧ ∃ ℤ i;
 0 ≤ i ≤ minimum(n-1, minimum(strlen(s1), strlen(s2))) ∧
 s1[i] < s2[i] ⇒
 (∀ ℤ k; 0 ≤ k < i ⇒ s1[k] ≡ s2[k]));
ensures \result > 0 ⇒ (n > 0 ∧ ∃ ℤ i;
 0 ≤ i ≤ minimum(n-1, minimum(strlen(s1), strlen(s2))) ∧
 s1[i] > s2[i] ⇒
 (∀ ℤ k; 0 ≤ k < i ⇒ s1[k] ≡ s2[k]));*/
int strncmp(const char *s1, const char *s2, size_t n);
```

Листинг 5. Контракт для функции `strncmp`*. Проект по разработке спецификаций для OpenBSD*

Listing 5. Contract for function `strncmp`*. The project to develop specifications for OpenBSD*

6.1 Логические функции

Для некоторых функций спецификации намеренно избыточны и фактически дважды по-разному описывают то, как функция должна работать. Например, одна из таких функций — `strlen`. В её спецификации есть обычные функциональные требования и есть требование на соответствие возвращаемого результата вызову логической функции под тем же названием `strlen`. Подобный подход мотивирован тем, что логическую функцию `strlen` удобно использовать в спецификациях других функций, например, `strcmp` (а логическую функцию, описывающую поведение функции `strcmp` — при описании функциональных требований к `strcpy`). При этом все основные свойства логической функции задаются с помощью аксиом и лемм (леммы на данном этапе не доказывались). Однако такие спецификации не во всех случаях удобны. Например, их в общем случае невозможно отобразить в проверки времени исполнения E-ACSL [18]. Поэтому для тех функций, которым в спецификациях ставилась в соответствие логическая функция, обязательно разрабатывались и «обычные» спецификации.

Функции на языке Си можно сопоставить логическую функцию (один-в-один), только в случае, если Си-функция «чистая» (pure). Логическую функцию рационально писать в том случае, если она пригодится для

разработки спецификаций для других функций. Например, в функциональных требованиях к `memcpy` можно выразить «одинаковость» `src` и `dest` посредством вызова логической функции `memcmp`.

```
/*@ requires valid_strn(cs, cnt);
requires valid_strn(ct, cnt);
assigns \nothing;
ensures \result ≡ -1 ∨ \result ≡ 0 ∨ \result ≡ 1;
behavior equal:
 assumes cnt ≡ 0 ∨
  cnt > 0 ∧
  (∀ ℤ i; 0 ≤ i < strnlen(cs, cnt) ⇒ (cs[i] ≡ ct[i])) ∧
  strnlen(cs, cnt) ≡ strnlen(ct, cnt);
 ensures \result ≡ 0;
behavior len_diff:
 assumes cnt > 0;
 assumes ∀ ℤ i; 0 ≤ i < strnlen(cs, cnt) ⇒ (cs[i] ≡ ct[i]);
 assumes strnlen(cs, cnt) ≢ strnlen(ct, cnt);
 ensures strnlen(cs, cnt) < strnlen(ct, cnt) ⇒ \result ≡ -1;
 ensures strnlen(cs, cnt) > strnlen(ct, cnt) ⇒ \result ≡ 1;
behavior not_equal:
 assumes cnt > 0;
 assumes ∃ ℤ i; 0 ≤ i < strnlen(cs, cnt) ∧ cs[i] != ct[i];
 ensures ∃ ℤ i; 0 ≤ i < strnlen(cs, cnt) ∧
  (∀ ℤ j; 0 ≤ j < i ⇒ cs[j] ≡ ct[j]) ∧
  (cs[i] ≢ ct[i]) ∧
  ((u8 *)cs[i] < (u8 *)ct[i] ? \result ≡ -1 : \result ≡ 1);
complete behaviors;
disjoint behaviors;*/
int strncmp(const char *cs, const char *ct, size_t cnt);
```

Листинг 6. Контракт для функции `strncmp`*. Проект по верификации библиотечных функций ядра Linux*

Listing 6. Conntract for function `strncmp`*. The Linux library functions verification project*

7. Нерешённые проблемы

На уровне спецификаций авторы столкнулись с рядом проблем, связанных с значительными неточностями моделирования операций с указателями, а также недостаточным уровнем поддержки языка ACSL инструментами.

Так, для функции `memmove` инструментами верификации генерируется условие верификации, которое проверяет, что указатели `dest` и `src` лежат в одном выделенном блоке памяти. Это необходимо для того, чтобы результат их сравнения был определён по стандарту [14]. Напомним, как работает функция `memmove`: она осуществляет копирование участка памяти размером *n* байт с адреса `src` по адресу `dst`, при условии, что два обозначенных региона памяти могут как пересекаться, так и не пересекаться. Чтобы реализовать последнее условие, в функции осуществляется порядковое сравнение

указателей `dest` и `src`. В том случае, если `dest` находится до `src`, осуществляется побайтовое копирование с начала `src` (таким образом, если регионы накладываются друг на друга, то в `src` будет затираться часть, ранее уже скопированная в `dest`); если же `dest` находится после `src`, то осуществляется копирование, начиная с конца региона памяти `src`.

Модель памяти, лежащая в основе инструментов верификации, позволяет осуществлять арифметические операции на указателях (в `memmove` это сравнение, реализованное через разницу между указателями) в том случае, если указатели принадлежат одному выделенному блоку памяти. Для `memmove` это не обязательно так. Если условие возможной разницы регионов записано в контракте, то соответствующее условие верификации, требующее совпадения блоков памяти, невозможно доказать. Это отражено в итоговых результатах в таблице 2.

Функция `strcat` осуществляет конкатенацию двух строк, добавляя строку `src` к строке `dest`. Для этого сначала находится конец строки `dest`, а после осуществляется копирование строк аналогично тому, как это происходит в функции `strcpy`. Для того, чтобы доказать условия верификации, проверяющие корректность работы с памятью в данной функции, было достаточно потребовать валидности строк `src` и `dest`, а также достаточного количества памяти за концом строки `dest` для того, чтобы вместить содержимое `src`. Однако при доказательстве корректности относительно функциональных требований к данной функции, потребовалось сформулировать дополнительное требование, утверждающее, что сумма длин строк `src` и `dest` вмещается в тип `size_t`. Функция реализована через итерацию по указателям. Следовательно, возможность доказательства корректности работы с памятью без привлечения дополнительного требования в данной функции означает, что в модели памяти инструмента верификации не учитывается возможность переполнения указателей.

```
void *memset(void *s, int c,
             size_t count) {
  char *xs = s;
  while (count--)
-   *xs++ = c;
+   *xs++ = (char) c;
  return s;
}
```

Листинг 7. Функция `memset`. Ядро Linux 4.12, файл lib/string.c
Listing 7. Function `memset`. Linux kernel 4.12, file lib/string.c

Несколько функций потребовали изменения кода для того, чтобы стало возможным доказать их корректность. Несмотря на то, что авторы ставили своей целью свести к минимуму внесение правок в код, в двух случаях этого избежать не удалось. В функциях `memset` и `strcmp` используется неявное приведение типов с переполнением. В `memset` тип `int` неявным образом

приводится к типу `char` (листинг 3), а в `strcmp` — `char` к `unsigned char`. Для того, чтобы добавить спецификацию о намеренном переполнении, требуется сделать приведение типа явным. В инструментах на уровне спецификаций не хватает конструкции неявного приведения типа с переполнением (например, `/*@ (unsigned int %) */`) для того, чтобы

```
axiomatic NotSupported {
 predicate less(ℤ a, ℤ b) = a < b;
 logic ℤ min(ℤ a, ℤ b) = less(a, b) ? a : b;
 lemma not_supported:
  ∀ ℤ a, b; less(a, b) ? min(a, b) ≡ a : min(a, b) ≡ b;
}

axiomatic Supported {
 predicate less(ℤ a, ℤ b) = a < b;
 logic ℤ min(ℤ a, ℤ b);
 lemma defn1:
  ∀ ℤ a, b; less(a, b) ⇔ min(a, b) ≡ a;
 lemma defn2:
  ∀ ℤ a, b; !less(a, b) ⇔ min(a, b) ≡ b;
}
```

стало возможным обойтись без изменения кода в данных случаях.

Листинг 8. Аксиоматические теории
Listing 8. Axiomatic theories

На уровне спецификаций инструментами не поддерживается использование предикатов в определениях логических функций, а также использование предикатов в леммах и аксиомах в тернарном операторе. Из-за этого иногда сложно дать явное определение логической функции и приходится использовать аксиоматическое определение. Эта особенность мешает дать явное определение логических функций `skip_spaces`, `strcspn`, `strpbrk` и `strspn`.

Функции из файла `ctype.h` (`isspace`, `isdigit`, `isalnum`, `isgraph`, `islower`, ...) определены как макросы, которые оперируют на массиве из 256 ячеек `_ctype`, в котором задаётся принадлежность каждого символа определённому классу. Чтобы упростить верификационную задачу данные макросы были заменены inline-функциями: инструменты верификации не позволяют писать спецификации на макроопределения, только на функции. Из-за того, что глобальная инициализация массивов не транслируется в модель для верификации (WhyML), массив `_ctype` был переопределён как строка (инициализация строк транслируется в модельные аксиомы). Однако доказать соответствие функций из файла `ctype.h` их спецификаций не удалось и после описанных трансформаций: солверы не справляются с доказательством, когда в модели есть аксиоматическое задание массива `_ctype` длиной 256 символов.

8. Результаты

В процессе доказательства полной корректности функций мы пользовались преимущественно солверами Alt-Ergo (1.30) и CVC4 (1.4). Их возможностей хватило, чтобы доказать все условия верификации. Для того, чтобы протестировать возможности других солверов в разных конфигурациях, нами был подготовлен тестовый стенд и специальная стратегия преобразования для каждого условия верификации.

8.1 Конфигурация тестового стенда

Солверы запускались на машине со следующей конфигурацией: AMD FX-8120 (Eight-Core Processor), 16GB RAM. С лимитами по времени в 60 секунд и по памяти в 6000 Mb. Параллельно работало не более трёх солверов. Операционная система — GNU/Linux (kernel: 4.12.12 (smp preempt) x86_64). Использовались инструменты следующих версий с сайта проекта AstraVer: WHY3 (0.87.3+git), FRAMA-C (Silicon-20161101), JESSIE2 (alpha3)

8.2 Стратегия трансформации условий верификации

Для того, чтобы все солверы были поставлены в максимально близкие условия и были доказаны все условия верификации (кроме одного для `memmove`), использовалась следующая стратегия трансформации условий верификации:

1. подразбить условие верификации по конъюнктам (split_goal_wp); перейти к шагу 2.
2. попробовать применить трансформацию из шага 1, иначе перейти к шагу 3;
3. встроить определения всех логических символов (inline_all);
4. попробовать применить трансформацию из шага 1, иначе перейти к шагу 5;
5. сколемизация условия верификации (introduce_premises).

Необходимо отметить, что в некоторых случаях трансформация inline_all наоборот затрудняет работу солверов. Это происходит в том случае, когда используется достаточно большое количество предикатов в спецификации функции, формируя длинную цепочку зависимостей. Однако разработанные нами спецификации не подходят под этот критерий, и эксперименты с запусками солверов показали обоснованность применения данной трансформации.

Эксперименты также показали, что решатели более эффективно разрешают выполнимость формул вида $f(\bar{x}) \wedge \neg g(\bar{x})$ (1), чем формул вида $\neg\forall\bar{x}.f(\bar{x}) \Rightarrow g(\bar{x})$ (2), несмотря на то, что переход от формул вида (2) к формулам вида (1) возможен с помощью простых преобразований, сохраняющих выполнимость (приведения к предваренной нормальной форме и сколемизации). Поэтому в

экспериментах использовалась трансформация WHY3 introduce_premises, соответствующая выполнению этих преобразований.

В остальном можно видеть, что приведённая стратегия максимально нацелена на подразбитие условия верификации на отдельные конъюнкты. Это облегчает работу солверов. При реальной работе с доказательствами подобная стратегия не используется. Некоторые из приведённых трансформаций используются лишь в том случае, когда солверы не могут сразу доказать условие верификации.

8.3 Результаты солверов

Результаты запуска солверов приведены в таблице 2. В запусках участвовали следующие солверы: ALT-ERGO (1.30), CVC3 (2.4.1), CVC4 (1.4), CVC4 (1.5), EPROVER (1.9.1-001), SPASS (3.9), Z3 (4.5.0).

Табл, 2. Запуски солверов
Table 2. Solver runs

Function	VC	Alt-Ergo 1.3		CVC3 2.4.1		CVC4 1.4		CVC4 1.5		Eprover 1.9.1-001		Spass 3.9		Z3 4.5.0	
	total	vc	atime	vc	atime	vc	atime	vc	atime	vc	atime	vc	atime	vc	atime
_parse_integ.	282	276	0.1	280	0.83	✓	0.18	✓	0.1	212	0.24	197	1.69	279	0.06
check_bytes8	50	49	0.55	49	0.09	49	0.09	✓	0.11	38	1.76	31	8.38	36	1.52
kstrtobool	1096	✓	0.05	✓	0.08	✓	0.1	✓	0.09	1006	0.13	937	0.38	1065	0.15
memchr	39	✓	6.05	11	0.22	✓	0.37	✓	0.15	31	2.58	11	5.73	29	0.12
memcmp	60	58	0.13	✓	0.15	58	0.1	✓	0.1	49	0.51	36	4.45	55	0.15
memcpy	43	✓	4.18	✓	0.35	✓	0.16	✓	0.14	30	1.05	16	6.85	30	0.06
memmove	**93***(92)	90	3.94	✓	0.88	87	0.16	✓	0.18	63	0.95	43	11.87	68	0.3
memscan	47	46	0.07	✓	0.1	✓	0.09	✓	0.09	41	0.59	34	4.55	42	0.06
memset	27	26	5.02	14	0.19	✓	0.19	✓	0.16	19	3.82	12	11.12	18	0.08
skip_spaces	34	30	0.76	32	1.96	✓	0.51	33	0.14	27	0.7	24	0.34	30	0.09
strcasecmp	58	50	0.43	52	1.65	57	0.79	✓	0.53	43	0.28	35	2.85	49	0.49
strcat	73	68	0.58	66	2.16	✓	1.13	71	0.17	54	2.56	39	0.67	60	0.94
strchr	43	35	4.57	23	0.17	✓	0.23	✓	0.22	31	1.03	24	3.65	32	0.11
strchrnul	46	42	2.07	37	0.26	✓	0.19	✓	0.16	40	1.91	31	2.27	39	0.31
strcmp	60	51	1.76	16	0.6	✓	1.75	59	1.08	47	1.05	36	1.65	47	0.1
strcpy	46	43	1.33	45	0.66	✓	0.48	✓	0.17	33	1.13	26	0.65	39	1.43
strcspn	78	68	0.38	69	0.37	74	2.95	75	1.82	58	1.85	46	1.68	61	0.11
strlcpy	84	82	0.15	82	0.14	✓	1.08	✓	0.24	67	1.2	52	1.74	78	0.42
strlen	26	✓	1.12	24	0.12	✓	0.16	✓	0.23	19	3.36	14	2.96	21	0.08
strnchr	49	38	4.44	19	0.23	46	3.34	✓	0.72	35	2.57	24	1.56	27	0.09
strncmp	102	81	2.57	25	0.25	94	2.39	99	2.32	76	1.06	55	2.56	76	0.57
strnlen	44	39	1.91	42	1.04	39	1.23	✓	1.31	31	2.4	26	5.52	32	0.08
strpbrk	70	57	0.64	58	1.54	62	3.18	67	1.57	48	1.89	39	0.75	53	0.09
strrchr	62	53	4.57	12	0.17	✓	1.09	60	0.85	46	2.33	31	4.67	46	0.11
strsep	62	60	0.25	60	0.09	✓	0.19	✓	0.15	55	0.12	51	1.48	58	0.06
strspn	107	99	0.84	100	0.69	104	1.32	103	0.61	89	1.37	75	1.59	91	0.13
TOTAL	2781	2645	0.9	2454	0.42	2740	0.61	2761	0.37	2288	0.76	1945	1.72	2461	0.22

В первой колонке таблица содержит имя функции ядра Linux, корректность которой доказывается. Во второй колонке содержится информация о

количестве условий верификации для указанной функции после применения описанной стратегии для тестирования солверов. В последующих колонках приводится информация о запусках солверов: количеству доказанных ими условий верификации и среднему времени работы на успешных запусках.

Там, где солвер сумел доказать все условия верификации для конкретной функции в таблице проставлена ✓. Солверы с максимальным количеством доказанных условий верификации помечаются зеленым цветом. Минимум среди остальных солверов выделяется красным цветом. Минимальное среднее время среди всех запусков отмечается синим цветом. Максимальное — жёлтым цветом.

Также солверы CVC4 (1.4, 1.5) и Z3 (4.5.0) дополнительно тестировались с драйвером noBV. Это специальный драйвер WHY3, который убирает теории для битовых векторов из задач солверов. Так как в нашем наборе функций отсутствуют побитовые операции и манипуляции с битовыми полями, предполагалось что использование данного драйвера себя оправдывает. Результаты данных конфигураций не вошли в итоговую таблицу, так как они оказались хуже по количеству доказанных условий верификации, чем другие конфигурации этих же солверов. При лимите по времени в 40 секунд и по памяти в 4000 Mb солверы CVC4 с драйвером noBV показывали результат лучше, однако при текущих лимитах картина радикально поменялась.

Необходимо отметить, что в нормальном цикле работ нами используются солверы ALT-ERGO (1.30), CVC4 (1.4), Z3 (4.5.0) в порядке частоты использования.

Солвер CVC4 версии 1.5 запускался с драйвером для версии солвера CVC4 1.4, так как в текущей версии WHY3 отсутствует драйвер для релизной версии солвера 1.5. Версия драйвера для пререлизной версии выдаёт результат хуже [19].

Табл. 3. Максимальное время работы солверов и случаи уникальных доказательств
Table 3. Maximum time of solvers and cases of unique proof

Alt-Ergo 1.3		CVC3 2.4.1		CVC4 1.4		CVC4 1.5		Eprover 1.9.1-001		Spass 3.9		Z3 4.5.0	
mtime	uniq	mtime	uniq	mtime	uniq	mtime	uniq	mtime	uniq	mtime	uniq	mtime	uniq
58.75	1	56.68	0	57.97	7	52.27	20	47.8	0	59.74	0	26.74	0

8.4 Интерпретация результатов

Все условия верификации, за исключением одного для `memmove`, успешно доказываются солверами. Лучше всего себя показали ALT-ERGO и CVC4, что легко объяснимо тем, что инструменты верификации тестируются, в основном, на этих двух солверах.

Лучший результат по количеству доказанных условий верификации показывает CVC4 версии 1.5.

Лучшие результаты по времени показывает Z3. Это можно частично объяснить тем, что учитываются только успешные запуски солверов. Z3 доказывает меньше условий верификации, чем ALT-ERGO и CVC4.

Также стоит отметить, что механизм запуска солверов WHY3 иногда давал сбои и некорректно учитывал лимит по времени. Так, для солвера Spass в 4х случаях было существенно (в 1.5-2 раза) превышено максимальное время доказательства, для солвера CVC4 (1.5) было одно превышение на 18 секунд. Несмотря на то, что солверы успешно доказали условия верификации, эти времени и для других солверов.

В таблице 3 приведены данные по максимальному времени работы солвера для успешного доказательства, а также по количеству уникальных доказательств. результаты не были им засчитаны и не учитывались в итоговой таблице. В процессе подготовки результатов мы сталкивались с некорректным учётом

Под уникальными доказательствами понимаются условия верификации, которые смог доказать только один солвер.

В целом, результаты показали, что версия солвера CVC4 (1.5) заслуживает пристального внимания к себе, и, возможно, перехода к использованию его как основного, при работе с данным стеком инструментов. Пока этому мешает отсутствие полноценной поддержки со стороны WHY3 и общая нестабильность его работы. Солвер часто падает с ошибкой сегментирования на условиях верификации, на которых не применялись вышеописанные трансформации.

9. Дальнейшая работа

Для части логических функций, реализованных к текущему моменту возможно доказать корректность лемм. Из-за ограничений инструментов, не все логические функции могли быть определены явно. Для тех из них, которые не заданы явно, доказать корректность лемм на текущий момент невозможно по очевидной причине. Для остальных корректность лемм должна быть доказана с помощью инструмента Coq.

Следующим логичным шагом является расширение набора функций для формальной верификации и включения в этот набор не только функций для работы со строками, но и, например, с битовой арифметикой, а также иных типов функций из папки `lib` ядра Linux. На этом этапе наибольший интерес для нас представляют функции, так или иначе использующие конструкции языка Си, которые сложны для формальной верификации.

После того, как контракты функций корректно сформулированы, имеет смысл проверить соблюдение предусловий в точках вызовов библиотечных функций. Ввиду того, что кода ядра Linux имеет существенный объем, осуществить подобную проверку методами дедуктивной верификации практически невозможно. Однако, с помощью плагина E-ACSL возможно часть

предусловий отобразить в проверки времени исполнения и осуществлять проверку соблюдения предусловий библиотечных функций в динамике. Это потребует существенной доработки плагина E-ACSL для работы с кодовой базой ядра Linux, а также частичного изменения самого кода ядра. Плагин использует несколько библиотек (для отслеживания состояния памяти, для работы с неограниченными числами), которые должны быть интегрированы в код ядра.

10. Заключение

В рамках работы были полностью доказаны 26 библиотечных функций ядра Linux. Большинство данных функций оперируют с данными строкового типа. На наборе этих функций удалось выявить значительное количество недостатков в использованных инструментах верификации, выработать подходы к доказательству и разработке спецификаций для функций аналогичного вида, предложить и реализовать расширение набора спецификационных конструкций языка ACSL. Авторы старались разрабатывать спецификации таким образом, чтобы не менять исходный код функций. Введение в язык ACSL двух дополнительных конструкций позволило доказать 11 функций, не изменяя их кода.

Итоговое соотношение спецификаций и размера кода составляет примерно ~2.6, то есть примерно две с половиной строчки спецификации на одну строчку кода.

Код функций, спецификации, протоколы доказательств выложены в открытом доступе вместе с инструкциями по воспроизведению результата [7]. Спецификации с исходным кодом могут в дальнейшем служить тестовым набором для инструментов дедуктивной верификации и солверов.

Список литературы

[1]. MISRA C: 2012. Guidelines for the Use of the C Language in Critical Systems, ISBN 978-1-906400-10-1 (paperback), ISBN 978-1-906400-11-8 (PDF), March 2013.

[2]. AstraVer Toolset: инструменты дедуктивной верификации моделей и механизмов защиты ОС. ИСП РАН. Доступно по ссылке: http://linuxtesting.org/astraver, 15.10.2017.

[3]. Мандрыкин М.У., Хорошилов А.В. Высокоуровневая модель памяти промежуточного языка Jessie с поддержкой произвольного приведения типов указателей. Программирование, 2015, т. 41, №4, стр. 23–29.

[4]. Мандрыкин М.У., Хорошилов А.В. Анализ регионов для дедуктивной верификации Си-программ. Программирование. 2016, т. 42, № 5, стр. 3–29.

[5]. Carvalho N., Silva Sousa C., Pinto J.S., Tomb A. Formal verification of kLIBC with the WP frama-C plug-in. In: Badger, J.M., Rozier, K.Y. (eds.) NFM 2014. LNCS, vol. 8430, pp. 343–358. Springer, Heidelberg (2014)

[6]. Torlakcik M. Contracts in OpenBSD. MSc thesis. University College Dublin. April, 2010.

[7]. Efremov D., Mandrykin M. VerKer: Verification of Linux Kernel Library Functions. Доступно по ссылке: http://forge.ispras.ru/projects/verker, 15.10.2017.
[8]. Cuoq P., Kirchner F., Kosmatov N., Prevosto V., Signoles J., Yakobowski B. Frama-C: A Software Analysis Perspective. Proceedings of the 10th International Conference on Software Engineering and Formal Methods. SEFM'12. Thessaloniki, Greece. 2012. Lecture Notes in Computer Science, vol. 7504, pp. 233–247. Springer-Verlag. Berlin, Heidelberg. 2012.
[9]. Baudin P., Cuoq P., Filliâtre J., Marché C., Monate B., Moy Y., Prevosto V. ACSL: ANSI/ISO C Specification Language. Version 1.12. CEA LIST and INRIA. May, 2017. Доступно по ссылке: https://frama-c.com/download/acsl_1.12.pdf, 15.10.2017.
[10]. Moy Y. Automatic Modular Static Safety Checking for C Programs. PhD thesis. Université Paris-Sud. January, 2009.
[11]. Filliâtre J., Paskevich A. Why3: Where Programs Meet Provers. Proceedings of the 22Nd European Conference on Programming Languages and Systems. ESOP'13. Rome, Italy. LNCS, vol. 7792, pp. 125–128. Springer-Verlag.
[12]. Burghardt J., Clausecker R., Gerlach J., Pohl H. ACSL By Example: Towards a Verified C Standard Library. Version 15.1.1. Доступно по ссылке: https://github.com/fraunhoferfokus/acsl-by-example/raw/master/ACSL-by-Example.pdf, 15.10.2017.
[13]. Cok D., Blissard I., Robbins J. C Library annotations in ACSL for Frama-C: experience report. GrammaTech, Inc. March, 2017. Доступно по ссылке: http://annotationsforall.org/resources/links/GT-libc-experience-report.pdf, 15.10.2017.
[14]. Hatcliff J., Leavens G. T., Leino K. R. M., Müller P., Parkinson M. Behavioral interface specification languages. ACM Comput. Surv. vol. 44, issue 3, article 16, 58 p. June 2012.
[15]. ISO/IEC 9899: 2011: C11 standard for C programming language. JTC and ISO. April 7, 2016.
[16]. Hubert T., Marché C. Separation Analysis for Deductive Verification. Heap Analysis and Verification (HAV'07). Braga, Portugal, March 2007, pp. 81–93.
[17]. Moy Y. Union and Cast in Deductive Verification. Proceedings of the C/C++ Verification Workshop. Technical Report ICIS-R07015, pp. 1–16. Radboud University Nijmegen. July, 2007.
[18]. Signoles J. E-ACSL Executable ANSI/ISO C Specification Language. Version 1.12. CEA LIST. May, 2017. Доступно по ссылке: https://frama-c.com/download/e-acsl/e-acsl.pdf, 15.10.2017.
[19]. Marché C. [Frama-c-discuss] Frama-C/WP and CVC4 (version 1.5). Доступно по ссылке: https://lists.gforge.inria.fr/pipermail/frama-c-discuss/2017-August/005338.html, 15.10.2017.

Formal Verification of Linux Kernel Library Functions

[1]D.V. Efremov <defremov@hse.ru >
[2]M.U. Mandrykin <mandrykin@ispras.ru>
[1]NRU Higher School of Economics,
20 Myasnitskaya Ulitsa, Moscow, 101000, Russia
[2]Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. The paper presents result of a study on deductive verification of 26 Linux kernel library functions with AstraVer toolset. The code includes primarily string-manipulating functions and is verified against contract specifications formalizing its functional correctness properties. The paper presents a brief review of the related earlier studies, discusses their results and indicates both the previous issues that were successfully solved in this study and the ones that remain and still prevent successful verification. The paper also presents several specification practices that were applied in the study, including some common specification patterns. The authors have successfully and fully proved functional correctness of 25 functions. The paper includes results of benchmarking 5 state-of-the-art SMT solvers on the resulting verification conditions.

Keywords: static analysis; formal verification; deductive verification; standard library.

DOI: 10.15514/ISPRAS-2017-29(6)-3

For citation: Efremov D.V., Mandrykin M.U. Formal Verification of Linux Kernel Library Functions. *Trudy ISP RAN/Proc. ISP RAS*, vol. 29, issue 6, 2017. pp. 49-76 (in Russian). DOI: 10.15514/ISPRAS-2017-29(6)-3

References

[1]. MISRA C: 2012. Guidelines for the Use of the C Language in Critical Systems, ISBN 978-1-906400-10-1 (paperback), ISBN 978-1-906400-11-8 (PDF), March 2013.

[2]. AstraVer Toolset: deductive verification of Linux kernel modules and security policy models. ISP RAS. Available at: http://linuxtesting.org/astraver, accessed 15.10.2017.S

[3]. Mandrykin M. U., Khoroshilov A. V. High-level memory model with low-level pointer cast support for Jessie intermediate language. Programming and Computer Software, 2015, vol. 41, no. 4, pp. 197–207. DOI: 10.1134/S0361768815040040

[4]. Mandrykin M. U., Khoroshilov A. V. Region analysis for deductive verification of C programs. Programming and Computer Software, 2016, vol. 42, no. 5, pp. 257–278. DOI: 10.1134/S0361768816050042

[5]. Carvalho N., Silva Sousa C., Pinto J.S., Tomb A. Formal verification of kLIBC with the WP frama-C plug-in. In: Badger, J.M., Rozier, K.Y. (eds.) NFM 2014. LNCS, vol. 8430, pp. 343–358. Springer, Heidelberg (2014).

[6]. Torlakcik M. Contracts in OpenBSD. MSc thesis. University College Dublin. April, 2010.

[7]. Efremov D., Mandrykin M. VerKer: Verification of Linux Kernel Library Functions. Available at: http://forge.ispras.ru/projects/verker, accessed 15.10.2017.

[8]. Cuoq P., Kirchner F., Kosmatov N., Prevosto V., Signoles J., Yakobowski B. Frama-C: A Software Analysis Perspective. Proceedings of the 10th International Conference on Software Engineering and Formal Methods. SEFM'12. Thessaloniki, Greece. 2012. Lecture Notes in Computer Science, vol. 7504, pp. 233–247. Springer-Verlag. Berlin, Heidelberg. 2012.

[9]. Baudin P., Cuoq P., Filliâtre J., Marché C., Monate B., Moy Y., Prevosto V. ACSL: ANSI/ISO C Specification Language. Version 1.12. CEA LIST and INRIA. May, 2017. Available at: https://frama-c.com/download/acsl_1.12.pdf, accessed 15.10.2017.

[10]. Moy Y. Automatic Modular Static Safety Checking for C Programs. PhD thesis. Université Paris-Sud. January, 2009.

[11]. Filliâtre J., Paskevich A. Why3: Where Programs Meet Provers. Proceedings of the 22Nd European Conference on Programming Languages and Systems. ESOP'13. Rome, Italy. LNCS, vol. 7792, pp. 125–128. Springer-Verlag.

[12]. Burghardt J., Clausecker R., Gerlach J., Pohl H. ACSL By Example: Towards a Verified C Standard Library. Version 15.1.1. Available at: https://github.com/fraunhoferfokus/acsl-by-example/raw/master/ACSL-by-Example.pdf, accessed 15.10.2017.

[13]. Cok D., Blissard I., Robbins J. C Library annotations in ACSL for Frama-C: experience report. GrammaTech, Inc. March, 2017. Available at: http://annotationsforall.org/resources/links/GT-libc-experience-report.pdf, accessed 15.10.2017.

[14]. Hatcliff J., Leavens G. T., Leino K. R. M., Müller P., Parkinson M. Behavioral interface specification languages. ACM Comput. Surv. vol. 44, issue 3, article 16, 58 pages. June 2012.

[15]. ISO/IEC 9899: 2011: C11 standard for C programming language. JTC and ISO. April 7, 2016.

[16]. Hubert T., Marché C. Separation Analysis for Deductive Verification. Heap Analysis and Verification (HAV'07). Braga, Portugal, March 2007, pp. 81–93.

[17]. Moy Y. Union and Cast in Deductive Verification. Proceedings of the C/C++ Verification Workshop. Technical Report ICIS-R07015, pp. 1–16. Radboud University Nijmegen. July, 2007.

[18]. Signoles J. E-ACSL Executable ANSI/ISO C Specification Language. Version 1.12. CEA LIST. May, 2017. Available at: https://frama-c.com/download/e-acsl/e-acsl.pdf, accessed 15.10.2017.

[19]. Marché C. [Frama-c-discuss] Frama-C/WP and CVC4 (version 1.5). Available at: https://lists.gforge.inria.fr/pipermail/frama-c-discuss/2017-August/005338.html, accessed 15.10.2017.

Автоматизация разработки моделей устройств и вычислительных машин для QEMU*

[1]*В.Ю. Ефимов <real@ispras.ru>*
[1]*А.А. Беззубиков <abezzubikov@ispras.ru>*
[1]*Д.А. Богомолов <bda@ispras.ru>*
[1]*О.В. Горемыкин <goremykin@ispras.ru>*
[1,2] *В.А. Падарян <vartan@ispras.ru>*
[1]*Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*
[2] *Московский государственный университет им. М.В. Ломоносова, 119991, Россия, г. Москва, Ленинские горы, д. 1*

Аннотация. Разработка виртуальных устройств и машин для QEMU — трудоёмкий процесс. С целью поддержки разработчика, в данной работе был проведён анализ архитектуры QEMU и процесса разработки моделей отдельных устройств и виртуальных машин для QEMU. Предлагается подход к разработке, в рамках которого начальный этап ощутимо автоматизируется, благодаря применению декларативного описания устройств и машин, а также средств графического представления разрабатываемых устройств и машин. Подход реализован в интегрированном инструменте, позволяющем разработчику QEMU получить компилируемый набор файлов с исходным Си-кодом. Разработчик задаёт параметры генерации устройств и описывает состав машины на языке Python или в графическом редакторе, обеспечивающем визуализацию текстового описания. Результатом применения инструмента при построении машины становится фактически готовый Си-код, требующий только уточнить конфигурацию процессора и обработать параметры командной строки. В случае периферийного устройства от разработчика потребуется реализовать поведенческий аспект. Проведённые эксперименты с платформами Q35 и Cisco 2621XM показали, что количество строк в описании устройства в 11-26 раз меньше числа строк получаемой заготовки на языке Си. Такая разница в объёме достигнута за счёт генерации формального кода, реализующего служебные интерфейсы QEMU. Такой код составляет ощутимую долю кода устройства, в то время как может быть сгенерирован по сравнительно небольшому описанию. Суммарный объём сгенерированного кода заготовок составил от ¼ до ¾. Исходный код разработанного инструмента доступен по адресу https://github.com/ispras/qdt.

* Работа поддержана грантом РФФИ № 16-29-09632

Ключевые слова: программный эмулятор; бинарный код; разработка виртуальных машин.

DOI: 10.15514/ISPRAS-2017-29(6)-4

Для цитирования: Ефимов В.Ю., Беззубиков А.А., Богомолов Д.А., Горемыкин О.В., Падарян В.А. Автоматизация разработки моделей устройств и вычислительных машин для QEMU. *Труды ИСП РАН*, том 29, вып. 6, 2017 г., стр. 77-104. DOI: 10.15514/ISPRAS-2017-29(6)-4

1. Введение

Виртуальные вычислительные машины (VM) применяются для решения разнообразных задач: включая исследования в рамках информационной безопасности. Одной из задач является организация контролируемого окружения для исследуемого машинного кода во время динамического анализа. Устоявшийся подход совмещает дизассемблер IDA Pro и интерактивную отладку, когда сервером удалённой отладки выступает эмулятор. Эмулятор даёт дополнительный «рубеж» изоляции между исследуемым кодом и инструментами анализа [1]. Поэтому он хорошо подходит для исследования компьютерных вирусов и другого вредоносного ПО. Одним из программных средств для организации VM является эмулятор QEMU [2].

Эмулятор QEMU наиболее подходит для этой цели, поскольку обладает рядом поолезных свойств: полностью открытый исходный код (лицензия GPL), поддержка разнообразных гостевых архитектур (Intel x86, AMD 64, ARM, MIPS, PowerPC, SPARC и др.), реализация важных, с точки зрения динамического анализа, технологий и возможностей.

Если возникает необходимость в динамическом анализе машинного кода для узкоспециализированных процессорных архитектур или малораспространённых машин, то, скорее всего, готовой VM не существует. Разработка такой VM становится сама по себе существенной проблемой, поскольку в отсутствие полностью готовой VM анализируемый код до конца не работоспособен, а работоспособность кода необходима для итеративной отладки VM. В случае QEMU, чтобы приступить к итеративной работе над VM, требуется предварительно написать значительный объем служебного кода.

Даже самые простые машины состоят из десятков устройств. Ввиду огромного разнообразия физических устройств даже в обширной библиотеке виртуальных устройств QEMU редко удаётся обнаружить требуемое или совместимое устройство, особенно, когда речь идёт об узкоспециализированных машинах. В этом случае разработчик вынужден реализовывать большое количество виртуальных устройств. Несмотря на то, что QEMU архитектурно приспособлен к добавлению новых моделей, процесс их разработки — трудоёмкая задача.

Поскольку развитие эмулятора ведётся распределенным сообществом, вопрос создания инструментальной поддержки для разработчика VM не получает должного приоритета. Нередко новые машины на основе QEMU разрабатываются закрыто, для внутренних нужд компании. Применительно к «разовой» разработке VM создание инструментов поддержки не актуально и, более того, не целесообразно в силу отвлечения ограниченных ресурсов. Тем не менее, появление автоматизированных методов разработки VM и соответствующих инструментов было бы полезно для всего сообщества разработчиков QEMU.

Для создания такого инструмента был исследован процесс разработки VM и отдельных виртуальных устройств. Используемый в настоящее время подход заключается в поиске похожей функциональности в существующих моделях и реализации требуемой по образу и подобию. При этом применяется непосредственное копирование кода, с последующими его правками и дополнениями. Функциональное наполнение вносится согласно документации, а служебный код обновляется, чтобы соответствовать требованиям актуальной версии эмулятора. В данной работе предложен метод ускорения, основывающийся на выявлении и автоматизации рутинных этапов данного процесса. Метод был реализован в программном инструменте на языке Python [4].

Дальнейший текст организован следующим образом:

- рассматриваются работы, решающие схожие проблемы;
- описывается подход, используемый в QEMU для эмуляции отдельных устройств и целых машин;
- с учётом подхода к эмуляции формулируется предлагаемый подход к автоматизации разработки моделей;
- описывается разработанный программный инструмент, реализующий подход к автоматизации;
- предлагается процесс разработки устройств и VM с использованием разработанного инструмента;
- приводятся экспериментальные данные об использовании инструмента при разработке виртуальных машин Q35 и C2621XM.

2. Обзор похожих работ

Необходимость поддержать разработку новых VM актуальна для любого развивающегося эмулятора. Даже если рассматривается эмулятор с единственной гостевой процессорной архитектурой, для работы системного кода может потребоваться определённый комплект устройств, с определённой конфигурацией регистров ввода/вывода, прерываниями, взаимодействием с внешней средой и др. Среди множества известных эмуляторов стоит выделить такие проекты, как SimNow [5], Simics [6], gem5 [7] и OVPsim [8]. Первые два эмулятора — коммерческое ПО, gem5 — ПО с открытым исходным кодом,

распространяемым по лицензии BSD. OVPsim состоит из коммерческого ядра эмуляции и открытой библиотеки VM. Все перечисленные эмуляторы используются на практике и поддерживают быстрое добавление новых VM.

2.1 AMD SimNow

Эмулятор AMD SimNow [5] предназначен для упреждающей разработки низкоуровневого системного ПО для выходящих на рынок x86 процессоров AMD, в силу чего библиотека компонент VM содержит только процессоры этой фирмы. Создание новой VM происходит в графическом редакторе, где задаются связи между компонентами машины. Связь между устройствами задаётся пользователем, причём требуется указать пару имен интерфейсов у связываемых устройств. При задании связи сразу происходит проверка её корректности. У эмулятора SimNow имеется комплект разработчика (SDK), позволяющий создавать на языке Си++ как инструменты анализа (трассировщики и т.п.), так и новые устройства. Модели устройств в SimNow реализуются на базе иерархии классов Си++, которая в основном используется для наследования библиотечных методов.

Типизация устройств крайне проста. Как правило, базовый тип устройства отсутствует, наибольшая часть методов класса реализует служебную логику, необходимую для работы эмулятора. Пример исключения из такой практики — класс *CUsbMouse*, который наследуется от классов *CUSBDevice* и *CAutomationLib*. Первый базовый класс представляет абстрактное USB-устройство, второй — реализует возможность получать управляющие команды из консоли эмулятора для конфигурации устройства или от сценария инициализации. В SDK включены исходные коды типовых моделей устройств (ПЗУ BIOS, аудио- и видео- адаптеры, мосты различных шин, контроллеры прерываний и др.), на основе которых предлагается разрабатывать собственные модели.

2.2 gem5

В эмуляторе gem5 [7] предложена более сложная иерархия типов устройств. Еще одним отличием от SimNow, стала возможность быстрого прототипирования VM на языке Python, для которого была реализована привязка Си++ API. Реализация моделей устройств, требующих большого количества вычислений, ведётся на Си++, а их интеграция и задание конфигурации VM выносится в сценарий. Поскольку gem5 разрабатывается для детального моделирования современных процессоров, особенности работы которых влияют на производительность системного и прикладного ПО, большая часть библиотечных компонентов эмулирует многоуровневую память и топологию связей между вычислительными ядрами. «Медленные» периферийные устройства в gem5 фактически не представлены.

2.3 Simics

По сравнению с предыдущими двумя эмуляторами, Simics обладает наибольшими возможностями по созданию новых моделей машин и отдельных устройств. Как и gem5 объектная модель и API Си++ имеют привязку к языку Python. Помимо того, доступен специализированный язык DML (Device Modeling Language), предназначенный для моделирования устройств в Simics. Описание устройства на DML транслируется компилятором dmlc в Си++, из которого собирается разделяемая библиотека.

В DML программа описывает *класс-устройство*, возможно наследованный от некоторого базового класса. Полями класса-устройства могут быть объекты, которые должны представлять один из встроенных классов.

Расширять перечень встроенных классов (в документации для них используют термин *тип объекта*) пользователь не может. Встроенные классы описывают как базовые примитивы виртуальной аппаратуры, так и служебные данные, используемые эмулятором в работе. Базовые примитивы сводятся к двум понятиям: регистр и соединение. Имеется пять вспомогательных классов, используемых для различных способов группировки регистров и соединений. Класс (тип), называемый attribute, описывает произвольное свойство объекта, которое необходимо сохранять при создании снимка состояния. При добавлении к устройству полей регистров и соединений среда разработки автоматически создает для их описания поля типа attribute.

```
device excalibur;

connect bus {
    interface pci;
}
...
bank databank {
    parameter function = 1;
    register r1 size 4 @ 0x0000 {
        field f1 {
            method read { ... }
            method write { ... }
        }
    }
    register r2 size 4 @ 0x0004 {
        field f2 {
            method read { ... }
            method write { ... }
        }
    }
}
```

Рис. 1. Пример DML-описания модели устройства `excalibur`
Fig. 1. `excalibur` *device model DML description example*

На рис. 1 представлено сокращённое описание учебного устройства excalibur, подключаемого к шине pci. У устройства имеется *банк регистров*, в которых размещаются некие данные. Размер регистров — 4 байта, регистр r1 размещён в банке по нулевому смещению, регистр r2 — по смещению 4. Содержимое каждого регистра полностью покрывается единственным *полем*, для которого определены методы *чтения* и *записи*. Для описания семантики действий, происходящих в устройстве, тела DML-методов выражаются на языке, расширяющем подмножество ISO Си. Добавлены некоторые конструкции, характерные для Си++, такие как new/delete, try, throw.

2.4 OVPSim

Поскольку эмулятор OVPSim изначально рассчитан на привлечение сторонних разработчиков для создания новых VM, он предлагает открытый и обширный API на языке Си, который распадается на три части:

- VMI — разработка новых процессоров,
- PPM/BHM — разработка новых периферийных устройств,
- OP — интеграция компонент VM и контролирование её работы.

```
ihwnew -name simpleCpuMemoryUart
ihwaddbus -instancename mainBus -addresswidth 32
ihwaddnet -instancename directWrite
ihwaddnet -instancename directRead
ihwaddprocessor -instancename cpu1 \
    -vendor ovpworld.org -library processor -type or1k -version 1.0 \
    -semihostname or1kNewlib \
    -variant generic
ihwconnect -bus mainBus -instancename cpu1 -busmasterport INSTRUCTION
ihwconnect -bus mainBus -instancename cpu1 -busmasterport DATA
ihwaddmemory -instancename ram1 -type ram
ihwconnect -bus mainBus -instancename ram1 \
    -busslaveport sp1 -loaddress 0x0 -hiaddress 0x0fffffff
ihwaddmemory -instancename ram2 -type ram
ihwconnect -bus mainBus -instancename ram2 \
    -busslaveport sp1 -loaddress 0x20000000 -hiaddress 0xffffffff
ihwaddperipheral -instancename periph0 \
    -vendor freescale.ovpworld.org -library peripheral \
    -version 1.0 -type KinetisUART
ihwsetparameter -handle periph0 -name outfile -value uartTTY0.log \
    –type string
ihwconnect -instancename periph0 \
    -busslaveport bport1 -bus mainBus \
    -loaddress 0x100003f8 -hiaddress 0x100013f7
ihwconnect -instancename periph0 -netport DirectWrite -net directWrite
ihwconnect -instancename periph0 -netport DirectRead -net directRead
```

Рис. 2. Пример TCL-сценария для создания заготовки VM
Fig. 2. VM draft creation TCL script example

Ускорение разработки на начальном этапе обеспечивает утилита iGen (входит в SDK), которая по декларативному описанию на языке TCL генерирует заготовки. Утилита поддерживает автоматическое создание набора Си-файлов для моделей процессоров и устройств, встраиваемых в эмулятор; способна интегрировать интерфейс разрабатываемого Си-модуля с SystemC TLM2. Результат работы утилиты — полный комплект файлов с исходным кодом, которые компилируются в динамически загружаемую библиотеку. Описание определяет перечень регистров устройства, их отображение на память, интерфейсы шин. Поведение устройства реализуется в OVPSim через функции обратного вызова, сгенерированные файлы содержат объявления функций, а их определения имеют пустые тела, которые разработчик реализует самостоятельно.

На рис. 2 приведён пример TCL-сценария, описывающего простую VM с одним процессором OpenRISC 1000, 32-х разрядным адресным пространством, на два диапазона которого отображено ОЗУ, и регистрами UART, которые, в свою очередь, отображены на диапазон адресов с 0x100003f8 по 0x100013f7.

3. Разработанный подход к автоматизации

Рассмотренные в предыдущем разделе подходы к ускорению разработки предполагают использование декларативного описания интерфейсов устройств и всей VM. По описанию строится компилируемый код, представляющий собой заготовку устройства без реализации поведенческих функций или готовую VM, требующую незначительной «ручной» доработки. Для этого инструменты сборки дополняются транслятором описаний в язык Си/Си++ или любой другой, поддерживаемый эмулятором.

Целесообразно применять графические средства разработки для улучшения наглядности компонуемых VM. К сожалению, базовая версия QEMU до сих пор не предлагает никаких средств ускорения разработки VM. Их создание потребует учёта особенностей внутренней архитектуры данного эмулятора и технологических особенностей его разработки, которую ведет многочисленное распределённое сообщество.

3.1 Объектная модель QEMU

Основой использованного в QEMU подхода к моделированию машин и их компонент является «объектная модель QEMU» (англ. *QEMU Object Model* или, далее по тексту, *QOM*). Данная модель применяется не только для моделирования машин и их компонент, но и для реализации вспомогательных возможностей. QOM формирует иерархию *типов* (type), являясь реализацией парадигмы ООП на языке Си. Каждый тип имеет

уникальное строковое *имя*. Тип описывает *класс* (class) и *экземпляр* (instance). Экземпляров может быть много, в то время как класс один.

Иерархия QOM получается путём *наследования* (по аналогии с ООП) — установления отношения между двумя типами, таким образом, что один тип называется *ребёнком*, а другой — *родителем*. Ребёнок копирует всю информацию из родителя — *наследует*. Множественное наследование не поддерживается. Тип может быть *абстрактным*: такой тип не может иметь экземпляров.

В отличие от распространённых объектно-ориентированных языков, в QOM тип object (объект) не является корнем всей иерархии типов, а только одним из них. Данный тип добавляет *свойства* к экземплярам и классам. Свойство описывается строковым именем (уникальным в пределах объекта или класса), функциями доступа (присваивания (set) и разыменования (get)) и типом этого свойства. Со свойством связана и другая информация, но её рассмотрение выходит за рамки данной работы. Тип свойства ограничивает область его допустимых значений.

Моделирование VM и её элементов основано на потомках типа object. К ним относятся:

- машина (machine),
- устройство (device),
- шина (bus),
- запрос прерывания (irq),
- участок памяти (qemu:memory-region).

Модели VM, шин и устройств должны быть, соответственно, потомками machine, bus и device. Типы irq и участка памяти являются инфраструктурными, они используются в общем API, и их уточнение обычно не требуется. Дальнейшее наследование шин и устройств происходит по принадлежности к стандарту шины. При этом вводятся промежуточные типы, реализующие общий функционал. Конкретная модель устройства наследуется от типа соответствующего стандарту её шины. При реализации модельного ряда устройств добавляется промежуточный тип с общими для всего ряда особенностями.

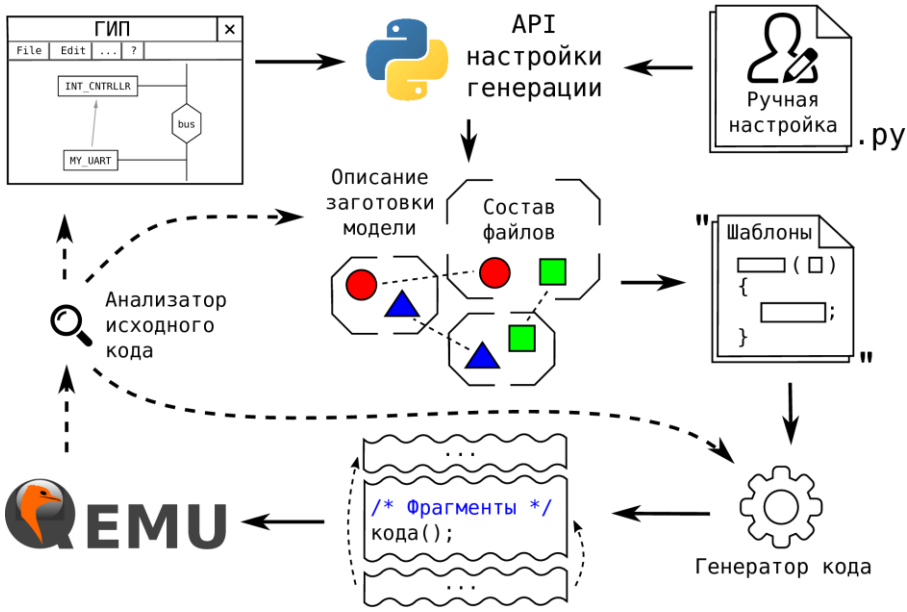


Рис. 3. Схема работы генератора заготовок моделей устройств и машин
Fig. 3. Machine & device draft generator system scheme

3.2 Метод автоматизации

На рис. 3 схематично изображён метод автоматизированной разработки эмулятора QEMU. Вручную или с помощью разработанного графического редактора разработчик задаёт основные параметры разрабатываемых моделей. На основе указанных параметров инструмент автоматически генерирует заготовки программного кода моделей. Эти заготовки не обладают законченной функциональностью, но готовы к компиляции.

Генерация основана на шаблонах — фрагментах программного кода, которые подстановками строковых параметров приводятся в синтаксически корректный код на языке Си. По исходному коду текущей версии QEMU генерируются необходимые включения заголовочных файлов и используемые макросы. Дальнейшая работа разработчика заключается во внесении в заготовку кода, соответствующего не формализованным особенностям работы устройства.

Метод шаблонов основан на следующей особенности кода QEMU. Любая модель, как устройство, так и машина, является частью эмулятора. Следовательно, её код, среди прочего, содержит фрагменты, обусловленные требованиями эмулятора (интеграция с QOM и т. д.), а не особенностями реального объекта, который эта модель описывает. Таким образом, условно

код модели можно разделить на две части: *индивидуальную* и *интерфейсную*.

В первую очередь рассмотрим это разделение для устройств. Индивидуальная часть задаёт поведение устройства в его программной реализации. Именно эта часть определяет то, как именно будет эмулироваться присутствие конкретно этого устройства в системе; делает его модель особенной, относительно других устройств, уникальной. При этом модель устройства является частью инфраструктуры QEMU, и индивидуальная часть должна эмулировать поведение устройства, используя возможности, предоставленные посредством API QEMU. В модели всегда можно выделить часть кода, взаимодействующую с этим API, она и называется *интерфейсной*. То есть, интерфейсная часть кода служит связкой между кодом индивидуальной части модели и остальным кодом эмулятора.

Индивидуальная часть обычно сформулирована на естественном языке в документации на устройство. При этом отсутствует единый формат формального описания, которого бы придерживались производители. Ввиду этого автоматизация разработки этой части весьма затруднительна и выходит за рамки этой работы.

С другой стороны, интерфейсная часть всех устройств очень похожа. Отличия заключаются, в основном, в перечне и количестве используемых моделью внешних интерфейсов, а также в именах собственных. Т. е. её параметры хорошо формализуемы.

Отдельно нужно сказать о применимости данного подхода к целой машине. В QEMU присутствует развитый API для интеграции устройств в единое целое, то есть в VM. Разработанный API во многом похож на API из QEMU. С помощью разработанного API можно формально описать полноценную VM. При этом имеются следующие ограничения.

1. Устройства, входящие в VM, должны иметь интерфейсную часть, реализованную в полном соответствии с принятым в QEMU подходом к написанию моделей устройств. Иначе несоответствие придётся компенсировать вручную.
2. Сгенерированная машина не поддаётся настройке, так как все её параметры зафиксированы на уровне исходного кода. Реализация возможности настраивать машины требует внесения кода вручную.

Первое ограничение не существенно, если преобладающая часть устройств VM реализуется вместе с ней по формальному описанию с помощью разработанного инструмента. Такие устройства будут иметь совместимую интерфейсную часть. Но в QEMU присутствует ряд устройств, которые были реализованы еще до того, как был выработан текущий подход к написанию моделей устройств. На данный момент не все из них были переписаны в соответствии с этим подходом. То есть они реализуют свою индивидуальную часть в обход новейших возможностей API для интерфейсной части. Если же

нужно использовать подобную модель устройства, то можно использовать разработанный инструмент, сгенерировав с его помощью новую интерфейсную часть, а затем скопировать индивидуальную часть из оригинала.

Второе ограничение заключается в следующем. Часть параметров машины обычно задается пользователем (а не разработчиком) только в момент запуска эмулятора через CLI (Command Line Interface):

- файл-образ ПЗУ (HDD, микросхема CMOS-памяти, CD, DVD, и т.п.),
- оконечную точку UART (виртуальный терминал, файл, и т.п.),
- способ подключения сетевого порта (TAP-адаптер, Ethernet по UDP и т. п.) и т. д.

Поддержка CLI требует внесения в код заготовки машины специального кода. Автоматизация этого процесса для наиболее часто используемых опций CLI является направлением дальнейших исследований.

3.3 API генератора заготовок

Состав интерфейсной части устройства определяется:

- обязательно используемыми служебными интерфейсами QEMU;
- потребностями индивидуальной части.

QEMU предоставляет ряд интерфейсов, из которых в интерфейсную часть выбираются только нужные. Код, соответствующий отдельно взятому интерфейсу, единообразен. Его можно получать из некоторого набора строковых заготовок, путём подстановки параметров. Часть параметров одних интерфейсов может быть связана с параметрами других интерфейсов. Часто для одного интерфейса необходимо сгенерировать несколько фрагментов кода. При этом фрагменты должны следовать в правильном порядке как относительно друг друга, так и относительно фрагментов других интерфейсов. Это требование следует из синтаксиса языка Си. Например, если один из аргументов функции является указателем на структуру, то сама структура должна быть объявлена выше. Кроме этого следует учитывать семантику фрагментов (близкие по смыслу фрагменты должны располагаться близко) и требования стиля программирования QEMU.

Генератор заготовок предоставляет API, упрощающий учёт этих и других особенностей. Состав API генератора условно можно разбить на две части:

- для *использования* шаблонов;
- для *добавления* шаблонов.

Каждый шаблон использует обе части. Через API использования шаблонов он добавляет себя в инфраструктуру инструмента, давая возможность применять себя для генерации. А с помощью API добавления шаблон реализует генерацию кода в соответствии с переданными ему параметрами.

3.1.1 Интерфейс использования шаблонов

Интерфейс использования шаблонов ориентирован на получение от пользователя перечня интерфейсов QEMU, требуемых разрабатываемой моделью, и их параметров. Для этого интерфейс использования предоставляет иерархию классов. Она построена по тому же принципу, что и иерархия классов QOM.

- *QOMDescription*
  - *SysBusDeviceDescription*
  - *PCIExpressDeviceDescription*
  - *MachineNode*

QOMDescription является базовым классом. Он не предназначен для использования. *SysBusDeviceDescription* описывает устройство на системной шине. *PCIExpressDeviceDescription* описывает PCI устройство QEMU.

Перечисленные классы выполняют роль контейнеров для параметров. Но, в то время как для описания устройств системной шины или PCI достаточно одного объекта перечисленных классов, для описания VM требуется ещё одна иерархия, рассмотренная ниже. Объект класса *MachineNode* служит контейнером для объектов той иерархии и прочих параметров генерации. Совместно объекты классов-потомков *QOMDescription* образуют *проект* (*QProject*), аккумулирующий данные для генерации кода. Классы, описывающие проект, устройства, VM и её состав, поддерживают сохранение этой информации в файл.

3.1.2 Модель вычислительной машины

Содержимое VM описывается с использованием следующей иерархии классов.

- *Node*
  - *BusNode*
    - *SystemBusNode*
    - *PCIExpressBusNode*
    - *ISABusNode*
    - *IDEBusNode*
    - *I2CBusNode*
  - *DeviceNode*
    - *SystemBusDeviceNode*
    - *PCIExpressDeviceNode*
  - *IRQLine*
  - *IRQHub*
  - *MemoryNode*
    - *MemoryLeafNode*
      - *MemoryAliasNode*
      - *MemoryRAMNode*

- *MemoryROMNode*

Node содержит уникальный идентификатор узла VM.

BusNode содержит все данные, описывающие шину любого типа. Все дочерние классы конкретизируют эти данные. Они были введены для сокращения объёма кода, необходимого для ручной работы с программным интерфейсом. При использовании графического редактора это не актуально.

DeviceNode описывает параметры самого устройства и настройки его интеграции. Классы, наследуемые от *DeviceNode,* расширяют этот список в соответствии с шаблонами устройств для конкретных стандартов шин.

IRQLine (линия) и *IRQHub* (концентратор) описывают распространение прерываний между устройствами (ту его часть, которая по какой-то причине не инкапсулирована в шину). Концентратор прерываний используется в случаях, когда одно прерывание должно быть доставлено в несколько устройств и/или может быть получено из нескольких устройств (так как линия прерывания соединяет строго два конца).

Следующие классы используются для явного создания участков памяти. Хотя большую часть адресного пространства машины определяют сами устройства, некоторые участки памяти, соответствующие ОЗУ, некоторым ПЗУ и т. п., должны быть добавлены явно. Для ОЗУ используется *MemoryRAMNode*, а для ПЗУ *MemoryROMNode*. *MemoryAliasNode* используется для ссылки на один участок адресного пространства из другого диапазона адресов. *MemoryLeafNode* (лист) является служебным промежуточным классом, запрещающим добавлять участки в участки, не являющиеся контейнерами.

3.1.3 Хранение настроек генерации

Объекты перечисленных выше классов объединены в проект (QProject). Формат файла, хранящего проект, основывается на возможности интерпретатора языка Python динамически добавлять код в программу. Сохранённый проект представляет собой код на языке Python. Поскольку инструмент сам написан на языке Python, данное решение существенно упростило разработку, поскольку не требует разработки специальных лексического, синтаксического и семантического анализаторов. Загрузка проекта представляет собой выполнение кода, в результате которого вырабатывается определение переменной, ссылающейся на объект класса *QProject*, который эквивалентен ранее сохранённому объекту. Генератор файлов разрабатывался таким образом, чтобы форматирование выводимого кода было удобно для человека.

Однако при сохранении *не* гарантируются:

- равенство *незначащих* пробельных символов;
- синтаксически незначащий порядок фрагментов кода (порядок определения аргументов конструкторов со значениями по умолчанию, порядок восстановления несвязанных объектов, и т.п.);

- сохранность имён переменных;
- использование тех же самых конструкций языка (например, программист может создавать несколько объектов в цикле, в то время как инструмент сгенерирует для этих объектов развёрнутый код);
- сохранность комментариев, если таковые были внесены вручную в файл проекта.

Эти особенности создают сложности при ручной работе с файлами проекта, и особенно при хранении файлов с использованием системы контроля версий. Решение этой проблемы является одним из направлений дальнейших исследований.

3.1.4 Способ генерации Си-кода и макрокоманд

Шаблоны реализованы с помощью форматных строк. Генерация кода заключается в подстановке параметров в форматные строки. Из полученных фрагментов кода затем составляются файлы. Форматные строки являются низкоуровневым способом определения шаблонов. Они ограничены в возможностях программной обработки. В связи с этим для разработки шаблонов был введён вспомогательный интерфейс. Будем называть его *интерфейсом добавления шаблонов*. Он основан на форматных строках, но предоставляемые им инструменты ориентированы на генерацию базовых конструкций языка Си. Поэтому интерфейс напоминает программную реализацию АСД, однако есть принципиальное отличие: он поддерживает и язык препроцессора.

Параллельно была предпринята попытка использовать для определения шаблонов АСД языка Си. Библиотека PyCParser [9], реализует двустороннее преобразование. Однако у применения АСД были обнаружены следующие недостатки:

- не поддерживается препроцессор, макросы которого активно используются в интерфейсной (и не только) части кода устройства в силу принятого в сообществе разработчиков QEMU стиля программирования;
- не учитываются незначащие символы (пробелы, переносы строк, комментарии), что требует доработки генератора PyCParser, чтобы он генерировал код, не противоречащий стилю программирования QEMU.

Основой интерфейса добавления шаблонов является модель *гибридного языка*, сочетающего подмножества языков Си и препроцессора. Выбор конструкций, которые попали в гибридный язык, обусловлен требованиями к оформлению кода QEMU, подробное рассмотрение этого вопроса выходит за рамки данной работы. Модель гибридного языка описывает содержимое заголовков и модулей языка Си. Разработчик описывает содержимое шаблона с точки зрения того, какие конструкции должны быть добавлены в файл в

соответствии с этим шаблоном. Причём один шаблон может касаться нескольких файлов, равно как и один файл может содержать конструкции из нескольких шаблонов. Некоторые конструкции из шаблонов связаны с уже существующими в QEMU конструкциями. Поэтому интерфейс позволяет определять содержимое существующих файлов. Описание существующих конструкций носит декларативный характер. Многие подробности могут быть пропущены, так как генерация не предполагается. Иными словами, используется принцип минимально достаточной информации.

При описании содержимого файла используются следующие классы.

- *Type*
  - *Structure*
  - *Function*
  - *Enumeration*
  - *Pointer*
  - *Macro*
  - *TypeReference*
- *Initializer*
- *Variable*
- *Usage*

Structure, Function, Enumeration и *Pointer* соответствуют структуре, функции, перечислению и указателю в языке Си. *Macro* используется для директив *#define* препроцессора.

Объект *TypeReference* (ссылка на тип) используется для ссылок на типы из других файлов. Любой тип может присутствовать непосредственно только в файле, где он объявлен. Но, когда один файл включается в другой с помощью директивы *#include,* второй косвенно содержит все типы первого. Чтобы отличить включённые типы от непосредственно определённых в данном файле применяется *TypeReference*. Ссылка создаётся для каждого включённого типа, включая те, которые уже были ссылками. Такой подход даёт возможность при генерации кода однозначно определить следует ли сгенерировать непосредственное определение типа по шаблону, или сгенерировать включение заголовочного файла, где он определён.

Одним из направлений дальнейших исследований является синтаксический анализ файлов QEMU с целью автоматического создания объектов, описывающих существующие типы. В настоящий момент такая функциональность реализована только для макросов препроцессора и опирается на функционал модифицированного препроцессора из библиотеки PyCParser.

Класс *Variable* (переменная) описывает пару: «тип, имя». Если к переменной нужно добавить начальное значение, то применяется класс *Initializer* (инициализатор).

Класс *Usage* (использование) применяется, когда требуется добавить инициализатор к *muny*, а не к переменной. Единственным примером в настоящее время является макрос. Инициализатор используется для расстановки значений при генерации вызова макроса.

Рассмотренная модель не является законченной. Но даже в таком варианте она позволяет с достаточной гибкостью описывать шаблоны устройств и машину, используемые при генерации интерфейсной части их кода. Развитие этой модели является направлением дальнейших исследований.

3.4 Генерация кода

Генерация кода заключается в получении из шаблонов фрагментов кода и объединении их в файлы. Причём полученные фрагменты должны быть упорядочены. Для этого была разработана вспомогательная модель файла с исходным кодом. Она оперирует неделимыми текстовыми фрагментами, из которых состоит файл, и порядковыми связями между ними. Элементы этой модели конструируются из элементов модели языка Си и препроцессора.

3.4.1 Модель файла с исходным кодом

Описанная выше модель языка Си и препроцессора не обеспечивает генерацию синтаксически корректного файла. Её задача: сгенерировать законченные *фрагменты* генерируемого файла. При этом остаётся решить следующие задачи:

- расположить фрагменты в синтаксически корректном порядке;
- обеспечить смысловую группировку фрагментов;
- соблюсти требования стиля программирования QEMU;
- минимизировать количество директив включения заголовков и др.

Далее эти задачи рассматриваются подробнее.

3.4.2 Сортировка фрагментов

Язык Си накладывает жёсткие ограничения на порядок определения различных символов. Например, тип должен быть объявлен до того, как будет создана переменная этого типа, или будет объявлена функция, принимающая аргумент такого типа. Подробное рассмотрение всех возможных примеров выходит за пределы данной статьи. Важно заметить, что почти все фрагменты связаны друг с другом, образуя ациклический граф, и для обеспечения синтаксически корректного порядка используется топологическая сортировка.

Помимо требований синтаксиса есть требования стиля программирования и здравого смысла, согласно которым, фрагменты должны следовать в следующем порядке:

1. включение заголовков;
2. объявление типов языка Си и макросов;
3. объявления функций;

4. определение функций, глобальных переменных и прочий код.

3.4.3 Учёт зависимостей и взаимосвязь с существующим кодом

Как уже отмечалось, для обеспечения видимости символов, объявленных в других файлах, генерируются директивы *include*. При этом имеется ряд тонкостей. Например, заголовочные файлы сами используют *include* для подключения других файлов, поэтому подключение одного файла может заменить подключение нескольких.

Эта особенность используется инструментом для сокращения количества подключаемых заголовков на основе анализа графа включения заголовков. Граф строится автоматически с использованием модифицированного препроцессора из библиотеки PyCParser.

Инструмент учитывает и другие особенности, рассмотрение которых выходит за рамки данной статьи.

3.4.4 Встраивание кода в QEMU

Для добавления заготовки устройства или платформы в QEMU кроме создания соответствующего исходного кода на Си, нужно зарегистрировать новые модули компиляции в системе сборки. Инструмент имеет соответствующую функциональность.

3.4.5 Адаптация к изменениям QEMU

QEMU является развивающимся проектом. Это приводит к тому, что в нём периодически происходят изменения, делающие шаблоны несовместимыми с новой версией.

Для решения этой проблемы используется эвристический подход. Все аспекты поведения инструмента, зависящие от версии QEMU, называются *эвристиками*. Так как один аспект работы может меняться многократно, то каждая эвристика представлена одной или несколькими записями в базе данных.

Код инструмента получает доступ к требуемой эвристике по *строковому ключу* — уникальному имени эвристики. Значением эвристики может быть любая сущность языка Python: от целочисленной константы до класса или модуля. Таким образом, при необходимости, можно *подменить* почти всю реализацию инструмента.

Каждая запись об эвристике имеет как минимум два значения: *новое* и *старое*. Запись привязывается к SHA1-идентификатору изменения в Git-графе [10] истории QEMU.

В инструменте реализован алгоритм, позволяющий для заданных SHA1, базы эвристик и Git-истории вычислить значения для всех имеющихся в базе ключей. Хранение обоих значений в каждой записи об эвристике избыточно. Но эта избыточность используется для проверки непротиворечивости записей.

3.5 Графический редактор

Все возможности инструмента доступны разработчику посредством интерфейса использования шаблонов. Для описания устройств и VM достаточно произвольного текстового редактора. Однако применение графического редактора, спроектированного специально для работы с этим интерфейсом, имеет следующие преимущества.

- Исключены лексические ошибки в именах переменных, названиях элементов интерфейса, а также синтаксические ошибки: разработчик вводит только значения параметров.
- Для многих значений параметров в QEMU определены макросы, использование которых предпочтительнее, согласно стилю программирования QEMU. Редактор, проанализировав код QEMU, может предоставить разработчику список доступных макросов, обычно применяемых с данным типом параметра. Например:
    - идентификатор PCI,
    - имя типа QOM,
    - список свойств выбранного устройства и т. п.
- Исключены некоторые семантические ошибки (например, в редакторе не предусмотрена возможность соединения линией прерывания двух шин, в то время как разработчик может написать подобное по ошибке). Имеется возможность дополнить редактор средствами поиска менее очевидных семантических ошибок.
- Все доступные параметры сосредоточены в *виджетах* и сопровождаются названиями на естественном языке. В большинстве случаев знания QEMU достаточно, чтобы понять суть параметра, не обращаясь к справочной информации.
- Интерпретация машины в виде схемы. Эта возможность особенно актуальна при разработке многоэлементных VM с большим количеством связей, так как на схеме легче ориентироваться, чем в тексте.

Для реализации графического редактора используется API Tkinter [11]. Данный интерфейс содержит всю необходимую функциональность и прост в развёртывании, так как включён в дистрибутивы Python.

3.6 Автоматизированный процесс разработки

С использованием разработанного инструмента процесс разработки как модели устройства, так и VM принимает следующую форму. Его можно разбить на 4 этапа: *ознакомление* с документацией, *подготовка* эмулятора, *генерация* заготовок интерфейсной части и *реализация* индивидуальной части.

При ознакомлении с документацией задача разработчика: оценить состав машины: какие устройства потребуется реализовать, как их связать в единую VM и т. д. Эта информация потребуется для генерации заготовок. Поскольку

инструмент генерирует заготовки с учётом текущей версии кода QEMU, может потребоваться внесение подготовительных изменений в код, чтобы задействовать максимум возможностей инструмента. Например, добавление новых идентификаторов PCI. Затем, разработчик задаёт параметры и получает заготовки — выполняет этап генерации. Наконец, он переходит к реализации индивидуальных частей кода устройств и уточнению заготовок VM. При этом происходит более детальное изучение документации, отладка и корректирование кода.

Проделанные в данной работе оценочные эксперименты проходили именно по такому сценарию.

4. Экспериментальные результаты

Для проверки состоятельности предложенного подхода с помощью разработанного инструмента были реализованы две VM:

- IBM PC совместимая машина Q35 на базе одноимённого набора микроконтроллеров фирмы Intel;
- маршрутизатор CISCO серии 2600 (C2621XM).

В качестве основной метрики эффективности автоматизации было выбрано количество строк. Подсчёты производились по истории Git и сгруппированы поэтапно. Замеры производились по разнице между начальной и конечной версией каждого этапа. Кроме этого для каждой VM приведена суммарная разница между базовой версией QEMU и полностью реализованной машиной. Важно иметь ввиду, что в приведённой ниже статистике изменение одной строки представлено как 1 удаление и 1 добавление в силу особенностей Git.

4.1 Q35

За основу Q35 была взята её реализация, уже имеющаяся в QEMU версии 2.9.5. Целью данного эксперимента была проверка возможностей разработанного инструмента. Выбор Q35 обусловлен тем, что эта машина является одной из самых сложных машин, реализованных в QEMU.

В ходе анализа исходного кода реализации Q35 был составлен перечень устройств и выявлена их взаимосвязь. Эти данные были формально описаны средствами разработанного инструмента. Полученная схема машины представлена на рис. 4.

Поскольку реализация Q35 не полностью соответствует всем требованиям QEMU, потребовался подготовительный этап. На этапе подготовки в QEMU вносились изменения, которые могут ограничить возможности инструмента:

- в заголовочный файл вынесена структура, описывающая устройство MC146818 RTC, и макрос динамического приведения к типу данного устройства;

- добавлены функции инициализации глобальных переменных *slave_pic* и *isa_pic*, в которые записываются ссылки на два устройства «i8259».

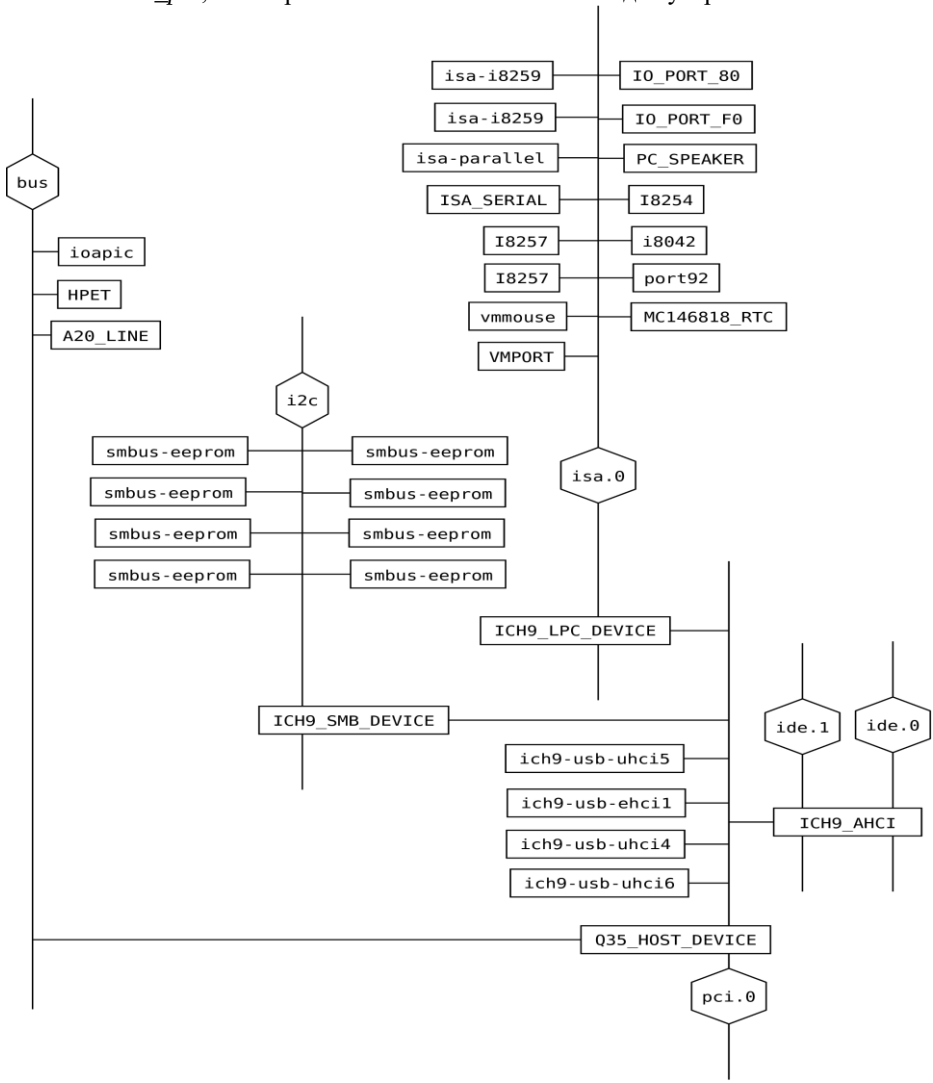


Рис. 4. Схема Q35
Fig. 4. Q35 scheme

В табл. 1 приведены оценки объёма работы по реализации Q35.

Табл. 1. Оценка объёма реализации Q35
Table 1. Q35 development steps evaluation

Этап	**Затронуто файлов**	**Вставлено строк**	**Удалено строк**
Подготовка	4	42	31
Генерация	8	599	0
Реализация	5	162	93
Суммарно	12	803	31

На этапе генерации с помощью инструмента были сгенерированы заготовки VM Q35 и трёх устройств, входящих в её состав: «A20 line», «port 80» и «port F0». В оригинальной VM Q35 эти устройства реализованы в коде самой машины.

На этапе реализации в заготовку VM были внесены следующие изменения:

- добавлен процессор;
- добавлена настройка BIOS;
- добавлен режим совместимости MS-DOS с исключениями FPU;
- выделена память под eeprom;
- добавлены к машине свойства, хранящие указатели на объекты устройств IOAPIC и PCI HOST;
- произведена инициализация ICH9 PM, IDE, PC CMOS, VGA, NIC и ACPI;
- скорректированы имена переменных.

Необходимость инициализации некоторых устройств на этапе реализации связана с тем, что эти устройства поддерживают настройку посредством CLI.

Стоит заметить, что в настоящее время инструмент не поддерживает группировку связанных по смыслу переменных в массивы и инициализацию их в цикле. Ввиду этого, инициализация прерываний была вынесена на этап реализации. Это позволило произвести регистрацию прерываний в цикле, тем самым обеспечив краткость кода VM. Инициализация прерываний заняла 20 строк кода, что составляет менее 3% от кода всей модели.

Таким образом, приблизительно 75% кода платформы было сгенерировано автоматически и лишь 11% пришлось модифицировать.

Итоговая версия платформы Q35 была успешно протестирована. Тестирование состояло из загрузки ОС Windows 7 и запуска Internet Explorer с последующим открытием страницы *google.com*.

4.2 C2621XM

За основу C2621XM взята его модель из эмулятора с открытым исходным кодом Dynamips [13]. Его разработка на данный момент заморожена, если не считать проект GNS3 [14], который использует Dynamips для эмуляции маршрутизаторов, коммутаторов и концентраторов, исправляя в нём ошибки.

На этапе подготовки в QEMU вносились следующие изменения:

- реализован MMU;
- реализована внутренняя коммутация прерываний процессора;
- исправлены некоторые регистры процессора специального назначения согласно с документацией [15];
- добавлены идентификаторы новых PCI устройств.

Первые два изменения позволили использовать процессор в режиме полносистемной эмуляции: до этого поддерживалась только эмуляция ABI ОС.

На этапе ознакомления, исходя из анализа исходного кода Dynamips, был составлен перечень устройств и выявлена их взаимосвязь. Схема VM представлена на рис. 5.

Начальной версией QEMU был выбран последний на тот момент выпуск 2.9.0. Все устройства, использованные в C2621XM, отсутствовали в QEMU и были перенесены из Dynamips. При этом все заготовки были сгенерированы с помощью инструмента. Стоит отметить, что устройства в Dynamips и, как следствие, их перенесённые в QEMU версии реализованы не полноценно, а лишь до той степени, чтобы удовлетворять потребностям некоторых версий системного ПО.

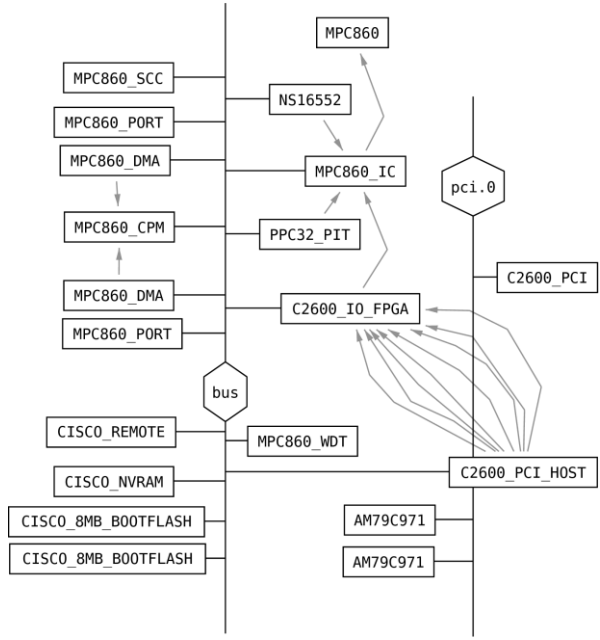


Рис. 5. Схема маршрутизатора C2621XM
Fig. 5. C2621XM router scheme

На этапе реализации была перенесена только индивидуальная часть устройств из Dynamips, а также скорректирована заготовка VM. Корректировка VM заключалась в следующем:

- связка параметров VM с CLI QEMU:
  - образы ПЗУ,
  - подключение сетевых интерфейсов,
  - подключение символьных устройств;
- реализация специального кода;
- корректировка имён переменных.

В табл. 2 приведена общая оценка объёма работы, а в табл. 3 дано сравнение количеств строк настроек генерации и полученных из них заготовок для некоторых устройств.

Табл. 2. Оценка объёма реализации C2621XM
Table 2. C2621XM development steps evaluation.

Этап	**Затронуто файлов**	**Вставлено строк**	**Удалено строк**
Подготовка	8	128	35
Генерация	37	2186	0
Реализация	31	4747	419
Суммарно	45	6642	35

Таким образом, приблизительно 1/3 кода всего маршрутизатора была сгенерирована автоматически. Причём только 1/5 часть сгенерированного кода потребовала модификации.

Табл. 3. Объёмы конфигурации и заготовок для устройств из C2621XM
Table. 3. Device generation configuration & resulting draft sizes for C2621XM

Устройство	**Конфигурация**	**Сгенерировано**	**Отношение**
MPC860_IC	6	125	20,8
C2600_PCI	7	82	11,7
AM79C971	12	175	14,6
NS16552	7	181	25,9

Итоговая версия маршрутизатора была успешно настроена для маршрутизации пакетов между двумя сетями (по одной на каждый интерфейс) и обеспечивала стабильное соединение в течении тестового времени (приблизительно 12 часов). В качестве генератора трафика использовалась утилита *ping*, настроенная на отправку ICMP запросов длиной 60кБ и 50кБ с машин из противоположных сетей. Сбоев замечено не было. Среднее время запроса составило 12мс.

5. Заключение

В ходе данной работы был исследован процесс разработки моделей устройств и машин для эмулятора QEMU. В изученном процессе был выделен рутинный начальный этап, хорошо поддающийся формализации. А именно, логику модели можно условно разделить на две части: индивидуальную и интерфейсную. Последняя служит прослойкой между индивидуальной частью и остальной VM, а также внешней средой. При этом интерфейсная часть содержит много формального кода, который может быть сгенерирован по сравнительно небольшому количеству параметров. Беглый анализ документации на устройство позволяет сформулировать эти параметры. Таким образом, в разработке устройства можно выделить промежуточный этап между изучением документации и реализацией. В течение этого этапа происходит генерация интерфейсного кода по заданному набору параметров.

Был разработан программный инструмент, автоматизирующий этап генерации. Инструмент поддерживает генерацию заготовок для устройств системной шины и шины PCI. Файлы настройки, используемые при генерации модели устройства, содержат в 11-26 раз меньше строк в сравнении с получаемой заготовкой. Этот подход может быть применён не только к устройству, но и ко всей VM в целом. Однако для VM подобной разницы в количестве строк достичь не удалось, ввиду того, что QEMU использует объектную модель для компоновки машин, что уже является шагом в сторону сокращения объёма кода.

По аналогии с этой объектной моделью в инструменте была разработана схожая модель, которая позволила представить VM в виде схемы в рамках графического редактора. Схема облегчает восприятие состава и связей VM для разработчика. Кроме того, имеется возможность применить малые автоматизации компоновки VM. Предложенный подход пригоден как для полной реализации VM вместе со всеми устройствами (при условии, что в QEMU уже есть поддержка соответствующей архитектуры процессора), так и для реализации VM с использованием уже имеющихся в QEMU устройств.

Инструмент был протестирован на двух VM: машина на базе Intel Q35 и машина C2600, являющаяся маршрутизатором фирмы CISCO серии 2600 (C2621XM). Они представляют две обозначенные крайности применимости данного инструмента, а именно, для Q35 в QEMU уже присутствовали все требуемые устройства, хотя некоторые из них пришлось привести в соответствие с требованиями объектной модели QEMU, используя данный инструмент. Для C2621XM, напротив, в QEMU отсутствовали все требуемые устройства. Доля сгенерированного кода составляет от ¼ до ¾ (в зависимости от количества уже реализованных устройств).

Перспективные направления исследований упоминались по тексту статьи. К числу наиболее важных из них относятся следующие направления.

- Автоматизация разработки поддержки процессорной архитектуры, что позволило бы полностью покрыть инструментом начальный этап разработки даже VM с новой для QEMU процессорной архитектурой.
- Поддержка добавления новых стандартов шин и генерации заготовок для устройств остальных, реже используемых шин, уже имеющихся в QEMU.
- Развитие графического редактора с целью внедрения в него малых автоматизаций, в совокупности облегчающих и ускоряющих процесс разработки.
- Реализация обратной связи о состоянии VM из запущенного QEMU с отображением информации времени выполнения на схеме машины, что позволит использовать инструмент ещё и для отладки в течение цикла итеративной разработки. Обратную связь предполагается организовать, запустив QEMU под отладчиком и контролируя состояние переменных времени выполнения и потока управления. Поскольку инструмент сам генерирует код VM, то информация, необходимая для сопоставления переменных времени выполнения и элементов схемы для него доступна.

Список литературы

[1]. Довгалюк П.М., Макаров В.А., Падарян В.А., Романеев М.С., Фурсова Н.И. Применение программных эмуляторов в задачах анализа бинарного кода. Труды ИСП РАН, том 26, вып. 1, 2014 г., стр. 277-296. DOI: 10.15514/ISPRAS-2014-26(1)-9

[2]. F. Bellard QEMU, a Fast and Portable Dynamic Translator. USENIX Annual Technical Conference, FREENIX Track. USENIX, 2007. P. 41-465 p.

[3]. J. Bennett. Howto: GDB Remote Serial Protocol. Writing a RSP Server. Application Note 4. Issue 2, Embecosm, Доступно по ссылке: http://www.embecosm.com/appnotes/ean4/embecosm-howto-rsp-server-ean4-issue-2.pdf, ноябрь 2008.

[4]. Страница языка программирования Python. Доступна по ссылке: https://www.python.org/, дата обращения 06.07.2017.

[5]. Страница AMD SimNow Simulator. Доступна по ссылке: http://developer.amd.com/simnow-simulator/, дата обращения 12.10.2017.

[6]. D. Aarno, J. Engblom. Software and System Development using Virtual Platforms. Full-System Simulation with Wind River Simics. Elsevier Inc. 15.09.2014. 366p.

[7]. N. Binkert, B. Beckmann, G. Black, S.K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D.R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M.D. Hill, and D.A. Wood. The gem5 simulator. SIGARCH Comput. Archit. News 39, 2, August 2011, 1-7pp.

[8]. Страница с документацией на Open Virtual Platforms. Доступна по ссылке: http://www.ovpworld.org/documentation/, дата обращения: 17.11.2017.

[9]. Страница библиотека PyCParser на GitHub. Доступна по ссылке: https://github.com/eliben/pycparser, дата обращения 20.03.2017.

[10]. Страница системы контроля версий Git. Доступна по ссылке: https://git-scm.com/about, дата обращения 09.03.2017.
[11]. Страница библиотеки для создания графического интерфейса «Tkinter». Доступна оп ссылке: https://wiki.python.org/moin/TkInter, дата обращения 2017.03.13.
[12]. В.Ю. Ефимов, К.А. Батузов, В.А. Падарян. Об особенностях детерминированного воспроизведения при минимальном наборе устройств. Труды ИСП РАН, том 27, вып. 2, 2015, стр. 65-92. DOI: 10.15514/ISPRAS-2015-27(2)-5
[13]. Руководство по работе с Dynamips. Доступно по ссылке: http://www.iteasypass.com/Dynamips.htm, дата обращения 30.06.2017.
[14]. Страница инструмента эмуляции сети GNS3. Доступна по ссылке https://gns3.com, дата обращения 30.06.2017.
[15]. Руководство пользователя для микроконтроллеров серии MPC860. Доступно по ссылке: http://www.nxp.com/docs/en/reference-manual/MPC860UM.pdf, дата обращения 30.06.2017.

Automation of device and machine development for QEMU

[1] *V.Yu. Efimov <real@ispras.ru>*
[1] *A.A. Bezzubikov <abezzubikov@ispras.ru>*
[1] *D.A. Bogomolov <bda@ispras.ru>*
[1] *O.V Goremykin <goremykin@ispras.ru>*
[1,2] *V.A. Padaryan <vartan@ispras.ru>*
[1]*Ivannikov Institute for System Programming of the RAS*
109004, Moscow, Alexander Solzhenitsyn st., 25
[2] *Lomonosov Moscow State University (MSU),*
GSP-1, Leninskie Gory, Moscow, 119991, Russia.

Abstract. Both virtual device and machine development for QEMU are difficult. To simplify the work of a developer we had analyzed both QEMU architecture and the development workflow. In this paper we suggest the new development approach which uses a declarative description for both machine and devices. The approach is implemented as an integrated software tool that returns a set of files containing a C code which could be compiled. Resulting code of machine is ready to use except for CPU configuration and CLI input. In case of a device, a developer has to implement the behavior of the device. Both device draft generation settings and machine content description are given to the tool in Python. A machine visual representation by a GUI is also implemented. A developer could use either GUI or a text editor (or both) to specify the settings. This way, the first stage of the development is automated. The tool was evaluated on Q35-based PC and Cisco 2621XM. The amount of device generation settings lines is 11-26 times smaller than the amount of the result code lines. This difference is achieved by generation of device model auxiliary code part which has a significant size because of QEMU API, while it could be generated using relatively small amount of settings. Generated code part is ¼ - ¾ of final machine code. The source code of the tool is available at https://github.com/ispras/qdt.

Keywords: software emulator; binary code; virtual machine development.

DOI: 10.15514/ISPRAS-2017-29(6)-4

For citation: Efimov V.Yu., Bezzubikov A.A., Bogomolov D.A., Goremykin O.V., Padaryan V.A. Automation of device and machine development for QEMU. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 6, 2017, pp. 77-104 (In Russian). DOI: 10.15514/ISPRAS-2017-29(6)-4

References

[1]. Dovgalyuk P.M., Makarov V.A., Padaryan V.A., Romaneev M.S., Fursova N.I. Application of software emulators for the binary code analysis. Trudy ISP RAN / Proc. ISP RAS, vol. 26, issue 1, pp. 277-296 (In Russian). DOI: 10.15514/ISPRAS-2014-26(1)-9

[2]. F. Bellard QEMU, a Fast and Portable Dynamic Translator. USENIX Annual Technical Conference, FREENIX Track. USENIX, 2007. P. 41-465 p.

[3]. J. Bennett. Howto: GDB Remote Serial Protocol. Writing a RSP Server. Application Note 4. Issue 2, Embecosm, Available at: http://www.embecosm.com/appnotes/ean4/embecosm-howto-rsp-server-ean4-issue-2.pdf, November 2008.

[4]. Python programing language site. Available at: https://www.python.org/, accessed: 06.07.2017.

[5]. AMD SimNow Simulator site. Available at: http://developer.amd.com/simnow-simulator/, accessed: 12.10.2017.

[6]. D. Aarno, J. Engblom. Software and System Development using Virtual Platforms. Full-System Simulation with Wind River Simics. Elsevier Inc. 15.09.2014. 366c.

[7]. N. Binkert, B. Beckmann, G. Black, S.K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D.R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M.D. Hill, and D.A. Wood. The gem5 simulator. SIGARCH Comput. Archit. News 39, 2, August 2011, 1-7pp.

[8]. Open Virtual Platforms documentation web page. Available at: http://www.ovpworld.org/documentation/, accessed: 17.11.2017.

[9]. PyCParser library page at GitHub. Available at: https://github.com/eliben/pycparser, retrieved: 20.03.2017.

[10]. Git SCM. Available at: https://git-scm.com/about, retrieved: 09.03.2017.

[11]. Tkinter GUI library site. Available at: https://wiki.python.org/moin/TkInter, accessed: 2017.03.13.

[12]. V. Yu. Efimov, K. A. Batuzov, V. A. Padaryan, A. I. Avetisyan. Features of the deterministic replay in the case of a minimum device set. Programming and Computer Software, April 2016, Volume 42, Issue 3, pp. 174-186. DOI: 10.1134/S0361768816030038

[13]. Dynamips Manual. Available at: http://www.iteasypass.com/Dynamips.htm, accessed: 30.06.2017.

[14]. GNS3 network simulator site. Available at: https://gns3.com, 30.06.2017.

[15]. MPC860 chip series user manual. Available at: http://www.nxp.com/docs/en/reference-manual/MPC860UM.pdf, accessed: 30.06.2017.

Декомпиляция объектных файлов *.dcuil

А.А. Михайлов <mikhailov@icc.ru>
А.Е. Хмельнов <alex@icc.ru>
Институт динамики систем и теории управления
имени В.М. Матросова СО РАН,
664033, Россия, Иркутск, ул. Лермонтова, 134

Аннотация. Работа посвящена решению задачи декомпиляции одного из разновидностей формата DCU – файлов .dcuil, создаваемых компиляторами тех версий Delphi, которые работали для платформы .NET. Разработан метод решения этой задачи, состоящий из ряда этапов: синтаксический анализ кода CIL; формирование графа потока управления; генерация промежуточного представления; структурирование графа потоков управления; анализ потоков данных с учётом семантики команд CIL; улучшение промежуточного представления с учётом особенностей работы компилятора Delphi; генерация кода.

Ключевые слова: обратная инженерия; объектный код; Delphi.

DOI: 10.15514/ISPRAS-2017-29(6)-5

Для цитирования: Михайлов А.А., Хмельнов А.Е. Декомпиляция объектных файлов DCUIL. Труды ИСП РАН, том 29, вып. 6, 2017 г., стр. 105-116. DOI: 10.15514/ISPRAS-2017-29(6)-5

1. Введение

Задача декомпиляции программного кода до сих пор не решена в полном объёме. Хотя существуют примеры декомпиляторов, корректно восстанавливающих исходный код для некоторых типов файлов (в основном это декомпиляторы кода виртуальных машин), для исполняемых файлов, содержащих машинный код, возможности всех существующих реализаций декомпиляторов очень ограничены, что затрудняет использование результатов их работы на практике. Сложность разработки таких декомпиляторов объясняется тем, что для полноценной декомпиляции исполняемых файлов требуется решить такие задачи, как: разделение кода и данных, учёт семантики машинных команд, вывод типов, распознавание системных библиотек. При этом, например, задача разделения кода и данных в общем случае является алгоритмически неразрешимой [1].

Объектные файлы содержат информацию о программном коде и данных, необходимую для сборки исполняемого файла редактором связей. Эти файлы более структурированы, чем исполняемые: код и данные в значительно

большей степени разделены, сохранена информация об именах подпрограмм и глобальных переменных (как определяемых, так и используемых), при этом объектные файлы содержат такой же машинный код, что и соответствующие исполняемые файлы. Таким образом, задача декомпиляции для объектных файлов должна решаться проще. Однако задача декомпиляции объектных файлов обычно не рассматривается, поскольку такие файлы в основном воспринимаются программистами, как некоторый кэш компилятора – вспомогательные данные, формируемые компилятором в ходе работы и не представляющие самостоятельной ценности.

Исключение составляет формат DCU [2] объектных файлов Delphi. Помимо образов памяти подпрограмм и глобальных данных, в файлах .dcu кодируется вся информация из интерфейсной части модуля, то есть они совмещают функции .obj и .h файлов, поэтому очень часто использующие Delphi разработчики распространяют свои модули и целые библиотеки модулей в формате DCU, без предоставления исходных текстов. При этом формат DCU частично изменяется с каждой новой версией продукта, поэтому программисты, зависящие от чужих модулей, должны полагаться на то, что разработчик такого модуля не прекратит свою работу и скомпилирует его для следующих версий компилятора, когда это потребуется. Время показывает, что эта надежда очень часто не оправдывается. Таким образом, декомпиляция объектных файлов DCU является актуальной для тех разработчиков, которые используют объектные файлы DCU без исходных кодов.

Несмотря на то, что в последних версиях Delphi платформа .NET не поддерживается, разработка метода декомпиляции для одной из разновидностей формата DCU открывает дорогу к разработке аналогичных методов для других разновидностей этого формата. Кроме того, декомпиляция файлов DCUIL может быть непосредственно востребована при решении задач, связанных с поддержкой унаследованного программного кода, скомпилированного для этой платформы.

2. Общая схема декомпиляции объектных файлов DCUIL

Рисунок 1. Схема декомпиляции объектного кода Delphi
Figure 1. Decompilation scheme of the Delphi object code

Декомпиляция кода в файлах DCU может выполняться отдельно для каждой подпрограммы, что существенно облегчает решение этой задачи. Для реализации декомпилятора объектных файлов DCUIL необходимо решить следующие основные задачи, (рис. 1): восстановление высокоуровневых операторов, генерация промежуточного представления, генерация кода, и выполнение оптимизаций, нацеленных на улучшение результата декомпиляции.

2.1 Загрузчик файла

Загрузчик осуществляет разбор входного файла в формате DCUIL, DCU. В качестве загрузчика использована программа DCU32INT, которая выполняет разбор файла в соответствии со спецификацией формата DCU на языке FlexT [3]. Загрузчик считывает последовательность тегов в исходном файле и ставит им в соответствие структуры данных, описывающие прочитанные утверждения. В ходе чтения теговых структур данных для каждого файла DCU формируется две таблицы: таблица адресов и таблица типов. Большинство структур данных файла DCU ссылаются на другие структуры по индексам в этих таблицах.

Описания подпрограмм содержат информацию о смещении образа памяти с кодом подпрограммы в блоке памяти модуля и размере этого образа.

За блоком памяти следует запись с таблицей перемещаемых адресов (FixUp), которая содержит информацию о том, в какие места блока памяти должны быть подставлены адреса различных процедур, переменных, описаний типов данных и других определений после их назначения редактором связей. Эта информация используется дизассемблером при разборе байт-кода для контроля правильности работы и отображения информации. Так, перемещаемые адреса могут встречаться только в операндах инструкций и не могут пересекаться с кодами команд. При наличии перемещаемого адреса в операнде информация об этом адресе используется при выводе операнда.

2.2 Процедура дизассемблирования

В программе DCU32INT реализован примитивный статический дизассемблер, который умеет:

- определять размер, занимаемый одной машинной командой;
- определять, что команда безвозвратно передаёт управление;
- обнаруживать ссылки из машинной команды (переходы на другие команды);
- отображать машинные команды.

Таких возможностей недостаточно для реализации декомпилятора, которому необходимо иметь всю информацию о семантике команды, доступную виртуальной машине исполняющей её.

Для получения семантики инструкций был использован проект Mono, реализованный на языке C#. Для этого был модифицирован декомпилятор ILSpy таким образом, чтобы на выходе он производил код близкий языку Delphi.

В результате декомпиляции части библиотеки Mono были получены две таблицы опкодов CIL:

- OneByteOpCodes – опкоды длиной один байт;
- TwoByteOpCodes – опкоды длиной два байта.

Каждое значение в таблице представляет собой объект, который содержит в себе всю необходимую информацию о семантике опкода:

- имя опкода;
- информацию о типе передачи управления;
- тип самого опкода;
- типы операндов;
- информацию о состоянии стека до выполнения команды и после;

В декомпиляторе стадия дизассемблирования сводится к сопоставлению последовательности байтов, кодирующих машинную команду некоторого выражения, описывающего её семантику. Команда CIL представляет собой закодированное по определённым правилам [4] указание для виртуальной машины на выполнение некоторой операции. Команда всегда начинается с кода команды. Код команды может занимать от одного до двух байтов, у двухбайтовых кодов первый байт всегда буден равен `0xFE` (т.е. ряд команд закодирован в дополнительной таблице, т.к. они не поместились в основной).

2.3 Генерация промежуточного представления

Промежуточное представление CILIR [5] (CIL Intermediate Representation), разработанное авторами в декомпиляторе DCUIL2PAS [6] реализовано в виде иерархии классов (рис. 2), с использованием техники подсчёта ссылок. Счётчики ссылок используются для экономии памяти и в дальнейшем для вычисления результатов выражений.

Промежуточный код для базовых блоков строится путём символьной интерпретации каждой команды CIL и сопоставления ей выражения (экземпляра класса), реализующего семантику. Начальное состояние каждого линейного участка кода характеризуется значениями параметров подпрограммы и локальных переменных, а также состоянием стека. Конечное состояние определяется в результате символьной интерпретации.

Далее применяется итерационный алгоритм анализа потоков данных для достигающих определений. Состояние локальных переменных и аргументов подпрограммы являются общим контекстом для всех базовых блоков, и используется только для хранения результатов, а не вычисления выражений.

Поэтому в качестве входного и выходного множества для передаточной функции рассматривается только состояние стека.

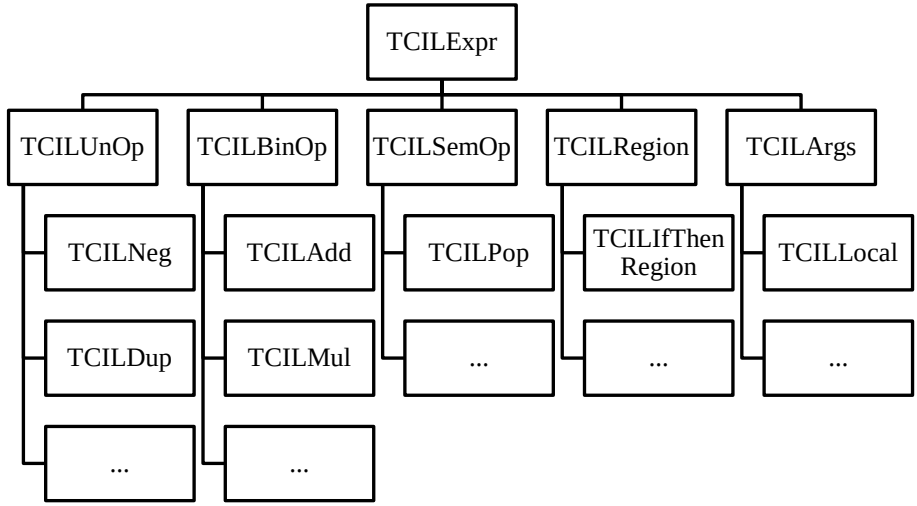


Рисунок 2. Иерархия классов промежуточного представления
Figure 2. Intermediate Representation Class Hierarchy

2.4 Упрощение сокращенных логических выражений

Для логических выражений, включающих логические связки `and` и `or`, компилятор может генерировать код для операторов условного перехода двумя разными способами:

- Полное вычисление выражений. Условие преобразуется в последовательность инструкций вычисления выражения, помещающих вычисленный результат на стек для последующего извлечения в качестве аргумента для опкода условного перехода.
- Сокращённое вычисление логических выражений (short-circuit boolean evaluation) с помощью условных выражений, основанных на следующих правилах:
    1. $A\ and\ B \Rightarrow if\ A\ then\ B\ else\ False,$
    2. $A\ or\ B \Rightarrow if\ A\ then\ True\ else\ B.$

Этот режим используется компилятором по умолчанию и позволяет не выполнять часть вычислений, если становится известно, что они уже не могут повлиять на результирующее значение выражения.

Случай полного вычисления логических выражений является наиболее простым для декомпилятора, поскольку результатом вычисления выражения,

извлеченного со стека в качестве операнда команды условного перехода, будет исходное логическое условие.

При декомпиляции логического выражения, вычисляемого сокращённым способом, на этапе улучшения промежуточного кода выполняется объединение логических условий по заранее определённому набору правил:

1. $if\ A\ then\ B\ else\ False \Rightarrow A\ and\ B,$
2. $if\ A\ then\ True\ else\ B \Rightarrow A\ or\ B.$

2.5 Генерация управляющего графа

Одной из наиболее важных задач декомпиляции является восстановление высокоуровневых операторов, таких как `if-then-else`, `if-then`, `while`, `for`, `case` и т.д.

Алгоритм для построения базовых блоков, представленный в [8], получает на вход последовательность трёхадресных команд. Далее в последовательности команд выделяются лидеры, которые разбивают её на линейные подпрограммы, которые начинаются с команды-лидера.

Для разбиения дизассемблируемой программы на базовые блоки используется информация о типе передачи управления, которая содержится в описывающей инструкцию структуре данных.

Генерация управляющего графа совершается в один проход и объединена с процессом дизассемблирования. В качестве параметра в функцию разбора передаётся указатель на процедуру, которая производит разбиение блока памяти на последовательности инструкций, которые соответствуют базовым блокам управляющего графа, и находит переходы между ними.

В процессе построения графа потоков управления каждый узел снабжается меткой со счетчиком ссылок и все инструкции условного и безусловного перехода приводятся к единому виду:

- *If x op y Then goto label* – переход по условию;
- *goto label* – безусловный переход.

Особого внимания заслуживает обработка операторов обработки исключительных ситуаций. В Delphi для обработки исключительных ситуаций используется два оператора:

- *try/finally* – применяется, когда необходимо, чтобы код в секции finally выполнился в любом случае.
- *try/except* – при возникновении исключительной ситуации исполнение основного фрагмент кода прекращается, и выполнение передается в секцию *except*.

Для каждого блока кода подпрограммы, если в нём используются операторы обработки исключительных ситуаций, в объектных и исполняемых файлах хранится специальная таблица. В ней содержится вся необходимая

информация для их обработки: смещение до `try`; размер защищаемого блока кода; адрес и размер кода обработчика исключительных ситуаций; тип оператора (`finally`, `except`). В зависимости от размера кода подпрограммы могут применяться более компактные версии кодирования записей таблицы Small или менее компактные Fat.

При генерации управляющего графа оператор `try/finally` обрабатывается достаточно просто: необходимо разбивать последовательность операторов на части, соответствующие блокам `try` и `finally`, в соответствии с информацией об адресах и размерах защищаемого блока и обработчика исключительных ситуаций. При обработке `try/except` необходимо сформировать новый базовый блок, поскольку поток управления в случае ошибки не достигает кода, следующего за оператором `except`.

2.6 Восстановление высокоуровневых операторов

На практике в большинстве работ, посвященных декомпиляции, используются два подхода к анализу потока управления отдельных процедур. Первый подход использует дерево доминаторов для поиска естественных циклов, и в дальнейшем использует их для оптимизации. Второй подход, называемый интервальным анализом, включает методы, которые позволяют анализировать структуру процедуры в целом и разбивать её на вложенные участки, называемые интервалами. Теория интервалов была предложена Алленом [9] в начале 1970-х годов и использовалась для проведения оптимизаций при более тщательном анализе потоков данных. Наиболее глубокий вариант интервального анализа, называемый структурным анализом, был предложен Цифуентес [10]. Данный метод классифицирует абсолютно все структуры потока управления в процедуре. На первом этапе метод производит выделение и структурирование циклов. Далее, в порядке, обратном обходу в глубину, на граф накладываются шаблоны, соответствующие высокоуровневым операторам, и с помощью семантически эквивалентных преобразований граф сводится к одной абстрактной вершине, которая содержит в себе всю иерархию вложенных интервалов. В теории компиляции методы анализа потока данных на основе анализа интервалов называются методами устранения.

На основе анализа дерева доминирующих вершин разработан алгоритм структурирования управляющего графа, в основе которого лежит предложенный Джонсоном с коллегами [11] в 1994 году метод структурирования управляющего графа путем представления его в виде иерархии SESE-регионов (Single Entry Single Exit).

В работе [11] отмечено, что два любых SESE-региона в управляющем графе должны быть, либо вложенными друг в друга, либо непересекающимися. Структурный анализ на основе SESE-регионов и PST (Program Structure Tree) обычно используется для эффективного построения промежуточного

представления в SSA (Static Single Assignment) форме, а также для более эффективного анализа потоков данных в процессе компиляции, и в нём не предусматривается классификация выделяемых регионов. Для решения этих задач существенным является требование, что SESE-регион образует именно пара дуг, т. е. узел схождения имеет только одну входящую дугу, а узел расхождения – только одну исходящую дугу.

В данной работе выделяются двухтерминальные регионы (TT-регион), которые соответствуют схождению потока управления в уграфе и его последующему расхождению. Требования к TT-региону являются более слабыми, чем требования к SESE-региону. При этом, каждый SESE-регион является и TT-регионом, но не наоборот.

После того, как выделены все TT-регионы, полученный граф с помощью семантически эквивалентных преобразований сводится в одну абстрактную вершину, содержащую в себе иерархию подпрограммы. Для этого применяется итеративный алгоритм наложения шаблонов, на каждой итерации которого рассматривается TT-регион, имеющий наибольший уровень вложенности.

В качестве выделяемых шаблонов в основном используются подграфы, которые соответствуют высокоуровневым конструкциям языка Delphi: `block`, `while`, `repeat`, `if-then`, `if-then-else`, `case`, `unresolved` .

Для вычисления дерева доминаторов и постдоминаторов использовался алгоритм [12]. Хотя он имеет не самую лучшую теоретическую оценку сложности из алгоритмов, представленных в работах [13][14][15], но на практике использование его оказывается предпочтительней на графах с менее 30 000 вершин из-за маленькой скрытой константы и простоты реализации.

2.7 Результаты тестирования

Разработанный декомпилятор DCUIL2PAS был протестирован на специально подготовленном наборе процедур, взятых из модификации алгоритма LZW [16], написанного на языке Delphi. Поскольку прямых аналогов декомпилятору объектных файлов dcuil нет, было принято решение провести сравнительное тестирование с инструментов ILSpy, поскольку он наиболее близок по своим характеристикам к DCUIL2PAS и распространяется по свободной лицензии, для оценки качества – мера качества декомпиляции [17]:

$$C_{decom} = \sum_{prog \in TS} \frac{\max(0, K' - K)}{KLOC(prog)},$$

где TS – тестовый набор программ; prog – исходная программа; KLOC(prog) – количество тысяч значимых строк кода программы prog; K – сумма штрафов исходной программы; K' – сумма штрафов восстановленной исходной программы.

Штрафы за артефакты трансляции и неполноту восстановления (табл. 1) были изменены в соответствии с требованиями декомпиляции объектного кода Delphi. Подсчет меры качества производился для каждой процедуры отдельно, при этом не учитывалось качество восстановления интерфейсной части модуля.

Табл. 1. Штрафы за артефакты трансляции и неполноту восстановления
Table 1. Penalties for translation artifacts and incompleteness of restoration

Конструкции программы	**Назначаемые штрафы**
Невосстановление имени переменной	1
Оператор перехода goto	3
Выход из середины цикла break	1
Оператор прерывания цикла continue	1
Невосстановленный оператор for	1

Полученные оценки качества декомпиляции представлены в табл. 2. На всех примерах мера качества разработанного декомпилятора оказалась выше, чем у ILSpy. Это связано в первую очередь с тем, что в процессе обработки объектных модулей линкером теряется часть информации об исходной программе, а также с тем, что декомпилятор ILSpy изначально разрабатывался, исходя из соображений, что исходная программа была написана не на языке Delphi.

Табл. 2. Мера качества декомпиляции
Table 2. Measure of decompilation quality

Название	**DCUIL2PAS**	**ILSpy**
BitWise	62.5	133.3
Compression	18.6	146
LZRW1KHCompressor	75	140
GetMatch	0	166.6

Помимо сравнительного анализа декомпилятором в пакетном режиме была разобрана стандартная библиотека VCL Delphi 8. Результаты, которые в большей степени демонстрируют производительность работы декомпилятора, приведены в табл. 3.

Табл. 3. Результат пакетной обработки (производительность)
Table 3. Batch processing result (performance)

Название	**Кол-во файлов**	**Размер (мб)**	**Время обработки (с)**
Delphi 8 VCL	325	39	396

Для оценки качества декомпиляции стандартной библиотеки VCL Delphi 8 в автоматическом режиме было просчитано количество процедур и функций, восстановленных в структурном виде (без использования оператора `goto`).

Тестирование показало (табл. 4), что в 98,7% случаях удается восстановить программу без операторов `goto`.

Таблица 4. Результат пакетной обработки (качество)
Table 4. Batch processing result (quality)

Название	**Кол-во процедур**	**Без goto**	C goto	%
Delphi 8 VCL	9003	8879	124	1.3

Разработанный декомпилятор объектных файлов DCUIL позволяет восстанавливать исходный код на языке Delphi, который в большинстве случаев пригоден для дальнейшей его компиляции и полностью семантически эквивалентен исходному представлению программы. Данное программное средство позволяет существенно сократить время на решение задач, связанных с поддержкой и переработкой унаследованного и стороннего программного обеспечения, исходные тексты которого не предоставлялись или были утрачены; позволяет находить закладки в готовых модулях и компонентах Delphi под .NET; искать и исправлять ошибки.

3. Заключение

Результат декомпиляции файлов .dcuil оказывается более качественным, более понятным для исследователя кода по сравнению с результатами декомпиляции исполняемых файлов .NET, поскольку в нём отображается дополнительная информация, не попадающая в исполняемые файлы, например, имена переменных. Кроме того, учёт таких особенностей компилятора, как сокращённое оценивание логических выражений, на стадии улучшения промежуточного представления позволяет получить более понятный код.

Многие этапы разработанного метода и реализованные для их работы подпрограммы не зависят от особенностей кода для платформы .NET. Основную сложность для распространения метода на другие платформы представляет описание семантики машинных инструкций реальных процессоров, система команд которых сложнее байт-кода виртуальной машины, как по количеству инструкций, так и по их эффектам.

Список литературы

[1]. Turing A. M. On computable numbers, with an application to the Entscheidungsproblem. Proceedings of the London mathematical society, vol. 2, no. 1, 1937, pp. 230–265.

[2]. Спецификация формата DCU на языке FlexT. 2017. URL: http://geos.icc.ru:8080/scripts/WWWBinV.dll/ShowR?DCU32.rfi (дата обращения: 2017-10-09).

[3]. А. Е. Хмельнов, И. В. Бычков, А. А. Михайлов. Декларативный язык FlexT — инструмент анализа и документирования бинарных форматов данных. Труды ИСП РАН, том 28, вып. 5. 2016 г., pp. 239–268. DOI: 10.15514/ISPRAS-2016-28(5)-15

[4]. Necula G. C., McPeak S., Rahul S. P. et al. CIL: Intermediate language and tools for analysis and transformation of C programs. International Conference on Compiler Construction. Springer, 2002, pp. 213–228.
[5]. Михайлов А. А. Промежуточное представление подпрограмм в задаче декомпиляции объектных файлов dcuil. Вестник Бурятского государственного университета, №. 9-3, 2014 г., стр. 32-38.
[6]. Михайлов А. А. Анализ графа потоков управления в задаче декомпиляции подпрограмм объектных файлов dcuil. Вестник Новосибирского государственного университета. Серия: Информационные технологии, том 12, №. 2, 2014 г., стр. 74-79.
[7]. Allen F. E., Cocke J. A program data flow analysis procedure. Communications of the ACM, vol. 19, №. 3, 1976, p. 137.
[8]. Aho A. V. Compilers: Principles, Techniques and Tools (for Anna University), 2/e. Pearson Education India, 2003.
[9]. Allen F. E. Control flow analysis. ACM Sigplan Notices, ACM, vol. 5, № 7, 1970, pp. 1-19.
[10]. Cifuentes C. Structuring decompiled graphs. Compiler Construction. Springer Berlin/Heidelberg, 1996, pp. 91-105.
[11]. Johnson R., Pearson D., Pingali K. The program structure tree: Computing control regions in linear time. ACM SigPlan Notices, vol. 29, №. 6, 1994, pp. 171-185.
[12]. Cooper K. D., Harvey T. J., Kennedy K. A simple, fast dominance algorithm. Software Practice & Experience, vol. 4, №. 1-10, 2001, pp. 1–8.
[13]. Ахо А, Дж Ульман. Теория синтаксического анализа, перевода и компиляции. Том 1. Компиляция. М., Мир, 1978.
[14]. Lengauer T., Tarjan R. E. A fast algorithm for finding dominators in a flowgraph. ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 1, № 1, 1979, pp. 121–141.
[15]. Кнут Д. Э. Искусство программирования. Т. 3. Сортировка и поиск. Издательский дом Вильямс, 2000.
[16]. Williams R. N. An extremely fast Ziv-Lempel data compression algorithm. Data Compression Conference, 1991. DCC'91, IEEE, 1991, pp. 362–371.
[17]. Трошина Е. Н. Исследование и разработка методов декомпиляции программ: Кандидатская диссертация. Московский государственный университет имени М. В. Ломоносова. 2009.

Delphi object files decompiler

A.A. Mikhailov <mikhailov@icc.ru>
A.E. Hmelnov <petrov@icc.ru>
Matrosov Institute for System Dynamics and Control Theory of Siberian Branch of Russian Academy of Sciences,
Lermontov str., 134, Post Box 292 664033, Irkutsk, Russia

Abstract. The work is devoted to solving the problem of decompiling one of the types of DCU - .dcuil format files created by the compilers of those versions of Delphi that worked for the .NET plat-form. A method for solving this problem is developed, consisting of a number of steps: syntactic analysis of the CIL code; control flow graph generation; intermediate representation generation; structuring control flow graph; dataflow analysis; intermediate representation optimization; code generation.

Keywords: reverse engineering; object code; Delphi.

DOI: 10.15514/ISPRAS-2017-29(6)-5

For citation: Mikhailov A.A., Hmelnov A.E. Delphi object files decompiler. *Trudy ISP RAN/Proc. ISP RAS*, vol. 29, issue 6, 2017, pp. 105-116 (in Russian). DOI: 10.15514/ISPRAS-2017-29(6)-5

References

[1]. Turing A. M. On computable numbers, with an application to the Entscheidungsproblem. Proceedings of the London mathematical society, vol. 2, no. 1, 1937, pp. 230–265.

[2]. DCU format specification in FlexT. 2017. URL: http://geos.icc.ru:8080/scripts/WWWBinV.dll/ShowR?DCU32.rfi. accessed: 10.09.2017

[3]. A.Y. Hmelnov, I.V. Bychkov, A.A. Mikhailov. A declarative language FlexT for analysis and documenting of binary data formats. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 5, 2016, pp. 239-268. (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-15

[4]. Necula G. C., McPeak S., Rahul S. P. et al. CIL: Intermediate language and tools for analysis and transformation of C programs. International Conference on Compiler Construction. Springer, 2002, pp. 213–228.

[5]. Mikhailov A. A. Intermediate representation for dcuil files decompilation. Vestnik Buryatskogo gosudarstvennogo universiteta [Bulletin of the Buryat State University], No. 9-3, 2014, pp 32-38 (in Russian).

[6]. Mikhailov A. A. Control flow analysis for dcuil files decompilation. Vestnik Novosibirskogo gosudarstvennogo universiteta. Seriya: Informacionnye tekhnologii [Novosibirsk State University Journal of Information Technologies], vol. 12, no 2. 2014, pp 74-79 (in Russian).

[7]. Allen F. E., Cocke J. A program data flow analysis procedure. Communications of the ACM, vol. 19, №. 3, 1976, p. 137.

[8]. Aho A. V. Compilers: Principles, Techniques and Tools (for Anna University), 2/e. Pearson Education India, 2003.

[9]. Allen F. E. Control flow analysis. ACM Sigplan Notices, ACM, vol. 5, № 7, 1970, pp. 1-19.

[10]. Cifuentes C. Structuring decompiled graphs. Compiler Construction. Springer Berlin/Heidelberg, 1996, pp. 91-105.

[11]. Johnson R., Pearson D., Pingali K. The program structure tree: Computing control regions in linear time. ACM SigPlan Notices, vol. 29, №. 6, 1994, pp. 171-185.

[12]. Cooper K. D., Harvey T. J., Kennedy K. A simple, fast dominance algorithm. Software Practice & Experience, vol. 4, №. 1-10, 2001, pp. 1–8.

[13]. Alfred V. Aho, Jeffrey D. Ullman. The theory of parsing, translation, and compiling. 1978.

[14]. Lengauer T., Tarjan R. E. A fast algorithm for finding dominators in a flowgraph. ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 1, № 1, 1979, pp. 121–141.

[15]. Donald E. Knuth. The Art of Computer Programming: Sorting and Searching. Addison Wesley Series in Computer Science and Information Processing, 2000. Vol. 3.

[16]. Williams R. N. An extremely fast Ziv-Lempel data compression algorithm. Data Compression Conference, 1991. DCC'91, IEEE, 1991, pp. 362–371.

[17]. Troshina E. N. Decompilation methods. Phd dissertation. Lomonosov Moscow State University. 2009 (in Russian).

Поиск недостающих вызовов библиотечных функций с использованием машинного обучения

И.А.. Якимов <ivan.yakimov.research@yandex.ru>
А.С. Кузнецов <ASKuznetsov@sfu-kras.ru>
Институт космических и информационных технологий,
Сибирский федеральный университет,
660074, Россия, г. Красноярск, ул. Академика Киренского, д. 26

Аннотация. Разработка программного обеспечения является сложным и подверженным ошибкам процессом. В целях снижения сложности разработки ПО создаются сторонние библиотеки. Примеры исходных кодов для популярных библиотек доступны в литературе и интернет-ресурсах. В данной работе представлена гипотеза о том, что большинство подобных примеров содержат повторяющиеся шаблоны. Более того, данные шаблоны могут быть использованы для построения моделей, способных предсказать наличие (либо отсутствие) недостающих вызовов определенных библиотечных функций с использование машинного обучения. В целях проверки данной гипотезы была реализована система, реализующая описанный функционал. Экспериментальные исследования, проведенные на примерах для библиотеки OpenGL, говорят в поддержку выдвинутой гипотезы. Точность результатов достигает 80%, при условии рассмотрения уже первых 4-х ответов, предлагаемых системой. Можно сделать вывод о том, что данная система при дальнейшем развитии может найти индустриальное применение.

Ключевые слова: OpenGL; качество программного обеспечения; рекомендательные системы; машинное обучение; нейронные сети;

DOI: 10.15514/ISPRAS-2017-29(6)-6

Для цитирования: Якимов И.А., Кузнецов А.С. Поиск недостающих вызовов библиотечных функций с использованием машинного обучения. Труды ИСП РАН, том 29, вып. 6, 2017 г., стр. 117-134. DOI: 10.15514/ISPRAS-2017-29(6)-6

1. Введение

Разработка программного обеспечения (ПО) является не только дорогостоящим, но и сложным процессом. Для упрощения разработки ПО создаются библиотеки функций, скрывающие сложность за программным интерфейсом (*application programming interface* — API). При использовании сторонних библиотек программисты зачастую применяют некоторые готовые

решения — *шаблоны*, полученные из ресурсов интернета и литературы. При этом начинающие программисты имеют тенденцию пропускать различные части данных шаблонов при реализации собственных приложений.

Ключевая идея. В предложенной работе исследуется проблема восстановления пропущенных частей шаблонов, используемых программистами при применении сторонних библиотек. В основу данного исследования положена гипотеза о том, что большая часть корректно работающих приложений, использующих сторонние библиотеки, основана на шаблонах, свойства которых поддаются анализу средствами машинного обучения. Данная гипотеза основана на результатах работ, ранее проведенных в смежных областях, таких как автоматическая генерация патчей [1] и автодополнение кода [2].

1.1 Обзор литературы

Статический анализ и динамические символьные вычисления. Во время динамических символьных вычислений [3,4,5] систематически исследуются различные пути выполнения программы. При этом, либо производится многократный перезапуск целевой программы, либо происходит параллельное выполнения различных ее путей посредством интерпретатора с копированием состояний. Статический анализ [6] в свою очередь производится без запуска программы. Результатом статического анализа является абстрактное представление, содержащее информацию о множестве различных путей выполнения программы.

Информация, полученная средствами динамического и статического анализа, используется для поиска программных ошибок. Данные инструменты нацелены на поиск низкоуровневых ошибок, таких как переполнение буфера, разыменовывание нулевого указателя и т.д. При условии моделирования работы внешних функций также может проводиться проверка корректности алгоритма использования библиотек (например, память, выделенная malloc, должна быть высвобождена с помощью free), предупреждение использования небезопасных функций (например, strlen вместо strnlen). Статический и динамический анализ может также применяться для проверки моделей и решения задач формальной верификации.

Таким образом, средства статического и динамического анализа предоставляют мощный инструментарий для проверки корректности программ. Однако в данной работе акцент смещен в сторону обнаружения недостающих вызовов библиотечных функций. Дополнительным требованием является то, чтобы разработчик не задавал правила обнаружения недостающих вызовов вручную. Система должна самостоятельно извлечь необходимые знания из подготовленных для нее примеров.

Автоматическая генерация патчей. Задача автоматической генерации патчей близка к задаче, решаемой в данном исследовании. Генерация патчей предполагает автоматическое исправление ошибок в программах. Семантика

верных программ, используемая при оценке качества патчей строится средствами машинного обучения [1]. Однако для проверки патчей все же предполагается наличие тестов, позволяющих определить их корректность. В данной работе для оценки верности предлагаемых системой рекомендаций не предполагается наличие тестов, поэтому системы автоматической генерации патчей не будут детальнее рассматриваться.

Рекомендательные системы. Наиболее близкими к предлагаемой в данной работе системе являются рекомендательные системы, построенные с использованием машинного обучения [7], в частности системы для улучшения механизма автодополнения кода.

В случае контекстно-зависимых вероятностных моделей автодополнения происходит анализ всех случаев применения объекта и затем моделируется распределение вероятности для следующего вызова [8, 9]. Развитием данной концепции является применение n-граммной языковой моделей исходного кода [10] для предоставления контекста автодополнения. Возможно также использование n-граммной модели с поддержкой кэша использованных токенов [11], а также дополнение языковой модели исходного кода статистической семантикой [12]. Помимо языковых моделей завершения кода на основе токенов также применяются вероятностные модели на основе абстрактных синтаксических деревьев [13,14].

Альтернативным подходом является автодополнение в указанных пользователем местах, в которых предположительно расположены недостающие фрагменты кода. В данном случае производится не только подбор нужного токена, но синтез целой строки кода. Пользователь помечает (пустые) строки, в которых предположительно должен располагаться фрагмент кода. В указанных строках система синтезируются наиболее вероятный недостающий фрагмент [15]. При этом используется языковая модель исходного кода, построенная на основе рекуррентной нейронной сети.

Таким образом, в существующих на данный момент рекомендательных системах пользователь должен в явном, либо неявном виде указывать конкретные участки кода, в которых предполагается применение автодополнения. Однако в данной работе фокус смещен на способность системы самостоятельно определять факт наличия (либо отсутствия) пропущенных вызовов библиотечных функций без активного участия пользователя.

Обучение с учителем. В первом приближении, машинное обучение с учителем предполагает построение моделей, тренеруемых устанавливать соответствие между некоторым входом и выходом на основе тренеровочной выборки примеров [16]. Одной из разновидностей моделей, применяемых в задачах машинного обучения является искусственная нейронная сеть (artificial neural network — ANN). В данной работе применяется рекуррентная нейронная сеть, построенная на ядре LSTM (long-short term memory) [17]. Архитектура LSTM была выбрана из-за своей широкой применимости в

задачах статистического моделирования языка. Рекуррентные нейронные сети способны аппроксимировать работу алгоритмов [18], что важно при построении системы, выявляющей закономерности в исходном коде.

1.2 Постановка задачи

Пусть дана некоторая пользовательская процедура, содержащая вызовы функций сторонней библиотеки. С помощью заранее заданного алгоритма из данной процедуры извлекается последовательность вызовов функций w = f1,f2,...,fn, где fi — имя функции. Предполагается, что в данной последовательности либо пропущена одна функция, необходимая для завершения определенного шаблона, либо пропущенные функции отсутствуют. Необходимо построить модель, обладающую следующими свойствами. Если в последовательности недостает вызова функции, завершающего некоторый шаблон, система должна восстановить имя пропущенной функции. В том же случае, когда пропуски отсутствуют, и последовательность представляет собой завершенный шаблон использования сторонней библиотеки, система должна установить данный факт.

2. Методы

2.1 Обзор системы

В качестве целевой была выбрана библиотека OpenGL [19]. Данный выбор обусловлен тем, что для OpenGL существует большое количество примеров с открытым исходным кодом, которые являются сравнительно однотипными, что делает их хорошим материалом для решения задач машинного обучения.

В законченном виде система предоставляет достаточно простой для использования интерфейс. Пользователь подает ей на вход исходный код программы, получая на выходе отчет. Отчет содержит в себе записи о каждой из пользовательских процедур. Каждая запись содержит несколько предположений системы о том, какая функция была пропущена. Особым видом предположения является отсутствие пропущенных функций.

Пример. Рассмотрим несколько примеров, полученных в ходе экспериментальных исследований работы системы. В первом примере приведен код процедуры по отображению сцены на экране. В данной процедуре был пропущен вызов функции glPushMatrix. Пропущенный вызов помещен в комментарий в целях улучшения наглядности данного примера. Без предварительных знаний о свойствах библиотечных функций и закономерностях их вызовов, система выучила шаблон, согласно которому за вызовом glPushMatrix должен следовать вызов glPopMatrix. Система предоставляет пользователю распределение вероятностей, содержащее имена 10 наиболее вероятных недостающих функций. Следует отметить, что в задачу системы не входит определение местоположения пропущенной функции.

Среди предложенных ответов присутствует и верный — по оценке системы, с вероятностью 98,84% была пропущена функция glPushMatrix. Специальный токен __none__ в данном распределении символизирует отсутствие пропусков. В примере с пропуском функций, реализующих логику push-pop, можно предложить простую реализацию алгоритма обнаружения пропущенных вызовов. Достаточно использовать счетчик push- и pop-вызовов и затем сравнить результат. Однако во многих случаях система должна быть способна распознавать сложные алгоритмы использования библиотечных функций, и обойтись алгоритмом со счетчиком вызовов крайне затруднительно. Во втором примере приведена процедура Reshape, в которой отсутствуют пропуски вызовов функций, то есть подаваемый на вход системе алгоритм реализован верно. Данный факт подтверждается в отчете, предлагаемом системой, согласно которому с вероятностью 55.27% пропуски отсутствуют.

Табл. 1. Пример работы системы
Table 1. Example

Исходный код	Вывод системы
void display(void) { glClear (GL_COLOR_BUFFER_BIT); glColor3f (1.0, 1.0, 1.0); **// glPushMatrix(); - недостающая функция** glutWireSphere(1.0, 20, 16); glRotatef ((GLfloat) year, 0.0, 1.0, 0.0); glTranslatef (2.0, 0.0, 0.0); glRotatef ((GLfloat) day, 0.0, 1.0, 0.0); glutWireSphere(0.2, 10, 8); glPopMatrix(); glutSwapBuffers(); }	**glPushMatrix 98,84%** glRotatef 0,22% __none__ 0,10% glFlush 0,09% gluPerspective 0,09% glFrustum 0,08% glScalef 0,07% glEnable 0,07% gluOrtho2D 0,06% glNewList 0,06%
// пропущенных вызовов нет static void Reshape(int width, int height){ glViewport(0, 0, width, height); glMatrixMode(GL_PROJECTION); glLoadIdentity(); glFrustum(-2.0, 2.0, -2.0, 2.0, 0.8, 10.0); gluLookAt(7.0, 4.5, 4.0, 4.5, 4.5, 2.5, 6.0, -3.0, 2.0); glMatrixMode(GL_MODELVIEW); }	**__none__ 55,27%** glLoadIdentity 18,47% glOrtho 6,72% gluOrtho2D 4,67% glEnable 3,04% glTranslatef 1,64% gluPerspective 1,25% glScalef 1,22% glMatrixMode 0,94% glDisable 0,76%

2.2 Архитектура

Рассмотрим внутреннее устройство обученной системы, сам процесс обучения будет показан в следующем разделе. На вход системы подается файл с исходным кодом на языке Си. Данный файл передается трассировщику. На

выходе трассировщика получается n трасс вызовов библиотечных функций «трасса 1», «трасса 2», …, «трасса n», где n — число пользовательских процедур, содержащих вызовы библиотечных функций. Каждая трасса представляет собой последовательность имен библиотечных функций, расположенных в порядке их вызова внутри одной пользовательской процедуры. Полученные трассы далее поступают на вход препроцессору, который преобразует их в матрицу, содержащую признаки каждой отдельной трассы. Далее полученная матрица поступает на вход классификатору. Структурная схема изображена на рис. 1.

Рис. 1. Архитектура системы
Fig. 1. Architecture of the system

Остановимся подробнее на задаче, решаемой классификатором. В конечном итоге, классификатор должен определять пропущенные в трассе функции, либо же отсутствие пропуска. Иначе говоря, он должен поставить в соответствие каждой трассе метку, которая либо хранит имя пропущенной функции, либо информацию о том, что она отсутствует. Следует отметить, что в случае обучения классификатора препроцессор не только преобразует трассы к требуемому формату, но и генерирует на их основе примеры.

2.3 Анализ исходных данных

Для тренировки классификатора средствами машинного обучения необходим соответствующий набор данных. Для получения набора данных были использован примеры исходных кодов программ из следующих источников [20], далее данный набор будет именоваться *бенчмарком*. Примеры содержат исходный код учебных OpenGL программ. Далее понятия OpenGL-функция и функция будут применяться как взаимозаменяемые там, где это не противоречит контексту. В общей сложности было использовано 214 файлов с исходным кодом программ. Отметим, что данные примеры изначально не предназначены для решения задач машинного обучения и не являются частью готового набора данных.

В данной работе применяется обучение с учителем, и набор данных разбивается на тренировочную и тестовую выборку. Таким образом, обучение классификатора производится на тренировочной выборке, каждый пример из которой содержит корректный ответ в виде метки. Проверка, в свою очередь, производится на тестовой выборке. Таким образом, набор данных должен быть размечен, и состоять из матрицы признаков и вектора меток.

Объектами, подлежащими классификации, являются пользовательские процедуры. Единственным признаком, характеризующим каждую процедуру, является полученная из нее трасса. Соответственно, метка, которая классификатором ставится в соответствие каждой процедуре, содержит либо имя пропущенной функции, либо «__none__» — запись о том, что пропущенные функции отсутствуют. Таким образом, признаки являются списками имен функций, а метки — либо именами функций, либо __none__.

В данной работе используется следующий принцип получения примеров: пусть дана трасса w = f1,f2,...,fn, содержащая последовательность из n вызовов функций. Для данной трассы может быть получен n+1 пример. Первые n примеров содержат трассы, в которых последовательно пропущены вызовы функций. Последний пример содержит исходную версию трассы, в которой пропуски отсутствуют. Набор данных формируется из большого количества примеров, получаемых из трасс. Из принципа построения набора данных следует, что для каждой функции число порождаемых примеров равно числу вызовов данной функции внутри бенчмарка. В общей сложности, в бенчмарке производятся вызовы 329 различных функций, некоторые из них более популярны, некоторые — менее. При этом, если функция glEnable была вызвана ~700 раз, то менее популярные функции, такие как glBitmap вызываются менее 30 раз. Таким образом, число вызовов распределено неравномерно между различными функциями. Однако в целях улучшения обучаемости классификатора желательно, чтобы число примеров для каждой из функций было примерно одинаковым. Таким образом, необходимо произвести анализ распределения частоты вызовов для отдельных функций. Данный анализ был проведен, и его результаты показали, что распределение вызовов обладает свойствами, схожими с принципом Парето. В частности, первые 20% вызванных функций обеспечивают 82% всех вызовов функций.

В табл. 2. приведено распределение частоты вызовов функций. В первом столбце указана взятая рассматриваемая доля наиболее популярных функций. Во втором столбце указана общая доля вызовов данных функций среди всех вызовов функций в бенчмарке.

Табл. 2. Распределение частоты вызовов функций
Table. 2. Function calls frequency distribution

Процент самых популярных функций	Процент вхождений в примеры
5%	45%
10%	68%
20%	83%
30%	90%

40%	94%
50%	96%

Можно судить о наличии проблемы неравномерности распределения количества примеров использования различных функций. В целях ее решения наименее популярные функции были сгруппированы в наборы. Для этого было задано отображение, переводящее имя каждой отдельной функции в имя соответствующего ей предопределенного набора. Разбиение функций в наборы было произведено эвристическим способом. При этом мы опирались на назначение функции (например, работа с текстурами) и распределение вызовов функций. Результаты классификации не учитывались при разбиении. Далее набор функций будет называться «метафункцией», в целях удобства.

Используя описанное выше отображение, исходный набор данных может быть поэлементно отображен в новый набор. В новом наборе все функции, входящие состав трасс и меток, отображаются на соответствующие им метафункции. Классификатор, обученный на наборе данных, составленном из метафункций, также для удобства назовем «метаклассификатором». В свою очередь классификатор, который обучен на наборе, построенном из необработанных функциях, назовем «наивным классификатором».

Каждая метафункция имеет определенную мощность — число функций, которые в нее отображаются. Мощность может быть равна нулю, единице, либо некоторому целому n > 1. *Пустой* назовем метафункцию, в которую не отображается ни одна из функций; данная метафункция используется как метка __none__. *Именные* метафункции соответствуют единственной функции. В *обобщенную* метафункцию отображается несколько функций. Еще одной важной характеристикой является вес метафункции – число примеров, в которые она входит в качестве метки.

Разберем введенное для удобства понятие метафункции на примере. Пустая метафункция __none__ не содержит ни одной функции, ее мощность равна нулю, а вес равен числу трасс, в которых не пропущена ни одна функция. Выделенная метафункция *glEnable* содержит единственную функцию — собственно glEnable, ее мощность равна единице, и вес равен числу вызовов glEnable в исходном коде программы. Обобщенная метафункция *draw* включает множество функций по отрисовке, выбранных эвристическим путем, ее мощность больше 1 т. к. она включает в себя несколько функций, а вес равен суммарному количеству вызовов входящих в нее функций.

В общей сложности, эвристическим путем было выделено 50 непустых метафункций. Среди данных метафункций 35 являются *именными*, и 15 *обобщенными*, соответственно.

2.4 Создание набора данных

Набор данных для обучения классификатора получается из бенчмарка путем ряда преобразований. Первым этапом является получение трасс вызовов функций внутри пользовательских процедур. При этом условные операторы и циклы не учитываются. Для получения трасс используется интерпретатор IR-кода [21]. Интерпретатор работает со списком пользовательских функций, объявленных внутри модуля, генерируемого clang [22] из файла с исходным кодом. Для каждой функции производится обход входящих в нее базовых блоков. Порядок следования базовых блоков определяется clang-ом на этапе трансляции в IR-код. Алгоритм получения трасс представлен на листинге 1.

Алгоритм: построение трасс вызовов OpenGL-функций

Вход: исходный код

Выход: набор трасс пользовательских процедур

для каждой пользовательской процедуры *proc*:

  trace = новая пустая трасса

  для каждого базового блока *block* процедуры function:

    для каждого вызова внешней функции *func* внутри блока *block*:

    если *func* — OpenGL-функция, добавить ее имя в трассу

Листинг 1. Построения трасс вызовов функций
Algorithm 1. Function calls tracing

Суммарная длинна полученных трасс составляет 15182 вызовов, что теоретически позволяет получить 15 тысяч примеров для каждой функции.

Отсечение трасс. Трассы, полученные описанным выше методом, отличаются по длине, то есть имеют различное количество вызовов. Разброс значения длины лежит в диапазоне от 1 до 338 вызовов. Обучение классификатора на основе рекуррентной нейронной сети на последовательностях различной длины вызывает затруднения. По данной причине из полученных трасс были отобраны трассы длиной от 5 до 25 вызовов, что составляет ~50% всех трасс. В результате, количество примеров сократилось до 7531.

Отображение функций в метафункции. Для обучения классификатора используется два набора данных. В первом наборе использованы оригинальные имена функции. Таким образом, поскольку число меток в наборе данных равно числу различных функций плюс дополнительная пустая функция __none, то мы имеем 329 + 1 метку. Во втором случае имена функций отображены на соответствующие им метафункции. В результате получается 50 + 1 метка, соответственно.

Группировка трасс. Данный шаг предшествует шагу получения примеров из каждой трассы. Трассы группируются так, чтобы получить n наборов трасс приблизительно равного размера, где n = 10. Далее n-1 набор составит тренировочную выборку и 1 набор — тестовую выборку, соответственно.

Таким образом, тестовая выборка составляет ~10% от объема полного набора данных. Обязательным условием получения тестовой выборки является то, чтобы в нее входили случайные примеры (примеры будут получены из трасс на последнем шаге). Однако полностью случайный выбор примеров невозможен. Примеры, полученные из одного файла с исходным кодом, не должны попадать в разные наборы — это нарушило бы достоверность проверки работы классификатора на тестовой выборке. По этой причине, при формировании наборов трасс случайным образом компонуются только трассы, полученные из одного файла с исходным кодом. Случайность тестовой выборки обеспечивается за счет случайного выбора наборов трасс.

Получение примеров и заполнение. Следующей фазой является получение примеров из каждой трассы с помощью описанного ниже алгоритма. На вход ему подается трасса из n символов, на выходе массив из n+1 примера. В основном цикле (строки 5-9) из трассы длиной производится n примеров. Далее добавляются исходная трасса и соответствующий ей пропуск (строки 10-11).

1. **Алгоритм**: получение примеров из трассы
2. **Вход**: трасса *trace* длинной n вызовов
3. **Выход**: n+1 пример
4. examples = []
5. для каждого *k* в диапазоне от 0 до длина(*trace*)-1:
6. *current_trace* = клонировать(*trace*)
7. удалить *current_trace*[*k*]
8. y = *trace*[*k*]
9. добавить (current_trace, *y*) к *examples*
10. xs = *trace*
11. y = '_' // пропуск означает отсутствие пропущенных функций
12. добавить (*xs, y*) к *examples*

Листинг 2. Получение примеров из трассы
Algorithm 2. Example extraction

После того как набор данных получен, все имена функций заменяются целочисленными идентификаторами — id, где id > 0. Следует отметить, что вследствие различий в длине трасс, входящих в состав примеров, их необходимо выравнивать по длине. Это достигается за счет добавления слева нужного числа нулей.

2.5 Архитектура классификатора

Готовый набор данных используется для обучения и тестирования классификатора. Нами был использован классификатор, построенный на базе фреймворка keras [23]. Данный классификатор состоит из нескольких слоев.

Структурная схема классификатора изображена на рис. 2. При обсуждении архитектуры классификатора понятия функции и метафункции считаются взаимозаменяемыми.

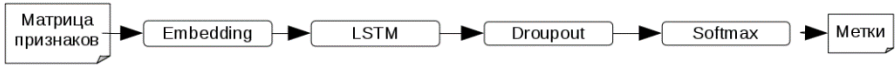


Рис. 2. Архитектура классификатора
Fig. 2. Classifier

Рассмотрим каждый из слоев более подробно. **Embedding**, или встраивание — преобразует поступающие в него трассы в векторное представление. Векторизация позволяет улучшить обучаемость нейронной сети при работе с последовательностями. С помощью векторизации каждый id преобразуется в вектор длиной в 32 элемента. **LSTM** — слой, реализующий рекурсивную нейронную сеть. Данный слой построен на базе модуля долгой краткосрочной памяти (Long short-term memory, LSTM). Отметим, в данной работе в качестве функции активации использована тангенциальная функция. **Dropout —** слой прореживания, необходимый для регуляризации нейросети с целью снижения эффектов переобучения. **Softmax** — функция активации, вычисляющая мягкий максимум. Данная функция преобразует выходы от предыдущего слоя в распределение вероятностей, описывающего вероятность принадлежности трассы к каждому из классов. Далее полученное распределение используется для определения класса, к которому принадлежит трасса.

3. Результаты

3.1 Описание эксперимента

Обучение классификатора было произведено на выборке, состоящей из 6623 примеров. Размер тестовой выборки, в свою очередь, составил 833 примеров. Число циклов (эпох) обучения нейронной сети равняется 100 циклам. При анализе выхода классификаторов была использована оценка точности top-k. На рис. 3 показан график, отображающий результаты проведенного эксперимента. По оси абсцисс отложено число k — количество взятых за рассмотрение ответов. Для каждого классификатора по оси ординат отложена линия, обозначающая график некоторой зависящей от k функции, соответствующей доле верных ответов при рассмотрении первых k ответов.

Линия S(k) на данном графике соответствует выходу классификатора, обученного на наборе данных из необработанных функций. Линии M(k) и M*(k) соответствуют выходу метаклассификатора, то есть классификатору, обученному на наборе данных из метафункций. При этом линия M*(k) соответствует доле корректно классифицированных *именных* метафункций.

3.2 Обсуждение

Анализ результатов позволяет выявить определенные закономерности. Так при k = 1 (то есть когда рассматривается только первое предположение о пропущенной функции), оба классификатора приблизительно в половине случаев дают верные ответы. Однако при значениях k > 1 проявляется выгода от использования метаклассификатора, то есть S(k) < M(k) для k = 2..10. В случае метаклассификатора доля верных ответов достигает 80% уже при k = 4; в свою очередь, в случае же простого классификатора — при k = 7. Таким образом, пользователь может с большей уверенностью полагаться на ответы системы в режиме метаклассификатора, так как чаще получает верную информацию о наличии (либо отсутствии) пропусков.

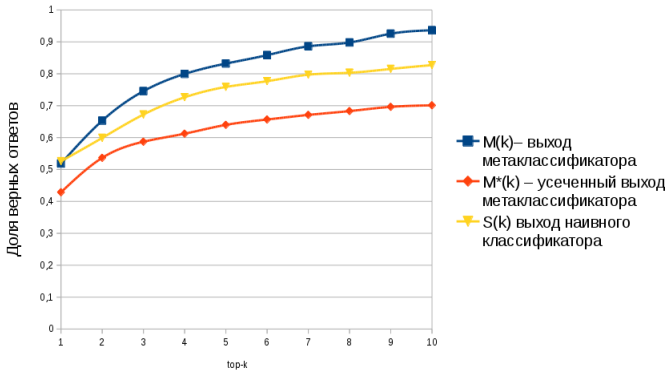


Рис. 3. Результаты
Fig. 3. Results

Однако, график, соответствующий доле верно классифицированных именных метафункций, лежит ниже графика, полученного для выхода наивного классификатора, то есть для всех k=1..10 M*(k)<S(k). Это вызвано тем, что метаклассификатор в качестве ответа приводит не только именные, но и обобщенные метафункции. При этом зачастую имя пропущенной функции не может быть точно определено, как в случае обобщенных метафункций. Таким образом, метаклассификатор чаще выдает верные ответы, однако наивный классификатор чаще выдает имена конкретных функций (строго говоря, он всегда выдает только имена конкретных функций). Следовательно, отображение редко используемых функций на наборы (метафункции) увеличивает число верных ответов, но при этом снижает среди них долю ответов с определенным именем функции. Это является закономерным, так как в данном случае многие функций объединяются в метафункции. Например, метаклассификатор может корректно определить факт отсутствия функции по работе с текстурами, но без уточнения того, какой именно

функции из набора не хватает — пользователю будет предоставлен список возможных вариантов.

3.3 Достоверность

Выбранная в качестве целевой, библиотека OpenGL является удобным объектом для данного исследования. Во-первых, вызовы входящих в нее функций образуют шаблоны, и, во-вторых, для данной библиотеки доступен широкий набор примеров в открытых источниках. Однако не все библиотеки обладают тем свойством, что их вызовы образуют шаблоны; примером может служить часть стандартной библиотеки языка Си по работе со строками — cstring. Также не для всех библиотек доступен столь же широкий набор примеров, как для OpenGL. Однако, на наш взгляд, предлагаемый метод может быть распространен на достаточно большой набор библиотек, для чего требуются дальнейшие экспериментальные и теоретические исследования.

Необходимо отметить ряд факторов, которые могли оказать влияние на достоверность результатов. Во-первых, в набор данных были включены трассы длинной от 5-25 функций. Однако на наш взгляд в большинстве случаев этого достаточно. Так, например, если функция содержит более 25 вызовов, то пользователь может произвести ее декомпозицию, сократив число вызовов; функция с 4 и менее вызовами маловероятно будет содержать какой-либо шаблон. Так или иначе, пользователю доступна информация об ограничении количества вызовов. Также следует отметить, что проверка классификатора была произведена на единственной тестовой выборке. Более надежным методом является перекрестная проверка, однако данный метод является слишком ресурсозатратным. Так как наборы трасс для тестовой выборки были отобраны случайным образом, можно с уверенностью утверждать, что результаты достаточно надежны и воспроизводимы. Далее, на данной стадии исследований в систему не заложен функционал по определению нескольких пропусков вызовов библиотечных функций. Однако данный недостаток может быть устранен при инкрементальном сценарии использовании системы, параллельно с разработкой программы. Например, на законченном фрагменте программного кода пользователь может его проверять на наличие пропуска и лишь затем приступать к новому фрагменту.

4. Заключение

В данной работе была разработана система, позволяющая определять наличие (либо отсутствие) пропусков вызовов библиотечных функций. Данная системы была апробирована на наборе данных, полученном из бенчмарка, включающего 239 файлов с исходным кодом примеров для OpenGL. Полученная система демонстрирует достаточно высокую точность работы. Так, в случае группировки редко используемых библиотечных функций в наборы, доля верных ответов равная 80% достигается уже при рассмотрении первых четырех ответов (top-4) системы. В базовом случае, аналогичная

точность, равная 80%, достигается для top-7 ответов. Несмотря на определенные ограничения, уже на данном этапе полученная система может быть использована в качестве ассистента при разработке программ с использованием библиотеки OpenGL, например, при обучении студентов. В настоящий момент проводится работа по увеличению набора данных, используемого при обучении классификатора, улучшению архитектуры положенной в его основу нейронной сети, а также добавлению возможности по определению наличия большего числа пропусков вызовов библиотечных функций.

Список литературы

[1]. Long F., Rinard M. Automatic patch generation by learning correct code. Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2016, pp 298-312. DOI: 10.1145/2837614.2837617

[2]. Bruch M., Bodden E., Monperrus M., Mezini M. 2010. IDE 2.0: collective intelligence in software development. Proceedings of the FSE/SDP workshop on Future of software engineering research, 2010, pp. 53-58. DOI: 10.1145/1882362.1882374

[3]. Cadar C., Godefroid P., Khurshid S., Păsăreanu C.S., Sen K., Tillmann N., Visser W. Symbolic execution for software testing in practice: preliminary assessment. Proceedings of the 33rd International Conference on Software Engineering, 2011, pp 1066-1071. DOI: https://doi.org/10.1145/1985793.1985995

[4]. Anand S., Burke E.K., Chen T.Y., Clark J., Cohen M.B., Grieskamp W., Harman M., Harrold M.J., Mcminn P. 2013. An orchestrated survey of methodologies for automated software test case generation. J. Syst. Softw. 86, 8 (August 2013), pp 1978-2001. DOI: 10.1016/j.jss.2013.02.061

[5]. Вертанов С.П., Герасимов А.Ю. Динамический анализ программ с целью поиска ошибок и уязвимостей при помощи целенаправленной генерации входных данных. Труды ИСП РАН, том 26, вып. 1, 2014 г., стр 375-394. DOI: 10.15514/ISPRAS-2014-26(1)-15

[6]. Герасимов А.Ю. Обзор подходов к улучшению качества результатов статического анализа программ. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 75-98. DOI: 10.15514/ISPRAS-2017-29(3)-6

[7]. Allamanis M., Barr E.T., Devanbu P., Sutton C. A Survey of Machine Learning for Big Code and Naturalness. Размещено на сайте arxiv.org 18 сентября 2017 г. Режим доступа: https://arxiv.org/abs/1709.06182

[8]. Bruch M., Monperrus M., Mezini M. Learning from examples to improve code completion systems. Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on foundations of software engineering, 2009, pp. 213-222. DOI: 10.1145/1595696.1595728

[9]. Proksch S., Lerch J., Mezini M. 2015. Intelligent Code Completion with Bayesian Networks. ACM Trans. Softw. Eng. Methodol. 25, 1, Article 3 (December 2015), 31 pages. DOI: 10.1145/2744200

[10]. Hindle A., Barr E.T., Su Z., Gabel M., Devanbu P. On the naturalness of software. Proceedings of the 34th International Conference on Software Engineering, 2012, pp. 837-847.

[11]. Franks C., Tu Z., Devanbu P., Hellendoorn V. CACHECA: a cache language model based code suggestion tool. Proceedings of the 37th International Conference on Software Engineering - Volume 2, 2015, pp. 705-708.
[12]. Nguyen T.T., Nguyen A.T., Nguyen H.A., Nguyen T.N. A statistical semantic language model for source code. Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, 2013, pp. 532-542. DOI: 10.1145/2491411.2491458
[13]. Bielik P., Raychev V., Vechev M. PHOG: probabilistic model for code. Proceedings of the 33rd International Conference on International Conference on Machine Learning,Vol. 48, 2016, pp. 2933-2942.
[14]. Maddison C.J., Tarlow D. Structured generative models of natural source code. Proceedings of the 31st International Conference on International Conference on Machine Learning,Vol. 32, 2014, II-649-II-657.
[15]. Raychev V., Vechev M., Yahav E. Code completion with statistical language models. Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2014, pp. 419-428. DOI: 10.1145/2594291.2594321
[16]. Smola Alex., Vishwanathan S.V.N. Introduction to Machine Learning. Cambridge University Press (2008).
[17]. Hochreiter S., Schmidhuber J. Long Short-Term Memory. Neural Comput. 9, 8 (November 1997), 1997, 1735-1780. DOI: 10.1162/neco.1997.9.8.1735
[18]. Siegelmann HT. Computation beyond the turing limit. Science. 1995 Apr 28; 268 (5210):545-8. DOI: 10.1126/science.268.5210.545
[19]. Библиотека OpenGL. Режим доступа: https://www.opengl.org/
[20]. Примеры использования C API OpenGL. Режим доступа: https://www.khronos.org/opengl/wiki/Code_Resources
[21]. LLVM — компиляторная инфраструктура. Режим доступа: https://llvm.org/
[22]. Clang — фронт-енд для семейства Си-подобных языков. Режим доступа: https://clang.llvm.org/
[23]. Keras - фреймворк для создания нейронных сетей. Режим доступа: https://keras.io/

Searching for missing library function calls using machine learning

I.A. Yakimov <ivan.yakimov.research@yandex.ru>
A.S. Kuznetsov <ASKuznetsov@sfu-kras.ru>

Institute of space and informatics technologies, Siberian Federal University, Akademika Kirenskogo 26 st., Krasnoyarsk, 660074, Russia

Abstract. Software development is a complex and error-prone process. In order to reduce the complexity of software development, third-party libraries are being created. Examples of source codes for popular libraries are available in the literature and online resources. In this paper, we present a hypothesis that most of these examples contain repetitive patterns. Moreover, these patterns can be used to construct models capable of predicting the presence (or absence) of missing calls of certain library functions using machine learning. To confirm this hypothesis, a system was implemented that implements the described functional. Experimental studies confirm the hypothesis. The accuracy of the results reaches 80% with a

top-4 accuracy. It can be concluded that this system, with further development, can find industrial application.

Keywords: OpenGL; software quality; recommender systems; machine learning; neural networks;

DOI: 10.15514/ISPRAS-2017-29(6)-6

For citation: Yakimov I.A., Kuznetsov A.S. Searching for missing library function calls using machine learning. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 6, 2017. pp. 117-134 (in Russian). DOI: 10.15514/ISPRAS-2017-29(6)-6

References

[1]. Long F., Rinard M. Automatic patch generation by learning correct code. Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2016, pp 298-312. DOI: 10.1145/2837614.2837617

[2]. Bruch M., Bodden E., Monperrus M., Mezini M. 2010. IDE 2.0: collective intelligence in software development. Proceedings of the FSE/SDP workshop on Future of software engineering research, 2010, pp. 53-58. DOI: 10.1145/1882362.1882374

[3]. Cadar C., Godefroid P., Khurshid S., Păsăreanu C.S., Sen K., Tillmann N., Visser W. Symbolic execution for software testing in practice: preliminary assessment. Proceedings of the 33rd International Conference on Software Engineering, 2011, pp 1066-1071. DOI: https://doi.org/10.1145/1985793.1985995

[4]. Anand S., Burke E.K., Chen T.Y., Clark J., Cohen M.B., Grieskamp W., Harman M., Harrold M.J., Mcminn P. 2013. An orchestrated survey of methodologies for automated software test case generation. J. Syst. Softw. 86, 8 (August 2013), pp 1978-2001. DOI: 10.1016/j.jss.2013.02.061

[5]. Vartanov S. P., Gerasimov A. Y. Dynamic program analysis for error detection using goal-seeking input data generation. Trudy ISP RAN / Proc. of ISP RAS, vol. 26, issue 1, 2014, pp. 375-394 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-15

[6]. Gerasimov A.Y. Survey on static program analysis results refinement approaches. Trudy ISP RAN / Proc. of ISP RAS, vol. 29, issue 3, 2017, pp. 75-98. DOI: 10.15514/ISPRAS-2017-29(3)-6 (in Russian)

[7]. Allamanis M., Barr E.T., Devanbu P., Sutton C. A Survey of Machine Learning for Big Code and Naturalness. Размещено на сайте arxiv.org 18 сентября 2017 г. Режим доступа: https://arxiv.org/abs/1709.06182

[8]. Bruch M., Monperrus M., Mezini M. Learning from examples to improve code completion systems. Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on foundations of software engineering, 2009, pp. 213-222. DOI: 10.1145/1595696.1595728

[9]. Proksch S., Lerch J., Mezini M. 2015. Intelligent Code Completion with Bayesian Networks. ACM Trans. Softw. Eng. Methodol. 25, 1, Article 3 (December 2015), 31 pages. DOI: 10.1145/2744200

[10]. Hindle A., Barr E.T., Su Z., Gabel M., Devanbu P. On the naturalness of software. Proceedings of the 34th International Conference on Software Engineering, 2012, pp. 837-847.

[11]. Franks C., Tu Z., Devanbu P., Hellendoorn V. CACHECA: a cache language model based code suggestion tool. Proceedings of the 37th International Conference on Software Engineering - Volume 2, 2015, pp. 705-708.
[12]. Nguyen T.T., Nguyen A.T., Nguyen H.A., Nguyen T.N. A statistical semantic language model for source code. Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, 2013, pp. 532-542. DOI: 10.1145/2491411.2491458
[13]. Bielik P., Raychev V., Vechev M. PHOG: probabilistic model for code. Proceedings of the 33rd International Conference on International Conference on Machine Learning,Vol. 48, 2016, pp. 2933-2942.
[14]. Maddison C.J., Tarlow D. Structured generative models of natural source code. Proceedings of the 31st International Conference on International Conference on Machine Learning,Vol. 32, 2014, II-649-II-657.
[15]. Raychev V., Vechev M., Yahav E. Code completion with statistical language models. Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2014, pp. 419-428. DOI: 10.1145/2594291.2594321
[16]. Smola Alex., Vishwanathan S.V.N. Introduction to Machine Learning. Cambridge University Press (2008).
[17]. Hochreiter S., Schmidhuber J. Long Short-Term Memory. Neural Comput. 9, 8 (November 1997), 1997, 1735-1780. DOI: 10.1162/neco.1997.9.8.1735
[18]. Siegelmann HT. Computation beyond the turing limit. Science. 1995 Apr 28; 268 (5210):545-8. DOI: 10.1126/science.268.5210.545
[19]. OpenGL: https://www.opengl.org/
[20]. OpenGL code resource: https://www.khronos.org/opengl/wiki/Code_Resources
[21]. LLVM — compiler Infrastructure: https://llvm.org/
[22]. Clang — a C language family frontend for LLVM: https://clang.llvm.org/
[23]. Keras – the Python Deep Learning library: https://keras.io/

Null safety benchmarks for object initialization

A.V. Kogtenkov <kwaxer@mail.ru>
Independent scientist,
Podolsk, Russia

Abstract. Null pointer dereferencing remains one of the major issues in modern object-oriented languages. An obvious addition of keywords to distinguish between never null and possibly null references appears to be insufficient during object initialization when some fields declared as never null may be temporary null before the initialization completes. There are several proposals to solve the object initialization problem. How can they be compared in practice? Are the implementations sound? This work presents a set of examples distilling out the use cases from publications on the subject and open source libraries and explains the criteria behind. Then it discusses expected results for a selected set of tools performing null safety checks for Eiffel, Java, and Kotlin, and concludes with the actual outcomes demonstrating immaturity of the solutions.

Keywords: null pointer dereferencing; null safety; void safety; object initialization; static analysis; null safety benchmarks.

DOI: 10.15514/ISPRAS-2017-29(6)-7

For citation: Kogtenkov A.V. Null safety benchmarks for object initialization. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 6, 2017. pp. 135-150 (in Russian). DOI: 10.15514/ISPRAS-2017-29(6)-7

1. Introduction

To construct a sound null-safe type system, most solutions of the problem add a notion of non-null and maybe-null types, usually expressed with additional type annotations. Such annotations would be sufficient to solve the null safety problem if objects could be created in an atomic operation, so that all fields marked as non-null were initialized with object references. Unfortunately, sequential initialization of the fields breaks the solution. Several proposals solving the object initialization issue suggest extending the type systems further to identify objects that are not completely initialized. Another group of approaches is based on static analysis that does not require any additional annotations at the expense of more sophisticated checks.

The theoretical solutions in the cited works specify a subset of an object-oriented language rather than a full language. The omission of the real-life language constructs leads to omission of all required checks in the actual implementations. This causes the following issues:

- It is hard to reason about the extended model for a real language because all the consequences of such an extension are unclear and difficult to grasp when they go beyond intuition and when combination of such features requires exhaustive examination of possible interactions. The models used in the theoretical frameworks are limited for exactly this reason: they are not easily scalable and an addition of a new construct increases the size of the associated proofs proportionally, i.e. the factor is multiplicative rather than additive. This can be seen on the mechanically checked formalization of the null safety in the local context where a combination of conditional expressions, loops and exceptions greatly increased the size of the proofs. Moreover, some language features, such as value types, exceptions, concurrency or garbage collection may require a complete redesign of the model with much higher complexity impact.
- The model and the real language are decoupled informally – they are developed by different people – and formally – the proofs for the model and the implementation for the real language are carried out using different tools. There is no guarantee that an implementation is correct with respect to its model.

This leads to slow adoption of the total safety guarantees in production. The total guarantees are replaced by the "partial" ones. E.g., null safety is guaranteed only when the program does not do anything "wrong". Unfortunately, this defeats the whole purpose of the safety guarantees because null dereferencing errors in such "wrong" programs may happen at unexpected places, including trusted ones.

This work is a case study of the null safety guarantees provided by existing implementations rather than by the theoretical findings. It presents a set of examples that can be used to

- check if a particular programming pattern is handled by the implementation;
- perform measurable comparison of different solutions in terms of their soundness, expressiveness and verbosity.

The examples are written in Java, Kotlin, and Eiffel. They are available for independent analysis at https://bitbucket.org/kwaxer/null-safety-benchmark/src/?at=2017-ispras. I evaluate the benchmarks on the *Kotlin* and *Eiffel* compilers and the *Checker Framework* for Java.

The main contributions of this work are:

- development of execution scenarios of object initialization to benchmark different null safety solutions and to provide measurable comparison;

- evaluation of production solutions with null safety guarantees against the benchmarks.

The rest of the paper is organized as follows. Section 2 identifies the key reasons of the object initialization problem and specifies additional requirements to the benchmarks. Section 3 gives an overview of the existing work in the area. Section 4 classifies scenarios to test implementations with null safety guarantees and reviews the structure of the proposed benchmark suite. Section 5 presents the results of the evaluation on some production environments. Section 6 provides a quick summary and concludes with the directions of future work.

2. Overview

2.1 Reasons of the object initialization problem

Null safety is complicated for object initialization. To understand why, I suggest to look at how program execution can lead to the null reference exception. Firstly, the object that causes the problem should not complete its initialization, i.e., some of its fields of non-null types should be null. Secondly, this object should be accessible – either directly or through some variables. Thirdly, the information that its initialization is incomplete should be lost. Otherwise, it would be easy to report the error at compile time. Finally, the reference retrieved from the uninitialized field of the object should be dereferenced to trigger the exception. To summarize, there are the following roots of the problem:

- *Non-atomic initialization* of an object leads to the possibility to have fields with null values even when their type is non-null.
- *Aliasing* allows for accessing the same object from an arbitrary point of the program, in particular, from the code that does not expect an incompletely initialized object.
- *Uncontrollable control flow*, interrupting the regular one, makes sequential reasoning about program execution useless.
- *Dereferencing* of an uninitialized field of the incompletely initialized object triggers the exception.

The solutions extending the type system with new types limit the operations on incompletely initialized objects. The solutions based on static analysis, such as *practical void safety* , specify the conditions, when dereferencing may be unsafe, and forbid such dereferencing altogether.

2.2 Restrictions on the benchmarks

In order to focus only on the object initialization problem, I put the following restrictions on the code in the benchmarks:

- Self-containment – the code should not rely on or assume any null-safe properties of the classes outside the examples.
- Limited null safety scope – the code should reflect only issues with object initialization. Other mechanisms, such as array initialization and initialization of static fields in Java, companion objects in Kotlin and once features in Eiffel, as well as general data flow analysis for null safety are out of scope.

3. Related work

Alexander J. Summers and Peter Müller set the following design goals of the type system they propose to use for null safety:

1. *Modularity:* the type system can check each class type separately.
2. *Soundness:* the type system is safe, i.e., null pointer exceptions are impossible at run-time.
3. *Expressiveness:* the type system handles common initialization patterns. In particular, it allows objects to escape from constructors and supports the initialization of cyclic structures.
4. *Simplicity:* the type system is simple and requires little annotation overhead.

I use this list in section 4 as the base for further fine-grained classification of solution properties. Then, the properties can be evaluated with specific examples, thus allowing for measurable comparison of different implementations. The detailed classification of the solution properties is reviewed in the earlier publication . The review is based on the associated algorithms and descriptions. Compared to this work, it considers expected behavior of the solutions with wider spectrum of properties, because it also includes qualitative ones, such as modularity. On the other hand, that review does not provide actual code to check the tools, nor does it verify that the proposed test cases can be expressed in a specific language.

Every publication on the problem of object initialization has few examples that authors use either to demonstrate issues with their approach, or to explain how the issues are resolved. I collected all the examples from the publications I know about as well as found in class libraries and divided them into the following groups.

3.1 Polymorphic call from a constructor

When a constructor of a superclass is invoked in C#, a call to a virtual method on `this` is considered a bad practice. At this moment, subclass fields of the object are not initialized yet and using them in the polymorphic call is unsafe. Manuel Fähndrich and Rustan Leino describe an example of this situation. In a descendant class, before one of its fields is set, the superclass constructor is called. The

constructor invokes a virtual method. The override of the method in the descendant accesses the uninitialized field leading to `NullReferenceException`.

Xin Qi and Andrew C. Myers give a similar example where they consider a class `Point` and its subclass `CPoint` that adds a color attribute.

3.2 Polymorphic callback from a constructor

Fig. 1 shows a sample object structure taken from the portable GUI library *EiffelVision*[1] employing a bridge pattern to support different platforms. A client of the library directly works only with the interface objects. Upon its creation, the interface object creates an implementation object that, on completion of initialization, notifies the interface object via a callback. If at this point some non-null fields of the interface object are unset, access on them causes a null dereferencing error.

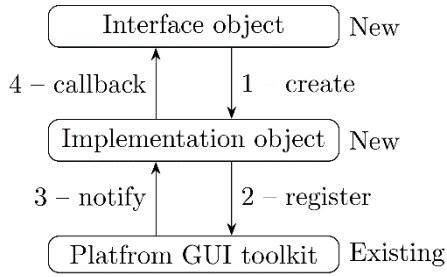


Fig. 1. Object structure in a portable GUI library.

3.3 Modification of existing structures

The ability to invoke regular procedures inside a creation procedure is convenient, e.g., for a mediator pattern . This pattern decouples objects so that they do not know about each other, but still can communicate using an intermediate object, *mediator*. Concrete types of the communicating objects are unknown to the mediator, and, therefore, the mediator cannot create them. A mediator's client is responsible for creating necessary communicating objects instead.

The communicating objects know about the mediator and can register themselves in the mediator according to their role. If the registration is done in the constructors of the communicating objects, the mediator's clients do not need to clutter the code with calls to a special feature *register* every time they create a new communicating object. An assignment like `x = new Comm (mediator)` should do both actions: the recording of the mediator object in the new communicating object, and

[1] https://www.eiffel.org/doc/solutions/EiffelVision%202

the registration of the communicating object in the mediator. A chat room adapted from and shown in Fig. 2 is an example implementing a mediator pattern.

```
class ROOM create make feature
        users: ARRAYED_LIST [USER]
        make
                do
                        create users.make (0)
                end
        join (a: USER)
                do
                        users.extend (a)
                end
        send (s: STRING)
                do
                        across users as u loop
                                u.item.receive (s)
                        end
                end
end
```

```
class USER create make feature
        room: ROOM
        make (r: ROOM)
                do
                        room := r
                        r.join (Current)
                end
        send (s: STRING)
                do
                        room.send (s)
                end
        receive (s: STRING)
                do
                        io.put_string (s)
                        io.put_new_line
                end
end
```

Fig. 2. Example of a mediator pattern (in Eiffel).

When the feature *join* is called in the creation procedure *make* of a *USER* object, all fields of the object should be set.

The need to register a newly created object in an existing one is also present in the earlier example with the GUI library in Fig. 1 where the newly created implementation object has to register itself in the existing GUI toolkit object to dispatch events from the underlying GUI toolkit to the implementation object and then to the interface object.

3.4 Circular references

An issue arises when two objects reference each other. If the corresponding fields have non-null types, access to them should be protected to avoid retrieving **`null`** by the code that relies on the field types and, therefore, expects non-null values. Manuel Fähndrich and Songtao Xia demonstrate the problem on a linked list example with a sentinel.

When a new list is constructed, a special sentinel node is created. The sentinel should reference the original list object. In other words, an incompletely initialized list object has to be passed to the sentinel node constructor as an argument. An attempt to access the field that is expected to reference the sentinel node inside the sentinel constructor would compromise null safety. Therefore, there should be means to prevent such accesses or to make them safe (e.g., by treating field values as possibly null and as referring to uninitialized objects).

There are similar circular references in the classes included in the library *Gobo*[2] to model XML documents. According to the XML specification , every document has one root element and every XML element has one parent, coinciding with the document node for the root element.

3.5 Self-referencing

A particular case of circular references concerns an object that references itself rather than another object. Xin Qi and Andrew C. Myers give the example of a binary tree where every node has a parent, and the root is a parent to itself. At a binary node creation, left and right nodes should get a new parent and the parent should reference itself. With any initialization order there are states where the new binary node should be used to initialize either its own field or the field `parent` of its left or right nodes before it is completely initialized. Therefore, arbitrary accesses to this node should be protected.

3.5 Safety violations

In addition to valid cases, authors usually mention examples that should trigger a compiler error. This aims at the original goal: a sound solution should catch potential null dereferencing at compile time.

4. Benchmark criteria and null safety suite

The most important goal of the null safety design is soundness. It limits the possibilities to write arbitrary code that is still null safe. The general problem of safe object initialization is undecidable. Therefore, the code can be checked in finite time only when some restrictions are imposed on it.

Soundness and expressiveness work against each other: the simpler the language rules, the less code can be written without violating them. If the rules are too strict, some scenarios found in real software can become extinct. E.g., according to the theory, the *raw types* do not allow for creation of circular structures, the *free and committed types* rule out registration of objects in existing object structures inside constructors, and *practical void safety* disallows any qualified calls, including potentially safe ones, as soon as there are some incompletely initialized objects in the current execution context.

The proposed benchmark suite ignores some important properties of the solutions. Measuring such properties requires significantly different code with tight dependency on the underlying language and involved tools. Consequently, it would be difficult to do the comparison. The ignored criteria include

- Modularity, including scope, telling whether it is sufficient to analyze (recursively) ancestors and suppliers of the class to be checked, and

[2] http://www.gobosoft.com/eiffel/gobo/xml/

incrementality, telling whether changes to previously checked code require a partial recheck rather than a complete one;

- Simplicity, including ease of use, telling whether few new simple rules are added to the language to make object initialization null safe, and performance, telling the resource consumption (algorithmic space and time complexity) to support the additional checks.

The assessment of different solutions from the theoretical point of view, based on the analysis of the corresponding algorithms, is given in the earlier paper that, in particular, elaborates on the criteria listed above. The current work focuses on the behavior of the actual implementations instead. The benchmarks are applicable only to existing tools and allow for measuring the number of additional type marks that need to be specified besides the marks “non-null” and “maybe-null”.

The test suite consists of two main parts: one for soundness examples and one for expressiveness examples. Because there are certain differences between languages in syntax and semantics, every example is equipped with an accompanying document describing the execution scenario. The document also lists possible variations of the example if present. In the text below, the names of the examples are underlined.

4.1 Soundness

Authors of all the solutions mentioned in section claim them to be sound. Unfortunately, not all aspects of a real programming environment are usually reflected in a formal model. In particular, none of the null safety formalizations reflects garbage collection that is an important channel to compromise safety guarantees.

The roots of the object initialization problem, mentioned in section , are mapped to the programming language constructs as follows:

- *Non-atomic initialization* corresponds to the order of initialization of the object fields intermingled with other computations. This work does not consider languages that support atomic (transactional) object initialization.
- Explicit *aliasing* becomes possible when an object is assigned to a field of an existing object, either passed to the constructor as an argument or directly reachable from the current context, or when the new object is thrown as an exception. Implicit *aliasing* happens when the class declares a finalizer that gets access to the object.
- *Uncontrollable control flow* can be caused by concurrent execution, preemptive execution (with exception and signal handlers), cooperative execution (coroutines).

- *Dereferencing* is done by a qualified call of the form `target.access` where `access` stands for a field or method name and `target` is a name of a reference corresponding to one of uninitialized fields of the object.

The soundness examples demonstrate potential scenarios where execution can lead to a null dereference error at run-time. The language tools should be able to detect and to report the possible error at compile time – this constitutes the null safety guarantees. If the error is not detected, the corresponding implementation is unsound.

If a program context does not expect an uninitialized object, there should be no channels that allow for the object to escape to this context. The following language mechanisms can make such escaping unexpected:

- exceptions;
- concurrency;
- cooperative execution.

These mechanisms are used in the examples in an attempt to demonstrate unsoundness of implementations. This is done by performing an unsafe dereferencing using either unqualified calls (of the form `field_name`) that access fields of the current object, or qualified calls (of the form `expression.field_name`), that access fields of the current or some other object. The values of the fields may be null or (recursively) have null values in non-null fields of referenced objects.

The escaping channels depend on the programming language. The most common ones are discussed next.

4.1.1 Registration in an existing object

A program can register a new object in an existing one. When this is done before the new object is completely initialized, there is a problem: the incompletely initialized object can be accessed via the exiting object because of aliasing. Thus, such registration should be disallowed or no accesses to the fields can assume that they are non-null. The scenario can be further classified by

1) the *source* of the reference to the existing object that can be either (a) passed as an argument to the constructor with a variant that the passed reference could correspond to the newly created (fresh) object rather than a fully initialized one, (b) retrieved from the current execution context (static fields, companion and singleton objects, once functions);
2) the *type* of the object in which the new one is registered: it can be (a) a user-defined one or (b) a built-in one (e.g., an array, a tuple, etc.).

4.1.2 Reclamation of incompletely initialized objects

Finalizers are the methods called before object's memory is reused. The finalizers are registered for calling by the run-time after object's memory is allocated and before the constructor is invoked. If the object initialization does not complete (due to an uncaught exception in the constructor), the finalizer is invoked on the incompletely initialized object. Unless a programmer keeps track of object initialization, there is no way to figure out what state the object is in. Therefore, the current object in a finalizer should be treated like at the beginning of a constructor. The reclamation example accesses non-null fields that are not initialized and, therefore, should be flagged as erroneous.

4.1.3 Out-of-order object transfer

Most programming languages allow for transferring references to objects bypassing regular control flow. The most familiar mechanism is exceptions. If a new exception object referencing an incompletely initialized one is thrown, the reference to the incompletely initialized object becomes accessible in the code that relies on the type system rules and does not expect uninitialized fields. The transfer example attempts to throw such an exception object.

4.2 Expressiveness

The examples in this group do not lead to null pointer exceptions and can be accepted as valid by the tools performing null safety checks. If the checks are too strict, they can rule out legitimate use cases.

4.2.1 Access to an initialized object

As soon as an object is completely initialized, it can be safely accessed even when it is used inside its constructor. In particular, it is safe to perform

- a callback on this object from the code to where it is passed. This pattern is discussed in section .
- registration of any form (argument, context, fresh) listed in the section about soundness. (See also section .)
- out-of-order transfer as soon as the exception object refers to the completely initialized one. Because the object is completely initialized, it does not cause a problem in the exception handling context.

4.2.2 Access to an uninitialized object

Calls on incompletely initialized objects require special precaution. They may enable somewhat higher code reuse, but are “viral”, because any reference obtained from an incompletely initialized object should be considered as maybe-null and incompletely initialized according to the rule *Field Read* . Therefore, the usability of

this coding technique is limited, and the example "uninitialized" is included in the suite merely for completeness.

4.2.3 Circular references

Object structures with references that make a cycle appear to be a common design pattern (see section). The corresponding benchmark checks whether the rules are flexible enough to allow for self-referencing of one object or mutual-referencing of two objects.

5. Evaluation results

The examples listed in section 4 are used to check the following language tools:

- *Checker Framework* version 2.2.1 from September 29, 2017
- *EiffelStudio* compiler version 17.05 from May 29, 2017
- *Kotlin* compiler version 1.1.50 from September 22, 2017

At the time of writing, the benchmark contains 6 expressiveness examples and 8 soundness examples (registration examples have 2 variants each: one with exceptions and one with threads), all in 3 programming languages, 2574 total lines of code.

Table 1. Evaluation results

(a) Results of expressiveness tests

Tool	Example: cycle: callback	cycle: self	cycle: mutual	uninitialized	registration: argument	registration: context
Checker Framework	$+_{1/2}$	$+_{1/1}$	$+_{2/2}$	$+_{1/2}$	$\oplus_{1/1}$	$\oplus_{1/1}$
EiffelStudio compiler	+	+	+	−	+	+
Kotlin compiler	+	+	+	+	+	+

(b) Results of soundness tests

Tool	Example: registration: argument	registration: fresh	registration: context	uninitialized	transfer
Checker Framework	$\ominus_{1/2}$	$\ominus_{2/2}$	$\ominus_{1/2}$	−	$\ominus_{2/3}$
EiffelStudio compiler	+	+	+	−	+
Kotlin compiler	−	−	−	−	−

Legend:

+ passed as expected	⊕passed unexpectedly	$_{m/}$ number of different additional annotations
− failed as expected	⊖failed unexpectedly	$_{/n}$ total number of additional annotations

The results for the expressiveness examples are summarized in Table 1(a). The *Checker Framework* employs the theory of *free and committed types*, and requires additional annotations to support the scenarios when **`this`** is passed from within the constructor as an argument to another constructor or method.

The theory states that all the registration examples should fail with compile-time errors, i.e., only 4 out of 6 tests should pass. However, the *Checker Framework* unexpectedly allows for assigning a non-completely initialized object to a field of a completely initialized one either received as an argument or retrieved from a static field. This deviation from the original set of rules might be the reason of the analysis unsoundness as discussed below.

The *practical void safety*, implemented in the *EiffelStudio* compiler, does not handle access on incompletely initialized objects, therefore, as expected, the example "uninitialized" does not pass. The expected score of 5 passing tests out of 6 tests matches the one obtained from the actual runs. Kotlin, on the other hand, has no special restrictions on uses of **`this`**, so all the expressiveness examples can be compiled.

The results for the soundness examples are presented in Table 1(b). In accordance with the documentation, the *Kotlin* compiler does not detect any invalid uses of the reference **`this`** in any of the examples and is unsound with respect to null safety.

As pointed out in section, object reclamation has not been mentioned in theoretical works as a potential source of null dereferencing caused by incomplete object initialization. Therefore, none of the tested tools applies special rules to finalizers. Therefore, all tested tools are unsound in this case.

The failure of the *Checker Framework* to catch errors in all other tests is surprising. Most probably, one bug is caused by missing language constructs in the simplified model language used to prove the soundness of the type system. Indeed, it has no equivalent of the statement **`throw`**. In order to preserve soundness, the statement should not satisfy the typing rules as soon as its argument is not of a committed type.

The origin of the *Checker Framework* bugs in the registration examples is unclear and demonstrates a gap between the typing rules specified in the proposed solution and the actual implementation. As a result, instead of 6 or 7 passing tests out of 8 tests, the *Checker Framework* fails in all soundness tests.

The *EiffelStudio* compiler passes 7 out of 8 soundness tests that matches the expected ratio of the *practical void safety* mechanism.

6. Conclusion and future work

None of the tested solutions guarantee complete null safety. It should be possible to fix the implementations of the *Checker Framework* and the *EiffelStudio* compiler, but at the time of writing the *Kotlin* compiler has no provisions for fixing the object initialization issue and can be thought of as “null safety aware”, rather than “null safety complete” product.

The gap between theoretical works and practical implementations is caused by a simplified model of the target language and absence of verification of the implementations against the models.

The work reveals the following areas of future development:

- application of the proposed benchmark to other frameworks claiming null safety guarantees;
- collaboration with tool developers to eliminate the deficiencies found in the tools;
- extension of the suite with new examples to cover more coding patterns found in the field.

References

[1]. Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). Fifth Edition of a Recommendation. W3C, Nov. 2008. URL: http://www.w3.org/TR/2008/REC-xml-20081126/.

[2]. Manuel Fähndrich and K. Rustan M. Leino. Declaring and Checking Non-null Types in an Object-oriented Language. In: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications. OOPSLA’03. ACM, 2003, pp. 302–312. DOI: 10.1145/949305.949332.

[3]. Manuel Fähndrich and Songtao Xia. Establishing Object Invariants with Delayed Types. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications. OOPSLA’07. ACM, 2007, pp. 337–350. DOI: 10.1145/1297027.1297052.

[4]. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., 1995.

[5]. Alexander Kogtenkov. Practical Void Safety. In: Verified Software. Theories, Tools, and Experiments. 9th International Conference, VSTTE 2017, Heidelberg, Germany, July 22–23, 2017, Revised Selected Papers. Ed. by Andrei Paskevich and Thomas Wies. Vol. 10712. Lecture Notes in Computer Science. Springer International Publishing, 2017. DOI: 10.1007/978-3-319-72308-2_9.

[6]. Alexander Kogtenkov. Towards null safety benchmarks for object initialization. In: Modeling and Analysis of Information Systems 24.6 (2017).

[7]. A.V. Kogtenkov. Mechanically Proved Practical Local Null Safety. In: Trudy ISP RAN / Proc. ISP RAS, vol. 28, issue 5, pp. 27–54. DOI: 10.15514/ISPRAS-2016-28(5)-2.

[8]. Mediator pattern. 2017. URL: https://en.wikipedia.org/wiki/Mediator_pattern (visited on 2017-11-20).
[9]. Bertrand Meyer. Targeted expressions: safe object creation with void safety. July 30, 2012. URL: http://se.ethz.ch/~meyer/publications/online/targeted.pdf (visited on 2017-05-08).
[10]. Xin Qi and Andrew C. Myers. Masked Types for Sound Object Initialization. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL'09. ACM, 2009, pp. 53–65. DOI: 10.1145/1480881.1480890.
[11]. Alexander J. Summers and Peter Müller. Freedom Before Commitment: A Lightweight Type System for Object Initialisation. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications. OOPSLA'11. ACM, 2011, pp. 1013–1032. DOI: 10.1145/2048066.2048142.

Эталонные тесты безопасности нулевых ссылок при инициализации объекта

А.В. Когтенков <kwaxer@mail.ru>
Независимый учёный,
Россия, г. Подольск

Аннотация. Разыменование нулевого указателя остаётся одной из основных проблем в современных объектно-ориентированных языках. Очевидное добавление ключевых слов, чтобы различать между всегда ненулевыми и возможно нулевыми ссылками, оказывается недостаточным во время инициализации объекта, когда некоторые поля, объявленные всегда ненулевыми, могут временно быть нулевыми до окончания инициализации. Существует несколько подходов к решению проблемы инициализации объекта. Каким образом их можно сравнить практически? Являются ли реализации обоснованными? Данная работа представляет набор примеров, выделяя сценарии использования из публикаций по теме и библиотек с открытым кодом, и объясняет стоящие за ними критерии. Затем она обсуждает ожидаемые результаты для выбранного набора инструментов, производящих проверки безопасности нулевых ссылок для Eiffel, Java и Kotlin, и завершается фактическими результатами, демонстрирующими незрелость решений.

Ключевые слова: разыменование нулевого указателя; безопасность нулевых ссылок; безопасность пустых ссылок; инициализация объектов; статический анализ; эталонные тесты безопасности нулевых ссылок.

DOI: 10.15514/ISPRAS-2017-29(6)-7

Для цитирования: Когтенков А.В. Эталонные тесты безопасности нулевых ссылок при инициализации объекта. Труды ИСП РАН, том 29, вып. 6, 2017 г., стр. 135-150 (на английском языке). DOI: 10.15514/ISPRAS-2017-29(6)-7

Список литературы

[1] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). Fifth Edition of a Recommendation. W3C, Nov. 2008. URL: http://www.w3.org/TR/2008/REC-xml-20081126/.

[2] Manuel Fähndrich and K. Rustan M. Leino. Declaring and Checking Non-null Types in an Object-oriented Language. In: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications. OOPSLA'03. ACM, 2003, pp. 302–312. DOI: 10.1145/949305.949332.

[3] Manuel Fähndrich and Songtao Xia. Establishing Object Invariants with Delayed Types. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications. OOPSLA'07. ACM, 2007, pp. 337–350. DOI: 10.1145/1297027.1297052.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc., 1995.

[5] Alexander Kogtenkov. Practical Void Safety. In: Verified Software. Theories, Tools, and Experiments. 9th International Conference, VSTTE 2017, Heidelberg, Germany, July 22–23, 2017, Revised Selected Papers. Ed. by Andrei Paskevich and Thomas Wies. Vol. 10712. Lecture Notes in Computer Science. Springer International Publishing, 2017. DOI: 10.1007/978-3-319-72308-2_9.

[6] Alexander Kogtenkov. Towards null safety benchmarks for object initialization. In: Modeling and Analysis of Information Systems 24.6 (2017).

[7] A.V. Kogtenkov. Mechanically Proved Practical Local Null Safety. In: Trudy ISP RAN / Proc. ISP RAS, vol. 28, issue 5, pp. 27–54. DOI: 10.15514/ISPRAS-2016-28(5)-2.

[8] Mediator pattern. 2017. URL: https://en.wikipedia.org/wiki/Mediator_pattern (visited on 2017-11-20).

[9] Bertrand Meyer. Targeted expressions: safe object creation with void safety. July 30, 2012. URL: http://se.ethz.ch/~meyer/publications/online/targeted.pdf (visited on 2017-05-08).

[10] Xin Qi and Andrew C. Myers. Masked Types for Sound Object Initialization. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL'09. ACM, 2009, pp. 53–65. DOI: 10.1145/1480881.1480890.

[11] Alexander J. Summers and Peter Müller. Freedom Before Commitment: A Lightweight Type System for Object Initialisation. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications. OOPSLA'11. ACM, 2011, pp. 1013–1032. DOI: 10.1145/2048066.2048142.

Построение предикатов безопасности для некоторых типов программных дефектов*

[1] *А.Н. Федотов <fedotoff@ispras.ru>*
[1] *В.В. Каушан <korpse@ispras.ru>*
[1,2,3,4] *С.С. Гайсарян <ssg@ispras.ru>*
[1] *Ш.Ф. Курмангалеев <kursh@ispras.ru>*
[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*
[2] *2119991 ГСП-1 Москва, Ленинские горы, МГУ имени М.В. Ломоносова, 2-й учебный корпус, факультет ВМК*
[3] *Московский физико-технический институт, 141700, Московская область, г. Долгопрудный, Институтский пер., 9*
[4] *Национальный исследовательский университет «Высшая школа экономики» 101000, Россия, г. Москва, ул. Мясницкая, д. 20*

Аннотация. В статье рассматриваются подходы и способы выполнения кода с использованием уязвимостей в программах. В частности, рассмотрены способы выполнения кода для переполнения буфера на стеке и в динамической памяти, для уязвимости использования памяти после освобождения и для уязвимости форматной строки. Описываются методы и подходы, позволяющие автоматически получать наборы входных данных, которые приводят к выполнению произвольного кода. В основе этих подходов лежит использование символьной интерпретации. Динамическое символьное выполнение предоставляет набор входных данных, который направляет программу по пути активации уязвимости. Предикат безопасности представляет собой дополнительные символьные уравнения и неравенства, описывающие требуемое состояние программы, наступающее при обработке пакета данных, например, передача управления на требуемый адрес. Объединив предикаты пути и безопасности, а затем решив полученную систему уравнений, можно получить набор входных данных, приводящий программу к выполнению кода. В работе представлены предикаты безопасности для перезаписи указателя, перезаписи указателя на функцию и уязвимости форматной строки, которая приводит к переполнению буфера на стеке. Описанные предикаты безопасности использовались в методе оценки критичности программных дефектов. Проверка работоспособности предикатов безопасности оценивалась на наборе тестов, который использовался в конкурсе *Darpa Cyber Grand*

* Работа поддержана грантом РФФИ № 17-01-00600 А

Challenge. Тестирование предиката безопасности для уязвимости форматной строки, приводящей к переполнению буфера, проводилось на программе Ollydbg, содержащей эту уязвимость. Для некоторых примеров удалось получить входные данные, приводящие к выполнению кода, что подтверждает работоспособность предикатов безопасности.

Ключевые слова: ошибки; символьное выполнение; предикат безопасности; анализ бинарного кода; динамический анализ.

DOI: 10.15514/ISPRAS-2017-29(6)-8

Для цитирования: Федотов А.Н., Каушан В.В., Гайсарян С.С., Курмангалеев Ш.Ф. Построение предикатов безопасности для некоторых типов программных дефектов. Труды ИСП РАН, том 29, вып. 6, 2017 г., стр. 151-162. DOI: 10.15514/ISPRAS-2017-29(6)-8

1. Введение

В современном мире безопасности программного обеспечения уделяют большое внимание. Важным аспектом обеспечения безопасности является поиск ошибок и уязвимостей. Поиск ошибок и уязвимостей может осуществляться на разных стадиях жизненного цикла программного обеспечения: в виде стороннего аудита во время эксплуатации ПО или на стадии разработки. В настоящие время применение безопасного цикла разработки (SDLC) становится неотъемлемой частью при создании программ промышленного уровня. ГОСТ Р 56939-2016 «Разработка безопасного программного обеспечения. Общие требования», регламентирует требования к разработке такого ПО. В соответствии с ГОСТ разработчик должен производить: моделирование угроз информационной безопасности, статический анализ кода, экспертизу исходного кода, функциональное тестирование программы, тестирование на проникновение, динамический анализ кода программы, фаззинг-тестирование программы.

Среди множества уязвимостей наиболее опасными являются уязвимости, позволяющие атакующему выполнить произвольный код [1]. Такие уязвимости достаточно сложно обнаружить автоматическими средствами, и поэтому довольно часто приходится привлекать аналитика. Кроме этого, возникает дополнительная сложность в оценке степени критичности проявления уязвимости.

Как правило, для активации уязвимости требуется специально сформированный набор входных данных, при обработке которого, программа перейдёт в состояние проявления этой уязвимости. Для автоматического формирования таких наборов входных данных, прежде всего, необходимо изучить и формализовать действия аналитика. На сегодняшний день существуют подходы и программные средства, позволяющие для некоторых типов уязвимостей получать такие входные данные [2, 4]. Эти подходы основываются на использовании динамического символьного выполнения.

Применение этой техники обусловлено несколькими аспектами. Для успешной передачи управления и выполнения произвольного кода, необходимо обладать сведениями о том, какие ячейки памяти и какие регистры процессора находятся под влиянием входных данных в момент активации уязвимости. Кроме того, в результате динамического выполнения формируются ограничения, описывающие путь в программе, на котором может произойти активация уязвимости (предикат пути). Добавив необходимые условия, требуемые для выполнения кода (предикат безопасности), и затем, решив полученный набор ограничений, можно получить искомый набор входных данных. Существующее системы имеют в своём арсенале набор предикатов безопасности для некоторых способов эксплуатации уязвимостей переполнения буфера на стеке и куче, а также уязвимости форматной строки [3-5]. В данной работе представлен расширенный набор предикатов безопасности для таких уязвимостей, как: переполнение буфера на стеке и куче, уязвимость форматной строки и использование памяти после освобождения. Разработанные предикаты применялись для оценки критичности программных дефектов [6,7].

Статья организована следующим образом. Во втором разделе приводится обзор некоторых типов дефектов программного обеспечения. В третьем разделе описаны методы и инструменты используемые для оценки критичности программных дефектов. В четвёртом разделе предложены разработанные предикаты безопасности. В пятом разделе рассматриваются результаты применения разработанных предикатов безопасности. В заключении обсуждаются полученные результаты и дальнейшие направления исследований.

2. Обзор критических программных дефектов

Перехват потока управления программы или выполнение произвольного кода в контексте программы является одной из наиболее опасных атак, а дефекты, срабатывания которых могут привести к реализации подобной атаки будем называть критическими. Для выполнения кода необходимо решить следующие задачи:

- разместить целевой код в одном из контролируемых буферов памяти;
- передать управление на адрес буфера с кодом.

Выполнение этих задач зависит от типа уязвимостей, а также состояния программы при котором она проявляется.

Переполнение буфера на стеке. Возможность выполнения произвольного кода при возникновении переполнения буфера на стеке известна довольно давно и описана в статье [8]. В результате переполнения буфера под контролем атакующего могут оказаться данные, манипуляция с которыми может позволить ему передать управление на свой код. Эти данные можно разделить на два типа: служебные данные и данные программы. К служебным данным относятся, например, адрес возврата из функции и указатель на кадр стека. К

данным программы можно отнести указатели, в том числе указатели на функции. На рис. 1 представлена организация стека и способы атаки, результатом которой является выполнение кода.

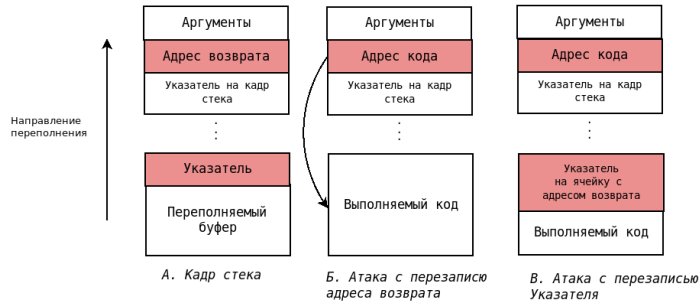


Рис. 1. Организация стека и способы атак для уязвимости переполнения буфера на стеке

Fig. 1. The organization of the stack and methods of attack for the buffer overflow vulnerability on the stack

Наиболее распространённой является атака, приведённая на Рис. 1Б. Атакующий размещает в переполняемом буфере свой код, а адрес возврата подменяет адресом этого буфера. Для данной атаки уже описаны предикаты безопасности, и они успешно реализованы в инструментах [9,10]. В случае, когда на стеке находится указатель на какую-либо переменную, то его можно перезаписать значением адреса ячейки, содержащей адрес возврата из функции. Для успешной реализации атаки необходимо, чтобы в дальнейшем по этому указателю произошла запись контролируемого атакующим значения. В этом случае возможно подменить записываемое значение на адрес буфера, в котором находится его код, и таким образом произойдёт перезапись адреса возврата из функции. Возврат из функции приведёт к передаче управления на код атакующего. Частный случай − перезапись указателя на функцию. В этом случае нарушителю достаточно перезаписать значение указателя адресом буфера с кодом и после вызова функции по указателю произойдёт передача управления на код полезной нагрузки.

Переполнение буфера на куче. В некоторых менеджерах динамической памяти в ОС Linux, основанных на менеджере памяти Doug Lea's Malloc (dlmalloc) существуют возможности для проведения атак, приводящих к выполнению кода. Одной из первых техник эксплуатация была техника *unlink***,** описанная в работе [11]. В результате переполнения буфера портятся служебные данные выделенных объектов в динамической памяти и после вызова функции освобождения может возникнуть ситуация, при которой атакующий может писать произвольное значение по произвольному адресу (CWE-123) [12], передача управления и выполнение кода здесь осуществляется подобно ситуации с перезаписью указателя. Таким образом, подавляющее большинство уязвимостей переполнения буфера на куче

приводит к выполнению произвольного кода, но в 2003 году было разработано исправление, препятствующее возникновению ситуации CWE-123 путем аварийного завершения программы. В работе [13] описаны различные способы, позволяющее эксплуатировать уязвимость переполнения буфера на куче, некоторые из них работают в современных версиях менеджеров памяти. Эти способы требуют соблюдения большого числа условий, и в открытом доступе находятся примеры эксплуатации модельных программ. Как и для переполнения буфера на стеке, остаются доступными способы эксплуатации, основанные на перезаписи указателя.

Использование памяти после освобождения. Один из способов проведения атаки с использованием памяти после освобождения применяется к программам, которые написаны на языке Си++ и содержат вызовов виртуального метода. Для проведения успешной атаки должны выполниться следующие условия:

- на куче выделен объект, содержащий виртуальный метод;
- объект удаляется (освобождается память, выделенная под этот объект);
- выделяется новый блок памяти, таким образом, что он пересекается с ранее выделенным объектом;
- в новый блок записываются данные, контролируемые атакующим;
- вызов виртуального метода, ранее выделенного объекта (факт использования памяти после освобождения).

В самом начале области памяти объекта, первые 4(8) байт, содержат указатель на таблицу виртуальных функций. Таким образом, у атакующего появляется возможность перезаписать указатель на свою, специально сформированную таблицу виртуальных функций. В результате вызова метода из этой таблицы произойдёт передача управления на код атакующего. На Рис. 2 представлен фрагмент кода на языке ассемблера, осуществляющего вызов виртуального метода.

```
MOV EAX, DWORD PTR[EBP - 0x1C]
    MOV EAX, DWORD PTR[EAX]
     CALL DWORD PTR[EAX+8]
```

Рис. 2. Вызов виртуального метода
Fig. 2. Invoking the virtual method

Первая инструкция загружает указатель на объект в регистр *EAX*. Вторая инструкция загружает на регистр указатель на таблицу виртуальных функций. Третья инструкция производит вызов виртуальной функции из таблицы.

Уязвимость форматной строки. Способы выполнения кода, используя уязвимость форматной строки известны и описаны в работах [1, 14]. Кроме того, есть методы, позволяющие это делать автоматически [2-4, 15].

Уязвимость существует в случае, если атакующий контролирует строку формата. Манипулируя спецификаторами в строке формата атакующий может передать управление на свой код. Спецификатор *%n* используется для записи количества выведенных символов по заданному в строке формата адресу. Таким образом, атакующей может перезаписать, например, адрес возврата из функции адресом буфера со своим кодом. В некоторых функциях работы с форматной строкой количество выводимых символов ограничено максимальным значением 32-битного беззнакового целого числа. В связи с этим возникает проблема при эксплуатации уязвимостей в программах, работающих под управлением 64-разрядных операционных систем. В этом случае, возможна только частичная перезапись ячейки памяти, содержащий адрес. Существует ещё один способ выполнения произвольного кода, используя уязвимость форматной строки. Для этого способа подходят функции, которые печатают строку не на стандартный поток вывода, а в буфер, располагающийся, например, на стеке. Манипулируя спецификаторами строки формата, атакующий может переполнить буфер и перезаписать адрес возврата, что в последствии приведёт к выполнению кода.

3 Методы генерации критических входных данных

Под критическими входными данными будем понимать входные данные, которые приводят программу либо к аварийному завершению, либо к выполнению заданного кода нарушителя. Наиболее результативным методом генерации критических входных данных является динамический анализ, который можно рассматривать как совокупное применение метода фаззинга для автоматической генерации тестовых наборов и поиска входных данных приводящих к аварийному завершению программы, технологий динамической бинарной трансляции и/или инструментации для получения трасс выполнения и информации о покрытии кода, конкретного и символьного выполнения для анализа предикатов безопасности, генерации входных данных обеспечивающих отказ в обслуживании или перехват потока управления.

Традиционно упомянутые средства применяются в следующей последовательности: фаззинг для получения входных данных приводящих к воспроизводимому аварийному завершению, снятие трассы выполнения на найденных данных и трансляции ее в символьную форму. Затем совместно решая полученную систему уравнений и предикат безопасности описывающий необходимые условия для эксплуатации ошибки, получаем данные, которые необходимо подать на вход программе. Полученные данные, реализуют условия, описываемые в предикате безопасности, например, перехват потока управления.

Различают два основных подхода к такому анализу: полносистемный и уровня приложения.

Полносистемный подход к анализу. В этом случае для получения трассы выполнения программы используется полносистемный эмулятор, например,

Qemu [16] данный подход применятся в таких платформах как S2E [17], PANDA [18], Avatar [19] и других. Положительная сторона такого подхода заключается в обеспечении возможности детерминированного воспроизведения в рамках системы и полноте трассы выполнения, включающей в себя как код анализируемой программы, так и системный код. Минусом является скорость работы и требования к системным ресурсам.

Анализ на уровне приложения. При анализе на уровне приложения, для трассировки используется динамическая бинарная трансляция или инструментация кода программы, при этом в трассу выполнения попадают только инструкции, относящиеся к коду исполняющемся в пользовательском пространстве целевого процесса. К подобным системам относятся Angr[5], Mayhem [3], Manticore [20] и д.р.

4 Построение предикатов безопасности

При построении предикатов безопасности для перехвата потока управления требуется описать размещение кода полезной нагрузки и передачу управления на этот код.

Построение предиката безопасности для перезаписи указателя (CWE-123):

- в контролируемом буфере памяти располагается код полезной нагрузки;
- адрес указателя равен адресу ячейки памяти, содержащий адрес возврата из функции;
- значение указателя равно адресу буфера с кодом полезной нагрузки.

Построение предиката безопасности для перезаписи указателя на функцию:

- в контролируемом буфере памяти располагается код полезной нагрузки;
- значение указателя на функцию равно адресу буфера с кодом полезной нагрузки.

Построение предиката безопасности для уязвимости форматной строки, которая приводит к переполнению буфера на стеке:

- в контролируемом буфере памяти располагается код полезной нагрузки;
- форматная строка составлена таким образом, что её обработка приводит к переполнению буфера на стеке и перезаписывает адрес возврата.

При составлении форматной строки для вывода значений используется форматный спецификатор *%x*. Как правило, этот спецификатор используется с параметром ширины и имеет вид *%8x*. Присутствие в форматной строке необходимого количества таких спецификаторов вызовет переполнение буфера.

5. Применение предикатов безопасности

Для многих областей программирования, таких как компиляторные технологии, машинное обучение и др. существуют тестовые наборы, на которых можно оценить эффективность подходов. В качестве примера можно привести наборы SPEC для тестирования компиляторов и «MNIST база изображений рукописных цифр», наборы Juliet от NIST для тестирования статических анализаторов. Однако для области компьютерной безопасности, в частности комплексного тестирования средств, входящих в процесс безопасной разработки ПО такого набора не было. Более того, все средства, как правило, тестировались на различных наборах программы и платформ, что затрудняло качественную оценку систем.

В августе 2016 года прошел финал DARPA Cyber Grand Challenge[21] – соревнования по построению автономной системы поиска дефекта в программе и генерации критических входных данных для найденных уязвимостей и генерации исправлений для найденных дефектов. Победителем, в котором стала система Mayhem. Организаторами соревнования был сформирован набор программ, содержащих уязвимости. Программы различаются по сложности, функционалу и типам уязвимости. В качестве единой платформы используется основанный на Linux дистрибутив, названный DECREE OS. После окончания соревнований данный тестовый набор был перенесен на другие ОС [22].

Разработанные предикаты безопасности применялись в методе оценки критичности программных дефектов [6]. Тестирование предикатов безопасности для переполнения буфера на стеке и уязвимости использования памяти после освобождения проводилось на нескольких программах из тестового набора [22]. Этот тестовый набор состоит из 241 исполняемого файла. Данные, приводящие к аварийному завершению, были получены аналитиком для 11 программ. Используя метод предварительной фильтрации аварийных завершений из работы [7], были выбраны 8 аварийных завершений, как наиболее опасные. Для 6 из них удалось получить входные данные, приводящие к выполнение заданного кода. Ниже приводится список этих программ:

- Movie_Rental_Service;
- Multi_User_Calendar;
- Palindrome;
- PKK_Steganography;
- Sample_Shipgame;
- ValveChecks.

При анализе программы *Movie_Rental_Service* применялось построение предиката безопасности для уязвимости использования памяти после освобождения. При анализе остальных программ применялся предикат безопасности для перезаписи адреса возврата из функции во время переполнения буфера на стеке. Для программ *Bloomy_Sunday* и *Charter* не

удалось получить входные данные, приводящие к выполнению кода. В результате анализа *Bloomy_Sunday* были получены входные данные, при которых выполнение заданного кода не произошло. Такая ситуация может быть связана с тем, что в результате анализа помеченных данных в предикат пути не попали ограничения, связанные адресными зависимостями. При анализе программы *Charter* выяснилось, что при переполнении буфера переписывается указатель, но в дальнейшем записи контролируемых данных по этому указателю не происходит. Таким образом, применить предикат безопасности для перезаписи указателя не удалось.

Тестирование предиката безопасности для уязвимости форматной строки, приводящей к переполнению буфера на стеке, производилось на программе OllyDbg. Описание уязвимости приводится в [23]. Для этой программы удалось получить входные данные, приводящие к выполнению заданного кода.

6. Заключение

В статье рассмотрены способы выполнения кода, используя уязвимости в программах. Описаны современные методы и средства, позволяющие решать эту задачу автоматически. Представлены новые предикаты безопасности для перезаписи указателя, перезаписи указателя на функцию и уязвимости форматной строки, которая приводит к переполнению буфера на стеке. Разработанные предикаты безопасности применялись в методе оценки критичности программных дефектов [6]. Тестирование предикатов безопасности проводилось на наборе тестов для *Darpa Cyber Grand Challenge,* а также на программе Ollydbg, содержащей уязвимость форматной строки. В результате тестирования удалось получить входные данные, приводящие к выполнению кода.

В качестве дальнейшего направление развития можно выделить разработку предикатов безопасности, которые учитывают современные защитные механизмы, препятствующие перехвату потока выполнения. Эти механизмы можно разделить на две группы: механизмы защиты операционной системы (DEP и ASLR), и защитные механизмы, встраиваемые компилятором ("канарейка" и безопасные функции работы со строками). Эти механизмы повсеместно применяются в современных операционных системах общего назначения, а опции компилятора, обеспечивающие встраивание защит, как правило, включены в опции компиляции по умолчанию. Тем не менее, иногда защиты намеренно отключаются разработчиками, а в некоторых случаях эти защиты можно обойти.

Список литературы

[1]. C. Anley, J. Heasman, F. Lindner, G. Richarte. The shellcoder’s handbook: discovering and exploiting security holes. John Wiley & Sons, 2011, 61 c.

[2]. Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert et al. Automatic exploit generation. Communications of the ACM, vol. 57, no. 2, 2014, pp. 74–84.

[3]. Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, David Brumley. Unleashing mayhem on binary code. 2012 IEEE Symposium on Security and Privacy (SP), 2012, pp. 380–394.
[4]. Shih-Kun Huang, Min-Hsiang Huang, Po-Yen Huang et al. Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. Software 2012 IEEE Sixth International Conference on Security and Reliability (SERE), 2012, pp. 78–87.
[5]. Y Shoshitaishvili, R Wang, C Salls et al. The Art of War: Offensive Techniques in Binary Analysis. IEEE Symposium on Security and Privacy (S&P), 2016, pp. 138-157.
[6]. А.Н. Федотов, В.А. Падарян, В.В. Каушан и др. Оценка критичности программных дефектов в условиях работы современных защитных механизмов. Труды ИСП РАН, том 28, вып. 5, 2016 г., стр. 73–92. DOI: 10.15514/ISPRAS-2016-28(5)-4
[7]. А.Н Федотов. Метод оценки эксплуатируемости программных дефектов. Труды ИСП РАН, том 28, вып. 4, 2016 г., стр. 137-148. 10.15514/ISPRAS-2016-28(4)-8
[8]. One Aleph. Smashing The Stack For Fun And Profit. 1996. URL: http://phrack.org/issues/49/14.html#article (дата обращения: 08.10.2017).
[9]. Падарян В.А., Каушан В.В., Федотов А.Н. Автоматизированный метод построения эксплойтов для уязвимости переполнения буфера на стеке. Труды ИСП РАН, том 26, вып. 3, 2914 г., стр. 127-144. DOI: 10.15514/ISPRAS-2014-26(3)-7
[10]. Padaryan V.A., Kaushan V.V., Fedotov A.N. Automated exploit generation for stack buffer overflow vulnerabilities. Programming and Computer Software, vol. 41, № 6, pp. 373–380. DOI: 10.1134/S0361768815060055
[11]. Once upon a free(). 2001. URL: http://phrack.org/issues/57/9.html#article (дата обращения: 08.10.2017).
[12]. CWE-123. URL: https://cwe.mitre.org/data/definitions/123.html (дата обращения:12.05.2017).
[13]. Malloc Des-Maleficarum. 2009. URL: http://phrack.org/issues/66/10.html (дата обращения: 08.10.2017).
[14]. gera. Advances in format string exploitation. 2002. URL: http://phrack.org/issues/59/7.html#article (дата обращения: 08.10.2017).
[15]. И.А. Вахрушев, В.В. Каушан, В.А. Падарян, А.Н. Федотов. Метод поиска уязвимости форматной строки. Труды ИСП РАН, том 27, вып 4, 2015 г., стр. 23-38. DOI: 10.15514/ISPRAS-2015-27(4)-2
[16]. Bellard F. QEMU, a fast and portable dynamic translator. USENIX Annual Technical Conference, FREENIX Track, 2005, pp. 41-46.
[17]. Chipounov V., Kuznetsov V., Candea G. S2E: A platform for in-vivo multi-path analysis of software systems. ACM SIGPLAN Notices, vol. 46, №. 3, 2011, pp. 265-278.
[18]. Dolan-Gavitt B. et al. Repeatable reverse engineering with PANDA. Proceedings of the 5th Program Protection and Reverse Engineering Workshop, ACM, 2015, p. 4.
[19]. Zaddach J. et al. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. NDSS Symposium 2014.
[20]. Manticore. URL: https://github.com/trailofbits/manticore (дата обращения: 08.10.2017).
[21]. Darpa Cyber Grand Challenge. URL: http://archive.darpa.mil/cybergrandchallenge/ (дата обращения: 08.10.2017).
[22]. Darpa Cyber Grand Challenge tests pack. URL:https://github.com/trailofbits/cb-multios (дата обращения: 08.10.2017).
[23]. Ollydbg bug. URL: https://www.exploit-db.com/exploits/388/ (дата обращения: 08.10.2017).

Building security predicates for some types of vulnerabilities

[1] *A.N. Fedotov <fedotoff@ispras.ru>*
[1] *V.V. Kaushan <korpse@ispras.ru>*
[1,2,3,4] *S.S. Gaissaryan <ssg@ispras.ru>*
[1] *Sh.F. Kurmangaleev <kursh@ispras.ru>*
[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia*
[2] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia*
[3] *Moscow Institute of Physics and Technology (State University), 9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia*
[4] *National Research University Higher School of Economics (HSE) 11 Myasnitskaya Ulitsa, Moscow, 101000, Russia*

Abstract. Approaches for code execution using program vulnerabilities are considered in this paper. Particularly, ways of code execution using buffer overflow on stack and on heap, using use-after-free vulnerabilities and format string vulnerabilities are examined in section 2. Methods for automatic generation input data, leading to code execution are described in section 3. This methods are based on dynamic symbolic execution. Dynamic symbolic execution allows to gain input data, which leads program along the path of triggering vulnerability. The security predicate is an extra set of symbolic formulas, describing program's state in which code execution is possible. To get input data, leading to code execution, path and security predicates need to be united, and then the whole system should be solved. Security predicates for pointer overwrite, function pointer overwrite and format string vulnerability, that leads to stack buffer overflow are presented in the paper. Represented security predicates were used in method for software defect severity estimation. The method was applied to several binaries from Darpa Cyber Grand Challenge. Testing security predicate for format string vulnerability, that leads to buffer overflow was conducted on vulnerable version of Ollydbg. As a result of testing it was possible to obtain input data that leads to code execution.

Keywords: software bugs; symbolic execution; security predicates; binary analysis; dynamic analysis.

DOI: 10.15514/ISPRAS-2017-29(6)-8

For citation: Fedotov A.N., Kaushan V.V., Gaissaryan S.S., Kurmangaleev Sh.F. Building security predicates for some types of vulnerabilities. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 6, 2017. pp. 151-162 (in Russian). DOI: 10.15514/ISPRAS-2017-29(6)-8

References

[1]. C. Anley, J. Heasman, F. Lindner, G. Richarte. The shellcoder's handbook: discovering and exploiting security holes. John Wiley & Sons, 2011, 61 pp.

[2]. Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert et al. Automatic exploit generation. Communications of the ACM, vol. 57, no. 2, 2014, pp. 74–84.
[3]. Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, David Brumley. Unleashing mayhem on binary code. 2012 IEEE Symposium on Security and Privacy (SP), 2012, pp. 380–394.
[4]. Shih-Kun Huang, Min-Hsiang Huang, Po-Yen Huang et al. Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. Software 2012 IEEE Sixth International Conference on Security and Reliability (SERE), 2012, pp. 78–87.
[5]. Y Shoshitaishvili, R Wang, C Salls et al. The Art of War: Offensive Techniques in Binary Analysis. IEEE Symposium on Security and Privacy (S&P), 2016, pp. 138-157.
[6]. A.N. Fedotov, V.A. Padarjan, V.V. Kaushan et al. Software defect severity estimation inpresence of modern defense mechanisms. Trudy ISP RAN / Proc. ISP RAS, vol. 28, issue 5, 2016, pp. 73–92 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-4
[7]. Fedotov A.N. Method for exploitability estimation of program bugs. Trudy ISP RAN / Proc. ISP RAS, vol. 28, issue 5, 2016, pp. 137-148 (in Russian). 10.15514/ISPRAS-2016-28(4)-8
[8]. One Aleph. Smashing The Stack For Fun And Profit. 1996. URL: http://phrack.org/issues/49/14.html#article (accessed 08.10.2017).
[9]. Padarjan V.A., Kaushan V.V., Fedotov A.N. Automated exploit generation method for stack buffer overflow vulnerabilities. Trudy ISP RAN / Proc. ISP RAS, vol. 26, issue 3, 2014, pp. 127-144 (in Russian). DOI: 10.15514/ISPRAS-2014-26(3)-7
[10]. Padaryan V.A., Kaushan V.V., Fedotov A.N. Automated exploit generation for stack buffer overflow vulnerabilities. Programming and Computer Software, vol. 41, № 6, pp. 373–380. DOI: 10.1134/S0361768815060055
[11]. Once upon a free(). 2001. URL: http://phrack.org/issues/57/9.html#article (accessed 08.10.2017).
[12]. CWE-123. URL: https://cwe.mitre.org/data/definitions/123.html (accessed 12.05.2017).
[13]. Malloc Des-Maleficarum. 2009. URL: http://phrack.org/issues/66/10.html (accessed 08.10.2017).
[14]. gera. Advances in format string exploitation. 2002. URL: http://phrack.org/issues/59/7.html#article (accessed 08.10.2017).
[15]. I.A. Vahrushev, V.V. Kaushan, V.A. Padarjan, A.N. Fedotov. Search method for format string vulnerabilities. Trudy ISP RAN / Proc. ISP RAS, vol. 27, issue 4, 2015, pp. 23-38 (in Russian). DOI: 10.15514/ISPRAS-2015-27(4)-2
[16]. Bellard F. QEMU, a fast and portable dynamic translator. USENIX Annual Technical Conference, FREENIX Track, 2005, pp. 41-46.
[17]. Chipounov V., Kuznetsov V., Candea G. S2E: A platform for in-vivo multi-path analysis of software systems. ACM SIGPLAN Notices, vol. 46, №. 3, 2011, pp. 265-278.
[18]. Dolan-Gavitt B. et al. Repeatable reverse engineering with PANDA. Proceedings of the 5th Program Protection and Reverse Engineering Workshop, ACM, 2015, p. 4.
[19]. Zaddach J. et al. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. NDSS Symposium 2014.
[20]. Manticore. URL: https://github.com/trailofbits/manticore (accessed 08.10.2017).
[21]. Darpa Cyber Grand Challenge. URL: http://archive.darpa.mil/cybergrandchallenge/ (accessed: 08.10.2017).
[22]. Darpa Cyber Grand Challenge tests pack. URL:https://github.com/trailofbits/cb-multios (accessed 08.10.2017).
[23]. Ollydbg bug. URL: https://www.exploit-db.com/exploits/388/ (accessed 08.10.2017).

Мелкогранулярная рандомизация адресного пространства программы при запуске[1]

[1] *А.Р. Нурмухаметов <oleshka@ispras.ru>*
[1] *Е.А. Жаботинский <ezhabotinskiy@ispras.ru>*
[1] *Ш.Ф. Курмангалеев <kursh@ispras.ru>*
[1,2,3,4] *С.С. Гайсарян <ssg@ispras.ru>*
[1] *А.В. Вишняков <vishnya@ispras.ru>*
[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*
[2] *2119991 ГСП-1 Москва, Ленинские горы, МГУ имени М.В. Ломоносова, 2-й учебный корпус, факультет ВМК*
[3] *Московский физико-технический институт, 141700, Московская область, г. Долгопрудный, Институтский пер., 9*
[4] *Национальный исследовательский университет «Высшая школа экономики» 101000, Россия, г. Москва, ул. Мясницкая, д. 20*

Аннотация. Уязвимости программного обеспечения являются серьезной угрозой безопасности. Важной задачей является развитие методов противодействия их эксплуатации. Она приобретает особую актуальность с развитием ROP-атак. Имеющиеся средства защиты обладают некоторыми недостатками, которые могут быть использованы атакующими. В данной работе предлагается метод защиты от атак такого типа, который называется мелкогранулярной рандомизацией адресного пространства программы при запуске. Приводятся результаты по разработке и реализации данного метода на базе операционной системы CentOS 7. Рандомизацию на уровне перестановки функций осуществляет динамический загрузчик с помощью дополнительной информации, сохраненной с этапа статического связывания. Описываются детали реализации и приводятся результаты тестирования производительности, изменения времени запуска и размера файла. Отдельное внимание уделяется оценке эффективности противодействия эксплуатации с помощью ROP атак. Строятся две численных метрики: процент выживших гаджетов и оценка работоспособности примеров ROP цепочек. Приводимая в статье реализация применима в масштабах всей операционной системы и не имеет проблем совместимости с точки зрения работоспособности программ. По результатам проведенных работ была продемонстрирована работоспособность данного подхода на

[1] Работа поддержана грантом РФФИ 17-01-00600 А

реальных примерах, обнаружены преимущества и недостатки и намечены пути дальнейшего развития.

Ключевые слова: рандомизация адресного пространства; диверсификация; ASLR; ROP.

DOI: 10.15514/ISPRAS-2017-29(6)-9

Для цитирования: Нурмухаметов А.Р., Жаботинский Е.А., Курмангалеев Ш.Ф., Гайсарян С.С., Вишняков А.В Мелкогранулярная рандомизация адресного пространства программы при запуске. Труды ИСП РАН, том 29, вып. 6, 2017 г., стр. 163-182. DOI: 10.15514/ISPRAS-2017-29(6)-9

1. Введение

Современное программного обеспечения попадает к конечным пользователям с некоторым количеством ошибок. Инструменты статического анализа и разнообразное тестирование не позволяют устранить из ПО все ошибки. Среди ошибок имеются такие, которые потенциально позволяют при некоторых условиях и входных данных контролировать поведение программы или раскрывать ее данные. Существует статистика (CVE [1]) о публично известных уязвимостях в ПО. Она показывает на рис. 1, что количество обнаруживаемых уязвимостей как минимум не уменьшается. Невозможно в условиях ограниченности ресурсов найти и устранить их все. Успешная эксплуатация уязвимостей может привести к утечке конфиденциальных данных и сбоям в работе информационных систем. Из этого следует, что актуальной проблемой является предотвращение эксплуатации уязвимостей.

Классификация CVE содержит большое количество видов уязвимостей. Для их эксплуатации придуманы разнообразные методы [2–7]. Техника эксплуатации также сильно зависит от механизмов защиты, применяемых в операционных системах и на аппаратном уровне (ASLR, DEP) [8–11].

Наиболее опасным механизмом эксплуатации является ROP [2] и ему подобные [4]. Идея таких методов заключается в том, что вредоносный код составляется из гаджетов — небольших фрагментов кода самой программы, каждый из которых заканчивается инструкцией передачи управления. В случае ROP — это инструкция возврата из функции. При этом адреса гаджетов размещаются подряд на стеке, чередуясь с аргументами, которые эти гаджеты снимают со стека. Из гаджетов составляются цепочки, которые позволяют выполнять произвольный код [2].

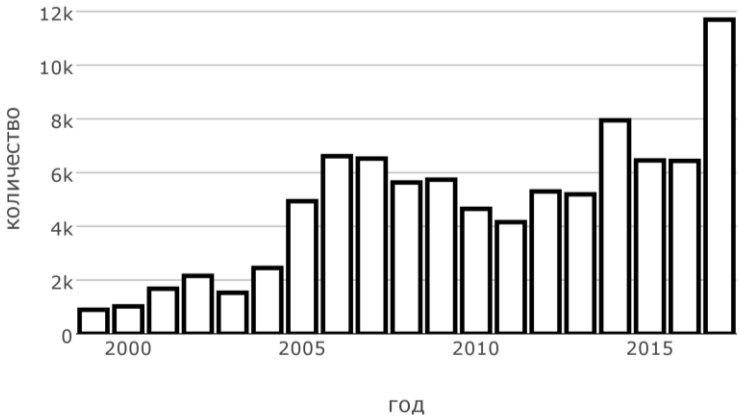


Рис. 1. Количество уязвимостей, занесенных в базу CVE с 1999 по 2017 год
Fig. 1. The number of vulnerabilities in the CVE database from 1999 to 2017 year

Таким образом, если атакующий использует уязвимость с возможностью передачи управления по произвольному адресу, то он может выполнять произвольный код в контексте атакуемой программы даже при наличии DEP (набор программных и аппаратных технологий, запрещающий приложению исполнять код из областей памяти, содержащих данные). Однако, чтобы выполнить полезный для атакующего код, нужно знать местоположение этого кода в памяти атакуемого процесса. Для построения ROP цепочки нужно знать адреса используемых гаджетов. Это означает, что для защиты от эксплуатации ошибок можно использовать рандомизацию адресного пространства процесса. Различные подходы к рандомизации адресного пространства предлагались в статьях [12–20]. Некоторые из них применяются на практике (ASLR). Однако перечисленные подходы обладают недостатками или не дают достаточной степени защиты.

Текст данной работы состоит из 6 частей. Первая глава представляет собой введение, в котором обосновывается актуальность данной работы. Во второй главе приводится обзор существующих решений. В третьей главе приводится описание предлагаемого метода и детали его реализации. Четвертая глава посвящена оценке влияния предлагаемого метода на производительность и размер программ. В пятой главе обсуждается эффективность противодействия эксплуатации уязвимостей и приводятся экспериментальные результаты. В шестой главе подводятся итоги и намечаются планы дальнейшего развития.

2. Обзор существующих решений

Из-за повсеместного присутствия программно-аппаратных защитных механизмов, предотвращающих выполнение данных (DEP), современные атаки ограничены использованием существующего кода для составления цепочек. Частично эту проблему решает технология рандомизации адресного пространства процесса (ASLR), которая позволяет задавать произвольный базовый адрес стека, кучи, сегментов кода и библиотек. Эта технология имеет два ключевых недостатка: изменяется только базовый адрес сегментов, внутренняя структура библиотеки или исполняемого файла сохраняется и относительные смещения между структурными элементами остаются постоянными; для поддержки такого механизма необходимо собирать позиционно-независимые исполняемые файлы, что отрицательно сказывается на производительности [21]. Сохранение относительных смещений внутри библиотек и исполняемых файлов приводит к тому, что атакующий может восстановить секцию кода программы или библиотеки по одному утекшему адресу. Для построения ROP цепочки необходимо знать адреса некоторого количества гаджетов. Для предотвращения вычисления всех этих адресов по одному раскрытому адресу можно сделать непредсказуемыми относительные смещения в коде.

Мелкогранулярная рандомизация во время компиляции. Существует подход по генерации диверсифицированной популяции исполняемых файлов во время компиляции и связывания [13]. Данный подход создает ряд инфраструктурных трудностей по распространению индивидуальных рандомизированных копий программы и создает непреодолимые сложности для сертификации, поскольку сертифицируется конкретный исполняемый файл. Кроме того, он обладает недостатками, связанными с возможностью утечки конкретного исполняемого файла, что компрометирует ту систему, откуда он был взят. Более того, от запуска к запуску карта памяти остается неизменной, что позволяет атакующему восстановить размещение кода в памяти в случае перезапускаемого сервиса. В свою очередь, рандомизация адресного пространства при запуске ограничивает период времени для раскрытия карты памяти процесса, а неудачные попытки, приводящие к аварийному завершению программы, делают все полученные данные бесполезными.

Мелкогранулярная рандомизация во время запуска. В рамках данной работы реализована рандомизация с гранулярностью до функций, то есть код каждой функции при запуске программы размещается в памяти по случайному адресу, после чего исправляются все упоминания адреса этой функции в других функциях. Аналогичное решение было предложено в работе Selfrando [15]. Однако их подход направлен на защиту отдельных приложений, в частности Tor Browser. Код, выполняющий рандомизацию, включается в сам исполняемый файл. Добавление необходимой для

рандомизации информации производится при помощи скриптовых оберток компилятора и компоновщика.

Сбор информации о функциях и релокациях извне компоновщика не позволяет поддерживать использование TLS. Включение кода для рандомизации в исполняемый файл увеличивает его размер и время загрузки, а кроме того мешает работе других средств защиты, так как делает исполняемый файл трудно отличимым от различных самомодифицирующихся программ, которые зачастую являются вредоносными. Подход Selfrando рассчитан на упрощение сборки отдельных приложений с поддержкой рандомизации, но плохо применим в масштабах всей системы.

Другой подход применен в инструменте XIFER [16]: библиотека, выполняющая рандомизацию, загружается при помощи LD_PRELOAD. Для рандомизации не требуется специальной подготовки исполняемого файла или библиотеки, так как вся необходимая информация получается посредством дизассемблирования. Более того, поддерживается гранулярность рандомизации на уровне базовых блоков и мельче. Это позволяет достичь большей энтропии, что делает защиту надежнее.

Однако такой подход замедляет запуск программ. Скорость обработки кода оценивается в 687 КБ/с. Это означает, что запуск приложения bash, имеющего в CentOS 7 секцию кода с размером 555 КБ, займет около одной секунды без учета используемых им динамических библиотек. Помимо этого, сильное уменьшение гранулярности рандомизации приведет к слишком частым промахам в кэше при исполнении рандомизированного кода, что уменьшит скорость работы приложения. Кроме того, дизассемблирование кода создает вероятность ошибок, особенно на больших программах, а разбиение функций на части может серьезно нарушить работу отдельных механизмов (eh_frame).

Постраничная рандомизация. При компиляции для программы генерируется позиционно-независимый код, который разбивается на фрагменты, дополняемые до целого количества страниц виртуальной памяти. Передача управления между этими фрагментами осуществляется через дополнительную таблицу (аналог таблицы глобальных смещений GOT). При запуске программы каждый фрагмент целиком загружается в случайные страницы виртуальной памяти, а адреса страниц записываются в эту таблицу.

Таким образом, хотя полученная энтропия и ниже, чем при мелкогранулярной рандомизации, но она все равно намного выше, чем при использовании обычного ASLR. При этом код не модифицируется при загрузке в память, а значит, запуск программ происходит немного быстрее. Одна физическая страница с кодом может разделяться между процессами, в то время как при мелкогранулярной рандомизации каждый процесс вынужден хранить свою рандомизированную копию библиотеки, что увеличивает использование памяти.

Первой попыткой такой реализации был oxymoron [17]. Он реализует постраничную рандомизацию для x86-64. Исполняемый код разбивается на страницы, в конце при необходимости добавляется межстраничная передача управления на следующую страницу. Адрес таблицы для межстраничной передачи управления хранится в сегментном регистре, значение которого архитектура не позволяет читать. Поэтому атакующий не может просто узнать адрес этой таблицы и считать ее для построения вредоносного кода. Каждая библиотека имеет свою таблицу, а межбиблиотечные вызовы осуществляются через функции-заглушки, которые загружают адрес таблицы конкретной библиотеки в сегментный регистр.

Другая реализация представлена в инструменте pagerando для ARM [18]. Функции переупорядочиваются для исключения лишних межстраничных переходов в конце страницы без чрезмерного количества неиспользуемой памяти в конце каждой страницы. Адрес таблицы хранится в регистре общего назначения и при межбиблиотечных вызовах сохраняется на стеке. Никаких мер для предотвращения утечки этого адреса не предпринимается. Влияние реализации постраничной рандомизации на производительность оценивается в 1-5 %.

Рандомизация во время работы программы. Бывают ситуации, при которых серверный процесс для обработки каждого запроса дублирует себя при помощи вызова fork. В таком случае атакующий может угадывать карту адресного пространства, не заботясь о стабильности работы атакуемого процесса. В случае завершения сервер создает новый процесс с той же самой картой адресного пространства, после чего можно продолжать перебор. Авторы предлагают отслеживать все указатели в памяти процесса при помощи инструментации машинного кода и динамического анализа помеченных данных, а затем с использованием этой информации заново выполнять обычный ASLR в дочернем процессе после выполнения fork. Анализ помеченных данных замедляет работу в 10-20 раз.

В нескольких работах для защиты предлагается использовать более частую перерандомизацию адресного пространства процесса. В работе [19] предлагается проводить рандомизацию перед системными вызовами, получающими информацию извне и следующими после вызовов, выводящих информацию. Таким образом, собранные данные о состоянии процесса устаревают к моменту, когда атакующий может воздействовать на поведение процесса. На SPEC2006 такой метод замедляет работу в среднем всего на 2 %, но это относительно компиляции с ключом -Og, с которым этот набор тестов работает на треть медленнее, чем с -O2. Кроме того, требуется модификация ядра ОС, а исполняемый файл должен быть дополнительно аннотирован для отслеживания указателей, что возможно только для программ, написанных на чистом Си и с некоторыми ограничениями на работу с указателями.

В работе [20] предлагается проводить рандомизацию через фиксированные интервалы времени и параллельно с работой основной программы из отдельного потока. Рандомизация производится на уровне функций с помощью таблицы символов, которую компоновщик оставляет в исполняемом файле, и дизассемблированного при ее помощи кода. Вместо отслеживания указателей происходит изменение их семантики. Указатель хранит индекс адреса в глобальной таблице. Адреса возврата на стеке шифруются уникальным для каждой функции ключом, который меняется при перерандомизации. Вызовы функций реализуются через относительные переходы, то есть программа статически связывается в памяти при запуске. Это устраняет GOT как возможный источник утечки адресов, но предотвращает использование dlopen и исключений Си++. Данный метод замедляет и запуск программ (из-за дизассемблирования и связывания), и выполнение (из-за шифрования адресов возврата). На SPEC2006 с единственной рандомизацией при запуске замедление составляет в среднем 8 %. При перерандомизации каждые 200 мс замедление составляет 13.5 % в среднем. Такое сравнительно небольшое замедление достигается благодаря тому, что рандомизация выполняется параллельно.

3. Предлагаемый метод и его реализация

В рамках данной работы предлагается реализации мелкогранулярной, с гранулярностью не крупнее функций, рандомизации адресного пространства программ при запуске. Для реализации этого подхода при сборке программ исполняемые и библиотечные файлы дополняются информацией о границах функций и релокациях (упоминаниях адресов кода или данных в программе, например, адреса одной функции в коде другой). При запуске программы системный динамический загрузчик использует эту информацию для случайного размещения отдельных функций в памяти. Данный метод требует доступа к исходному коду и процессу сборки. Рандомизация выполняется только при загрузке программы. Адресное пространство не изменяется, например, при вызове fork. Кроме того, предлагаемая рандомизация не затрагивает адресное пространство ядра.

Для реализации мелкогранулярной рандомизации на этапе запуска программы в динамический загрузчик и в инструментарий для сборки программ были внесены изменения. Рандомизация реализована для архитектуры x86-64 и операционной системы CentOS 7, использующей ELF как основной формат исполняемых и библиотечных файлов. Для минимизации потенциальных проблем совместимости в этот формат не было внесено никаких изменений. Необходимая для рандомизации информация хранится в дополнительной секции, которая игнорируется при использовании стандартного динамического загрузчика.

3.1 Хранение информации для рандомизации

Для выполнения мелкогранулярной рандомизации на уровне функций необходимо знать границы этих функций и релокации. Для хранения этой информации в ELF файле размещается дополнительная секция. В этой секции хранятся информация о релокациях, виртуальные адреса функций без рандомизации, их длина и выравнивание.

Кроме того, из соображений эффективности некоторые адреса в структуре самого ELF файла и в определенных частях программы также записаны в таблицу релокаций. К таким релокациям относятся адреса в секции eh_frame и адреса динамических символов.

Помимо дополнительной секции в сегмент NOTE добавляется запись, содержащая виртуальный адрес загрузки этой секции в память. Этот сегмент предназначен для хранения произвольной дополнительной информации, все загрузчики и инструменты просто игнорируют неизвестные им записи. Таким образом, ELF файл, собранный с поддержкой рандомизации, может быть загружен любым стандартным динамическим загрузчиком, а вся дополнительная информация будет просто проигнорирована.

3.2 Модификации динамического загрузчика

В Linux на x86-64 при запуске исполняемого ELF файла ядро операционной системы загружает в память все загружаемые сегменты из этого файла и из упомянутого в нем динамического загрузчика. Выполнение начинается с точки входа динамического загрузчика, который работает в контексте самого процесса. Он загружает все требуемые динамические библиотеки, подготавливает программу к запуску и передает управление на точку входа самой программы.

Для реализации рандомизации при запуске программы необходимо внесение изменений в динамический загрузчик. Динамический загрузчик является частью библиотеки glibc. В динамический загрузчик была добавлена функциональность, которая находит в ELF файле, загруженном в память, дополнительную секцию. С помощью найденной дополнительной секции выполняется случайное переупорядочивание функций загружаемого файла. После этого выполняется проход по списку релокаций и их исправление. Если дополнительная секция отсутствует, то рандомизация не совершается. Таким образом, модифицированный загрузчик может загружать программы и библиотеки, собранные без поддержки рандомизации.

Описанная функциональность модифицирует код программы. Код, как правило, загружается в память, для которой запрещается запись, поэтому для нее временно разрешается изменение. Если в системе присутствует система ограничения доступа (SELinux, PAX), то может потребоваться ее дополнительная настройка для разрешения такого поведения.

Дополнительная секция и выделенная во время рандомизации память освобождаются перед передачей управления программе. Таким образом, после завершения рандомизации в памяти процесса не остается никакой дополнительной информации, утечка которой могла бы раскрыть размещение функций.

3.3 Модификация инструментария сборки

Для создания дополнительной секции были внесены изменения в статический компоновщик. Информация о границах функций получалась от компилятора с помощью указания ключа командной строки -ffunction-sections. Статический компоновщик собирает информацию обо всех релокациях при их разрешении в процессе связывания и сохраняет ее в дополнительной секции. Кроме того, Некоторые типы релокаций, например, из TLS и eh_frame потребовали нетривиальной обработки, которую сложно провести вне компоновщика, что предотвратило их поддержку в схожей работе [15].

Для удобства использования статическому компоновщику был добавлен ключ командной строки, включающий поддержку рандомизации. При указании этого ключа при промежуточной сборке нескольких объектных файлов в один предотвращается слияние секций, содержащих код, и дополнительная секция не создается. При статической сборке этот ключ игнорируется. Рандомизация статически собранных файлов не поддерживается.

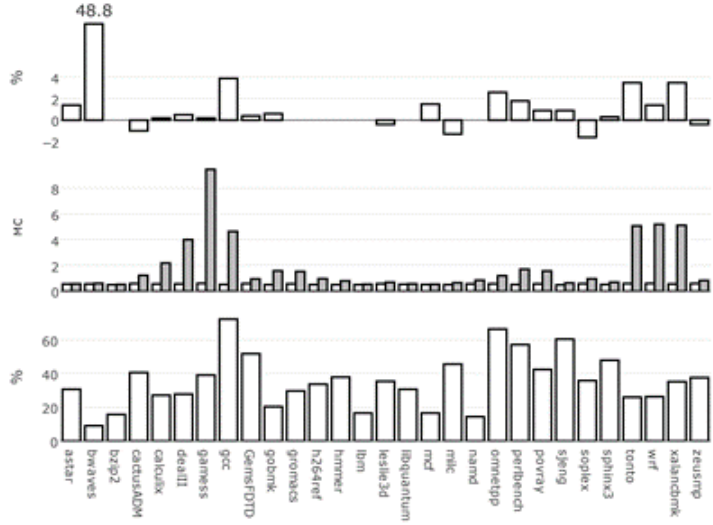


Рис. 2. Результаты тестов SPEC2006 на изменение производительности, увеличение времени запуска и увеличении размера (сверху вниз)
Fig. 2. Test results of SPEC2006 for performance, runtime startup and size increasing (from top to bottom)

4. Измерение производительности

Тестирование проводилось на системе CentOS 7. Процессор Intel Core i7-4790 (3.6 GHz), 16 ГБ оперативной памяти. Проверка корректности работы была проведена на промышленном наборе тестов SPEC2006 и минимальной сборке операционной системы CentOS 7. SPEC2006 отрабатывает корректно, почти все проверенные пакеты CentOS 7 проходят все тесты, аномалий в работе программ не наблюдается. Отдельные приложения могут модифицировать свой собственный исполняемый файл, что приводит к ошибкам при использовании рандомизации, но это необходимо исправлять на уровне самих приложений, а не инструментария, реализующего рандомизацию. Использовалась стандартная система сборки и выполнения этого набора тестов. Для одной и той же конфигурации SPEC2006 запускался со стандартными компоновщиком и загрузчиком без рандомизации и с модифицированными с включенной рандомизацией.

Время выполнения тестов. Изменение времени работы тестов приведено на рис. 2 на графике измерение производительности. Среднее геометрическое замедление составляет примерно 1.5 %. Большинство тестов показало незначительное изменение времени выполнения, за исключением bwaves. Для него замедление составило более 40 %, это объясняется тем, что для этого приложения критически важна локальность распределения кода. Некоторые из тестов по такой же причине даже показали незначительные улучшения производительности по сравнению с нерандомизированной версией программы.

Время загрузки программ. Время загрузки измерялось путем многократного запуска программ с прекращением исполнения перед передачей управления на точку входа. На рис. 2 приведены результаты измерений. Несмотря на достаточно большое относительное замедление процесса загрузки программ, в отдельных случаях эта величина достигает 10 раз, время загрузки остается пренебрежимо малым по сравнению с типичным временем работы нетривиальных программ. Самый медленный запуск программы из набора тестов при замедлении 15.3 раза занимает всего 9.5 миллисекунд.

Размер исполняемого файла. На рис. 2 приводится график изменения размера исполняемого файла при рандомизации. В среднем размер исполняемого файла увеличивается на 50 %, максимальная величина – 73 %. Следует отметить, что хранимая в исполняемом файле дополнительная информация занимает место только на диске. После завершения рандомизации, занимаемая ей память, освобождается. С учетом размера современных дисков и типичного суммарного размера исполняемых и библиотечных файлов в системе (3 ГБ на тестовой системе) – это не является проблемой.

5. Противодействие эксплуатации уязвимостей

Актуальным вопросом для данной работы является исследование эффективности реализованного метода по способности противодействовать эксплуатации уязвимостей. Для ее оценки существует два принципиальных подхода. Первый, который используется в большинстве статей аналогичной тематики, заключается в теоретико-логическом обосновании эффективности. Второй метод заключается в экспериментальной проверке реализованных методов защиты с приведением результатов статистических измерений. Наиболее полное исследование второго типа опубликовано в работе [22]. В данной работе прибегнем к второму методу оценки эффективности реализованного метода.

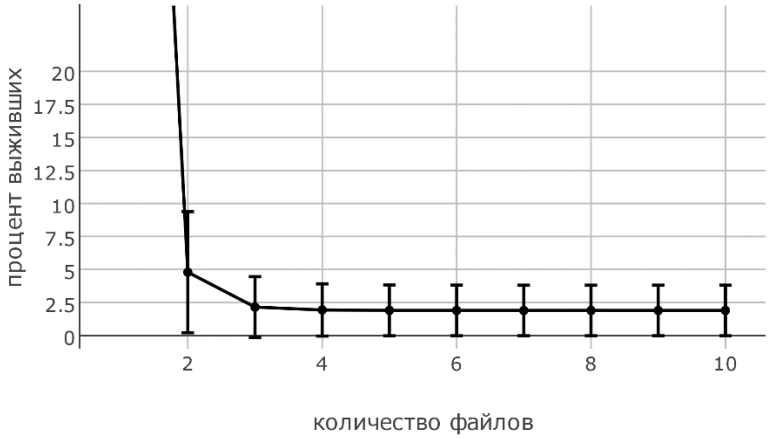


Рис. 3. Среднее относительное количество выживших гаджетов в зависимости от размера популяции
Fig. 3. The average percentage of survived gadgets depending on the size of population

Предполагаемый сценарий атаки на приложение заключается в следующем: атакующий имеет в своем распоряжении исполняемый файл приложения; из гаджетов этого файла строится ROP цепочка; данную цепочку пытаются выполнить на других экземплярах приложения.

Экспериментальная оценка производилась с помощью нескольких серий измерений. Исследуемым набором приложений были исполняемые файлы из стандартной минимальной установки CentOS 7, располагаемые в директориях /usr/bin и /usr/sbin. Для корректности полученных результатов брались только файлы, собранные без поддержки позиционно-независимого кода. Тестовый набор состоит из 487 файлов.

Для исследования необходимо было сохранять состояние адресного пространства приложения после момента его перемешивания. Это было

сделано с помощью сохранения дампа памяти процесса. Дамп памяти сохраняется в ELF формате, где каждому сегменту создается своя отдельная секция. Однако по умолчанию сохранять секции с кодом в дамп памяти не требуется, поскольку эта секция остается неизменной и всегда доступна в файле на диске. Для записи всех сегментов рабочей памяти процесса были внесены изменения в алгоритм сохранения дампов памяти отладчика gdb. С его помощью для каждого файла из тестового набора было получено по 10 дампов памяти. Собранные дампы памяти образовали вместе с оригинальными файлами тестовую популяцию над которой производились все эксперименты с помощью классификатора гаджетов [23].

Поиск и классификация гаджетов. Поиск гаджетов осуществляется при помощи инструмента с открытым исходным кодом ROPgadget [24]. Инструмент находит инструкции передачи управления в исполняемых секциях программы и дизассемблирует несколько байт, предшествующих найденным инструкциям. Все успешно дизассемблированные блоки инструкций добавляются в список потенциальных гаджетов.

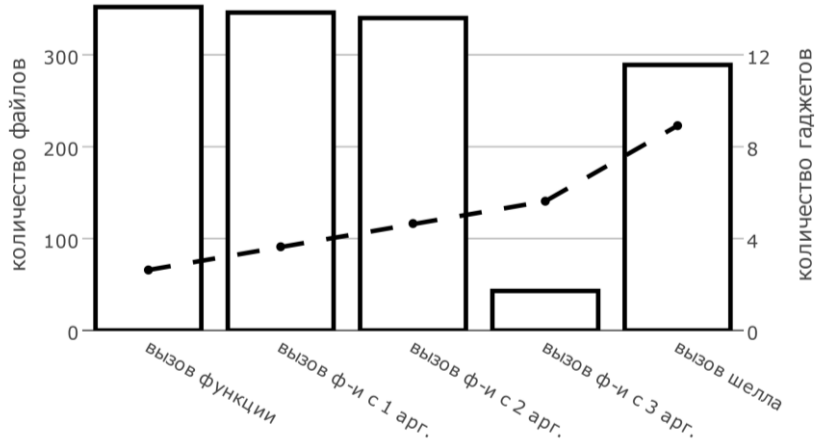


Рис. 4. Количество успешно созданных цепочек и их средний размер для различных модельных примеров ROP цепочек
Fig. 4. The number of successfully created chains and their average size for different model examples of ROP chains

Полученные кандидаты в гаджеты классифицируются согласно семантическим типам, описанным в статье [23]. Инструкции гаджета транслируются в промежуточное представление, которое в дальнейшем интерпретируется. Во время интерпретации отслеживаются обращения к регистрам и памяти на чтение и запись. Начальные считанные значения генерируются случайным образом. В результате интерпретации получаются начальные и конечные значения регистров и памяти, которые ограничивают

список возможных семантических типов, которым удовлетворяет гаджет. После этого производится еще несколько запусков процесса интерпретации с различными входными данными. В результате остаются только те типы, которым удовлетворял гаджет на всех запусках.

Классифицированные гаджеты сохраняются в базу данных вместе с дополнительной информацией о типе гаджета, об его адресе, о параметрах гаджетов и побочных эффектах. С помощью полученных баз данных возможно узнать, существует ли в данном файле на заданном адресе гаджет, какие параметры и побочные эффекты у гаджета на заданном адресе и так далее.

Оценка количества выживших гаджетов. Введем определение термину выживший гаджет. Пусть имеется некоторая популяция разных версий дампов памяти одной и той же программы. Тогда будем называть для нее выжившим гаджетом такой гаджет исходного исполняемого файла, который находится по одному и тому же адресу в каждом экземпляре популяции. Выжившие гаджеты важны для исследования по причине того, что составленная из таких гаджетов ROP цепочка работоспособна на каждом экземпляре в популяции.

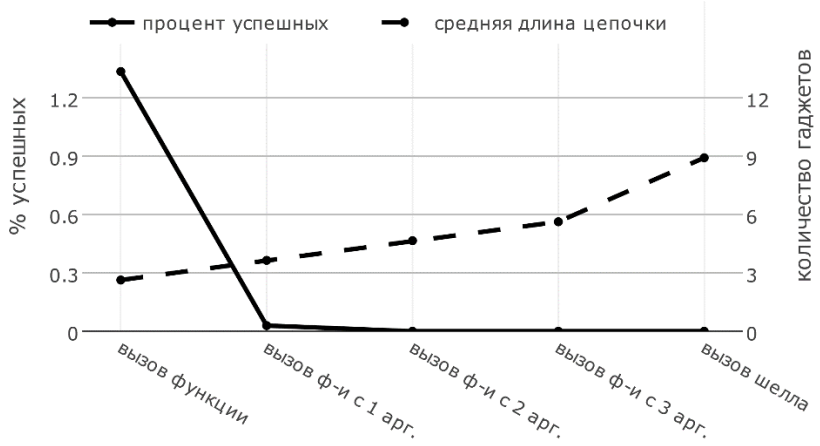


Рис. 5. Усредненная относительная работоспособность оригинальной ROP цепочки для других файлов популяции

Fig. 5. Average success rate of the original ROP chain for other population files

Измерение количества выживших гаджетов производилось путем обращения к базам данных гаджетов, полученных с помощью классификатора гаджетов. Зависимость количества выживших гаджетов от размера популяции представлена на рис. 3. На нем представлена кривая, отражающая среднее арифметическое значение доли выживших гаджетов по всем программам из тестового набора, кроме того у каждой точки отложено среднеквадратическое отклонение от среднего значения. Характер формы представленных кривых

напоминает экспоненциально убывающую последовательность с некоторым константным смещением по оси абсцисс вверх (примерно 2 %). Это остаточное количество выживших гаджетов, наблюдаемое независимо от размера популяции, объясняется следующим замечанием: в исполняемом сегменте кроме кода функций, местоположение которых меняется, находятся также вспомогательные секции: таблица связывания процедур PLT, INIT, FINI и другие. Они остаются неизменными, и гаджеты внутри них всегда являются выжившими.

Оценка работоспособности ROP цепочек. Важно оценить работоспособность ROP цепочек, построенных по оригинальному исполняемому файлу, для других экземпляров популяции. Размер ROP цепочек может варьироваться от нескольких гаджетов до десятков и более. Возьмем несколько примеров цепочек, возрастающего размера и сложности: вызов функции без аргументов, вызов функции с 1 аргументом, вызов функции с 2 аргументами, вызов функции с 3 аргументами и вызов оболочки командной строки.

Перечисленные примеры цепочек составляются по базам данных гаджетов. На рис. 4 приведены результаты построения ROP цепочек для тестового набора исполняемых файлов. Столбцы значений отвечают количеству файлов из тестового набора, для которых построение конкретного примера оказалось возможным. Пунктирная кривая показывает количество гаджетов в ROP цепочке для каждого примера. Затем проверяется работоспособность построенных цепочек для экземпляров в популяции соответствующего исходного файла. Процентное отношение работоспособных файлов к размеру популяции отображено на рис. 5. Из данного графика видно, что процент успешности резко падает с увеличением длины цепочки и для нетривиальных цепочек стремится к нулю. Стоит отметить, что относительно небольшое значение процента успешности для вызова функции без аргументов объясняется тем, что для реализации такой цепочки зачастую достаточно гаджетов из неизменяемых секций (PLT, INIT, FINI).

6. Заключение

В данной работе представлена реализация мелкогранулярной рандомизации адресного пространства программ, с гранулярностью на уровне функций, при их запуске. Функции исполняемых файлов и библиотек размещаются при их загрузке в случайном порядке. Это увеличивает энтропию рандомизации адресного пространства по сравнению с ASLR, что усложняет построение и проведение ROP атак на защищенные таким образом программы. Была экспериментально оценена эффективность противодействия эксплуатации методом ROP с помощью двух метрик: процент выживших гаджетов и оценка работоспособности примеров ROP цепочек. Приводимая в статье реализация показала свою пригодность для применения в масштабах всей системы. Кроме

того, она лишена проблем совместимости: исполняемый файл, собранный с поддержкой рандомизации, может быть загружен стандартным динамическим загрузчиком, а рандомизирующий динамический загрузчик может загружать обычные исполняемые файлы формата ELF без дополнительной секции. В ходе тестирования среднее замедление времени работы тестового набора SPEC2006 составило 1.5 %. Время загрузки программ остается пренебрежимо малым по сравнению с временем их работы.

У разработанной реализации на данный момент имеются незначительные недостатки, которые можно исправить в будущем. Самым существенным недостатком является несоответствие отладочной информации, что затрудняет отладку рандомизированного кода. В дальнейшем необходимо генерировать актуализированную отладочную информацию для исполняемых файлов в режиме отладки. Реализованная рандомизация проводится на уровне функций. Поддержка более мелкой гранулярности позволит увеличить энтропию, а значит, усилить защиту. Также имеет смысл реализовать рандомизацию размещения коротких функций с учетом связей между ними. Близкое размещение функций, часто вызывающих друг друга, может повысить производительность отдельных программ. Кроме того, результаты тестирования эффективности защиты показывают, что в дополнительной защите нуждаются также секции исполняемого файла (PLT, INIT, FINI).

Список литературы

[1]. CVE Details website: vulnerabilities by date. По состояние на 10.04.2017 г. http://www.cvedetails.com/browse-by-date.php

[2]. R. Roemer, E. Bbuchanan, H. Shacham, S. Savage. Return-oriented programming: Systems, languages, and applications. ACM Trans. Inf. Syst. Secur., vol. 15, no. 1, 2012, pp. 2:1–2:34.

[3]. A. Sadeghi, S. Niksefat, M. Rostamipour, Pure-Call Oriented Programming (PCOP): chaining the gadgets using call instructions. Journal of Computer Virology and Hacking Techniques, no. 434, 2017, pp. 1-18

[4]. T. Bletsch, X. Jiang, V. Freeh, W. Liang, Zh. Liang. Jump-oriented Programming: A New Class of Code-reuse Attack. Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, 2011, pp. 30-40.

[5]. H. Hu, Sh. Shinde, S. Adrian, Z.L. Chua, P. Saxena, Zh. Liang. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. IEEE Symposium on Security and Privacy (SP), 2016, pp. 969-986.

[6]. H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). Proceedings of the 14th ACM conference on Computer and communications security, 2007, pp 552-561.

[7]. A. Bittau, A. Belay, A. Mashtizadeh et al. Hacking blind. Proceedings of the 2014 IEEE Symposium on Security and Privacy, 2014, pp. 227–242.

[8]. M. Abadi, M. Budiu, ´U. Erlingsson, J. Ligatti. Control-flow integrity principles, implementations, and applications. ACM Trans. Inf. Syst. Secur, vol. 13, no. 1, 2009, pp. 4:1–4:40.

[9]. A.J. Mashtizadeh, A. Bittau, D. Boneh, D. Mazi`eres, Ccfi: Cryptographically enforced control flow integrity. Proceedings of the Sixth ACM SIGSAC Conference on Computer and Communications Security, 2015, pp. 941–951.

[10]. N. Christoulakis, G. Christou, E. Athanasopoulos, S. Ioannidis. Hcfi: Hardware-enforced control-flow integrity. Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, 2016, pp. 38–49.

[11]. N. Carlini, A. Barresi, M. Payer et al. Control-flow bending: On the effectiveness of control-flow integrity. Proceedings of the 24th USENIX Conference on Security Symposium, 2015, pp. 161–176.

[12]. K. Lu, S. N¨urnberger, M. Backes, W. Lee. How to make ASLR win the clone wars: Runtime re-randomization. 23nd Annual Network and Distributed System Security Symposium, 2016.

[13]. А.Р. Нурмухаметов, Ш.Ф. Курмангалеев, В.В. Каушан, С.С. Гайсарян. Применение компиляторных преобразований для противодействия эксплуатации уязвимостей программного обеспечения. Труды ИСП РАН, том 26, вып. 3, 2014, стр. 113-126. DOI: 10.15514/ISPRAS-2014-26(3)-6

[14]. A. Gupta, S. Kerr, M. Kirkpatrick, E. Bertino. Marlin: A fine grained randomization approach to defend against ROP attacks. Network and System Security, 7 th International Conference, 2013.

[15]. M. Conti, S. Crane, T. Frassetto et al. Selfrando: Securing the tor browser against de-anonymization exploits. PoPETs, no. 4, 2016, pp. 454–469.

[16]. L. Davi, A. Dmitrienko, S. N¨urnberger, A. Sadeghi. Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and ARM, 8th ACM Symposium on Information, Computer and Communications Security, 2013.

[17]. M. Backes, S. Nurberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. Proceedings of the 23rd USENIX Security Symposium, 2014, pp. 433–447.

[18]. S. Crane, A. Homescu, P. Larsen. Code randomization: Haven't we solved this problem yet? Cybersecurity Development (SecDev), IEEE, 2016.

[19]. D. Bigelow, T. Hobson, R. Rudd et al. Timely rerandomization for mitigating memory disclosures, Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, 2015, pp. 268–279.

[20]. D. Williams-King, G. Gobieski, K. Williams-King et al. Shuffler: Fast and deployable continuous code re-randomization. Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, 2016, pp. 367–382.

[21]. M. Payer. Too much PIE is bad for performance. Technical report.

[22]. J. Coffman, C. Wellons, C.C. Wellons. ROP Gadget Prevalence and Survival under Compiler-based Binary Diversification Schemes. Proceedings of the 2016 ACM Workshop on Software PROtection, 2016, pp. 15-26.

[23]. А.В. Вишняков. Классификация ROP гаджетов. Труды ИСП РАН, том 28, вып. 6, 2016 г., стр. 27-36. DOI: 10.15514/ISPRAS-2016-28(6)-2

[24]. ROPgadget. По состоянию на 16.10.2017 г. https://github.com/JonathanSalwan/ROPgadget

Fine-grained address space layout randomization on program load

[1] *A.R. Nurmukhametov <oleshka@ispras.ru>*
[1] *E.A. Zhabotinskiy <ezhabotinskiy@ispras.ru>*
[1] *Sh.F. Kurmangaleev <kursh@ispras.ru>*
[1,2,3,4] *S.S. Gaissaryan <ssg@ispras.ru>*
[1] *A.V. Vishnyakov <vishnya@ispras.ru>*
[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia*
[2] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia.*
[3] *Moscow Institute of Physics and Technology (State University), 9 Institutskiy per., Dolgoprudny, Moscow Region, 141700, Russia*
[4] *National Research University Higher School of Economics (HSE) 11 Myasnitskaya Ulitsa, Moscow, 101000, Russia*

Abstract. Program vulnerabilities are a serious security threat. It is important to develop defenses preventing their exploitation, especially with a rapid increase of ROP attacks. State of the art defenses have some drawbacks that can be used by attackers. In this paper we propose fine-grained address space layout randomization on program load that is able to protect from such kind of attacks. During the static linking stage executable and library files are supplemented with information about function boundaries and relocations. A system dynamic linker/loader uses this information to perform functions permutation. The proposed method was implemented for 64-bit programs on CentOS 7 operating system. The implemented method has shown good resistance to ROP attacks based on two metrics: the number of survived gadgets and the exploitability estimation of ROP chain examples. The implementation presented in this article is applicable across the entire operating system and has shown 1.5 % time overhead. The working capacity of proposed approach was demonstrated on real programs. The further research can cover forking randomization and finer granularity than on the function level. It also makes sense to implement the randomization of short functions placement, taking into account the relationships between them. The close arrangement of functions that often call each other can improve the performance of individual programs.

Keywords: address space layout randomization; diversification, ASLR; ROP.

DOI: 10.15514/ISPRAS-2017-29(6)-9

For citation: Nurmukhametov A.R., Zhabotinskiy E.A., Kurmangaleev Sh. F., Gaissaryan S.S., Vishnyakov A.V. Fine-grained address space layout randomization on program load. Trudy ISP RAN/Proc. ISP RAS, 2017, vol. 29, issue 6, pp. 163-182 (in Russian). DOI: 10.15514/ISPRAS-2017-29(6)-9

References

[1]. CVE Details website: vulnerabilities by date. Accessed 10.04.2017. http://www.cvedetails.com/browse-by-date.php

[2]. R. Roemer, E. Bbuchanan, H. Shacham, S. Savage. Return-oriented programming: Systems, languages, and applications. ACM Trans. Inf. Syst. Secur., vol. 15, no. 1, 2012, pp. 2:1–2:34.

[3]. A. Sadeghi, S. Niksefat, M. Rostamipour, Pure-Call Oriented Programming (PCOP): chaining the gadgets using call instructions. Journal of Computer Virology and Hacking Techniques, no. 434, 2017, pp. 1-18

[4]. T. Bletsch, X. Jiang, V. Freeh, W. Liang, Zh. Liang. Jump-oriented Programming: A New Class of Code-reuse Attack. Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, 2011, pp. 30-40.

[5]. H. Hu, Sh. Shinde, S. Adrian, Z.L. Chua, P. Saxena, Zh. Liang. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. IEEE Symposium on Security and Privacy (SP), 2016, pp. 969-986.

[6]. H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). Proceedings of the 14th ACM conference on Computer and communications security, 2007, pp 552-561.

[7]. A. Bittau, A. Belay, A. Mashtizadeh et al. Hacking blind. Proceedings of the 2014 IEEE Symposium on Security and Privacy, 2014, pp. 227–242.

[8]. M. Abadi, M. Budiu, ´U. Erlingsson, J. Ligatti. Control-flow integrity principles, implementations, and applications. ACM Trans. Inf. Syst. Secur, vol. 13, no. 1, 2009, pp. 4:1–4:40.

[9]. A.J. Mashtizadeh, A. Bittau, D. Boneh, D. Mazi`eres, Ccfi: Cryptographically enforced control flow integrity. Proceedings of the Sixth ACM SIGSAC Conference on Computer and Communications Security, 2015, pp. 941–951.

[10]. N. Christoulakis, G. Christou, E. Athanasopoulos, S. Ioannidis. Hcfi: Hardware-enforced control-flow integrity. Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, 2016, pp. 38–49.

[11]. N. Carlini, A. Barresi, M. Payer et al. Control-flow bending: On the effectiveness of control-flow integrity. Proceedings of the 24th USENIX Conference on Security Symposium, 2015, pp. 161–176.

[12]. K. Lu, S. N¨urnberger, M. Backes, W. Lee. How to make ASLR win the clone wars: Runtime re-randomization. 23nd Annual Network and Distributed System Security Symposium, 2016.

[13]. A. Nurmukhametov, Sh. Kurmangaleev, V. Kaushan, S. Gaissaryan. Application of compiler transformations against software vulnerabilities exploitation. Programming and Computer Software, vol. 41, no. 4, 2015, pp. 231-236. DOI: 10.1134/S0361768815040052

[14]. A. Gupta, S. Kerr, M. Kirkpatrick, E. Bertino. Marlin: A fine grained randomization approach to defend against ROP attacks. Network and System Security, 7 th International Conference, 2013.

[15]. M. Conti, S. Crane, T. Frassetto et al. Selfrando: Securing the tor browser against de-anonymization exploits. PoPETs, no. 4, 2016, pp. 454–469.

[16]. L. Davi, A. Dmitrienko, S. N¨urnberger, A. Sadeghi. Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and ARM, 8th ACM Symposium on Information, Computer and Communications Security, 2013.

[17]. M. Backes, S. Nurberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. Proceedings of the 23rd USENIX Security Symposium, 2014, pp. 433–447.
[18]. S. Crane, A. Homescu, P. Larsen. Code randomization: Haven't we solved this problem yet? Cybersecurity Development (SecDev), IEEE, 2016.
[19]. D. Bigelow, T. Hobson, R. Rudd et al. Timely rerandomization for mitigating memory disclosures, Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, 2015, pp. 268–279.
[20]. D. Williams-King, G. Gobieski, K. Williams-King et al. Shuffler: Fast and deployable continuous code re-randomization. Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, 2016, pp. 367–382.
[21]. M. Payer. Too much PIE is bad for performance. Technical report.
[22]. J. Coffman, C. Wellons, C.C. Wellons. ROP Gadget Prevalence and Survival under Compiler-based Binary Diversification Schemes. Proceedings of the 2016 ACM Workshop on Software PROtection, 2016, pp. 15-26.
[23]. Vishnyakov A.V. Classification of ROP gadgets. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 6, 2016, pp. 27-36 (in Russian). DOI: 10.15514/ISPRAS-2016-28(6)-2
[24]. ROPgadget. https://github.com/JonathanSalwan/ROPgadget. Accessed 16.10.2017

Критерий существования бесконфликтного расписания для системы строго периодических задач[1]

[1] *С.А. Зеленова <sophia@ispras.ru>*
[1,2] *С.В. Зеленов <zelenov@ispras.ru>*
[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*
[2] *Национальный исследовательский университет Высшая школа экономики, 101000, Москва, ул. Мясницкая, д. 20*

Аннотация. В критических системах выполнение жестких требований по времени взаимодействия между задачами обеспечивается строгой периодичностью запуска задач, когда каждая задача стартует через равные промежутки времени. При планировании строго периодических задач с прерываниями наиболее трудным этапом является выбор начальных стартовых точек задач. В настоящей работе предлагается новый подход к анализу расписаний, основанный на изучении раскрасок графов периодов задач и на решении систем линейных сравнений. Основным результатом является критерий существования бесконфликтного расписания для произвольного количества строго периодических задач с прерываниями на одном процессоре. Критерий позволяет либо направленно найти стартовые точки, либо быстро установить, что расписание построить невозможно.

Ключевые слова: системы реального времени; строго периодическая задача; планирование; раскраска графа; система линейных сравнений.

DOI: 10.15514/ISPRAS-2017-29(6)-10

Для цитирования: Зеленова С.А., Зеленов С.В. Критерий существования бесконфликтного расписания для системы строго периодических задач. Труды ИСП РАН, том 29, вып. 6, 2017 г., стр. 183-202. DOI: 10.15514/ISPRAS-2017-29(6)-10

1. Введение

В жестких системах реального времени, таких как автомобильная электроника, авионика, мобильная робототехника, телекоммуникация, и т.п.,

[1] Поддержано грантом РФФИ, Договор № 17-01-00504\17

жизненно важно, во-первых, чтобы в каждый момент времени обрабатываемая информация была как можно более актуальна, а во-вторых, чтобы система вовремя реагировала на поступающие входные данные, причем нарушение этих требований приводит к катастрофическим последствиям. В подобных системах требуется точная регулировка взаимодействия датчиков (sensors), приводов (actuators) и функций управления обратной связью (т.е. функций обработки информации и выдачи управляющих воздействий). Для этого такие задачи запускаются строго периодически, т.е. каждая из них должна периодически стартовать через равные промежутки времени.

Несколько десятилетий назад для повышения надежности каждому процессу отводилось свое устройство. Однако к настоящему времени размеры систем настолько выросли, что подобная организация архитектуры привела бы к непомерному весу и энергопотреблению системы. Выходом здесь является разделение ресурсов, т.е. выполнение нескольких задач на одном процессоре. В связи с этим большую актуальность получает задача составления совместного расписания для нескольких процессов, выполняемых на одном или нескольких устройствах.

Планирование задач может быть динамическим (т.е. осуществляемым в реальном масштабе времени в ходе работы системы) или статическим (когда расписание составляется до запуска системы). Выполнение задач с точки зрения планирования бывает двух видов: с прерываниями (preemptive), когда выполнение задачи разрешается на время отложить для запуска другой задачи, и без прерываний (non-preemptive), когда этого делать не разрешается.

В данной статье мы затронем вопросы статического планирования строго периодических задач с прерываниями.

Строго периодическая задача задается двумя параметрами: период (время между двумя последовательными стартами задачи) и длительность (время, которое необходимо задаче для ее выполнения в пределах одного периода).

В отличие от классического случая, когда периодичность не является строгой [8], [9], построение расписания для набора строго периодических задач с прерываниями разбивается на два этапа:

- построение расписания для начальных точек данной системы периодических процессов.
- распределение остальных точек в пределах заданных периодов.

По определению строго периодической задачи все ее точки старта образуют арифметическую прогрессию. Поскольку две задачи не могут стартовать одновременно, арифметические прогрессии для разных задач не должны конфликтовать, т.е. не должны иметь общих точек. Достаточно давно известно достаточное условие бесконфликтности, состоящее в том, что начальные точки должны иметь разные остатки по модулю наибольшего общего делителя (НОД) соответствующих периодов (см., например, [11]), в частности, взаимная простота периодов немедленно влечет невозможность построения

расписания. В случае, когда задач всего две, это условие является критерием, позволяющим эффективно построить расписание. Однако, для случая произвольного количества задач подобный критерий пока не найден.

Имеющиеся подходы к проблеме планирования строго периодических задач не используют в должной мере взаимосвязи между периодами (см., например, [5], [6], [7], [10], [12]), что приводит к существенному увеличению времени поиска. Существующие алгоритмы решают проблему поиска стартовых точек либо с применением грубых эвристик [12], либо перебором с различными оптимизациями [5], [6]. Однако, переборное решение обладает тем недостатком, что в случае, если расписание построить невозможно, требуется проанализировать все возможные варианты.

В настоящей работе мы решаем проблему первого этапа — поиска системы начальных точек без конфликтов. В такой постановке мы пренебрегаем длительностью задач. Используя результаты теории графов [1], [2], [3] и теории чисел [4], мы предлагаем новый подход к анализу построения расписания для строго периодических задач, основанный на изучении структуры групп их периодов. Основным результатом является критерий существования бесконфликтного расписания для произвольного количества задач. Анализируя периоды с помощью этого критерия, можно либо направленно найти стартовые точки, либо быстро установить, что расписание построить невозможно.

Дальнейшее изложение материала статьи построено так.

В разделе 2 вводятся основные определения и приводятся элементарные условия существования бесконфликтного расписания для строго периодических задач.

В разделе 3 исследуется взаимосвязь проблемы построения бесконфликтного расписания для произвольного количества строго периодических задач и проблемы раскраски некоторых графов специального вида, построенных на основе значений периодов этих задач.

В разделе 4 вводится понятие графа делимости и исследуется вопросы совместимости раскрасок графов периодов с точки зрения возможности построения бесконфликтного расписания. Доказывается теорема, устанавливающая необходимые и достаточные условия раскраски графов для существования бесконфликтного расписания.

В разделе 5 производится обзор существующих подходов к построению расписаний для строго периодических задач.

В разделе 6 подводятся итоги статьи и намечаются пути дальнейших исследований.

2. Необходимое условие существования расписания

Задача называется *строго периодической*, если

- задача выполняется повторно с некоторой периодичностью;

- повторные запуски задачи совершаются строго через фиксированный промежуток времени, называемый *периодом*.

Длительностью строго периодической задачи называется время, отводимое на ее выполнение в рамках одного периода.

Пусть дано множество строго периодических задач, каждая из которых имеет единичную длительность. Требуется составить расписание выполнения данных задач на одном процессоре.

Введем обозначение для множества периодов: $P = \{p_i | i = 1...N\}$. Заметим, что множество P является мультимножеством, т.е. периоды, относящиеся к разным процессам в нем не сливаются, а рассматриваются как отдельные сущности, даже если они равны.

Не теряя общности можно предположить, что начальные точки всех процессов отстоят от общей точки отсчета не дальше периода соответствующего процесса. Здесь точка отсчета соответствует нулю, а каждая начальная точка соответствует какому-либо целому неотрицательному числу.

Атомарным расписанием назовем все точки одного процесса с заданным началом и заданным периодом. Атомарные расписания *конфликтуют*, когда они содержат одинаковые точки, т.е. их пересечение непусто. *Конфликтным расписанием* назовем расписание, в котором есть два конфликтующих атомарных расписания.

Утверждение 1. Если периоды p_1 и p_2 двух процессов взаимно просты, то любые расписания для этой пары процессов являются конфликтными.

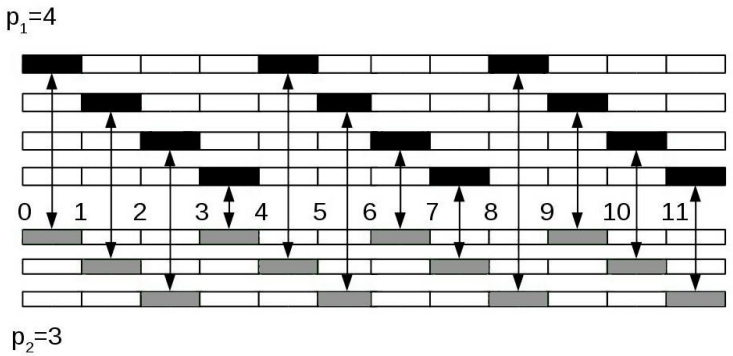


Рис. 1. Пример пары процессов, для которых не существует бесконфликтного расписания. Для каждого процесса указаны всевозможные варианты расписаний. Стрелками указаны конфликтующие позиции
Fig. 1. Example of two tasks that cannot be scheduled without conflicts. For each task, all possible variants of schedule are shown. Arrows point to conflicts.

Доказательство. Конфликтность расписания можно записать в виде равенства $t_1 + n_1 p_1 = t_2 + n_2 p_2$, где t_1, t_2 — начальные точки, а n_1, n_2 — неотрицательные целые числа. Из теории чисел известно, что для взаимно

простых чисел существует целочисленная линейная комбинация равная их наибольшему общему делителю, то есть единице: существуют целые числа k, m такие, что $kp_1 + mp_2 = 1$. Тогда $k(t_2 - t_1)p_1 + m(t_2 - t_1)p_2 = t_2 - t_1$, то есть $t_1 + k(t_2 - t_1)p_1 = t_2 + m(t_1 - t_2)p_2$. Взяв достаточно большое N, получаем $t_1 + (k(t_2 - t_1) + Np_2)p_1 = t_2 + (m(t_1 - t_2) + Np_1)p_2$, где коэффициенты $k(t_2 - t_1) + Np_2$ и $m(t_1 - t_2) + Np_1$ — положительные числа, то есть атомарные расписания имеют общую точку, а, значит, конфликтуют.▷

Утверждение 2. Трансформации расписания в виде сдвига или деления точек расписания на общий множитель не меняют свойства конфликтности или бесконфликтности расписания.

Доказательство. Для сдвига утверждение очевидно. Для случая деления точек на общий множитель: две различные точки дадут разные результаты при делении на общий множитель, так что свойство бесконфликтности или конфликтности сохранится для трансформированного расписания. ▷

Утверждение 3. Пусть p_1 и p_2 — периоды двух задач и пусть их наибольший общий делитель равен d. Тогда, если t_1, t_2 — начальные точки этих двух процессов и атомарные расписания не конфликтуют, то t_1 и t_2 не могут иметь одинаковые остатки по модулю d.

Доказательство. Пусть $t_1 \equiv t_2 (mod\ d)$ и $t_1 < t_2$, тогда можно трансформировать расписание следующим образом: сдвинем точку отсчета в t_1 и произведем соответствующую перенумерацию (т.е. $t_{1'} = 0$, $t_{2'} = t_2 - t_1$). Все точки нового расписания делятся на d, так что можно произвести вторую трансформацию — поделить все точки расписания на d. При этом новые периоды $p_{1'} = p_1 / d$, $p_{2'} = p_2 / d$ окажутся взаимно простыми, т.к. d является наибольшим общим делителем p_1 и p_2. И, таким образом, согласно утверждению 1, трансформированное расписание будет конфликтным, что означает (согласно утверждению 2) конфликтность первоначального расписания.▷

Утверждение 4. Пусть p_1 и p_2 — периоды двух процессов и пусть их наибольший общий делитель равен d. Тогда, если t_1, t_2 — начальные точки этих двух процессов и t_1 и t_2 имеют разные остатки по модулю d, то соответствующие атомарные расписания не конфликтуют.

Доказательство. Пусть $t_1 = dk_1 + r_1$ и $t_2 = dk_2 + r_2$, $r_1 \neq r_2$. Тогда наличие общей точки у атомарных расписаний можно записать в виде условия: $t_1 + n_1p_1 = t_2 + n_2p_2$. Однако, $t_1 + n_1p_1 = r_1 + dk_1 + n_1p_1 \equiv r_1 (mod\ d)$, а $t_2 + n_2p_2 = r_2 + dk_2 + n_2p_2 \equiv r_2 (mod\ d)$, отсюда, ввиду того, что $r_1 \neq r_2$, получаем невозможность указанного равенства, что и требовалось доказать.▷

Следствие 1 (необходимое и достаточное условие бесконфликтности расписания). Расписание является бесконфликтным тогда и только тогда, когда в заданной системе задач для любых двух процессов с периодами p_1, p_2 начальные точки t_1, t_2 этих двух процессов имеют разные остатки по модулю наибольшего общего делителя периодов p_1, p_2.

Доказательство. Согласно утверждению 4, выполнение условия означает бесконфликтность любой пары атомарных расписаний, что влечет бесконфликтность общего расписания. С другой стороны, в бесконфликтном расписании любая пара атомарных расписаний бесконфликтна, т.е., согласно утверждению 3, начальные точки t_1, t_2 соответствующих процессов не могут иметь одинаковые остатки по модулю наибольшего общего делителя периодов.▷

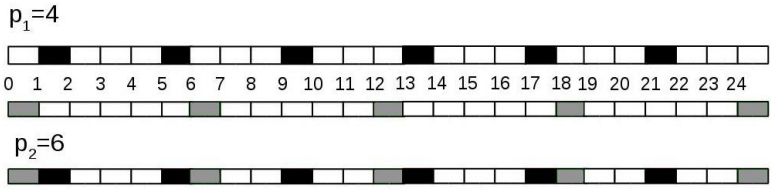


Рис. 2. Пример бесконфликтного расписания для пары процессов, у которых периоды имеют НОД не равный единице. Внизу указано общее расписание.
Fig. 2. Example of non-conflict schedule for two tasks with GCD ≠ 1. Bottom line shows the schedule itself.

Следствие 2 (необходимое условие существования бесконфликтного расписания). Если для подсистемы периодов p_i где $i = 1..k$ существует такое d, что $(p_i, p_j) = d$ для всех $i,j = 1..k$ и при этом $k > d$, то бесконфликтное расписание составить нельзя.

Доказательство. В силу следствия 1, начальные точки $t_1, \ldots, t_k$ процессов с периодами $p_1, \ldots, p_k$ должны иметь попарно различные остатки при делении на d, но таких остатков существует только $d < k$, так что, согласно известному принципу Дирихле, в любом расписании будут две начальные точки t_m, t_s с одинаковыми остатками по модулю d, стало быть такое расписание будет конфликтным.▷

3. Задача существования бесконфликтного расписания и ее сложность

Рассмотрим полный граф G вершинами которого являются периоды задач $p \in P$, а ребро, соединяющее вершину p_1 с вершиной p_2 помечено наибольшим общим делителем периодов $d = (p_1, p_2)$. Можно выделить в графе G подграф G_d, содержащий все ребра, помеченные фиксированным числом d, и вершины, которые соединены такими ребрами.

Правильной раскраской неориентированного графа будем называть раскраску, при которой две вершины, соединенные ребром, покрашены в разные цвета. Напомним, что *хроматическим числом* графа называется минимальное число красок, для которого существует правильная раскраска.

Утверждение 5. Если хроматическое число подграфа G_d больше d, то любое расписание для данной системы задач будет конфликтным.

Доказательство. Пусть вершины, входящие в подграф G_d, отвечают периодам p_1, ... , p_k и пусть заданы начальные точки t_1, ... , t_k для задач с этими периодами. Тогда можно разбить множество t_1, ... , t_k на классы по модулю d. Таких классов всего d . Сопоставим каждому классу свою краску. Бесконфликтность расписания (согласно следствию 1) должна означать, что полученная раскраска правильная, т.е. каждое ребро в графе G_d соединяет вершины разного цвета, но, поскольку по условию хроматическое число G_d больше d, т.е. не существует правильных раскрасок с числом красок меньше или равном d, то любая раскраска в d цветов окажется неправильной, а расписание — конфликтным.▷

Как нетрудно видеть, утверждение 5 является обобщением следствия 2.

Теперь зададимся следующим вопросом: какого рода графы могут встречаться в качестве подграфов G_d. Следующая теорема говорит о том, что подграф G_d может быть любым.

Теорема 1. Пусть дан неориентированный граф с n вершинами и произвольное натуральное число d. Можно разместить в вершинах данного графа систему чисел h_1, ... , h_n, такую, что $(h_i, h_j) = d$, если соответствующие вершины графа соединены ребром и $(h_i, h_j) \neq d$, если между соответствующими вершинами ребра нет.

Доказательство. Разместим в вершинах графа числа d . Далее возьмем различные простые числа α_i, $i = 1..n$ такие, что d не делится на α_i. Домножим числа в вершинах графа на эти простые числа (каждой вершине отвечает одно число α_i). Теперь любые два числа в вершинах имеют наибольший общий делитель, равный d . Далее, будем перебирать все пары вершин, между которыми нет ребра и домножать числа в данной паре на новое простое число α_s, $s > n$, которое будет отлично от всех предыдущих чисел α_1, ... , α_{s-1} и которое не является делителем d. Когда процесс закончится, мы получим систему, в которой выполнены следующие свойства:

- если между вершинами есть ребро, то числа в вершинах имеют наибольший общий делитель d, т.к. мы каждый раз домножали на различные простые числа и эти множители могут совпасть только если между вершинами ребра нет;
- если между вершинами ребра нет, то наибольший общий делитель соответствующих чисел больше d, т.к. у данных чисел имеется еще один общий множитель, не являющийся делителем d.

Итак, требуемая система чисел построена.▷

Таким образом, для решения вопроса существования расписания для произвольной системы периодов необходимо уметь решать следующую задачу.

Проблема. Дан произвольный неориентированный граф. Вопрос: при каких условиях можно раскрасить вершины графа в n цветов так, чтобы любые две соседние по ребру вершины были разного цвета.

Поиск условий существования правильных раскрасок с фиксированным числом красок весьма сложен. Так, известная проблема четырех красок является утверждением, что планарность графа является достаточным условием для существования правильной раскраски в 4 цвета.

Естественным условием существования правильной раскраски в n цветов является отсутствие полных подграфов с n вершинами, однако это условие не дает гарантии того, что правильная раскраска существует. Известны примеры графов (см. рис. 3), хроматическое число которых значительно больше мощности максимального полного подграфа.

Тем не менее, существуют алгоритмы нахождения хроматического числа для данного графа, а также построения соответствующей раскраски.

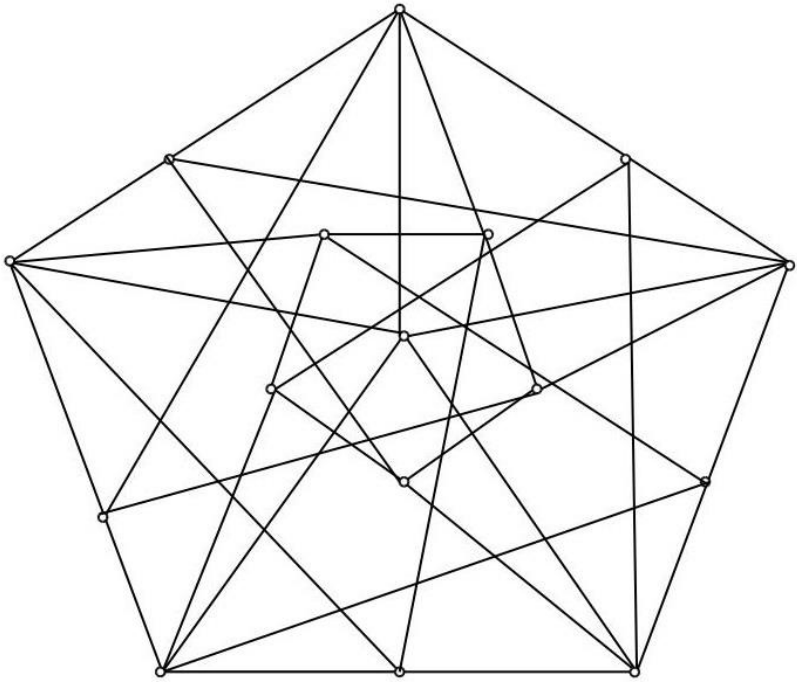


Рис. 3. Пример графа, не содержащего треугольников, с хроматическим числом, равным 5

Fig. 3. Example of triangle-free graph with chromatic number equal to 5.

4. Граф делимости и совместимость раскрасок

Пускай для всех подграфов G_d выполнено условие существования раскраски в d цветов. Естественно, возникает вопрос: существует ли в данном случае бесконфликтное расписание или нет. Следующий пример показывает, что это условие недостаточно.

Пример. Рассмотрим периоды 6, 12, 14, 18, 28, 30, 42, 154. Подграфы G_2, G_4, G_6, G_{14} изображены на рис. 4. Легко видеть, что любая правильная раскраска подграфа G_2 раскрашивает вершины $6, 12, 18, 30$ в один цвет: имеется путь по ребрам через вершины 6 – 14 – 12 – 154 – 18 – 28 – 30, а т.к. цветов только два, то они должны в этой цепочке чередоваться, т.е. вершины $6, 12, 18, 30$ будут окрашены в один цвет. Итак, начальные точки для вершин $6, 12, 18, 30$ должны иметь одинаковую четность, а это означает, что по модулю 6 для этих начальных точек имеется только три варианта (если начальные точки нечетны, то по модулю 6 они могут иметь остатки $1, 3, 5$, если четны — то остатки $0, 2, 4$). Однако, взглянув на подграф G_6 мы видим, что вершины $6, 12, 18, 30$ образуют полный подграф, так что по модулю 6 они должны быть раскрашены в четыре разных цвета, что невозможно, т.к. доступных цветов только три. Итак, бесконфликтное расписание в данном случае построить не удастся.

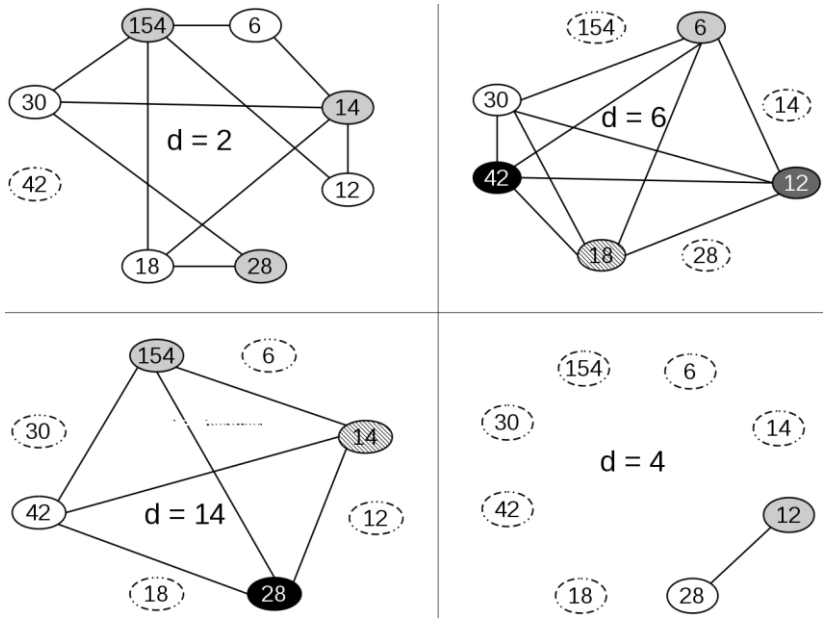


Рис. 4. Пример несовместимых правильных раскрасок графов G_d
Fig. 4. Example of incompatible proper coloring of graphs G_d

Далее, мы будем называть систему правильных раскрасок подграфов G_d *совместной*, если в этой системе возможно бесконфликтное расписание, если же такого расписания в данной системе раскрасок нет, то систему раскрасок назовем *несовместной*. Существование расписания в данной системе раскрасок означает наличие начальных точек $t_1, \ldots, t_N$ таких, что для всех d вершины подграфа G_d, покрашенные в разные цвета, имеют начальные точки с разными остатками по модулю d, а вершины, покрашенные в одинаковые цвета, имеют начальные точки с одинаковыми остатками по модулю d. Таким образом, каждому цвету соответствует ровно один остаток по модулю d.

Для дальнейшего нам понадобится еще один граф, который мы назовем *графом делимости*. Сначала построим множество вершин этого графа — множество H. Возьмем для каждой пары периодов их наибольший общий делитель d. Множество всех возможных d обозначим через D. Множество H — это замыкание множества D относительно операции взятия наибольшего общего делителя.

Множество H может быть построено стандартной процедурой замыкания:

1. В множество H добавляются все числа из D.
2. Для всех пар элементов из H вычисляются наибольшие общие делители и добавляются в множество H.
3. Шаг 2 повторяется до тех пор, пока множество H не перестанет увеличиваться.

Так как при добавлении в множество H попадают только числа, не превышающие максимального периода, то процесс построения H закончится, и количество шагов можно оценить максимальным периодом.

Множество H обладает следующими свойствами:

1. Для каждой пары чисел из H их наибольший общий делитель принадлежит H;
2. Множество H является минимальным среди множеств, обладающих свойством 1, и содержащих D.

Построим ориентированный граф делимости G_H на основе множества H. Вершины графа G_H — элементы множества H, между вершинами h_1 и h_2 проведено ребро с началом в h_1 и концом в h_2, если

- h_1 является собственным делителем h_2
- не существует промежуточной вершины h_3, для которой h_1 делит h_3 и h_3 делит h_2.

Если имеется ребро (h_1, h_2), то h_1 будем называть родителем, а h_2 — ребенком. Две вершины соединены путем, если имеется ориентированный путь из одной вершины в другую. Вершины, из которых есть путь в вершину h, будем называть предками вершины h. По построению, всякий предок является собственным делителем своего потомка.

Утверждение 6. Если две вершины h_1 и h_2 в графе G_H имеют общего родителя d, то d — их наибольший общий делитель.

Доказательство. Действительно, если d' — наибольший общий делитель h_1 и h_2, то он присутствует в графе G_H, так как множество H замкнуто относительно операции взятия наибольшего общего делителя. Родитель d является делителем и h_1, и h_2 по построению, то есть является их общим делителем. Если d отличен от d', то d является собственным делителем d'. Т.к. h_1 и h_2 отличны друг от друга, то кто-то из них отличен от d', например, h_1. Тогда получаем, что h_1 делится на d', а d' делится на d, причем все три числа отличны друг от друга, чего не может быть по свойствам построения ребра в графе G_H.▷

Рассмотрим для каждой вершины d графа G_H неориентированный граф G_d. Если d является наибольшим общим делителем какой-либо пары периодов, то G_d – граф, описанный выше (его вершины – периоды, ребра проведены там, где НОД равен d), во всех других случаях считаем граф G_d графом изолированных вершин-периодов. Пусть для каждого графа G_d дана раскраска вершин. По прежнему будем называть систему раскрасок совместной, если в этой системе можно составить бесконфликтное расписание, то есть существует такая бесконфликтная система начальных точек, что для любого графа G_d выполнено свойство: вершины, покрашенные одинаково, имеют начальные точки с одинаковыми остатками по модулю d, а вершины, покрашенные в разные цвета, имеют начальные точки с различными остатками по модулю d. Ниже мы доказываем несколько свойств, которые должны выполняться для совместимости раскрасок.

Свойство 1. В раскраске графа G_d должно участвовать не более d красок и раскраска эта должна быть правильной (т.е. если вершины соединены ребром, то они покрашены разными красками).

Доказательство. Так как остатков по модулю d ровно d, то и красок должно быть не больше d. То, что раскраска должна быть правильной, обсуждалось в предыдущем разделе.▷

Свойство 2. Если в графе G_d имеется раскрашенное одним цветом множество вершин, то в графе $G_{d'}$, где d' – предок d в графе G_H это множество должно быть также одного цвета.

Доказательство. Множество вершин одного цвета означает, что им приписан один и тот же остаток по модулю d, то есть для всех периодов из этого множества начальные точки сравнимы по модулю d, но тогда они сравнимы и по модулю d', так как d' – делитель d.▷

Свойство 3 (следствие свойства 2). Если в графе G_d две вершины разного цвета, то и в графе $G_{d'}$, где d' – потомок d в графе G_H, эти вершины разного цвета.

Доказательство. Если бы эти вершины в $G_{d'}$ были одинакового цвета, то, по свойству 2, в G_d они также должны были быть одного цвета, а это не так.▷

Свойство 4. Если в графе G_d подмножество S вершин раскрашено в один цвет, то в графе $G_{d'}$, где d' – потомок d в графе G_H, это подмножество вершин может быть раскрашено не более чем в $\frac{d'}{d}$ цветов.

Доказательство. Как уже говорилось в доказательстве свойства 2, для всех периодов из подмножества S начальные точки сравнимы по модулю d, таким образом, остатки по модулю d' должны быть сравнимы по модулю d, а таких остатков ровно $\frac{d'}{d}$ штук (r_0, $r_0 + d$, ... , $r_0 + (\frac{d'}{d} - 1)d$, где $r_0 < d$).▷

Свойство 5. Пусть d_1, ... , d_k – предки d в графе G_H, и пусть подмножество S вершин-периодов является одноцветным во всех графах G_{d_i}, i = 1,...,k. Тогда в графе G_d это подмножество вершин может быть раскрашено не более чем в $\frac{d}{HOK(d_1,...,d_k)}$ цветов.

Доказательство. Для всех периодов из подмножества S начальные точки сравнимы по модулю d_1, ... , d_k, то есть они (а также их остатки) сравнимы по модулю наименьшего общего кратного чисел d_1, ... , d_k, но таких остатков ровно $\frac{d}{HOK(d_1,...,d_k)}$, значит и цветов при раскрашивании S в графе G_d можно использовать не больше $\frac{d}{HOK(d_1,...,d_k)}$.▷

Итак, в совместной системе все указанные свойства должны быть выполнены. Заметим, что свойство 4 является частным случаем свойства 5. Здесь оно вынесено отдельно исключительно для удобства понимания.

Заметим, что для совместности системы раскрасок необходимо, чтобы существовала нумерация красок в графах G_d остатками по модулю d такая, что система сравнений $t \equiv r(mod\ d)$ для соответствующих начальных точек t была бы разрешима для всех периодов.

Докажем несколько фактов из теории чисел, полезных для дальнейшего изложения.

Лемма 1. Если $d = ab$, где a и b — взаимно простые числа, то сравнение $x \equiv r(mod\ d)$ эквивалентно системе из двух сравнений

$$\begin{cases} x \equiv r \pmod{a} \\ x \equiv r \pmod{b} \end{cases}$$

Доказательство. Если x — решение первого сравнения, то $x = ld + r$ для некоторого l. Отсюда, имеем: $x = lab + r$, то есть сравнения $x \equiv r(mod\ a)$ и $x \equiv r(mod\ b)$ верны. Обратно, если для x выполнены оба сравнения $x \equiv r(mod\ a)$ и $x \equiv r(mod\ b)$, то $x = l_1a + r = l_2b + r$, откуда имеем, что $l_1a = l_2b$, т.е., в силу взаимной простоты чисел a и b, l_1 делится на b, так что $l_1 = l_{1'}b$, а $x = l_1a + r = l_{1'}ab + r = l_{1'}d + r$, то есть $x \equiv r(mod\ d)$.▷

Лемма 2. Если $d = \alpha_1^{k_1}...\alpha_n^{k_n}$ — разложение числа d на различные простые множители α_i , то сравнение $x \equiv r(mod\ d)$ можно заменить на систему сравнений

$$\begin{cases} x \equiv r \pmod{\alpha_1^{k_1}} \\ \dots \\ x \equiv r \pmod{\alpha_n^{k_n}} \end{cases}$$

Доказательство. Для доказательства данного утверждения достаточно несколько раз применить лемму 1.▷

Лемма 3. Система сравнений $x \equiv r_i (mod\ \alpha^{k_i})$, где α — простое число, $i = 1,\dots,n$ и $k_1 < k_2 < \dots < k_n$, разрешима тогда и только тогда, когда $r_i \equiv r_j (mod\ \alpha^{k_j})$ при всех $i > j$. Если это условие выполнено, то данная система сравнений эквивалентна одному сравнению $x \equiv r_n (mod\ \alpha^{k_n})$.

Доказательство. Если система разрешима и $x = t$ — решение и $i > j$, то $t = l_i\alpha^{k_i} + r_i = l_j\alpha^{k_j} + r_j$, то есть $r_i - r_j$ делится на α^{k_j} или, что то же самое, $r_i \equiv r_j (mod\ \alpha^{k_j})$. Обратно, если $r_i \equiv r_j (mod\ \alpha^{k_j})$ при всех $i > j$ и $x = t$ — решение сравнения $x \equiv r_n (mod\ \alpha^{k_n})$, то $t = l\alpha^{k_n} + r_n \equiv r_j (mod\ \alpha^{k_j})$ для всех $j < n$, то есть t является решением всех остальных сравнений и их можно исключить, при этом множество решений не изменится.▷

Лемма 4. Пусть имеется система сравнений $x \equiv r_i (mod\ d_i)$ и $d_i = \alpha_{i1}^{k_{i1}} \dots \alpha_{in_i}^{k_{in_i}}$ — разложение на простые множители. По лемме 2 можно каждое сравнение заменить на систему сравнений $x \equiv r_i (mod\ \alpha_{ij}^{k_{ij}})$. Если при этом все цепочки сравнений по модулям степеней каждого простого числа удовлетворяют условию леммы 3, то первоначальная система сравнений разрешима и может быть заменена на систему $x \equiv r_i (mod\ \alpha_{is_i}^{k_{is_i}})$, где k_{is_i} — максимальная степень простого числа α_{is_i}.

Доказательство. По лемме 3 требуемая замена может быть произведена и, таким образом, наша система сведется к системе сравнений вида $x \equiv r_i (mod\ \alpha_{ik_i})$, где α_i — различные простые числа. Но такая система имеет решение (например, по китайской теореме об остатках, т.к. разные простые числа взаимно просты), значит имеет решение и первоначальная система.▷

Лемма 5. Если $a \equiv b (mod\ \alpha^s)$, а $b \equiv c (mod\ \alpha^{s'})$ для некоторого простого числа α и $s' \le s$, то $a \equiv c (mod\ \alpha^{s'})$.

Доказательство. По условию $a = \alpha^s l + b = \alpha^s l + \alpha^{s'} l' + c = \alpha^{s'} (\alpha^{s-s'} l + l') + c$. □

Теорема. Если для системы раскрасок выполнены указанные выше свойства 1, 2, 5, то эта система совместна.

Доказательство. Назовем корнем графа G_H вершину, в которую не входит ни одно ребро. Поскольку у любой пары вершин из G_H есть общий предок (их наибольший общий делитель), то корень у графа G_H один.

Упорядочим все вершины графа G_H естественным образом: по величине числа d, стоящего в вершине. Такой порядок обладает одним хорошим свойством:

предок всегда идет раньше потомка, — т.к. предок является собственным делителем потомка, то он меньше потомка по величине. Самым первым в ряду вершин графа G_H относительно данного порядка оказывается корень, т.к. все остальные вершины являются его потомками.

Введем некоторые обозначения. Пусть $d_1 < d_2 < \ldots < d_M$ — вершины графа G_H. Если p — некоторая раскрашенная вершина-период в графе G_d, то мы должны сопоставить ей некоторый остаток r(p,d) по модулю d, так что $r(p_1,d) = r(p_2,d)$, если p_1, p_2 покрашены одной краской и $r(p_1,d) \neq r(p_2,d)$, если p_1, p_2 покрашены в разные цвета. Наша цель — так пронумеровать остатками краски, чтобы система

$$\begin{cases} x \equiv r(p, d_1) \pmod{d_1} \\ \ldots \\ x \equiv r(p, d_M) \pmod{d_M} \end{cases}$$

имела решение для каждого $p \in P$.

Для построения такой нумерации будем действовать методом математической индукции.

База индукции: в корне d_1 по свойству 1 имеется не больше d_1 одноцветных множеств, и мы можем присвоить им произвольные различные остатки по модулю d_1. Очевидно, для любого остатка r по модулю d_1 существует решение сравнения $x \equiv r (mod\ d_1)$ — это сам остаток r.

Шаг индукции состоит в следующем. Предположим, что для $d_1, d_2, \ldots, d_{k-1}$ нумерация красок остатками уже произведена так, что система

$$\begin{cases} x \equiv r(p, d_1) \pmod{d_1} \\ \ldots \\ x \equiv r(p, d_{k-1}) \pmod{d_{k-1}} \end{cases}$$

имеет решение для каждого $p \in P$. Обозначим решение этой системы $t_{k-1}(p)$.

Построим нумерацию для красок, которыми покрашен граф G_{d_k}. Все вершины-периоды в G_{d_k} разбиваются на непересекающиеся одноцветные множества $S_1, \ldots, S_m$, каждому такому множеству должен быть поставлен в соответствие свой остаток.

По свойству 2, множество S_i в любом родителе вершины d_k является одноцветным и можно этому множеству сопоставить набор цветов в родителях d_k. Два множества S_{i_1} и S_{i_2} будем называть похожими, если в каждом из родителей d_k они покрашены в одинаковый цвет. Если же в каком-то из родителей их цвет различается, то такие множества будем называть непохожими.

Рассмотрим множество S_i для некоторого i. Выберем какой-то период p, лежащий в S_i и рассмотрим решение $t = t_{k-1(p)}$ соответствующей системы сравнений. Пусть $t \equiv r (mod\ d_k)$, где $r < d_k$ — остаток от деления t на d_k. Если d_i — предок d_k, то $r \equiv r(p, d_i) (mod\ d_i)$. Действительно, $t \equiv r(p, d_i) (mod\ d_i)$,

значит $t = d_i l' + r(p, d_i) = d_k l + r$, отсюда, $r = d_i l' - d_k l + r(p, d_i) = d_i l'' + r(p, d_i)$, т.к. d_i — предок d_k, т.е. является его собственным делителем. Больше того, если m — наименьшее общее кратное всех родителей d_k, то $r + am \equiv r(p, d_i)(mod\ d_i)$ для любого целого a. Это следует из того, что m делится на d_i. Имеется ровно $\frac{d_k}{m}$ чисел вида $r + am$, попадающих в промежуток от 0 до $d_k - 1$, т.е. являющихся остатками по модулю d_k. С другой стороны, по свойству 5 имеется не больше $\frac{d_k}{m}$ множеств S_j, похожих на S_i, т.е. покрашенных в каждом из родителей d_k в тот же цвет, что и множество S_i. Действительно, если объединить все похожие на S_i множества, то в каждом родителе d_k это объединение является одноцветным, а по свойству 5 такое множество не может быть покрашено больше чем в $\frac{d_k}{m}$ цветов в графе G_{d_k}. Таким образом, можно каждому похожему на S_i множеству сопоставить остаток вида $r + am$, причем все такие остатки можно сделать разными.

Итак, мы сопоставили каждому из множеств S_i некоторый остаток по модулю d_k. Причем для разных S_i эти остатки разные: если два множества S_{i_1} и S_{i_2} похожи, то остатки у них различны по построению, а если они непохожи, то остатки будут различны, т.к. по модулю одного из родителей d_k эти остатки сравнимы с разными числами (для того из родителей, для которого множества S_{i_1} и S_{i_2} покрашены в разные цвета).

Необходимо показать, что новая система

$$\begin{cases} x \equiv r(p, d_1) \pmod{d_1} \\ \dots \\ x \equiv r(p, d_k) \pmod{d_k} \end{cases}$$

имеет решение для любого $p \in P$.

Рассмотрим данную систему сравнений. По лемме 2 можно заменить ее на систему вида

$$\begin{cases} x \equiv r(p, d_1) \pmod{\alpha_i^{s_{i1}}} \\ \dots \\ x \equiv r(p, d_k) \pmod{\alpha_i^{s_{ik}}} \end{cases}$$

где α_i — простые числа, входящие в разложение чисел d_j.

Нужно проверить не нарушилась ли совместность системы при добавлении новых сравнений. Пусть α^s — наибольшая степень некоторого простого числа α, входящая в разложение числа d_k на простые множители. Тогда к первоначальной системе добавляется сравнение $x \equiv r(p, d_k)(mod\ \alpha^s)$. Есть два случая: или s — максимальная степень из степеней α, входящих в d_i или же эта степень не максимальная.

Пусть s — максимальная степень α и пусть некоторое $d = d_j$, $j < k$ имеет в своем разложении степень $\alpha^{s'}$, $s' \le s$. Тогда имеется общий предок d' вершин

d и d_k, который также имеет в своем разложении степень $\alpha^{s'}$. Пусть d'' — родитель d_k, такой, что d' делит d'', тогда d'' делится на некоторую степень $\alpha^{s''}$, так что $s' \le s'' \le s$. По построению, $r(p, d_k) \equiv r(p, d'')(mod\ d'')$, то есть $r(p, d_k) \equiv r(p, d'')(mod\ \alpha^{s''})$. По предположению индукции, $r(p,d) \equiv r(p, d')(mod\ \alpha^{s'})$ и $r(p, d'') \equiv r(p, d')(mod\ \alpha^{s'})$. Отсюда, по лемме 5, $r(p, d_k) \equiv r(p, d')(mod\ \alpha^{s'})$ и $r(p, d_k) \equiv r(p,d)(mod\ \alpha^{s'})$, таким образом условие леммы 3 не нарушается при добавлении нового сравнения.

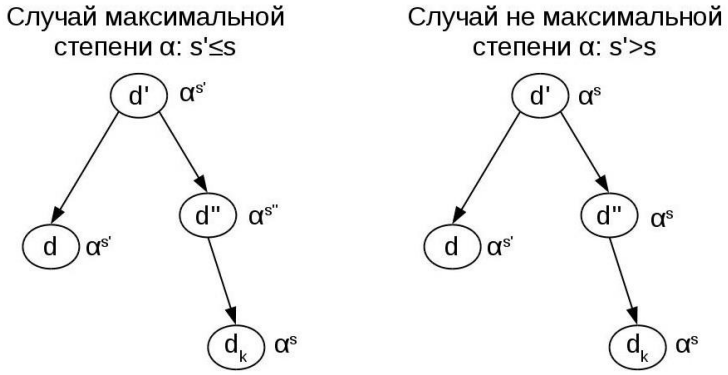


Рис. 5. Иллюстрация распределения степеней α в разных случаях
Fig. 5 Distribution of degrees of α in different cases.

Если же s — не максимальная степень α, то имеется $d = d_j$, $j < k$, имеющее в своем разложении степень $\alpha^{s'}$, $s' > s$. Аналогично предыдущему случаю, имеется общий предок d' вершин d и d_k, который имеет в своем разложении степень α^s. Пусть d'' — родитель d_k, такой, что d' делит d'', тогда d'' делится на степень α^s. По предположению индукции, $r(p,d) \equiv r(p, d')(mod\ \alpha^s)$. С другой стороны, по построению $r(p, d_k) \equiv r(p, d'')(mod\ d'')$, то есть $r(p, d_k) \equiv r(p, d'')(mod\ \alpha^s)$, и, таким образом, новое сравнение эквивалентно одному из уже имевшихся, так что условие леммы 3 снова выполнено.

Итак, при добавлении новых сравнений условие леммы 3 выполняется, то есть, по лемме 4 новая система сравнений будет иметь решение.

Осталось показать, что расписание, в котором решения $t_M(p)$ являются начальными точками будет бесконфликтным. Это вытекает из того, что $t_M(p) \equiv r(p,d)(mod\ d)$, и для p_1, p_2 с наибольшим общим делителем d в графе G_d вершины p_1, p_2 разного цвета, то есть, по построению, остатки $r(p_1,d)$, $r(p_2,d)$ также различны. То есть $t_M(p_1)$ и $t_M(p_2)$ имеют разные остатки по модулю $d = (p_1, p_2)$, и, согласно следствию 1, расписание бесконфликтно.▷

Из доказательства теоремы видно, что для совместности системы раскрасок достаточно выполнения свойства 5 для родителей вершины d, а не для произвольной системы предков этой вершины, так что окончательный результат можно сформулировать так:

Теорема о существовании бесконфликтного расписания. Для системы периодов P существует бесконфликтное расписание тогда и только тогда, когда существует система раскрасок графов G_d для всех $d \in H$ такая, что

- В раскраске графа G_d используется не более d красок и раскраска эта правильная.
- Если в графе G_d имеется раскрашенное одним цветом множество вершин, то в графе $G_{d'}$, где d' – предок d в графе G_H это множество также одного цвета.
- Пусть $d_1, \dots, d_k$ – родители d в графе G_H, и пусть подмножество S вершин-периодов является одноцветным во всех графах G_{d_i}, $i = 1,\dots,k$. Тогда в графе G_d это подмножество вершин раскрашено не более чем в $\frac{d}{НОК(d_1,\dots,d_k)}$ цветов. □

5. Обзор существующих подходов к построению расписаний для строго периодических задач

В работах [7], [10] исследуется случай планирования строго периодических задач без прерываний. Доказываются критерии бесконфликтности для случая двух задач на основе НОД их периодов. Также приводится ряд громоздких достаточных условий бесконфликтности в общем случае на основе рассмотрения НОД всех периодов.

Отметим, что НОД всех периодов является слишком грубой характеристикой. Например, для задач с периодами 6, 10, 15 расписание построить возможно, хотя их НОД = 1.

В работах [5], [6], [11], [12] исследуется случай планирования строго периодических задач с прерываниями.

В работе [11] доказаны условия невозможности построения расписания:

- какие-то два периоды взаимно-просты;
- НОД периодов каких-то двух задач делит разность их стартовых точек.

Кроме того, приводится громоздкое достаточное условие невозможности построения расписания в случае, когда для данных задач дополнительно введено отношение «предшествования».

В работе [12] предлагается эвристический алгоритм построения расписаний с условием, что выход из прерывания требует ненулевое время. Задачи планируются последовательно по мере возрастания их периодов, причем в

качестве точки старта для очередной задачи выбирается наименьшая из гарантированно свободных точек. Если этого сделать не удалось, то считается, что расписание построить невозможно. Отметим, что это вообще говоря не так, поскольку алгоритм не перебирает другие возможности для выбора стартовой точки. Тем не менее, алгоритм работает в случае, когда количество задач сравнительно невелико, а попарные НОД их периодов достаточно большие — тогда множество гарантированно свободных точек не успевает стать пустым к моменту планирования последней задачи.

В работах [5], [6] предлагаются переборные (с некоторыми оптимизациями) алгоритмы построения расписаний с целью минимизировать суммарное количество прерываний (при этом сами прерывания считаются «бесплатными»). Собственно перебор используется для поиска стартовых точек.

Отметим, что во всех приведенных алгоритмах (в т.ч. «переборной» модификации алгоритма из [12]) наличие реального конфликта в расписании можно установить только перебрав все возможные варианты.

6. Заключение

В настоящей работе предложен новый подход к анализу возможности построения расписания для строго периодических задач. На основе предложенного представления структуры группы периодов в терминах теории графов доказан критерий существования бесконфликтного набора стартовых точек для произвольного количества задач. Полученный критерий позволяет либо направленно найти стартовые точки, либо быстро установить, что расписание построить невозможно.

Результат настоящей работы позволяет продолжить исследования проблемы построения бесконфликтных расписаний и в то же время указывает на сложность этой проблемы в общем случае. В рамках данной проблемы возникают, в частности, следующие направления для дальнейших исследований:

- создание алгоритмов построения расписаний (в том числе для задач с произвольными длительностями);
- вопрос оптимизации распределения задач по нескольким устройствам с заданными характеристиками;
- вопрос изменения расписания при добавлении новых задач;
- выбор подходящих характеристик (например, периодов) задач, если есть возможность менять эти характеристики в некоторых пределах;
- составление расписания при наличии дополнительных требований, таких как синхронизация задач или дополнительные ограничения на время пересылки сообщений.

Исследованию этих и других смежных вопросов мы посвятим наши следующие работы.

Список литературы

[1] Зыков А.А. Основы теории графов. М: Вузовская книга, 2004.
[2] Кристофидес Н. Теория графов. Алгоритмический подход. М: Мир, 1978.
[3] Оре О. Теория графов. М: Наука, 1980.
[4] Виноградов И.М. Основы теории чисел. М.-Л.: Гостехиздат, 1952.
[5] Зеленов С.В. Планирование строго периодических задач в системах реального времени. Труды ИСП РАН, том 20, 2011 г., с. 113-122.
[6] Третьяков А. Автоматизация построения расписаний для периодических систем реального времени. Труды ИСП РАН, том 22, 2012 г., стр. 375-400. DOI: 10.15514/ISPRAS-2012-22-20.
[7] Kermia O., Sorel Y. Schedulability Analysis for Non-Preemptive Tasks under Strict Periodicity Constraints. Proceedings of the 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2008, p. 25-32.
[8] Liu C.L., Layland J.W. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. J. ACM, 1973, No 20, p. 46-61.
[9] Liu J.W.S.W. Real-Time Systems. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edn. 2000.
[10] Marouf M., Sorel Y. Schedulability conditions for non-preemptive hard real-time tasks with strict period. Proceedings of 18th International Conference on Real-Time and Network Systems, RTNS'10, 2010, p. 50-58.
[11] Yomsi P.M., Sorel Y. Non-Schedulability Conditions for Off-line Scheduling of Real-Time Systems Subject to Precedence and Strict Periodicity Constraints. Proceedings of 11th IEEE International Conference on Emerging technologies and Factory Automation, ETFA'06, WIP, Prague, Czech Republic, September 2006.
[12] Yomsi P.M., Sorel Y. Schedulability Analysis for non Necessarily Harmonic Real-Time Systems with Precedence and Strict Periodicity Constraints using the Exact Number of Preemptions and no Idle Time. Proceedings of the 4th Multidisciplinary International Scheduling Conference, MISTA'09, Dublin, Ireland, August, 2009.

Non-conflict scheduling criterion for strict periodic tasks

[1] *S.A. Zelenova <sophia@ispras.ru>*
[1,2] *S.V. Zelenov <zelenov@ispras.ru>*
[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia*
[2] *National Research University Higher School of Economics (HSE) 20 Myasnitskaya Ulitsa, Moscow, 101000, Russia*

Abstract. In the paper, we address mission critical systems, such as automobile, avionic, mobile robotic, telecommunication, etc. Such systems must meet hard real-time constraints in order to avoid catastrophic consequences. To meet the real-time constraints, strict periodicity is used (i.e. for any periodic task, time between release points is constant). Sensors, actuators and feedback control functions are typical examples of strict periodic tasks. We study a monoprocessor preemptive scheduling problem for arbitrary number of strict periodic tasks. In this context, we focus on the following problem: how to find non-conflict set of task release points (i.e. sequences of instance release points for different tasks must not intersect).

First, as a preliminaries, we introduce some fundamental definitions and prove several elementary schedulability conditions. Next, we investigate the correlation between the scheduling problem and a graph coloring problem for graphs of some special kinds. The graphs under consideration are built on the basis of the tasks' period values. We introduce a notion of divisibility graph for tasks' periods, and study compatibility of graphs' coloring with respect to the schedulability problem. At last, we prove a theorem that provides necessary and sufficient graph coloring conditions for schedulability of given strict periodic tasks. This theorem allows either to find non-conflict set of task release points directly, or to determine quickly that scheduling is impossible.

Keywords: real-time system; strict periodic task; scheduling; graph coloring; system of linear congruences.

DOI: 10.15514/ISPRAS-2017-29(6)-10

For citation: Zelenova S.A., Zelenov S.V. Non-conflict scheduling criterion for strict periodic tasks. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 6, 2017. pp. 183-202 (in Russian). DOI: 10.15514/ISPRAS-2017-29(6)-10

References

[1] A. A. Zykov. Fundamentals of Graph Theory. BCS Associates, 1990.
[2] N. Christofides. Graph Theory. An Algorithmic Approach. Academic Press, 1975.
[3] O. Ore. Theory of graphs. American Mathematical Society, 1962.
[4] I.M. Vinogradov. Elements of Number Theory. Dover Publications Inc. 1954.
[5] S.V. Zelenov. Scheduling of Strictly Periodic Tasks in Real-Time Systems. Trudy ISP RAN/Proc. ISP RAS, vol. 20, 2011, pp. 113-122 (in Russian).
[6] A.V. Tretyakov. Automation of scheduling for periodic real-time systems. Trudy ISP RAN/Proc. ISP RAS, vol. 22, 2012, pp. 375-400 (in Russian). DOI: 10.15514/ISPRAS-2012-22-20.
[7] Kermia O., Sorel Y. Schedulability Analysis for Non-Preemptive Tasks under Strict Periodicity Constraints. Proceedings of the 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2008, p. 25-32.
[8] Liu C.L., Layland J.W. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. J. ACM, 1973, No 20, p. 46-61.
[9] Liu J.W.S.W. Real-Time Systems. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edn. 2000.
[10] Marouf M., Sorel Y. Schedulability conditions for non-preemptive hard real-time tasks with strict period. Proceedings of 18th International Conference on Real-Time and Network Systems, RTNS'10, 2010, p. 50-58.
[11] Yomsi P.M., Sorel Y. Non-Schedulability Conditions for Off-line Scheduling of Real-Time Systems Subject to Precedence and Strict Periodicity Constraints. Proceedings of 11th IEEE International Conference on Emerging technologies and Factory Automation, ETFA'06, WIP, Prague, Czech Republic, September 2006.
[12] Yomsi P.M., Sorel Y. Schedulability Analysis for non Necessarily Harmonic Real-Time Systems with Precedence and Strict Periodicity Constraints using the Exact Number of Preemptions and no Idle Time. Proceedings of the 4th Multidisciplinary International Scheduling Conference, MISTA'09, Dublin, Ireland, August, 2009.

Реализация сервиса для замены Keystone в качестве центрального сервиса идентификации облачной платформы Openstack[1]

Е.Л. Аксенова <lenaaxenova@ispras.ru>
В.В. Швецова <shvetcova@ispras.ru>
О.Д. Борисенко <al@somestuff.ru>
И.В. Богомолов <bogomolov@ispras.ru>
Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. В данной работе рассматриваются проблемы масштабируемости проекта Keystone — центрального сервиса авторизации и аутентификации облачной платформы Openstack и новый принцип построения сервиса, позволяющий избегать этих проблем. В более ранних работах был предложен подход к масштабированию Openstack Keystone путем отказа от использования СУБД MySQL/MariaDB и PostgreSQL в качестве хранилища данных сервиса в пользу распределенных NoSQL решений. Данная работа представляет полноценную реализацию сервиса, обеспечивающего полную функциональность Openstack Keystone API V3, на базе API Gateway и использования Apache Cassandra.

Ключевые слова: Openstack Keystone; Apache Cassandra; Kong; API Gateway; Lua; облачные среды.

DOI: 10.15514/ISPRAS-2017-29(6)-11

Для цитирования: Аксенова Е.Л., Швецова В.В, Борисенко О.Д., Богомолов И.В. Реализация сервиса для замены Keystone в качестве центрального сервиса идентификации облачной платформы Openstack. Труды ИСП РАН, том 29, вып. 6, 2017 г., стр. 203-212. DOI: 10.15514/ISPRAS-2017-29(6)-11

1. Введение

В современном мире существует необходимость надежного хранения и оперативной обработки больших объемов данных. Одним из наиболее перспективных подходов в данной области является построение систем

[1] РФФИ 15-29-07111 офи_ м – Исследование методов обеспечения масштабируемости систем в облачных средах и разработка высокопроизводительного отказоустойчивого центрального сервиса идентификации.

хранения и обработки данных над облачными средами. Ресурсы облачной среды могут предоставляться клиенту в моделях обслуживания IaaS (инфраструктура как услуга), PaaS (платформа как услуга), SaaS (программное обеспечение как услуга), DaaS (рабочее окружение как услуга) [1]. В данной работе рассматривается облачная платформа Openstack и ее сервисы, обеспечивающие IaaS модель обслуживания.

Модель IaaS характеризуется предоставлением вычислительных виртуальных ресурсов, возможностью организации сетей, а также предоставлением ресурсов хранения (загрузочные образы операционных систем, хранилища объектных данных и виртуальные блочные устройства). Данная модель подразумевает полноценное использование клиентом вычислительных ресурсов выделенной виртуальной машины и является самой низкоуровневой моделью обслуживания в облачных средах.

Облачные услуги предоставляются на коммерческой основе такими провайдерами как Amazon EC2 [2], Microsoft Azure [3], Google Compute Engine [4] и рядом других. Однако существует множество ситуаций, где использование публичных облачных сервисов недопустимо из-за повышенных требований к контролю над пользовательскими данными. В таких условиях как правило используются приватные облачные среды, построенные на платформах с открытыми исходными кодами. Среди открытых платформ, предоставляющих выделяются следующие проекты: OpenStack [5], Eucalyptus [6], OpenNebula [7].

Кроме описанной выше функциональности облачные платформы должны поддерживать возможность масштабирования как относительно физических узлов системы, так и относительно количества пользователей системы с учетом разделения прав доступа. Архитектура облачных платформ представляет из себя большое число изолированных сервисов, каждый из которых реализует определенную функциональность облачной среды [8]. Все сервисы системы обязаны поддерживать различные варианты развертывания на множество физических узлов.

Обязательным компонентом облачных платформ является специальный модуль, отвечающий за аутентификацию и авторизацию пользователей и сервисов облачной среды (далее — сервис идентификации). С целью защиты от подмены аппаратных узлов и проверки прав доступа на запрашиваемые ресурсы все подсистемы проекта взаимодействуют с сервисом идентификации. В связи с этим данный сервис подвергается существенной нагрузке, которая растет в зависимости от числа работающих подсистем и пользователей проекта.

В данной работе исследуются проблемы открытого проекта OpenStack [9], предоставляющего наибольшую функциональность среди открытых облачных платформ [10]. Основным объектом исследований является сервис идентификации Keystone [11], недостатки которого оказывают существенное

влияние на работу всей системы. В частности, сервис Keystone является одной из основных причин невозможности масштабируемости системы как по количеству физических узлов, так и по числу активных пользователей [12].

В работе [13] подробно описана методика тестирования, в ходе которого были выявлены основные проблемы сервиса идентификации Keystone:

1. Низкая производительность сервиса с точки зрения обработки числа запросов к системе в секунду (RPS).
2. Деградация производительности со временем вплоть до отказа в обслуживании вне зависимости от используемых модулей, обеспечивающих его работу.
3. Невозможность построения монолитной облачной среды на базе открытых облачных платформ с более чем 200 физическими узлами в системе без использования специфических решений, которые не могут являться универсальными. К таким решениям относятся отказ от монолитности облачной среды с использованием федераций и отказ от встроенного сервиса идентификации Keystone.
4. Нелинейный характер масштабирования системы в зависимости от числа используемых ресурсов.

Также был проведен анализ причин подобного поведения. Одной из основных причин наблюдаемых проблем является построение сервиса над классическими РСУБД с ограничениями по количеству одновременных клиентских подключений. Одновременно с этим сама архитектура сервиса не позволяет производить группировку соединений с СУБД.

В других облачных платформах сервис идентификации построен по сходным принципам.

В крупных облачных системах данные проблемы являются существенными, поэтому цель данной работы заключается в устранении вышеперечисленных недостатков за счет перехода к новой модели устройства облачного сервиса идентификации на базе распределенных NoSQL СУБД.

2. Устройство предлагаемого решения

Данный раздел посвящен принципам построения масштабируемого сервиса для замены Openstack Keystone. В качестве основного принципа используется раздельное масштабирование узлов, отвечающих за построение токенов, и узлов, отвечающих за хранение пользовательских данных.

В рамках проверки идеи об использовании распределенных хранилищ для масштабирования сервиса, предоставляющего функциональность Openstack Keystone, был реализован прототип, реализующий функциональность для самого требовательного сценария нагрузки. С точки зрения алгоритма работы необходимых для сценария вызовов, прототип не отличался от Openstack Keystone. Однако для реализации был использован язык Lua и использован

Tarantool [14][15] в качестве распределенного хранилища. Результат тестирования приведен на рис. 1.

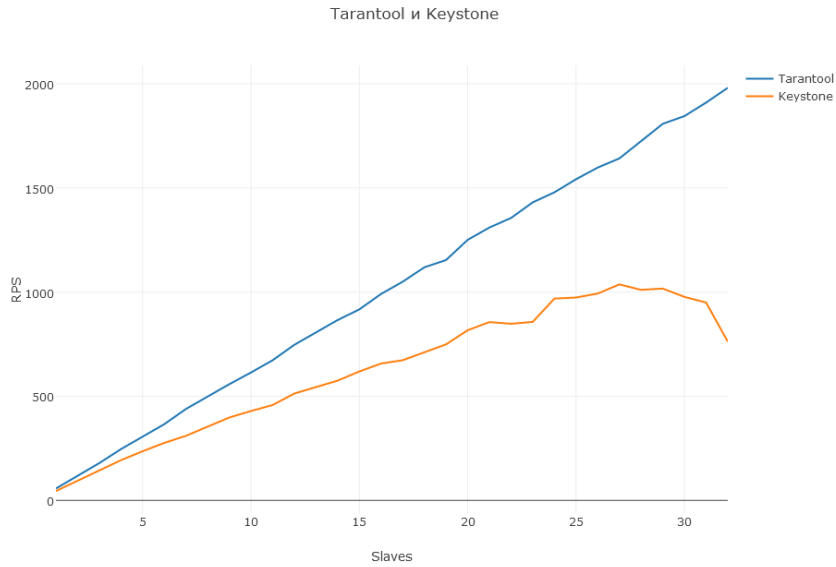


Рис. 1. Зависимость критического значения RPS от числа узлов в кластере для Keystone (PostgreSQL + tmpfs) и прототипа на базе Tarantool
Fig. 1. Critical RPS value to cluster nodes for Keystone (PostgreSQL on tmpfs) and Tarantool-based prototype

В отличие от результатов сервиса Keystone, реализованное решение демонстрирует линейный характер масштабируемости и более высокое значение RPS, что особенно заметно при большом числе узлов кластера. Таким образом, обеспечивается возможность масштабирования облачной платформы по физическим узлам и по числу пользователей, в отличии от сервиса OpenStack Keystone.

Положительные результаты, полученные при отказе от использования централизованной РСУБД в пользу распределенных решений, демонстрируют необходимость полноценной реализации сервиса идентификации, опирающегося на распределенную СУБД. Для решения данной задачи нами была выбрана система API Gateway Kong.

Как было отмечено выше, проект Openstack представлен в виде набора изолированных сервисов и развивается согласно принципам микросервисной архитектуры [16]: каждая из подсистем проекта работает в отдельном процессе и проектируется вне зависимости от других систем, взаимодействие между ними осуществляется с помощью HTTP-протокола. Ввиду

разнородности данных систем пользователю нередко приходится сталкиваться с тем, что интерфейс различных модулей отличается друг от друга, работа с такими системами может вызывать ряд затруднений. Популярным вариантом решения подобных проблем является использование технологии API Gateway, которая представляет из себя дополнительный программный и логический уровень между клиентами и внутренними интерфейсами сервисов [17].

Основными характеристиками таких систем являются простота масштабирования и добавления новых систем в проект, возможность проверки идентификационных данных непосредственно в слое API Gateway, удобная поддержка версий подсистем проекта, а также возможность кэширования запросов и ответов системы и регулирование трафика. Благодаря указанным свойствам, логика работы сервиса идентификации Keystone прозрачно переносится на слой API Gateway.

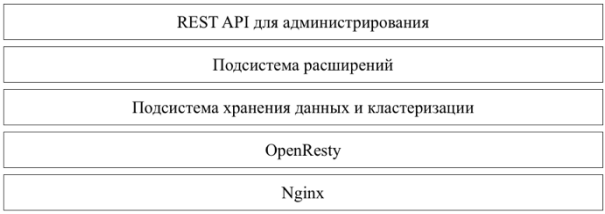


Рис. 2. Слои в API Gateway Kong
Fig. 2. Kong API Gateway layers

Среди открытых платформ, реализующих технологию API Gateway, система Kong [18] предоставляет наибольшую функциональность, а также поддерживает взаимодействие с как с реляционными, так и с нереляционными СУБД. Kong состоит из нескольких слоев (см. рис. 2).

- Верхний слой работает по принципам RESTful API (и перенаправляет полученные запросы и ответы согласно логике, реализованной в следующем слое – плагинах);
- Слой расширений представляет собой программы, написанные на языке Lua, они и обеспечивают работу API Gateway. Kong предоставляет набор готовых расширений, а также поддерживает возможность написания новых;
- Следующий слой отвечает за кластеризацию и хранение данных. Kong поддерживает работу с СУБД PostgreSQL и Apache Cassandra [19], а также допускает использование Redis [20] для кэширования данных. СУБД используются для хранения всех служебных данных проекта Kong и, кроме того, могут быть использованы для хранения пользовательских данных согласно логике, определенной каждым конкретным расширением;

- Следующий слой Openresty является расширением веб-сервера Nginx [21] с помощью функций, написанных на языке Lua. Данный слой позволяет контролировать и обрабатывать пользовательские запросы и ответы на различных стадиях их жизненного цикла (lifecycle);
- Нижний слой представлен веб-сервером Nginx и используется для обработки HTTP/HTTPS и проксирования запросов.

В качестве системы хранения данных для разработанного сервиса выбрана Apache Cassandra.

Авторами работы был разработан и реализован сервис идентификации для облачной среды Openstack в виде плагина на языке Lua, интегрируемого в систему Kong. Для хранения данных используется распределенная СУБД Apache Cassandra поскольку ее модель масштабирования широко известна своей эффективностью [22].

Данные в Cassandra хранятся в соответствии с моделью данных «семейство столбцов». Под семейством столбцов подразумевается структура, содержащая неограниченное число строк, доступ к которым осуществляется по уникальному ключу – имени строки. Основные преимущества Apache Cassandra заключаются в высокой масштабируемости и отказоустойчивости системы, очень высокой пропускной способности операции записи и хорошей пропускной способности для операций считывания, а также в возможности репликации с настраиваемым уровнем согласованности. Но при этом, являясь NoSQL системой, Cassandra не поддерживает транзакции ACID, операции соединения и возможность определения внешних ключей.

Схема данных в авторской реализации практически совпадает с оригинальной схемой данных в проекте Keystone — с поправкой на специфику Cassandra: в схеме отсутствуют внешние ключи.

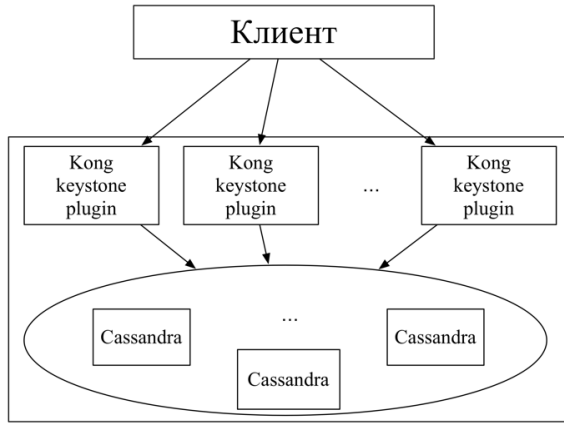


Рис. 3. Схема масштабирования разработанной системы
Fig. 3. Scaling scheme for the system

На рис. 3 схематично демонстрируется возможность масштабирования системы идентификации, реализованной с помощью Kong, на некоторое число узлов кластера. Каждый сервис Kong при этом осуществляет взаимодействие с распределенной СУБД Cassandra.

В следующих работах будет проведено тестирование производительности и сравнение с Openstack Keystone в контексте возможностей масштабирования.

Список литературы

[1]. Moreno-Vozmediano R., Montero R.S., Llorente I.M. Iaas cloud architecture: From virtualized datacenters to federated cloud infrastructures. Computer, vol. 45, no. 12, 2012, pp. 65-72.
[2]. Официальный сайт Amazon Elastic Compute Cloud. - https://aws.amazon.com/ec2/.
[3]. Официальный сайт Microsoft Azure. - https://azure.microsoft.com/en-us/.
[4]. Официальный сайт Google Compute Engine. – https://cloud.google.com/compute/.
[5]. Официальныйсайт проекта OpenStack. – https://www.openstack.org/.
[6]. Официальныйсайт проекта Eucalyptus. – https://www.eucalyptus.com/.
[7]. Официальныйсайт проекта OpenNebula. – https://opennebula.org.
[8]. Luo J.Z. et al. Cloud computing: architecture and key technologies. Journal of China Institute of Communications, vol. 32, no. 7, 2011. pp. 3-21.
[9]. Freet D. et al. Open source cloud management platforms and hypervisor technologies: A review and comparison. SoutheastCon, 2016. IEEE, 2016, pp. 1-8.
[10]. Lynn T. et al. A Comparative Study of Current Open-source Infrastructure as a Service Frameworks. CLOSER, 2015, pp. 95-104.
[11]. Описание архитектуры Openstack Keystone. http://docs.openstack.org/developer/keystone/architecture.html.
[12]. Богомолов И.В., Алексиянц А.В., Борисенко О.Д., Аветисян А.И. Проблемы масштабируемости облачных сред и поиск причин деградации центрального сервиса идентификации OpenStack Keystone. Известия ЮФУ. Технические науки, №12 (185), 2016 г., стр. 130-140. DOI: 10.18522/2311-3103-2016-12-130140
[13]. И.В. Богомолов, А.В. Алексиянц, А.В. Шер, О.Д. Борисенко, А.И. Аветисян. Метод тестирования производительности и стресс-тестирования центральных сервисов идентификации облачных систем на примере Openstack Keystone. Труды ИСП, том 27, вып. 5, 2015 г., стр. 49–58. DOI: 10.15514/ISPRAS-2015-27(5)-4
[14]. Официальный сайт проекта Tarantool. - https://tarantool.org/.
[15]. Abramova V., Bernardino J., Furtado P. Experimental evaluation of NoSQL databases, International Journal of Database Management Systems, No. 3, 2014, pp. 1-16.
[16]. Dmitry Namiot, Manfred Sneps-Sneppe. On Micro-services Architecture. International Journal of Open Information Technologies, vol. 2, no. 9, 2014, pp. 24-27.
[17]. Anton Fagerberg. Optimising clients with API Gateways. Department of Computer Science Faculty of Engineering LTH, 2015.
[18]. Официальный сайт проекта Kong. - https://getkong.org.
[19]. Официальный сайт проекта Cassandra. - http://cassandra.apache.org.
[20]. Официальный сайт проекта Redis. - https://redis.io.
[21]. Официальный сайт проекта Nginx. - http://nginx.org.
[22]. Lakshman A., Prashant M. Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review, vol. 4, issue 2, 2010, pp. 35-40.

Openstack Keystone identification service drop-in replacement

E.L. Axenova <lenaaxenova@ispras.ru>
V.V. Shvetsova <shvetcova@ispras.ru>
O.D. Borisenko <al@somestuff.ru>
I.V. Bogomolov <bogomolov@ispras.ru>
Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. The paper is dedicated to architecture and scalability principles for developed service intended to be a drop-in replace for Openstack Keystone. Openstack Keystone is the central identification and service catalogue service for clouds based on Openstack. Previous papers indicated problems of this service: it uses RDBMS (MariaDB/MySQL/PostgreSQL) as a data storage. Since each service and each user gets a token to have access to Openstack cloud and tokens are periodically revoked by the system, token generation is a critical function for the whole cloud. As soon as Keystone queries DBMS for getting user or service identification hashes and recomputes this hash upon the user-provided data, there is a bottleneck based on Keystone architecture. Each Keystone process has separate session with DBMS and since the recommended way is to use Galera cluster thus the DBMS part is limited to the slowest DBMS node since Galera provides High-Availability not the performance scale. Our approach is based on API Gateway Kong and its scalability through Apache Cassandra usage as a data store. Drop-in replacement is implemented as Lua plugin inside Kong API Gateway and implements Keystone V3 API.

Keywords: Openstack Keystone; Apache Cassandra; Kong; API Gateway; Lua; cloud platform.

DOI: 10.15514/ISPRAS-2017-29(6)-11

For citation: Axenova E.L., Shvetsova V.V., Borisenko O.D., Bogomolov I.V. Openstack Keystone identification service drop-in replacement. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 6, 2017, pp. 203-212 (in Russian). DOI: 10.15514/ISPRAS-2017-29(6)-11

References

[1]. Moreno-Vozmediano R., Montero R.S., Llorente I.M. Iaas cloud architecture: From virtualized datacenters to federated cloud infrastructures. Computer, vol. 45, no. 12, 2012, pp. 65-72.
[2]. Amazon Elastic Compute Cloud official page. Available at: https://aws.amazon.com/ec2/.
[3]. Microsoft Azure official page. Available at: https://azure.microsoft.com/en-us/.
[4]. Google Compute Engine official page. Available at: https://cloud.google.com/compute/.
[5]. OpenStack project official page. Available at: https://www.openstack.org/.
[6]. Eucalyptus project official page. Available at: https://www.eucalyptus.com/.
[7]. OpenNebula project official page. Available at: https://opennebula.org.

[8]. Luo J.Z. et al. Cloud computing: architecture and key technologies. Journal of China Institute of Communications, vol. 32, no. 7, 2011. pp. 3-21.
[9]. Freet D. et al. Open source cloud management platforms and hypervisor technologies: A review and comparison. SoutheastCon, 2016. IEEE, 2016, pp. 1-8.
[10]. Lynn T. et al. A Comparative Study of Current Open-source Infrastructure as a Service Frameworks. CLOSER, 2015, pp. 95-104.
[11]. Openstack Keystone architecture description. Available at: http://docs.openstack.org/developer/ keystone/architecture.html.
[12]. Bogomolov I.V.., Aleksiyants ., Borisenko O.D., Avetisyan A.I. Scalability problems in cloud environments and reasons for performance degradation on identity service Openstack Keystone. Izvestiya SFedU. Engineering Sciences, №12 (185), 2016, pp. 130-140 (in Russian). DOI: 10.18522/2311-3103-2016-12-130140
[13]. Bogomolov I.V., Aleksiyants A.V., Sher A.V., Borisenko O.D., Avetisyan A.I. A performance testing and stress testing of cloud platform central identity: OpenStack Keystone case study. Trudy ISP RAN / Proc. ISP RAS, vol. 27, issue 5, 2015, pp. 49–58 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)-4
[14]. Tarantool project official page. Available at: https://tarantool.org/.
[15]. Abramova V., Bernardino J., Furtado P. Experimental evaluation of NoSQL databases, International Journal of Database Management Systems, No. 3, 2014, pp. 1-16.
[16]. Dmitry Namiot, Manfred Sneps-Sneppe. On Micro-services Architecture. International Journal of Open Information Technologies, vol. 2, no. 9, 2014, pp. 24-27.
[17]. Anton Fagerberg. Optimising clients with API Gateways. Department of Computer Science Faculty of Engineering LTH, 2015.
[18]. Kong project official page. Available at: https://getkong.org.
[19]. Cassandra project official page. Available at: http://cassandra.apache.org.
[20]. Redis project official page. Available at: https://redis.io.
[21]. Nginx project official page. Available at: http://nginx.org.
[22]. Lakshman A., Prashant M. Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review, vol. 4, issue 2, 2010, pp. 35-40.

Исследование максимального размера плотного подграфа случайного графа

[1,2] *Н.Н. Кузюрин <nnkuz@ispras.ru>*
[2] *Д.О. Лазарев <dennis810@mail.ru>*
[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*
[2] *Московский физико-технический институт, 141700, Московская область, г. Долгопрудный, Институтский пер., 9*

Аннотация. Для описания случайных сетей используется модель случайного графа Эрдёша-Реньи $\boldsymbol{G(n,p)}$. При исследовании современных случайных сетей часто требуется определить размер максимальной плотной подсети. В настоящей статье приводятся оценки максимального размера с-плотного подграфа, асимптотически почти наверно содержащегося в случайном графе $G(n,\frac{1}{2})$. Было показано, что при $c<\frac{1}{2}, G(n,\frac{1}{2})$ — асимптотически почти наверно с-плотный; получены верхняя и нижняя оценки размера максимального $\frac{1}{2}$-плотного подграфа, асимптотически почти наверно. содержащегося в $G(n,\frac{1}{2})$; а при $c>\frac{1}{2}$ получена оценка сверху на размер максимального с-плотного подграфа асимптотически почти наверно содержащегося в $G(n,\frac{1}{2})$.

Ключевые слова: случайный граф; случайный граф Эрдёша-Реньи; плотность графа; максимальный плотный подграф.

DOI: 10.15514/ISPRAS-2017-29(6)-12

Для цитирования: Кузюрин Н.Н., Лазарев Д.О. Исследование максимального размера плотного подграфа случайного графа. Труды ИСП РАН, том 29, вып. 6, 2017 г., стр. 213-220. DOI: 10.15514/ISPRAS-2017-29(6)-12

1. Введение

Вероятностный метод — эффективный и широко применяемый математический метод, позволяющий доказывать существование объекта с заданными свойствами. Он заключается в оценке вероятности того, что случайный объект из заданного класса удовлетворяет нужному условию. Если доказано, что эта вероятность положительна, то объект с нужными свойствами существует. Несмотря на явную неконструктивность метода, на его основе могут быть созданы вероятностные алгоритмы построения объекта с параметрами, близкими к параметрам, оцененным с помощью вероятностного метода. Вероятностный метод — сравнительно молодой и динамически развивающийся метод, основоположником которого по праву считается выдающийся венгерский математик Пол Эрдёш. Несмотря на то, что метод применялся и до Эрдёша, например, Селёшем для доказательства существования турнира на n вершинах не менее чем с $\frac{(n-1)!}{2^n}$ гамильтоновыми циклами, именно Пол Эрдёш в полной мере осознал его потенциал, именно ему, его ученикам и соавторам принадлежит большее число доказательств, ставших впоследствии классическими.

Различные разделы науки постоянно ставят всё новые и новые задачи, так или иначе связанные со случайными графами. Впервые случайный граф был определен в рассматриваемой в работе модели Эрдёшем и Реньи в их совместной работе [1]. Данная модель используется и по сей день при исследовании свойств объектов, связанных со случайными сетями.

Широко изучается, в связи с его важной ролью в биологических и социальных сетях, также и понятие плотности графа. Например, в работе [2] приведён алгоритм нахождения плотного подграфа в гигантском графе, вроде графа цитирования в Интернете, а в работе [3] предложен алгоритм нахождения k самых плотных подграфов в динамическом графе.

Ниже приведены основные определения, которые будут многократно использоваться в работе.

Определение 1. Дан граф H с p (p>1) вершинами и q рёбрами. Его плотностью называется величина D, равная отношению количества рёбер графа к числу рёбер в полном графе с числом вершин, равным числу вершин в графе H:

$$D = \frac{2q}{p(p-1)}$$

Определение 2. Случайным графом в модели Эрдёша-Реньи, или просто случайным графом с n вершинами и вероятностью выпадения ребра, равной p, называется вероятностное пространство над пространством всех графов на n вершинах, в котором вероятность события, для любого ребра полного графа на n вершинах заключающегося в том, что оно принадлежит графу, не зависит

от аналогично определённых событий для других рёбер и равна p. Данное пространство обозначается $\boldsymbol{G(n,p)}$.

Определение 3. Скажем, что некоторое свойство выполняется асимптотически почти наверно (а.п.н.) для некоторого семейства случайных графов, если при $n \to \infty$ вероятность того, что граф на n вершинах из данного семейства обладает данным свойством стремится к 1.

Многократно используется в работе следующее неравенство, доказанное русским математиком А. А. Марковым:

Если X — случайная величина, принимающая целые неотрицательные значения, то $P(X > 0) \le E(X)$, где $P\langle Y\rangle$ — вероятность события Y, а $E(X)$ — математическое ожидание случайной величины X.

В настоящей работе исследовался размер максимального с-плотного подграфа случайного графа в модели Эрдёша-Реньи на n вершинах $G(n,\frac{1}{2})$. Было доказано, что

- При $c < \frac{1}{2}, G(n,\frac{1}{2})$ — асимптотически почти наверно с-плотный;
- Получены верхняя и нижняя оценки размера максимального $\frac{1}{2}$-плотного подграфа, а.п.н. содержащегося в $G(n,\frac{1}{2})$;
- При $c > \frac{1}{2}$ получена оценка сверху на размер максимального с-плотного подграфа, а.п.н. содержащегося в $G(n,\frac{1}{2})$.

2. Максимальный размер клики в случайном графе в модели Эрдёша-Реньи

В работе [4] Эрдёшем и Боллобашем исследовался размер максимальной клики в случайном графе $G(n,\frac{1}{2})$. Было установлено, что асимптотически почти наверно данная величина имеет плотную концентрацию.

Обозначим за k(n) верхнюю целую часть решения уравнения

$$\binom{n}{k} 2^{-\binom{k}{2}} = 1;\ k(n) = (1+o(1))\log_2 n$$

Тогда имеет место следующее утверждение:

Теорема.(Erdos and Bollobas, 1976). Размер максимальной клики в $G(n,\frac{1}{2})$ а.п.н. равен либо $k(n)$, либо $k(n)-1$.

Идея доказательства: оценка сверху получается вычислением математического ожидания числа клик размера k графа $G(n,\frac{1}{2})$; для оценки снизу требуется воспользоваться методом второго момента.

3. Оценки размера максимального c-плотного подграфа случайного графа G(n,c) при различных значениях c

Обозначим за $k(n,c)$, при $c \ge \frac{1}{2}$ наименьшее целое решение неравенства:

$$\binom{n}{k}\Pr\left\langle Bi\left(\binom{k}{2},\frac{1}{2}\right) \ge c\binom{k}{2}\right\rangle \ge 1,$$

где $Bi(n,p)$ — биномиальное распределение с параметрами n и p, а $\Pr\langle x\rangle$ — вероятность события X.

Теорема.

1) При $c < \frac{1}{2}$ сам случайный граф а.п.н. c-плотный.

2) При $c = \frac{1}{2}$ выполняется:

а) При $k = \frac{n}{f(n)}$, где $f(n) \to \infty$ произвольно медленно при $n \to \infty$,

граф $G(n,\frac{1}{2})$ а.п.н. содержит $\frac{1}{2}$- плотный подграф размера k.

б) При $k = \frac{n}{4} - \sqrt{\frac{n\ln n}{2}} - \sqrt{\frac{n}{\ln n}}\ln\ln n - \sqrt{\frac{n}{32\ln n}}\ln\ln n$

$\exists \varepsilon = \frac{1}{3}, \exists N_0 : \forall n > N_0$, верно: $\Pr\left\langle G(n,\frac{1}{2})\ содержит\ \frac{1}{2} - плотный\ подграф\ размера\ k \right\rangle > \varepsilon$

3) При $\frac{1}{2} < c < 1$, размер максимального c-плотного подграфа графа $G(n,\frac{1}{2})$ а.п.н. не превосходит $k(n,c)+1$.

Доказательство.

1) Оценив соотношение соседних биномиальных коэффициентов $\binom{n}{k}$ и $\binom{n}{k+1}$, можно установить верность следующей леммы:

Лемма.

$$\Pr\left\langle Bi\left(n,\frac{1}{2}\right) \ge cn\right\rangle \in \left[2^{-n}\binom{n}{cn}\frac{1-c}{c},\ 2^{-n}\binom{n}{cn}\right] \blacksquare$$

Пусть $m = \binom{n}{2}$. Число рёбер в случайном графе $G(n,p)$ имеет биномиальное распределение $Bi(m,p)$. Используя равенство $\binom{n}{k} = \binom{n}{n-k}$ и лемму, можно получить:

$$\Pr\left\langle Bi(m,\frac{1}{2}) \geq cm \right\rangle = 1 - \Pr\left\langle Bi(m,\frac{1}{2}) \geq (1-c)m \right\rangle \to 0,\ m \to \infty$$

Следовательно, $G(n,\frac{1}{2})$ — а.п.н. с-плотный.

2) а) Заметим, что

$$\Pr\left\langle плотность G(n,\frac{1}{2}) \geq \frac{1}{2} \right\rangle \geq \frac{1}{2}$$

Разобьём вершины графа $G(n,\frac{1}{2})$ на $[f(n)]$ множеств размера либо $\frac{n}{f(n)}]$, либо $\left[\frac{n}{f(n)}\right] + 1$. $\left[\frac{n}{f(n)}\right] + 1$. Вероятность того, что каждый из подграфов, порождённых этими множествами вершин, имеет плотность, меньшую $\frac{1}{2}$, не превосходит $\frac{1}{2}$. Стало быть, вероятность того, что ни один из подграфов не имеет плотность $D > \frac{1}{2}$ не превосходит $2^{-f(n)} \to 0,\ n \to \infty$. Следовательно, граф $G(n,\frac{1}{2})$ а.п.н. содержит $\frac{1}{2}$-плотный подграф размера k, что и требовалось доказать.

б) В работе [5] Боллобаш, в частности, показывает, что, если за A_n обозначить событие, заключающееся в том, что степень каждой вершины $G(n,\frac{1}{2})$ — не менее, чем $\frac{n}{2} - \sqrt{\frac{n \ln n}{2}} - \sqrt{\frac{n}{\ln n}} \ln\ln n - \sqrt{\frac{n}{32 \ln n}} \ln\ln n$, то $\Pr\langle A_n \rangle \to 1$ при $n \to \infty$; также известно, что

$$\Pr\left\langle D\left(G\left(n,\frac{1}{2}\right)\right) \leq \frac{1}{2} \right\rangle \geq \frac{1}{2},$$

следовательно $\exists N_0$:

$$\Pr\left\langle \left(\text{ни одна вершина } G\left(n,\frac{1}{2}\right) \text{ не имеет степень меньше } \frac{n}{2} - \sqrt{\frac{n\ln n}{2}} - \sqrt{\frac{n}{\ln n}} \ln\ln\ln n - \sqrt{\frac{n}{32\ln n}} \ln \right.\right.$$

$$\left.\left. \left(D\left(G\left(n,\frac{1}{2}\right)\right) \leq \frac{1}{2}\right) \right\rangle$$

для всех $n > N_0$. Рассмотрим подмножество H множества вершин графа $G(n,\frac{1}{2})$ размером не менее $M = \frac{3n}{4} + \sqrt{\frac{n\ln n}{2}} + \sqrt{\frac{n}{\ln n}}\ln\ln\ln n + \sqrt{\frac{n}{32\ln n}}\ln\ln n$ и дополнительное к нему множество H'. Из любой вершины H' в H идёт по крайней мере $\frac{n}{4}$ рёбер. Значит, суммарное число рёбер в подграфах $G(n,\frac{1}{2})$,порождённых множествами вершин H и H' не превосходит

$$\frac{n(n-1)}{4} - \frac{(n-M)n}{4} = \frac{n(M-1)}{4} \le \frac{(n-M-1)(n-M)}{4} + \frac{M(M-1)}{4}$$

Значит, либо плотность подграфа $G(n,\frac{1}{2})$, порождённого множеством вершин H менее $\frac{1}{2}$, либо плотность подграфа $G(n,\frac{1}{2})$, порождённого множеством вершин H'менее $\frac{1}{2}$, что и требовалось доказать.

3) Вычислим $E(n,k,c)$ – математическое ожидание количества порождённых подграфов размера *k* случайного графа $G(n,\frac{1}{2})$, имеющих плотность, не меньше *c*:

$$E(n,k,c) = \binom{n}{k}\Pr\langle \textit{плотность случайного графа на k вершинах} \ge c\rangle$$

$$= \binom{n}{k}\Pr\left\langle Bi\left(\binom{k}{2},\frac{1}{2}\right) \ge c\binom{k}{2}\right\rangle$$

Решая уравнение $E(n,k,c) = 1$, и учитывая, что, по Лемме(обозначим $\binom{k}{2}$за m) $\Pr\left\langle Bi\left(m,\frac{1}{2}\right) \ge cm\right\rangle \in \left[2^{-m}\binom{m}{cm}\frac{1-c}{c}, 2^{-m}\binom{m}{cm}\right]$, получаем:

$$k(n,c) = (1+o(1))2\log_{2c^c(1-c)^{(1-c)}} n.$$

Для нахождения верхней оценки размера максимального с-плотного подграфа случайного графа, учитывая соотношение, которое может быть получено анализом выражения $E(n,k,c)$:

$$\frac{E(n,k(n,c)+1,c)}{E(n,k(n,c),c)} < \frac{1}{n},$$

выполняющееся для любого $C > \frac{1}{2}$ при любом $n > N_c$, можем получить, что $E(n,k(n,c)+1,c) < \frac{1}{n}$ E(n, k(n, c) + 1, c) < $\frac{1}{n}$. Стало быть, по неравенству Маркова,

$$\Pr\left\langle \textit{существует подграф } G\left(n,\frac{1}{2}\right) \textit{ на } k(n,c)+1 \textit{ вершинах}\right\rangle \to 0,$$

при $n \to \infty$. ■

Стоит отметить, что, так как $2c^c(1-c)^{1-c} \to 2$ при $c \to 1$, то при $c = 1$ настоящий результат совпадает с результатом, полученным Боллобашем в работе [4]: $k = (1+o(1))2\log_2 n$ для плотности подграфа, равной 1.

Список литературы

[1]. P.Erdos and A. Renyi. On Random Graphs. Publicationes Mathematicae (Debrecen), Vol. 6, 1959, pp. 290-297.

[2]. D.Gibson, R.Kumar and A.Tomkins. Discovering Large Dense Subgraphs in Massive Graphs. Proceedings of the 31st international conference on Very large data bases, 2005, pp. 721-732**.**

[3]. Valari E., Kontaki M., Papadopoulos A.N. Discovery of Top-k Dense Subgraphs in Dynamic Graph Collections. In: Ailamaki A., Bowers S. (eds) Scientific and Statistical Database Management. SSDBM 2012, Lecture Notes in Computer Science, vol 7338. Springer, Berlin, Heidelberg, pp 213-230, DOI: 10.1007/978-3-642-31235-9_14.

[4]. B. Bollobas and P. Erdos. Cliques in Random Graphs. Mathematical Proceedings of the Cambridge Philosophical Society, Volume 80, Issue 3, 1976, pp. 419-427, DOI: 10.1017/S0305004100053056.

[5]. B. Bollobas. Degree sequences in Random Graphs. Discrete Mathematics, Volume 33, Issue 1, 1981, pp. 1-19, DOI: 10.1016/0012-365X(81)90253-3.

Analysis of size of the largest dense subgraph of random hypergraph

[1,2] *N.N. Kuzyrin <nnkuz@ispras.ru>*

[2] *D.O. Lazarev <dennis810@mail.ru>*

[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia*

[2] *Moscow Institute of Physics and Technology , Dolgoprudnyj, Institutskij alley, Moscow region, 141700, Russia*

Abstract. Random networks are often described using Erdos-Renyi model of random graph $\boldsymbol{G(n,p)}$. The concept of graph density is often used in random network analysis**.** In this article, the maximal size of c-dense subgraph almost surely included in random graph $G(n,\frac{1}{2})$ was evaluated. It was shown, that if $c < \frac{1}{2}$, then $G(n,\frac{1}{2})$ is almost surely c-dense; the upper and lower bounds for the size of maximal $\frac{1}{2}$-dense subgraph almost surely included in $G(n,\frac{1}{2})$ were determined**;** in case when $c > \frac{1}{2}$, the upper bound for the maximal size of c-dense subgraph almost surely included in $G(n,\frac{1}{2})$ was attained**.**

Keywords: random graph; Erdos-Renyi model; graph density; maximal dense subgraph.

DOI: 10.15514/ISPRAS-2017-29(6)-12

For citation Kuzyrin N.N., Lazarev D.O. Analysis of size of the largest dense subgraph of random hypergraph. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 6, 2017. pp. 213-220 (in Russian). DOI: 10.15514/ISPRAS-2017-29(6)-12

References

[1]. P.Erdos and A. Renyi. On Random Graphs. Publicationes Mathematicae (Debrecen), Vol. 6, 1959, pp. 290-297.

[2]. D.Gibson, R.Kumar and A.Tomkins. Discovering Large Dense Subgraphs in Massive Graphs. Proceedings of the 31st international conference on Very large data bases, 2005, pp. 721-732**.**

[3]. Valari E., Kontaki M., Papadopoulos A.N. Discovery of Top-k Dense Subgraphs in Dynamic Graph Collections. In: Ailamaki A., Bowers S. (eds) Scientific and Statistical Database Management. SSDBM 2012, Lecture Notes in Computer Science, vol 7338. Springer, Berlin, Heidelberg, pp 213-230, DOI: 10.1007/978-3-642-31235-9_14.

[4]. B. Bollobas and P. Erdos. Cliques in Random Graphs. Mathematical Proceedings of the Cambridge Philosophical Society, Volume 80, Issue 3, 1976, pp. 419-427, DOI: 10.1017/S0305004100053056.

[5]. B. Bollobas. Degree sequences in Random Graphs. Discrete Mathematics, Volume 33, Issue 1, 1981, pp. 1-19, DOI: 10.1016/0012-365X(81)90253-3.

Алгоритм упаковки прямоугольников в несколько полос и анализ его точности в среднем[1]

[2] *Д.О. Лазарев <dennis810@mail.ru>*
[1,2] *Н.Н. Кузюрин <nnkuz@ispras.ru>*
[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*
[2] *Московский физико-технический институт, 141700, Московская область, г. Долгопрудный, Институтский пер., 9*

Аннотация. В 2012 году М. А. Трушников предложил принципиально новый онлайновый алгоритм упаковки прямоугольников в полосу, а в 2013 году была получена оценка точности алгоритма в среднем, равной математическому ожиданию незаполненной прямоугольниками площади, равная $\boldsymbol{O}(\sqrt{\boldsymbol{N}}(\boldsymbol{lnN})^{\frac{3}{2}})$. Ранее известная оценка $\boldsymbol{O}(\boldsymbol{N}^{\frac{2}{3}})$ была существенно улучшена. В настоящей работе данная оценка улучшена до $\boldsymbol{O}(\sqrt{\boldsymbol{N}}\boldsymbol{lnN})$, а также алгоритм Трушникова обобщён на случай упаковки $\boldsymbol{N}$ прямоугольников в $\boldsymbol{k}$ полос, $\boldsymbol{k} \leq \sqrt{\boldsymbol{N}}$, с сохранением оценки $\boldsymbol{O}(\sqrt{\boldsymbol{N}}\boldsymbol{lnN})$.

Ключевые слова: новая эвристика, задача упаковки прямоугольников в полосу, задача упаковки прямоугольников в несколько полос, оценка в среднем.

DOI: 10.15514/ISPRAS-2017-29(6)-13

Для цитирования: Лазарев Д.О., Кузюрин Н.Н. Алгоритм упаковки прямоугольников в несколько полос и анализ его точности в среднем. Труды ИСП РАН, том 29, вып. 6, 2017 г., стр. 221-228. DOI: 10.15514/ISPRAS-2017-29(6)-13

1. Введение

Задача упаковки прямоугольников в несколько полос, в англоязычной литеретуре называемая Multiple Strip Packing Problem, заключается в упаковке открытых прямоугольников в некоторое число вертикальных полубесконечных полос единичной ширины. Основания полос — отрезки единичной длины, лежащие на одной горизонтальной прямой с ординатой, равной нулю. Прямоугольники не пересекаются, их стороны параллельны

[1] Работа выполнена при финансовой поддержке РФФИ, проект № 17-07-01006 А

сторонам полос, вращения запрещены. Требуется минимизировать высоту h упаковки — ординату самой высокой точки, принадлежащей замыканию одного из прямоугольников.

Практическая важность задачи обуславливается следующей её интерпретацией: каждый прямоугольник представляет собой задачу, вычисляемую на кластере. Каждая задача вычисляется на некотором вычислительном устройстве, то есть каждый прямоугольник кладётся в некую полосу. Высота прямоугольника — время выполнения задачи, а ширина — отношение мощности вычислительных ресурсов, задействованных для решения данной задачи к общей доступной на данном вычислительном устройстве мощности вычислительных ресурсов.

Будем рассматривать онлайновые алгоритмы, размещающие очередной прямоугольник в полосы без знания параметров выпадающих за ним прямоугольников. Предположим, что общее число прямоугольников известно алгоритму до начала работы. Пусть значения длин сторон всех прямоугольников – независимые в совокупности величины, имеющие равномерное распределение на [0,1]. Анализ алгоритма производим в среднем, т.е. минимизируем $E(S_{sp})$ — математическое ожидание площади полос, не заполненной прямоугольниками на высоте от 0 до h, где h — высота упаковки.

Задача упаковки прямоугольников в одну полосу, называемая в англоязычной литературе Strip Package Problem, хорошо исследована и для неё было построено множество эвристик. Известно, что для оптимальной упаковки в данной задаче $E(S_{sp}) = \Omega(\sqrt{N})$. В 1993 году в [1] Коффманом и Шором был предложен оффлайновый алгоритм упаковки прямоугольников с $E(S_{sp}) = O(\sqrt{N})$. Там же был предложен онлайновый алгоритм с $E(S_{sp}) = O(N^{\frac{2}{3}})$.

В 2012 году в [2] Трушниковым был предложен принципиально новый онлайновый алгоритм, а в следующем году в [3] было показано, что данный алгоритм работает лучше всех ранее известных алгоритмов: для него $E(S_{sp}) = O(\sqrt{N}(lnN)^{\frac{3}{2}})$.

Задача упаковки прямоугольников в несколько полос была рассмотрена С.Н. Жуком в работе [4], однако в ней рассматривается оценка точности алгоритма для худшего случая.

В настоящей работе исследовались алгоритмы, работающие по тому же принципу, что и алгоритм Трушникова. Для них улучшена ранее доказанная в работе [3] оценка: $E(S_{sp}) = O(\sqrt{N}lnN)$.

Также данная оценка обобщена на случай k полос, где $k \leq \sqrt{N}$.

2. Модифицированное доказательство Трушникова

В данном разделе мы изучаем алгоритмы онлайновой упаковки в полубесконечную полосу ширины w $(w \geq 1)$ N прямоугольников, длины

сторон которых имеют равномерное распределение на отрезке $[0,1]$. Считаем, что число N изначально известно алгоритму, и пусть параметр $d = d(N)$.

Предположим, что удалось без остатка свободного пространства разбить нижнюю часть полосы высотой $\frac{N}{4w}$ на d контейнеров шириной $\frac{i}{d+1}, i = \overline{1,d}$ и высотой $U = \frac{N}{2wd}$.

Рассмотрим алгоритм типа алгоритма Трушникова: прямоугольник кладём в контейнер минимальной ширины, в который он помещается; если же он не помещается, то назовём его прямоугольником переполнения. Нижнюю горизонтальную сторону такого прямоугольника кладём на самую высокую верхнюю грань предыдущего прямоугольника переполнения. Первый прямоугольник переполнения кладём нижней стороной на верхнюю сторону верхнего контейнера.

Тогда верно следующее утверждение: если $\exists N_0 : \forall N > N_0$ верно: $\exists C > c > 0 : c\sqrt{N} < d(N) < C\sqrt{N}$, то алгоритм данного типа даёт математическое ожидание незаполненной площади полосы снизу до верхней грани верхнего контейнера, равное $E(S_{sp}) = O(\sqrt{N} ln N)$.

Доказательство во многом повторяет оценку Трушникова $E(S_{sp})$ для его алгоритма в случае $w = 1$, полученную в работе [3].

Вначале заметим, что число прямоугольников с шириной $\geq \frac{d-1}{d}$ имеет мат. ожидание, равное $O(\sqrt{N})$.

В работе [3] Трушниковым была доказана следующая Лемма:

Лемма 1. Пусть случайная величина $X = \sum_{i=1}^{d} X_i$, где $X_i = \xi_i \eta_i$, где ξ_i принимает значения 1 с вероятностью p и 0 с вероятностью $1-p$, η_i- равномерно распределена на отрезке $[0,1]$. Также все η_i и ξ_j- независимы в совокупности. Тогда $\forall a \in (0,1)$ выполнено неравенство:

$$P\{X > (1+a)EX\} \leq e^{-\frac{5}{9}a^2 EX}$$

Рассмотрим динамический процесс выпадения прямоугольников.

Лемма 2. Пусть выпало $N - 2N^{\frac{3}{4}}\sqrt{2ClnN}$ прямоугольников. Тогда

$$E\{\textit{мощности переполнения}\} \to 0 (n \to \infty).$$

Доказательство. Покажем, что $Pr\{$в полосу попало несколько прямоугольников суммарной высотой более $U-1\} < \frac{1}{N^{2.1}}$. Отсюда получим, что $\Pr\{\textit{переполнение имеет место}\} < \frac{1}{N^{1.1}}$. А значит, $E(\text{переполнения}) < \frac{1}{N^{0.1}} \to 0 (N \to \infty)$.

Зафиксируем полосу. Найдём вероятность того, что в неё попало несколько прямоугольников суммарной высотой более чем $U-1$. Если это не так, то переполнения в полосе в отсутствии переполнений в других полосах точно нет.

Вычислим a из условия $U-1=E(X)(1+a)$. $E(X)=\frac{N-2N^{\frac{3}{4}}\sqrt{2ClnN}}{2d(N)}$.

В результате элементарных преобразований, получим, что $a=\frac{2N^{\frac{3}{4}}\sqrt{2ClnN}}{2N-2N^{\frac{3}{4}}\sqrt{2ClnN}}=$

$=(1+o(1))2N^{-\frac{1}{4}}\sqrt{2ClnN}$.

$$\exp\{-\frac{5}{9}a^2E(X)\}=\exp\{-\frac{5}{9}*4N^{-\frac{1}{2}}2c\ln N(\frac{1}{2}\frac{N}{d(N)})(1+0(1))\}<N^{-2.1}$$

для достаточно больших N. ■

Лемма 3. Пусть упаковали $N-\lfloor N^{\frac{1}{2}+b}\rfloor$ прямоугольников, $\frac{lnln10N}{lnN}\le b\le\frac{1}{4}$. Зафиксируем $\gamma=2b-\frac{ln(10lnN)}{lnN}$. Тогда вероятность того, что каждая из $\lfloor dN^{-\gamma}\rfloor$ самых широких полос заполнена до высоты $U-1$ равна $O(N^{-1.1})$.

Доказательство. Сначала покажем, что вероятность того, что в любые $\lfloor dN^{-\gamma}\rfloor$ подряд идущих контейнеров выпали прямоугольники суммарной высотой $\ge(U-1)\lfloor dN^{-\gamma}\rfloor$ равна $O(N^{-2.1})$. Далее заметим, что в каждые подряд идущие группы по $\lfloor dN^{-\gamma}\rfloor$ контейнеров выпали прямоугольники суммарной высотой $(U-1)\lfloor dN^{-\gamma}\rfloor$ с вероятностью $\le N^{-1.1}$. Предположим, что s самых широких контейнеров заполнены выше, чем на $U-1$, где $s\ge\lfloor dN^{-\gamma}\rfloor$. Рассмотрим множество M, состоящее из $\lfloor dN^{-\gamma}\rfloor$ самых узких из них. Так как высота заполнения $(s+1)$-ого по ширине контейнера ниже, чем $U-1$, то в M попали прямоугольники суммарной высотой $\ge(U-1)\lfloor dN^{-\gamma}\rfloor$. Получили противоречие. Значит, хотя бы один из самых широких $\lfloor dN^{-\gamma}\rfloor$ контейнеров заполнен ниже высоты $U-1$.

Итак, зафиксируем $\lfloor dN^{-\gamma}\rfloor$ контейнеров. Пусть высота X выпавших в них прямоугольников превосходит $(U-1)\lfloor dN^{-\gamma}\rfloor$. Оценим вероятность этого события с помощью Леммы 1.

$$EX=\frac{N-\lfloor N^{\frac{1}{2}+b}\rfloor}{2d}\lfloor dN^{-\gamma}\rfloor$$

Найдём a из условия $(1+a)EX=(U-1)\lfloor dN^{-\gamma}\rfloor$: $a=\frac{\lfloor N^{\frac{1}{2}+b}\rfloor-2d}{N-\lfloor N^{\frac{1}{2}+b}\rfloor}\ge\frac{N^{\frac{1}{2}+b}-2d-1}{N-N^{\frac{1}{2}+b}+1}$.

И из леммы 1) вероятность того, что контейнеры заполнены выше $U-1$ не превосходит

$$exp(-\frac{5}{9}(\frac{N^{\frac{1}{2}+b}-2d-1}{N-N^{\frac{1}{2}+b}+1})^2\frac{(N-2d)\lfloor dN^{\gamma}\rfloor}{d})\le exp(-\frac{4}{9}\frac{N^{2+2b-\gamma}}{N^2})\le\frac{1}{N^{-2.1}}$$

для достаточно больших N, что и требовалось доказать. ■

Назовём отрезком $[S,T], T > S$ множество прямоугольников, выпавших строго после -го и до T-го прямоугольника включительно.

Мы получим требуемую оценку, если покажем, что мат. ожидание числа прямоугольников из отрезка $[N - 2N^{\frac{3}{4}}\sqrt{2ClnN}, N - \sqrt{N}]$, попавших в переполнение равно $O(\sqrt{N}lnN)$. Это и доказывается в следующей лемме.

Лемма 4. Мат. ожидание числа прямоугольников из отрезка $[N - 2N^{\frac{3}{4}}\sqrt{2ClnN}, N - \sqrt{N}]$, попавших в переполнение равно $O(\sqrt{N}lnN)$.

Рассмотрим 1-ый отрезок: $[N - N^{\frac{1}{2}+\frac{lnln10N}{lnN}}, N - N^{\frac{1}{2}}]$. В него попало не более, чем $N^{\frac{1}{2}+\frac{lnln10N}{lnN}} = exp(lnln10N)\sqrt{N} = O(\sqrt{N}lnN)$ прямоугольников.

Рассмотрим 2-ой отрезок: $[N - N^{\frac{1}{2}+2\frac{lnln10N}{lnN}}, N - N^{\frac{1}{2}+\frac{lnln10N}{lnN}}]$. В него попало не более, чем $N^{\frac{1}{2}+2\frac{lnln10N}{lnN}}$прямоугольников. Вероятность же для прямоугольника не попасть в один из $dN^{-\gamma} = dN^{-\frac{lnln10N}{lnN}}$самых широких контейнеров, и, таким образом, попасть в переполнение с вероятностью не более $\frac{1}{N^{1.1}}$, равна $N^{-\frac{lnln10N}{lnN}}$. Таким образом,

$$E\{\textit{числа прямоугольников из данного отрезка в переполнении}\} = O(N^{\frac{1}{2}+\frac{\ln\ln 10N}{\ln N}}) = O(\sqrt{N}\ln N)$$

Зафиксируем $i_{max} = [\frac{1}{4}lnN/lnln10N] + 1$.

Рассмотрим -ый отрезок, где $(j = \overline{3, i_{max}})$: $[N - N^{\frac{1}{2}+j\frac{lnln10N}{lnN}}, N - N^{\frac{1}{2}+(j-1)\frac{lnln10N}{lnN}}]$. В него попало не более, чем $N^{\frac{1}{2}+j\frac{lnln10N}{lnN}}$прямоугольников. Вероятность же для прямоугольника не попасть в один из $dN^{-\gamma} = dN^{-(2j-3)\frac{lnln10N}{lnN}}$ самых широких контейнеров, и, таким образом, попасть в переполнение с вероятностью не более $\frac{1}{N^{1.1}}$, равна $N^{-(2j-1)\frac{lnln10N}{lnN}}$. Таким образом, $E\{$числа прямоугольников из данного отрезка в переполнении$\} = O(N^{\frac{1}{2}}) = O(\sqrt{N})$. Всего же отрезков — $O(lnN) \Rightarrow$ вклад данной области составляет $O(\sqrt{N}lnN)$.

Рассмотрим последний отрезок: $[N - 2N^{\frac{3}{4}}\sqrt{2ClnN}, N - N^{\frac{3}{4}}]$. В него попало не более, чем $2\sqrt{2ClnN}N^{\frac{3}{4}}$ прямоугольников. Вероятность же для прямоугольника не попасть в один из $dN^{-\gamma} <$ (для достаточно больших N) $< dN^{-\frac{3}{8}}$ самых широких контейнеров, и, таким образом, попасть в переполнение с вероятностью не более $\frac{1}{N^{1.1}}$, не более, чем $N^{-\frac{3}{8}}$. Таким образом, $E\{$числа

прямоугольников из данного отрезка в переполнении$\} = O(N^{\frac{1}{2}}) = O(\sqrt{N}lnN)$. ■

Лемма доказана, а с ней получена требуемая оценка.

3. Упаковка прямоугольников в большое число полос

3.1 Постановка задачи

Пусть дана функция $k = k(N)$. Требуется упаковать N прямоугольников в k полубесконечных полос единичный ширины в онлайновом режиме. Длины сторон прямоугольников — случайные величины, равномерно распределённые на отрезке $[0,1]$. Число прямоугольников N известно до выпадения первого прямоугольника.

Качество алгоритма упаковки будем оценивать, используя величину $E(S_{sp})$- мат. ожидание площади полос, незаполненной прямоугольниками, в части полос, начинающейся ото дна и заканчивающейся на самой большой высоте верхнего основания прямоугольника, помещённого в полосу.

3.2 Алгоритм А

Пусть $k(N)^2 \le N$для всех $N \ge N_0$. Зафиксируем некоторые $N > N_0$ и $k(N) = k$. За dобозначим число, равное $2k[\frac{\sqrt{N}}{k}]$. Разделим нижнюю область 1-ой полосы высотой $\frac{N}{4d}$ на два контейнера шириной $\frac{1}{d+1}$ и $\frac{d}{d+1}$. Над данной областью выделим следующую область той же высоты, которую разобьём на 2 контейнера шириной $\frac{2}{d+1}$ и $\frac{d-1}{d+1}$. Аналогично выделим ещё несколько областей той же высоты, чтобы всего их число стало $[\frac{\sqrt{N}}{k}]$, и каждую из полос разобьём на 2 контейнера так, чтобы два контейнера, граничащих по вертикальной стороне, дополняли бы друг друга до области и ширина узких контейнеров из области бы равнялась $\frac{i}{d+1}, i = \overline{1,[\frac{\sqrt{N}}{k}]}$.

Аналогично разобьём вторую и следующие за ней области до высоты $\frac{N}{4k}$, причём ширина самых узких контейнеров из области будет равна $\frac{(j-1)[\frac{\sqrt{N}}{k}]+i}{d+1}, i = \overline{1,[\frac{\sqrt{N}}{k}]}$.

В эти контейнеры будем укладывать прямоугольники следующим образом:

1) Прямоугольник кладётся в контейнер наименьшей ширины, в который он помещается.

2) Если же он никуда не помещается, так как все контейнеры, ширина которых больше ширины прямоугольника, не могут его вместить, то

назовём прямоугольник прямоугольником переполнения 1-го рода. Суммарную высоту всех таких прямоугольников обозначим за h_1.

3) Если же ширина прямоугольника больше, чем $\frac{d-1}{d}$, то назовём прямоугольник прямоугольником переполнения 2-го рода. Суммарная высота этих прямоугольников равна h_2.

4) Начнём последовательно упаковывать прямоугольники по полосам. Высотой полосы назовём величину, равную высоте самой высокой точки, принадлежащей замыканию одного из прямоугольников или контейнеров, лежащих в данной полосе. Предполагая, что разбиение полос на контейнеры выполнено до выпадения первого прямоугольника, очередной прямоугольник переполнения кладётся в одну из полос с минимальной высотой на верхнюю грань самого высокого алгоритма.

3.3 Оценка математического ожидания площади свободного пространства

Докажем, что $E(S_{sp}) = O(\sqrt{N}lnN)$. Очевидно следующее утверждение и следствие из него:

Утверждение. Если $b \le a$, то $\frac{a}{2} \le b[\frac{a}{b}] \le a$.

Следствие. $\sqrt{N} \le 2k[\frac{\sqrt{N}}{k}] \le 2\sqrt{N}$.

Заметим, что алгоритм A_1 упаковки в 1 полосу, пакующий прямоугольники в те же контейнеры, что и описанный ранее алгоритм, а прямоугольники переполнения кладущий один поверх другого, является алгоритмом типа алгоритма Трушникова и удовлетворяет требованиям из модифицированного доказательства Трушникова, а значит верно следующее:

1) E{площади незаполненного прямоугольниками пространства внутри контейнеров} = $O(\sqrt{N}lnN)$.

2) $E(h_1 + h_2) = O(\sqrt{N}lnN)$.

Теперь рассмотрим алгоритм A упаковки во много полос. Обозначим за Hразность высот самой высокой точки прямоугольников переполнения и высоты верхней стороны верхнего контейнера, равной $\frac{N}{4k}$. Если в переполнение не попали прямоугольники, то данная величина равна 0.

Так как любая полоса, по построению, заполнена не ниже $H-1$, то $(H-1)k \le h_1 + h_2$. Стало быть, $E(kH) = O(\sqrt{N}lnN)$.

Для алгоритма A: $E(S_{sp}) \le$E{площади незаполненного прямоугольниками пространства внутри контейнеров} + $Hk = O(\sqrt{N}lnN)$, что и требовалось доказать.

Список литературы

[1]. Coffman E.G., Jr, Shor P.W. Packing in two dimensions: Asymptotic average-case analysis of algotithms. Algorithmica, vol. 9, No 3, pp. 253-277
[2]. М.А. Трушников. Об одной задаче Коффмана-Шора, связанной с упаковкой прямоугольников в полосу. Труды ИСП РАН, том 22, 2012, стр. 456-462. DOI: 10.15514/ISPRAS-2012-22-24
[3]. М.А. Трушников. Вероятностный анализ нового алгоритма упаковки прямоугольников в полосу. Труды ИСП РАН, том 24, 2013, стр. 457-468, DOI: 10.15514/ISPRAS-2013-24-21
[4]. С.Н. Жук. Анализ некоторых эвристик в задаче упаковки прямоугольников в несколько полос. Труды ИСП РАН, том 6, 2005, стр. 13-26

An algorithm for Multiple Strip Package and its average case evaluation

[2] *D.O. Lazarev <dennis810@mail.ru>*
[1,2] *N. N. Kuzyrin <nnkuz@ispras.ru>*
[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia*
[2] *Moscow Institute of Physics and Technology, Dolgoprudnyj, Institutskij alley, Moscow region, 141700, Russia*

Abstract. In 2012 M.A. Trushnikov suggested a new online method for 2DSP Problem. The average case evaluation for a 2DSP algorithm equals the expected value of space of strip not filled with rectangles. In 2013 the average case evaluation for this method was attained and it equaled $\boldsymbol{O}(\sqrt{\boldsymbol{N}}(\boldsymbol{lnN})^{\frac{3}{2}})$. The best known before evaluation $\boldsymbol{O}(\boldsymbol{N}^{\frac{2}{3}})$ was improved. In present article this evaluation was improved to $\boldsymbol{O}(\sqrt{\boldsymbol{N}}\boldsymbol{lnN})$. Also a new method was constructed for MSP Problem, where $\boldsymbol{N}$ rectangles are packed in $\boldsymbol{k}$ strips, $\boldsymbol{k} \leq \sqrt{\boldsymbol{N}}$, with average case evaluation equaling $\boldsymbol{O}(\sqrt{\boldsymbol{N}}\boldsymbol{lnN})$.

Keywords: new heuristic, Strip Packing problem, Multiple Strip Packing problem, average case evaluation.

DOI: 10.15514/ISPRAS-2017-29(6)-13

For citation Lazarev D.O., Kuzyrin N.N. An algorithm for Multiple Strip Package and its average case evaluation. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 6, 2017. pp. 221-228 (in Russian). DOI: 10.15514/ISPRAS-2017-29(6)-13

References

[1]. Coffman E.G., Jr, Shor P.W. Packing in two dimensions: Asymptotic average-case analysis of algotithms. Algorithmica, vol. 9, No 3, pp. 253-277
[2]. M. A. Trushnikov. On one problem of Koffman-Shor connected to strip packing problem. Trudy ISP RAN/Proc. ISP RAS, vol. 22, 2012, pp. 456-462 (in Russian). DOI: 10.15514/ISPRAS-2012-22-24
[3]. M. A. Trushnikov. Probabilistic analysis of a new strip packing algorithm. Trudy ISP RAN/Proc. ISP RAS, vol. 24, 2013, pp. 457-468 (in Russian). DOI: 10.15514/ISPRAS-2013-24-21
[4]. S.N. Zhuk. Analysis of some heuristics in Multiple Strip Package Problem. Trudy ISP RAN/Proc. ISP RAS, vol. 6, 2004, pp. 13-26 (in Russian)

Задачи оптимизации размещения контейнеров MPI-приложений на вычислительных кластерах[1]

[1] *Д.А. Грушин <grushin@ispras.ru>*
[1,2] *Н.Н. Кузюрин <nnkuz@ispras.ru>*
[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*
[2] *Московский физико-технический институт, 141700, Московская область, г. Долгопрудный, Институтский пер., 9*

Аннотация. MPI — это хорошо зарекомендовавшая себя технология, которая широко используется в высокопроизводительной вычислительной среде. Однако настройка кластера MPI может быть сложной задачей. Контейнеры — это новый подход к виртуализации и простой упаковке приложений, который становится популярным инструментом для высокопроизводительных задач (HPC). Именно этот подход рассматривается в данной статье. Упаковка MPI-приложения в виде контейнера решает проблему конфликтных зависимостей, упрощает конфигурацию и управление запущенными приложениями. Для управления ресурсами кластера может использоваться обычная система очередей (например, SLURM) или системы управления контейнерами (Docker Swarm, Kubernetes, Mesos и др.). Контейнеры также дают больше возможностей для гибкого управления запущенными приложениями (остановка, повторный запуск, пауза, в некоторых случаях миграция между узлами), что позволяет получить преимущество при оптимизации размещения задач по узлам кластера по сравнению с классической схемой работы планировщика. В статье рассматриваются различные способы оптимизации размещения контейнеров при работе с HPC-приложениями. Предлагается вариант запуска MPI приложений в системе Fanlight, упрощающий работу пользователей. Рассматривается связанная с данным способом запуска задача оптимизации.

Ключевые слова: контейнеры, docker, оптимизация

DOI: 10.15514/ISPRAS-2017-29(6)-14

Для цитирования: Грушин Д.А., Кузюрин Н.Н. Задачи оптимизации размещения контейнеров MPI-приложений на вычислительных кластерах. Труды ИСП РАН, том 29, вып. 6, 2017 г., стр. 229-244. DOI: 10.15514/ISPRAS-2017-29(6)-14

[1] Работа выполнена в рамках гранта РФФИ 17-07-01006 А

1. Введение

Высокопроизводительные вычисления (HPC) – это важная область в информатике, которая предполагает решение больших задач, требующих высокой вычислительной мощности. Эти задачи слишком велики для одного компьютера, поэтому для HPC используют группу машин, которые работают вместе. Существуют различные модели параллельного программирования, разработанные для эффективной координации кластера компьютеров. Одной из популярных моделей является передача сообщений для оперативной связи и совместных вычислений.

В настоящее время MPI является «де-факто» стандартом передачи сообщений в HPC [1]. MPI имеет устойчивость к аппаратным или сетевым ошибкам, обеспечивает гибкий механизм для разработчиков распределенной программы или инструментов поверх нее.

MPI – это библиотека функций, которые можно вызывать из разных языков программирования для обеспечения различных функциональных возможностей для параллельных программ. Каждый процесс MPI хранит свои данные в локальной памяти и обменивается данными с другими процессами MPI, передавая сообщения через сеть. Существует много реализаций, соответствующих стандарту MPI. Стандарт определяет различные аспекты интерфейса передачи сообщений, включая передачу сообщений «точка-точка», коллективные коммуникации и привязки различных языков программирования. Существует множество версий MPI с открытым исходным кодом, таких как Open-MPI и MVAPICH. Существуют также коммерческие реализации MPI, такие как Intel MPI, ScaMPI (бывший HP-MPI) и Voltaire MPI.

Традиционно настройка кластера MPI является сложной задачей, которая требует от системных администраторов значительного времени, т.к. системы HPC обычно обслуживают большое количество пользователей, которым необходимо запускать разные приложения с противоречивыми требованиями и зависимостями.

Контейнеризация в Linux – это технология виртуализации на уровне операционной системы, которая предлагает "легкую" виртуализацию. Приложение, которое работает как контейнер, имеет свою собственную корневую файловую систему, но разделяет ядро с операционной системой хоста.

За последние несколько лет контейнерные технологии были быстро внедрены в IT отрасль, и эта технология почти стала синонимом Docker. Разработчики проекта взяли за основу довольно зрелую концепцию контейнеров (Solaris Zones, IBM LPAR, LXC и др.) и быстро разработали поверх нее удобный для

пользователя продукт с рабочим процессом, который идеально подходит под философию DevOps[2] и современных микросервисных архитектур.

В последние годы проект Docker с открытым исходным кодом завоевывает популярность в применении контейнеров для высокопроизводительных вычислений [2]. Контейнеры Docker упаковывают программное обеспечение в полную файловую систему, в которой содержится все необходимое для запуска: код, среда выполнения, системные инструменты и системные библиотеки.

Разработчики программного обеспечения могут создавать сертифицированные контейнеры для своих приложений, уменьшая накладные расходы для клиента. По умолчанию (при первом запуске) контейнер не имеет состояния. Это гарантирует, что каждый запуск одного и того же образа контейнера будет одинаковым. Измененные файлы или настройки будут очищены после завершения приложения. Это гарантирует, что программа будет работать как ожидалось, независимо от ее окружения. Вместо того, чтобы делить вычислительные ресурсы на разделы для удовлетворения различных требований, кластер может быть установлен с минимальной конфигурацией, а все требования к окружению будут обеспечены контейнерами приложений. Это значительно уменьшает накладные расходы на управление конфигурацией.

Поскольку контейнеры, в отличие от виртуальных машин, не требуют наличия полной ОС, потеря производительности оказывается незначительной. Тесты показывают среднюю потерю для приложений MPI в 1,5-2% [4]. Данный факт, а также неоспоримые плюсы упаковки MPI-приложения в контейнер объясняют растущую популярность применения контейнеров для высокопроизводительных вычислений.

Однако одной из важных проблем вычислительных кластеров является оптимизация загрузки. Общепринятым решением является отправка задачи в очередь планировщика, ожидание освобождения необходимых ресурсов и запуск задачи эксклюзивно на выделенных узлах.

В статье рассматриваются различные способы оптимизации размещения контейнеров при работе с HPC-приложениями. Также рассматривается вариант запуска MPI-приложений в системе Fanlight.

[2] DevOps (акроним от англ. development и operations) — набор практик, нацеленных на активное взаимодействие специалистов по разработке со специалистами по информационно-технологическому обслуживанию и взаимную интеграцию их рабочих процессов друг в друга. Базируется на идее о тесной взаимозависимости разработки и эксплуатации программного обеспечения и нацелен на то, чтобы помогать организациям быстрее создавать и обновлять программные продукты и услуги.

2. Распределение контейнеров по узлам кластера

Как известно, вычислительный кластер состоит из управляющего и вычислительных узлов, связанных высокопроизводительной сетью. Каждый узел содержит несколько процессоров, каждый процессор –. несколько ядер.

Для параллельной программы необходимо выделить заданное количество процессорных ядер в эксклюзивное пользование. В противном случае (две задачи используют одно ядро –. overbooking) производительность сильно снижается, что приводит к непредсказуемому увеличению времени работы задач [7].

Для управления доступом к вычислительным узлам используется планировщик. Планировщик устанавливается на все узлы, на управляющем узле находится очередь. Доступ на узлы в обход планировщика запрещается. При запуске задачи указывается время работы, по окончанию которого задача будет принудительно завершена.

Все поступающие на кластер задачи помещаются в очередь. Если необходимое количество ресурсов для первой в очереди задачи свободно, то ресурсы помечаются как занятые, задача забирается из очереди и запускается. Способ выбора необходимых задаче узлов из свободных определяется алгоритмом оптимизации. При этом оптимизация может проводиться по одному или нескольким критериям. Для вычислительного кластера обычно используют следующие критерии:

- минимизация времени ожидания в очереди,
- максимизация пропускной способности кластера (количество завершенных задач в единицу времени),
- минимизация энергопотребления и времени ожидания в очереди, и др.
- Наиболее распространенным является алгоритм Backfill, когда для заполнения "пустых мест" в расписании используются задачи не с начала очереди [8]. При этом время работы задачи неизвестно, либо задается пользователем. В последнем случае оказывается, что время работы пользователи задают с большой положительной погрешностью, так как система принудительно завершает задачи по истечении выделенного времени [9]. В [10] было предложено не использовать время, задаваемое пользователем, а предсказывать его на основе истории запуска (журнала) задач.
- Менеджер ресурсов (планировщик) вычислительного кластера использует shell-сценарий, который является описанием задания, и отвечает за настройку среды выполнения и передачу необходимой информации процессу `mpirun`. Для удаленного выполнения используется SSH. При запуске, процессу передается список адресов зарезервированных для данной задачи узлов кластера, который используется библиотекой MPI для запуска остальных процессов

программы. Данная схема запуска является стандартной для MPI-приложений и плотно интегрирована со всеми известными планировщиками.

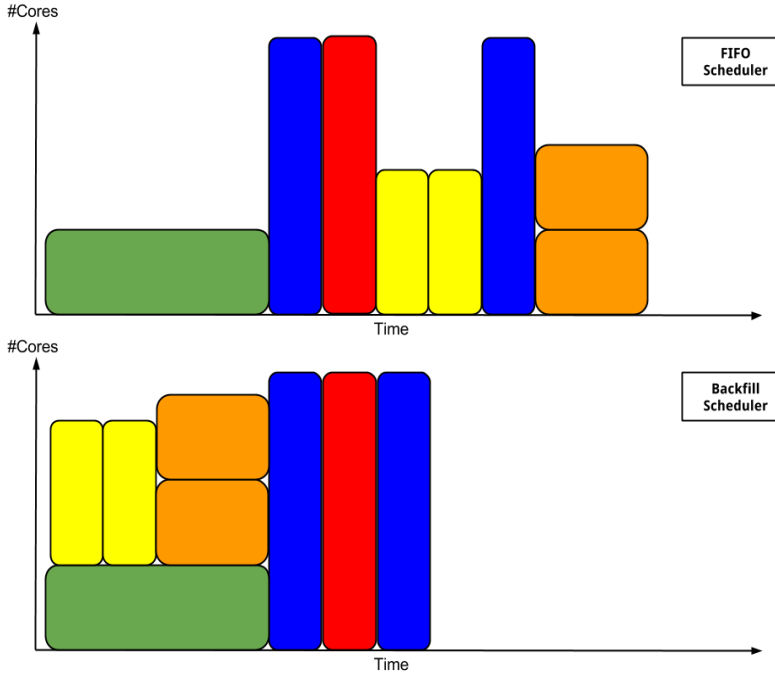


Рис. 1. Backfill
Fig. 1. Backfill

3. HPC-Контейнеры

Жизненным циклом контейнера управляет специальное ПО –. система управления контейнерами, в задачи которой входит:

- построение образа контейнера,
- хранение и обработка образов контейнеров,
- создание контейнера из образа,
- запуск, контроль выполнения, остановка, удаление контейнера и др.

Наиболее известными на сегодняшний день системами управления контейнерами являются: Docker, Kubernetes, Mesos, LXC, и др. Для формирования единого подхода в управлении контейнерами была создана организация OCI (Open Container Initiative) [11].

Управляющее ПО должно быть установлено на каждый из узлов кластера. Для запуска контейнера необходимо, чтобы образ контейнера находился на каждом из задействованных узлов кластера. Это означает, что образ необходимо скопировать на узлы заранее, или непосредственно перед запуском задачи. Средний размер образа параллельной программы может достигать нескольких гигабайт, что в определенных условиях вносит существенную задержку при запуске. Предпочтительной стратегией оптимизации в данном случае является регистрация и отслеживание образов и управление кешированными данными на узлах кластера [12].

При использовании HPC-контейнеров возможны несколько вариантов.

- Наиболее распространенной является схема запуска через существующий планировщик. В данном случае сохраняется совместимость –. кластер одновременно можно использовать для обычных приложений [13]. Разработчики некоторых планировщиков уже начали внедрять поддержку контейнеров [16].
- Управление распределением задач средствами системы управления контейнерами (Kubernetes, Docker, Mesos).
- Смешанный режим. Кластер используется как для высокопроизводительных вычислений, так и для микросервисов. В данном режиме кластер находится под управлением двух планировщиков: системы управления контейнерами и HPC-планировщика. Администратор кластера решает, какое количество ресурсов будет предоставлено под высокопроизводительные вычисления [18].

4. Оптимизация размещения HPC-контейнеров

В финансовом или инженерном моделировании HPC-задание может состоять из десятков тысяч коротких задач, требующих планирования с малой задержкой и высокой пропускной способностью для завершения моделирования в течение приемлемого периода времени. Задача вычислительной гидродинамики (CFD) может выполняться параллельно на многих сотнях или даже тысячах узлов, используя библиотеку передачи сообщений для синхронизации состояния. Для размещения и запуска таких заданий (а также проверки, приостановки, возобновления) требуются специализированные функции планирования и управления.

Другие HPC-задачи могут требовать специализированные ресурсы, такие как графические процессоры или доступ к ограниченным лицензиям на программное обеспечение. Организации могут проводить политику в отношении того, какие типы ресурсов могут быть использованы для обеспечения адекватного финансирования проектов и соблюдения сроков.

HPC-планировщики развивались и внедряли поддержку таких видов рабочей нагрузки. Поэтому, если говорить об универсальном решении для запуска

контейнеризированных программ на кластере, то запуск через HPC-планировщик остается единственным вариантом. Однако, универсальное решение требуется не всегда. Организовать и эффективно использовать кластер для определенного вида HPC-задач с использованием контейнеров можно и с помощью систем управления контейнерами.

В контейнерных распределенных вычислительных системах решается другой вид задач. Распределенная программная платформа (framework [3]) предоставляет абстракцию разработчикам, что позволяет им использовать большой объем ресурсов, рассматривая их как единый пул. Платформа предоставляет средства для распределения и обработки данных и задач на узлах, а также решает проблемы отказоустойчивости, такие как перезапуск задачи, сохранение состояния и др. Известными системами являются Spark[19], Storm[20], Tez[21] и др.

Существует три основные категории планировщиков для распределенных программных платформ:

- монолитные,
- двухуровневые,
- с разделяемым состоянием.

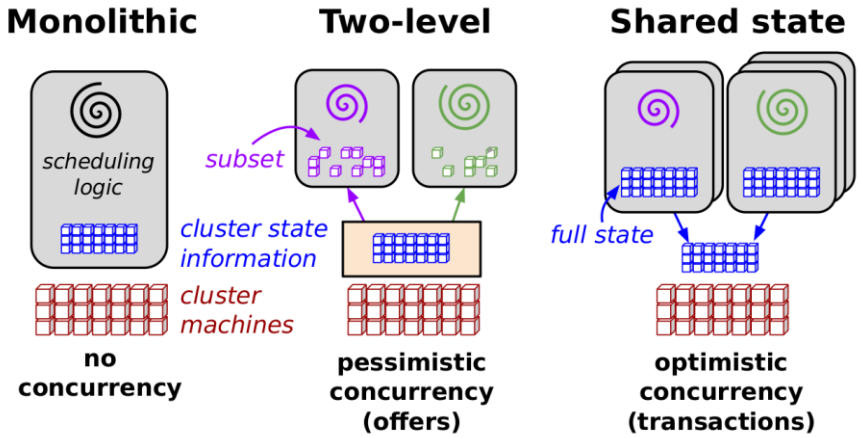


Рис. 2. Виды планировщиков
Fig. 2. Schedulers

Первый тип имеет глобальный пул ресурсов и должен реализовывать множество возможных политик распределения задач для каждой структуры

[3] Остов, каркас, структура — программная платформа, определяющая структуру программной системы; программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта.

кластера (предполагается, что кластер используется одновременно различными вычислительными платформами). По мере увеличения количества специализированных структур и приложений планировщик может стать узким местом.

Двухуровневые планировщики, такие как Mesos, выделяют пул ресурсов в структуры, которые затем реализуют свои собственные специализированные политики распределения. Каждая структура имеет частичное представление глобального пула ресурсов.

Планировщики с разделяемым состоянием предоставляют весь пул вычислительных ресурсов для набора специализированных планировщиков. За счет этого достигается высокий уровень консолидации ресурсов и появляется большая гибкость с точки зрения различий в политике распределения, но при этом необходимо тщательно управлять изменениями в глобальном пуле ресурсов для смягчения противоречивых решений о распределении.

С ростом популярности больших данных (big data) количество машин во многих кластерах увеличилось с сотен до десяти тысяч и более, а поток обрабатываемых задач характеризуется высокой интенсивностью и малым размером отдельной задачи. При этом планировщик должен обеспечивать высокую загрузку кластера, доступность, локальность данных. За последние годы было разработано множество планировщиков, некоторые используются в крупных компаниях: Microsoft Apollo [22], Google Borg [23], Facebook Tupperware [24], Twitter Aurora [25], Alibaba Fuxi [26], Omega [27], Sparrow [28], Hawk [29], Tarcil [30].

5. Разработка в ИСП РАН

В ИСП РАН разработана и развивается программная система Fanlight, позволяющая использовать разнородные вычислительные ресурсы в виде модели "Desktop as a Service" [31]. Работая с системой через Web-браузер пользователь получает доступ к графическому рабочему столу и возможность запускать на нем приложения, используя вычислительные ресурсы облака. Рабочий стол и приложения размещаются в отдельных контейнерах.

В ИСП РАН Fanlight используется пользователями в рамках программы Unihub для задач вычислительной гидродинамики (CFD).

Система поддерживает несколько видов приложений, запускаемых в отдельных контейнерах.

- **Графические Linux-приложения**. Приложение запускается внутри контейнера, окно приложения отображается в контейнере с запущенным Xvnc сервером, клиент HTML5 (noVnc) подсоединяется к Xvnc серверу через прокси.
- **Консольные**. Приложение запускается в контейнере, доступ к нему можно получить через HTML5 терминал.

- **Web-приложения**. Приложение доступно через браузер. Соединение происходит через прокси.

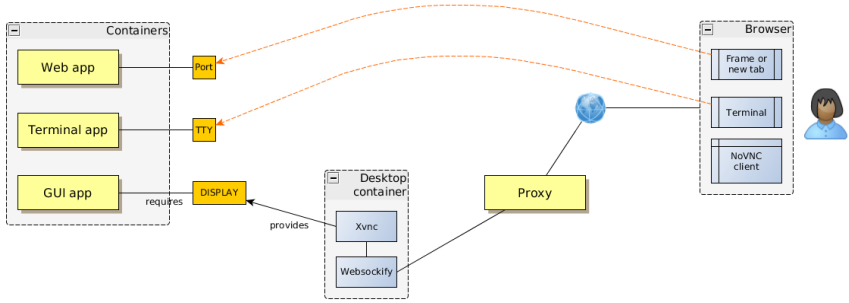


Рис. 3. Приложения Fanlight
Fig. 3. Fanlight applications

В основном, система используется для работы с графическими приложениями, такими как Salome, Paraview, Blender, Tnavigator и др. В системе реализована возможность работы с графическими ускорителями. OpenGL графика работает через VirtualGL, который перенаправляет команды 3D-рендеринга из OpenGL приложения в аппаратный 3D-ускоритель и отдает сжатые изображения VNC серверу.

Система Fanlight состоит из нескольких микросервисов.

- Controller
  - Взаимодействует с Docker API.
  - Отвечает за обработку REST API.
  - Управляет авторизацией и хранением данных.
- Proxy. Обеспечивает соединение между noVNC клиентом в браузере и VNC сервером в контейнере.
- Frontend. Реализует WEB-интерфейс.
- Admin Dashboard. WEB-интерфейс для администратора.

Для развертывания системы на вычислительных ресурсах необходимо установить на каждом узле Docker, далее процесс развертывания происходит в автоматическом режиме с помощью системы Ansible и/или Docker Compose. Все приложения добавляются в систему в виде Docker образов и не требуют дополнительной настройки системного окружения.

Домашние директории пользователей монтируются на все узлы средствами сетевой файловой системы. Все приложения и рабочие столы пользователя имеют доступ к его домашней директории.

Система Fanlight может работать как на одном, так и на нескольких серверах. Если в качестве ресурсов используется вычислительный кластер с сетью Infiniband, то система может запускать контейнеризированные MPI-приложения.

6. Запуск контейнера с MPI-задачей в Fanlight и стратегии размещения контейнеров

При работе с MPI-приложением на вычислительном кластере пользователь использует систему очередей планировщика. Подготовленный выполняемый скрипт специальной командой ставится в очередь (пакетное задание). Когда запрашиваемые задачей ресурсы освобождаются, планировщик запускает скрипт, передавая ему необходимое системное окружение. При большом числе пользователей задача может ожидать своей очереди несколько часов, и если в скрипте имеется ошибка, то все это время будет потрачено впустую. Такой сценарий случается достаточно часто. Для борьбы с этой проблемой обычно администраторы настраивают тестовую очередь, в которой задача принудительно завершается сразу после запуска. Это позволяет убедиться, что скрипт работает без ошибок, и программа начинает расчет.

Fanlight предлагает другой способ запуска. Поскольку контейнер может быть временно приостановлен, то ресурсы для задачи выделяются сразу, задача запускается и затем приостанавливается до освобождения узлов. Это позволяет решить проблему с ошибкой запуска и упростить работу для пользователей, особенно начинающих.

При распределении задач таким способом могут применятся различные стратегии. Наиболее простыми, имеющимися по умолчанию в системах управления контейнерами, например, Docker Swarm, являются:

- **random –**. выбирается случайный узел;
- **spread –**. выбирается наименее загруженный узел;
- **binpack –**. размещается как можно больше контейнеров на одном узле.

При большом количестве задач данные стратегии работают недостаточно эффективно.

HPC-планировщик со стратегией Backfill запускает задачу в момент освобождения необходимых ресурсов [8]. Это означает, что планировщик может выбирать, когда и где запустить задачу. В предлагаемом нами варианте запуска задач в системе Fanlight задача запускается сразу, а затем ждет своей очереди в "спящем" состоянии. В данном случае мы можем выбрать момент времени для пробуждения задачи, однако не можем переместить задачу на другие узлы. Это ограничение может в определенных ситуациях приводить к образованию пустых мест –. окон в расписании. Однако, проведенные нами наблюдения показали, что в большинстве случаев этого не происходит.

Тем не менее, данная особенность требует более детального исследования, выходящего за рамки данной статьи. Стратегия распределения должна учитывать вероятность возникновения таких окон и минимизировать ее, определенным образом размещая задачи.

7. Заключение

В статье были рассмотрены различные способы оптимизации размещения контейнеров при работе с HPC-приложениями. Также был показан вариант запуска MPI-приложений в системе Fanlight, разработанной в ИСП РАН [31].

В [32] мы предложили собирать данные о поведении пользователей и характере работы различных приложений с целью прогнозирования создаваемой контейнерами нагрузки. В данной работе мы предлагаем использовать полученную информацию о поведении пользователей и приложений для более точного предсказания времени работы задачи.

Предварительный анализ показывает, что чем точнее определяется время работы каждой задачи, тем меньше вероятность сдвига (раньше или позже запланированного) запуска следующей задачи и возникновения незаполненных окон в расписании. Таким образом, точность прогнозирования повышает эффективность использования ресурсов.

Дальнейшие исследования будут направлены на более детальный анализ факторов, влияющих на точность прогнозирования времени работы задач, а также на изучение факторов, приводящих к возникновению окон в расписании при распределении задач в системе Fanlight.

С использованием имеющихся вычислительных ресурсов ИСП РАН построена система из нескольких сотен контейнеров под управлением систем Docker Swarm, Kubernetes, Mesos. Исследовано поведение каждой из перечисленных систем в следующих сценариях работы: обновление приложения, развертывание приложения, добавление и удаление серверов. Разработан автоматический тест для оценки скорости работы систем в перечисленных сценариях.

Список литературы

[1]. Forum M.P. MPI: A message-passing interface standard. Knoxville, TN, USA: University of Tennessee, 1994.

[2]. Nguyen N., Bein D. Distributed mpi cluster with docker swarm mode. 2017 ieee 7th annual computing and communication workshop and conference (ccwc). 2017. Pp. 1–7.

[3]. Azab A. Enabling docker containers for high-performance and many-task computing. 2017 ieee international conference on cloud engineering (ic2e). 2017. Pp. 279–285.

[4]. Ermakov A., Vasyukov A. Testing docker performance for HPC applications. CoRR. 2017. Vol. abs/1704.05592.

[5]. Felter W. et al. An updated performance comparison of virtual machines and linux containers. 2014.

[6]. Di Tommaso P. et al. The impact of docker containers on the performance of genomic pipelines. PeerJ. 2015. Vol. 3. P. e1273.
[7]. Herbein S. et al. Resource management for running hpc applications in container clouds. High performance computing: 31st international conference, isc high performance 2016, frankfurt, germany, june 19-23, 2016, proceedings / ed. Kunkel J.M., Balaji P., Dongarra J. Cham: Springer International Publishing, 2016. Pp. 261–278.
[8]. Baraglia R. et al. Backfilling strategies for scheduling streams of jobs on computational farms. Making Grids Work. Springer, 2008. Pp. 103–115.
[9]. Mu'alem A.W., Feitelson D.G. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. IEEE Transactions on Parallel and Distributed Systems. 2001. Vol. 12, № 6. Pp. 529–543.
[10]. Nissimov A., Feitelson D.G. Probabilistic backfilling. Job scheduling strategies for parallel processing: 13th international workshop, jsspp 2007, seattle, wa, usa, june 17, 2007. revised papers. ed. Frachtenberg E., Schwiegelshohn U. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. Pp. 102–115.
[11]. https://www.opencontainers.org.
[12]. Harter T. et al. Slacker: Fast distribution with lazy docker containers. 14th USENIX conference on file and storage technologies, FAST 2016, santa clara, ca, usa, february 22-25, 2016. 2016. Pp. 181–195.
[13]. Higgins J., Holmes V., Venters C. Orchestrating docker containers in the hpc environment. High performance computing: 30th international conference, isc high performance 2015, Frankfurt, Germany, July 12-16, 2015, proceedings / ed. Kunkel J.M., Ludwig T. Cham: Springer International Publishing, 2015. Pp. 506–513.
[14]. Benedicic L. et al. Portable, high-performance containers for hpc. arXiv preprint arXiv:1704.03383. 2017.
[15]. Kurtzer G.M., Sochat V., Bauer M.W. Singularity: Scientific containers for mobility of compute. PLOS ONE. Public Library of Science, 2017. Vol. 12, № 5. Pp. 1–20.
[16]. http://www.univa.com/resources/files/gridengine_container_edition.pdf.
[17]. https://developer.ibm.com/storage/products/ibm-spectrum-lsf/.
[18]. https://navops.io/command.html.
[19]. https://spark.apache.org.
[20]. https://storm.apache.org.
[21]. https://tez.apache.org.
[22]. Boutin E. et al. Apollo: Scalable and coordinated scheduling for cloud-scale computing. Proceedings of the 11th usenix conference on operating systems design and implementation. Berkeley, CA, USA: USENIX Association, 2014. Pp. 285–300.
[23]. Verma A. et al. Large-scale cluster management at google with borg. Proceedings of the tenth european conference on computer systems. New York, NY, USA: ACM, 2015. Pp. 18:1–18:17.
[24]. http://www.slideshare.net/dotCloud/tupperware-containerized-deployment-at-facebook.
[25]. http://aurora.incubator.apache.org/.
[26]. Zhang Z. et al. Fuxi: A fault-tolerant resource management and job scheduling system at internet scale. Proc. VLDB Endow. VLDB Endowment, 2014. Vol. 7, № 13. Pp. 1393–1404.
[27]. Schwarzkopf M. et al. Omega: Flexible, scalable schedulers for large compute clusters. Proceedings of the 8th acm european conference on computer systems. New York, NY, USA: ACM, 2013. Pp. 351–364.

[28]. Ousterhout K. et al. Sparrow: Distributed, low latency scheduling. Proceedings of the twenty-fourth acm symposium on operating systems principles. New York, NY, USA: ACM, 2013. Pp. 69–84.
[29]. Delgado P. et al. Hawk: Hybrid datacenter scheduling. Proceedings of the 2015 usenix conference on usenix annual technical conference. Berkeley, CA, USA: USENIX Association, 2015. Pp. 499–510.
[30]. Delimitrou C., Sanchez D., Kozyrakis C. Tarcil: Reconciling scheduling speed and quality in large shared clusters. Proceedings of the sixth acm symposium on cloud computing. New York, NY, USA: ACM, 2015. Pp. 97–110.
[31]. http://fanlight.ispras.ru.
[32]. Д.А. Грушин, Н.Н. Кузюрин. Балансировка нагрузки в системе Unihub на основе предсказания поведения пользователей. Труды ИСП РАН, том 27, вып. 5, 2015 г., стр. 23–34. DOI: 10.15514/ISPRAS-2015-27(5)-2

Optimization problems running MPI-based HPC applications

[1]*D.A. Grushin <grushin@ispras.ru>*
[1,2]*N.N.Kuzjurin <nnkuz@ispras.ru>*
[1]*Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia*
[2] *Moscow Institute of Physics and Technology, Dolgoprudnyj, Institutskij alley, Moscow region, 141700, Russia*

Abstract. MPI is a well-proven technology that is widely used in a high-performance computing environment. However, configuring an MPI cluster can be a difficult task. Containers are a new approach to virtualization and simple application packaging, which is becoming a popular tool for high-performance tasks (HPC). This approach is considered in this article. Packaging an MPI application as a container solves the problem of conflicting dependencies, simplifies the configuration and management of running applications. A typical queue system (for example, SLURM) or a container management system (Docker Swarm, Kubernetes, Mesos, etc.) can be used to manage cluster resources. Containers also provide more options for flexible management of running applications (stop, restart, pause, in some cases, migration between nodes), which allows you to gain an advantage optimizing the allocation of tasks to cluster nodes in comparison with the classic scheduler. The article discusses various ways to optimize the placement of containers when working with HPC-applications. A variant of launching MPI applications in Fanlight system is proposed, which simplifies the work of users. The optimization problem associated with this method is considered also.

Keywords: docker, containers, scheduling

DOI: 10.15514/ISPRAS-2017-29(6)-14

For citation: Grushin D.A., Kuzjurin N.N. Optimization problems running MPI-based HPC applications. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 6, 2017. pp. 229-244 (in Russian). DOI: 10.15514/ISPRAS-2017-29(6)-14

References

[1]. Forum M.P. MPI: A message-passing interface standard. Knoxville, TN, USA: University of Tennessee, 1994.
[2]. Nguyen N., Bein D. Distributed mpi cluster with docker swarm mode. 2017 ieee 7th annual computing and communication workshop and conference (ccwc). 2017. Pp. 1–7.
[3]. Azab A. Enabling docker containers for high-performance and many-task computing. 2017 ieee international conference on cloud engineering (ic2e). 2017. Pp. 279–285.
[4]. Ermakov A., Vasyukov A. Testing docker performance for HPC applications. CoRR. 2017. Vol. abs/1704.05592.
[5]. Felter W. et al. An updated performance comparison of virtual machines and linux containers. 2014.
[6]. Di Tommaso P. et al. The impact of docker containers on the performance of genomic pipelines. PeerJ. 2015. Vol. 3. P. e1273.
[7]. Herbein S. et al. Resource management for running hpc applications in container clouds. High performance computing: 31st international conference, isc high performance 2016, frankfurt, germany, june 19-23, 2016, proceedings / ed. Kunkel J.M., Balaji P., Dongarra J. Cham: Springer International Publishing, 2016. Pp. 261–278.
[8]. Baraglia R. et al. Backfilling strategies for scheduling streams of jobs on computational farms. Making Grids Work. Springer, 2008. Pp. 103–115.
[9]. Mu'alem A.W., Feitelson D.G. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. IEEE Transactions on Parallel and Distributed Systems. 2001. Vol. 12, № 6. Pp. 529–543.
[10]. Nissimov A., Feitelson D.G. Probabilistic backfilling. Job scheduling strategies for parallel processing: 13th international workshop, jsspp 2007, seattle, wa, usa, june 17, 2007. revised papers. ed. Frachtenberg E., Schwiegelshohn U. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. Pp. 102–115.
[11]. https://www.opencontainers.org.
[12]. Harter T. et al. Slacker: Fast distribution with lazy docker containers. 14th USENIX conference on file and storage technologies, FAST 2016, santa clara, ca, usa, february 22-25, 2016. 2016. Pp. 181–195.
[13]. Higgins J., Holmes V., Venters C. Orchestrating docker containers in the hpc environment. High performance computing: 30th international conference, isc high performance 2015, Frankfurt, Germany, July 12-16, 2015, proceedings / ed. Kunkel J.M., Ludwig T. Cham: Springer International Publishing, 2015. Pp. 506–513.
[14]. Benedicic L. et al. Portable, high-performance containers for hpc. arXiv preprint arXiv:1704.03383. 2017.
[15]. Kurtzer G.M., Sochat V., Bauer M.W. Singularity: Scientific containers for mobility of compute. PLOS ONE. Public Library of Science, 2017. Vol. 12, № 5. Pp. 1–20.
[16]. http://www.univa.com/resources/files/gridengine_container_edition.pdf.
[17]. https://developer.ibm.com/storage/products/ibm-spectrum-lsf/.
[18]. https://navops.io/command.html.
[19]. https://spark.apache.org.
[20]. https://storm.apache.org.

[21]. https://tez.apache.org.
[22]. Boutin E. et al. Apollo: Scalable and coordinated scheduling for cloud-scale computing. Proceedings of the 11th usenix conference on operating systems design and implementation. Berkeley, CA, USA: USENIX Association, 2014. Pp. 285–300.
[23]. Verma A. et al. Large-scale cluster management at google with borg. Proceedings of the tenth european conference on computer systems. New York, NY, USA: ACM, 2015. Pp. 18:1–18:17.
[24]. http://www.slideshare.net/dotCloud/tupperware-containerized-deployment-at-facebook.
[25]. http://aurora.incubator.apache.org/.
[26]. Zhang Z. et al. Fuxi: A fault-tolerant resource management and job scheduling system at internet scale. Proc. VLDB Endow. VLDB Endowment, 2014. Vol. 7, № 13. Pp. 1393–1404.
[27]. Schwarzkopf M. et al. Omega: Flexible, scalable schedulers for large compute clusters. Proceedings of the 8th acm european conference on computer systems. New York, NY, USA: ACM, 2013. Pp. 351–364.
[28]. Ousterhout K. et al. Sparrow: Distributed, low latency scheduling. Proceedings of the twenty-fourth acm symposium on operating systems principles. New York, NY, USA: ACM, 2013. Pp. 69–84.
[29]. Delgado P. et al. Hawk: Hybrid datacenter scheduling. Proceedings of the 2015 usenix conference on usenix annual technical conference. Berkeley, CA, USA: USENIX Association, 2015. Pp. 499–510.
[30]. Delimitrou C., Sanchez D., Kozyrakis C. Tarcil: Reconciling scheduling speed and quality in large shared clusters. Proceedings of the sixth acm symposium on cloud computing. New York, NY, USA: ACM, 2015. Pp. 97–110.
[31]. http://fanlight.ispras.ru.
[32]. Grushin D., Kuzyurin N. Load balancing in unihub saas system based on user behavior prediction. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 5, 2015, pp. 23–34 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)-2

Simulation of mixed convection over horizontal plate

M.N. Nikitin <max@nikitin-pro.ru>
Samara State Technical University,
244, Molodogvardeyskaya st., Samara, 443001, Russia

Abstract. Mixed convection over horizontal heated plate was simulated with four numerical models based on Reynolds stress, large eddy simulation (LES) and eddy viscosity approximations. Temperature distributions over plate and in adjacent volume of fluid were the main criteria of results assessment. Three-dimensional computational domain was considered with symmetry boundary conditions. Simulation was performed with Code_Saturne software package in unsteady formulation. Three orthogonal meshes were evaluated to validate initial guess about optimal cell size. u2-f and Smagorinsky LES models appeared to yield the most adequate results. However, temperature distribution in high-buoyancy region, located in the middle of heated plate, was reproduced much more accurate with classical LES and elliptic blending Reynolds stress models. Obtained results are suitable for industrial applications (e.g. cooling jackets) and might be a base ground for further research.

Keywords: heat transfer; numerical modeling; temperature distribution; Reynolds stress model; large eddy simulation; eddy viscosity model; Code_Saturne

DOI: 10.15514/ISPRAS-2017-29(6)-15

For citation: Nikitin M.N. 1. Simulation of mixed convection over horizontal plate. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 6, 2017. pp. 245-252. DOI: 10.15514/ISPRAS-2017-29(6)-15

1. Introduction

Efficient heat exchange is usually associated with highly forced convection, while natural convection is negligible. However, maintenance of forced convection might be too expensive and complex, especially with liquid heat carriers. Thus many industrial heat exchangers (e.g. cooling jackets) tend to operate in mixed convection mode, when pressure and buoyant forces contribute to convection at approximately the same rate. Prediction of flow patterns and temperature distribution is crucial for effective implementation of mixed convection in heat exchange equipment. Understanding of flow affected by mixed convection can minimize thermal stresses and eliminate stagnation zones.

Mixed convection can occur when a downward flow impinges on horizontal heated plate and stagnation point flow (Hiemenz flow) above the plate provides action of buoyant motion perpendicular to forced motion. This common case has been studied extensively, but with a little insight into flow instability and actual temperature distribution. Most of recent studies of mixed convection over heated horizontal plate [1-3] reveal Hiemenz flow structure in two-dimensional formulation. At the same time some researchers [4-6] show asymmetric nature of Hiemenz flow. The most recent experimental study of Hiemenz flow over horizontal plate [7] provides an important insight into flow patterns and temperature distribution. However, flow instability is described in steady state formulation only.

Numerical simulation was conducted to supplement basic dependencies which are based on experimental data [7]. Three-dimensional computational domain was considered since preliminary simulations revealed dramatic effect of dimension reduction on results. Preliminary simulations also gave erroneous results when problem was formulated in steady state. Considering preliminary results and suggestions of previous researchers, numerical simulation of downward water flow impinging on horizontal heated plate was conducted for three-dimensional domain in unsteady formulation. Obtained numerical model can be used for engineering of equipment that utilizes mixed convection over horizontal plate.

2. Materials and methods

2.1 Computational domain and mesh

Main part of experimental rig [7] is square-profiled duct with heated plate which is perpendicular to downward water flow. Smooth confuser along with calming meshes provide a uniform downward flow. Upstream part of tested duct was considered as computational domain (fig. 1). Symmetry conditions were applied to both sides due to prolate form of heated plate. Symmetry allowed reduction of computational domain down to 250x125x150 mm box.

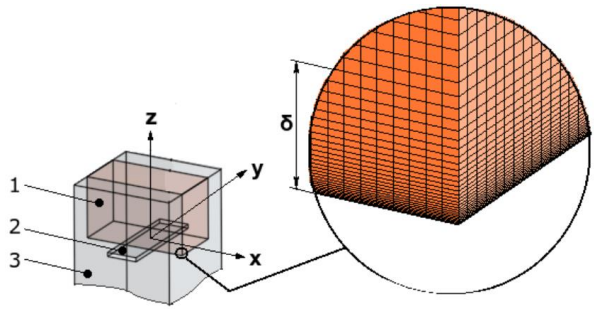


Fig. 1. Computational domain: 1 – Computational domain; 2 – Heated plate (75x250 mm); 3 – Tested duct (250x250 mm).

Computational mesh was generated with SALOME – an open source software platform designated for pre- and post-processing in numerical simulations. Three hexahedral meshes with cell sizes of 1.25, 1.5 and 2 mm were considered. Meshes consisted of 2.9, 1.8 and 0.83 million cells respectively. Each mesh was inflated near heat transfer surface ensuring $y^+<1$. Considering flow velocity, first row of cells was limited to be 0.12 mm thick. Total thickness of inflated rows (delta) was in a range of 10–15 mm (see fig. 1).

2.2 Numerical model

Numerical model was developed with Code_Saturne – free proprietary software designated for computational fluid dynamics and based on finite volume approach. All simulations were carried out in unsteady formulation. Water was used as working medium with temperature-dependent physical properties. Viscosity, specific heat and thermal conductivity were defined by interpolation of tabular data while density was defined as:

$$\rho = 1001.1 - 0.0867T - 0.0035T^2,$$

where T is local temperature in degrees Celsius.

Boundary conditions were defined as follows: top face of computational domain was set as inlet with uniform velocity U=0.02 m/s; bottom face was split into two equal outlets (87.5x250 mm each) and heated wall (75x250 mm, q=7 kW/m^2) between them; side faces that are normal to x-axis were set as adiabatic walls with no-slip condition; side faces that are normal to y-axis were set as symmetry planes. Inlet temperature was set to \$19 ^{o}C. Initial temperature in computational domain was equal to inlet temperature.

Iterative method of gradient calculation was used. Unlimited iterative reconstruction of non-orthogonalities was used to receive distinctive feedback on mesh quality. Continuity and momentum equations were not combined since the continuity equation does not have temporal progress term. Transposed gradients and divergence source terms were handled in momentum equation to ensure convergence on relatively coarse meshes. Pressure relaxation (R=0.9) was used to stabilize calculation. Since high pressure gradients were not expected a standard pressure interpolation over computational domain was used.

Very robust geometric-algebraic multigrid solver (GAMG) was utilized to solve linearized pressure equation. Momentum and energy equations were solved by diagonal solver (Jacobi). All linear solvers were limited by maximum number of iterations ($N_{max}=10^5$) and maximum precision (epsilon = 10^{-5}). To ensure reasonable accuracy at such strict limitations three test runs were carried out with epsilon 5, 6 and 7. These test runs showed reasonable deterioration of accuracy with an average rate of 3.8 % per an order of epsilon magnitude. Least stable, but most accurate second-order (centered) discretization scheme was used.

Corrected semi-implicit algorithm (SIMPLEC) was utilized to solve Navier-Stokes equations. This algorithm allowed to set relatively large time step (Delta t = 0.01 s), ensuring recommended value of Courant number (Cr_{max} = 5). Providing constant time step, 50 s of experiment were simulated at each run. Residuals analysis along with graphical evaluation of stability of velocity and temperature profiles showed that solution was usually converged in 40 s.

Navier-Stokes and conservation equations were closed with Reynolds stress models (RSM) and scale-resolving simulation (SRS) models. Eddy viscosity models showed inadequate results except u2-f model, which was considered for mesh resolution analysis.

Elliptic blending RSM (EBRSM) was preferred to Speziale-Sarkar-Gatski (SSG) model since its ability to handle inflated meshes ($y^+ < 1$). Classical dynamic large eddy simulation (LES) model and Smagorinsky LES model were considered as part of SRS. Wall-adapting local eddy (WALE) model was unable to reproduce required temperature distribution, whereas other SRS models are not implemented in current version of Code_Saturne (v 4.0).

All considered models included convection and diffusion terms in every unknown variable. Reconstruction of convective and diffusive fluxes at the faces was enabled since mesh was orthogonal. Despite second-order terms of convection, diffusion and source terms were expressed in first-order by setting theta = 1:

$$\phi^{n+\theta} = (1-\theta)\phi^n + \theta\phi^{n+1}.$$

2.3 Data reduction

Models adequacy was assessed by flow patterns and temperature distribution over heated plate. Adopted experimental study [7] revealed distinctive crests of heated water being pushed by buoyancy forces against downward flow. Consistency and height of these crests and corresponding hot stripes on heated plate were major criteria of reproduction assessment of flow patterns. Crests height was assessed by filtering out volume regions with temperature more than T_{in}+1 ^oC. Distribution of plate temperature was derived from a series of measurements along y-oriented lines, which were discretized by 100 data points. Arithmetic mean of each line data over five time steps (46-50 s) was considered. Spatial-temporal standard deviations were obtained for each dataset.

3. Results

In general all considered models yielded adequate results (fig. 2). At the same time one can mention significant spatial and temporal deviations of calculated temperature of heated plate represented on fig. 2 as vertical bars. These deviations were a result of crests movement across the plate.

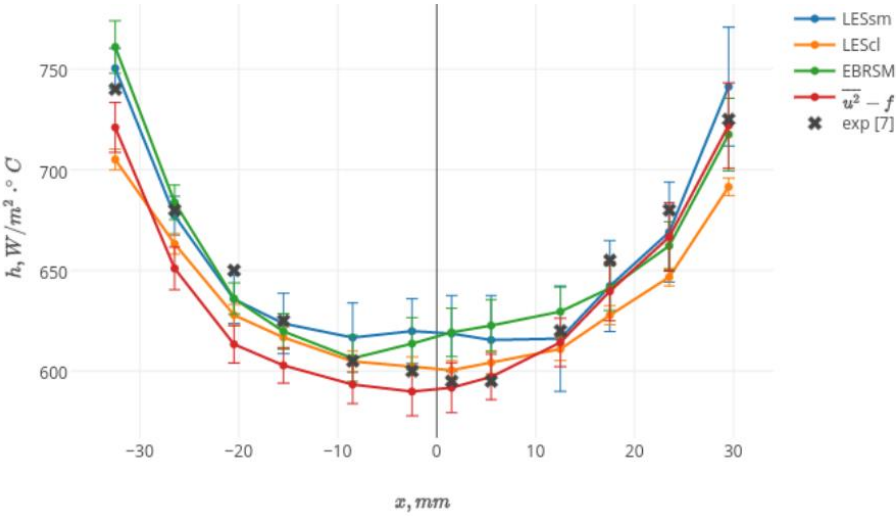


Fig. 2. Models comparison (error bars show temporal standard deviation).

U2-f and LES models reproduced slight oscillations of hot crests, which were distributed over heated plate in mostly stationary manner. Hot crests modeled with EBRSM (see fig. 3) were moving with approximately constant speed of 0.01 m/s from the center of the plate towards side walls of the duct.

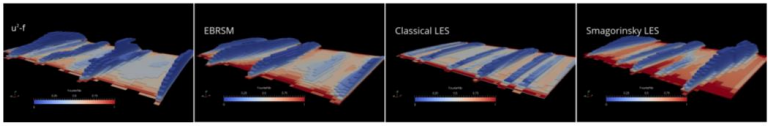
While Smagorinsky model provided the lowest crests, other models reproduced well proportioned crests, which were in a good agreement with experimental vitalizations [7]. These crests were rising and falling, forming wave-like structures. Filtered volume of working flow was represented on fig. 3 in color scale of Fourier values to provide more distinctive picture of crests distribution.

Smagorinsky model elliptic stress model (EBRSM) yielded overestimated values of local heat transfer coefficient closer to the middle of the plate. In contrast, classical LES accurately predicts plate temperature in the middle, whereas obviously overestimates temperatures near the edges of the plate. Also one can see overall underestimation of heat transfer efficiency with u2-f model.

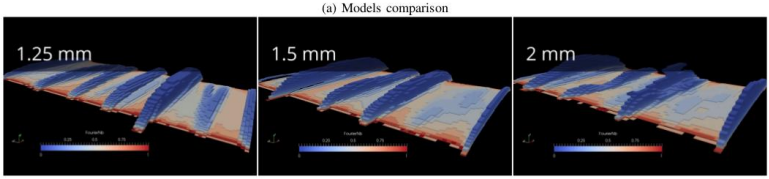
4. Discussion

Four numerical models based on LES, RSM and Eddy viscosity turbulence approximations were developed to simulate mixed convection over horizontal plate. EBRSM and Smagorinsky models yielded the most accurate results, providing adequate temperature distribution on heat transfer surface and corresponding pattern of buoyancy driven flow over it. Heat transfer coefficient overestimation in the middle of the plate might be the result of large mesh size. As one can see on fig. 3 (b) higher mesh resolution provides more even crests distribution, though these

crests are less prominent. Lower profile of crests means higher surface temperatures and consequently lower heat transfer intensity.



(a) Models comparison



(b) Cell size comparison

Fig. 3. Hot crests height and distribution by volume of fluid filtered with temperature more than T_{in}+1 ^{o}C.

Nevertheless, mesh size had relatively low impact on obtained results, namely a negligible variation of controlled parameters. It is a known fact that mesh inflation affects computation results to a certain extent, providing dimensionless distance $y^{+}<1$ at heat transfer surface. Therefore, obtained results supported initial guess about optimal cell size (2 mm).

None of tested models was able to predict distribution of local heat transfer coefficient in exact manner. However, all models gave mean error below 2 %. One should mention asymmetry of experimental data [7] at points with $|x|<10$ mm, which were adopted «as is». Temporal deviations of obtained results are represented on fig. 2 by vertical bars, which illustrate uncertainty rate. Clearly, u2-f and classical LES models adequately reproduced a drop of heat transfer efficiency that occurs in the middle of the plate, whereas EBRSM and Smagorinsky models has failed to make consistent prediction in that region. Closer to the plate edges the situation was an opposite. Considering overall error values EBRSM and Smagorinsky models have been selected as more adequate.

Numerical modeling of mixed convection over heated horizontal plate revealed some caveats in terms of handling high-buoyancy driven flow. Nevertheless, EBRSM and Smagorinsky models considered fairly adequate for engineering purposes. Clearly, further research is required to bridge the gap between modeling natural-biased and forced-biased regimes of mixed convection.

5. Conclusion

The study addressed mixed convection over plate heated with constant heat flux. Temperature distributions over plate and in adjacent volume of fluid were the main criteria of results assessment. Four numerical models based on LES, SRS and eddy viscosity approximations were developed to reproduce mixed convection in domain of three-dimensional orthogonal mesh.

EBRSM and Smagorinsky models produced the most adequate results in terms of temperature distribution and characteristics of hot crests. However, Smagorinsky model yielded less distinctive crests.

Obtained results ensured good accuracy level for industrial applications (e.g. cooling jackets). However, further research is required to potentially increase accuracy of mixed convection simulation in a region of high buoyancy.

References

[1]. M. C. K. Chen and C. Sohn. Thermal instability of two-dimensional stagnation-point boundary layers. Journal of Fluid Mechanics, vol. 132, pp. 49–63, 2006.

[2]. M. Amaouche and D. Boukari. Influence of thermal convection on non-orthogonal stagnation point flow. International Journal of Thermal Sciences, vol. 42, pp. 303–310, 2003.

[3]. M. Nobari and A. Beshkani. A numerical study of mixed convection in a vertical channel flow impinging on a horizontal surface. International Journal of Thermal Sciences, vol. 46, pp. 989–997, 2007.

[4]. F. B. F. Mendil and D. Sadaoui. Effect of temperature dependent viscosity on the thermal instability of two-dimensional stagnation point flow. Mechanics & Industry, vol. 16, pp. 1–8, 2015.

[5]. V. Calmidi and R. Mahajan. Mixed convection over a heated horizontal surface in a partial enclosure. International Journal of Heat and Fluid Flow, vol. 19, pp. 358–367, 1998.

[6]. A. Ishak. Mixed convection boundary layer flow over a horizontal plate with thermal radiation. Heat and Mass Transfer, vol. 46, pp. 147–151, 2009.

[7]. K. Kitamura and A. Mitsuishi. Fluid flow and heat transfer of mixed convection over heated horizontal plate placed in vertical downward flow. International Journal of Heat and Mass Transfer, vol. 53, pp. 2327–2336, 2010.

Моделирование смешанной конвекции над горизонтальной пластиной

М.Н. Никитин <max@nikitin-pro.ru>
Самарский государственный технический университет,
443100, Россия, г. Самара, ул. Молодогвардейская, 244

Аннотация. Смешанная конвекция над нагреваемой горизонтальной пластиной была смоделирована с использованием четырех численных моделей на базе рейнольдсовых, вихревых и вязкостных моделей турбулентности. Основным критерием оценки адекватности моделей было распределение температуры на поверхности пластины и в слое жидкости над ней. В расчетах использовалась трехмерная расчетная область с условиями симметрии. Моделирование проводилось с использованием программного пакета Code_Saturne в нестационарной постановке. Для оценки сеточной сходимости решения были использованы три ортогональные сетки. Вязкостная модель u2-f и вихревая модель Смагоринского показали наиболее адекватные результаты. Однако

наиболее точные распределения температуры в области преимущественного действия архимедовой силы были получены при использовании вихревой k-модели и эллиптической рейнольдсовой модели EBRSM. Разработанные модели пригодны для промышленного использования (например, для проектирования охлаждающих рубашек) и могут быть использованы в качестве основы для дальнейших исследований смешанной конвекции.

Ключевые слова: теплопередача; численное моделирование; распределение температуры; модели рейнольдсовых напряжений; вихревые модели; вызкостные модели; Code_Saturn

DOI: 10.15514/ISPRAS-2017-29(6)-15

For citation: Никитин М.Н. Моделирование смешанной конвекции над горизонтальной пластиной. Труды ИСП РАН, том 29, вып. 6, 2017 г., стр. 245-252 (на английском языке). DOI: 10.15514/ISPRAS-2017-29(6)-15

Список литературы

[1]. M. C. K. Chen and C. Sohn. Thermal instability of two-dimensional stagnation-point boundary layers. Journal of Fluid Mechanics, vol. 132, pp. 49–63, 2006.

[2]. M. Amaouche and D. Boukari. Influence of thermal convection on non-orthogonal stagnation point flow. International Journal of Thermal Sciences, vol. 42, pp. 303–310, 2003.

[3]. M. Nobari and A. Beshkani. A numerical study of mixed convection in a vertical channel flow impinging on a horizontal surface. International Journal of Thermal Sciences, vol. 46, pp. 989–997, 2007.

[4]. F. B. F. Mendil and D. Sadaoui. Effect of temperature dependent viscosity on the thermal instability of two-dimensional stagnation point flow. Mechanics & Industry, vol. 16, pp. 1–8, 2015.

[5]. V. Calmidi and R. Mahajan. Mixed convection over a heated horizontal surface in a partial enclosure. International Journal of Heat and Fluid Flow, vol. 19, pp. 358–367, 1998.

[6]. A. Ishak. Mixed convection boundary layer flow over a horizontal plate with thermal radiation. Heat and Mass Transfer, vol. 46, pp. 147–151, 2009.

[7]. K. Kitamura and A. Mitsuishi. Fluid flow and heat transfer of mixed convection over heated horizontal plate placed in vertical downward flow. International Journal of Heat and Mass Transfer, vol. 53, pp. 2327–2336, 2010.

Методика решения задач аэроупругости для лопасти ветроустановки с использованием СПО

[1, 2] *П.С. Лукашин <ski1l@mail.ru>*
[1, 2] *В.Г. Мельникова <vg-melnikova@yandex.ru>*
[2] *С.В. Стрижак <strijhak@yandex.ru>*
[1] *Г.А. Щеглов <shcheglov_ga@bmstu.ru>*
[1] *МГТУ им. Н.Э. Баумана,*
105005, г. Москва, ул. 2-я Бауманская, д.5. стр.1
[2] *Институт системного программирования им. В.П. Иванникова РАН,*
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. В связи с развитием ветроэнергетики и строительством новых ветропарков в РФ возникает потребность в решении ряда прикладных задач и разработке эффективных методик расчета элементов ветроустановок. Одно из направлений в задачах механики сплошной среды связано с задачами аэроупругости. В данной статье показана возможность решения связанной задачи аэроупругости с использованием программного комплекса на базе свободного программного обеспечения OpenFOAM и Code_Aster. На примере лопасти ветроустановки длиной 61.5 метра рассмотрены методики решения задач статической и динамической аэроупругости, в которых расчет обтекания лопасти дозвуковым набегающим потоком воздуха производится в пакете OpenFOAM (решатели simpleFOAM и pimpleFOAM), а расчет напряженно-деформированного состояния лопасти производится в пакете Code_Aster. В статье приводятся блок-схемы для трех различных подходов решения задачи аэроупругости, примеры скриптов и командных файлов для передачи данных между пакетами в процессе расчета. Контрольно-объемная сетка, состоящая их гексаэдральных элементов, для расчета обтекания лопасти построена в пакете OpenFOAM, конечно-элементная сетка**,** состоящая из треугольных оболочечных элементов первого порядка, для расчета напряженно-деформированного состояния построена в пакете Salome-Meca. Результаты расчета представлены в форме полей для давления и скорости; зависимостей для невязок давления, скорости, турбулентной вязкости; проекций аэродинамической силы от времени; эпюр перемещения и напряжения; значений давления и перемещения для двух точек на поверхности лопасти от времени. Расчеты выполнены с использованием ресурсов web-лаборатории UniHUB ИСП РАН.

Ключевые слова: аэроупругость; ветроустановка; лопасть; Code_Aster; OpenFOAM; модель турбулентности; решатель; статический расчет; динамический расчет; частота, перемещения, напряжение, расчеты, web-лаборатория.

DOI: 10.15514/ISPRAS-2017-29(6)-16

Для цитирования: Лукашин П.С., Мельникова В.Г., Стрижак С.В., Щеглов Г.А. Методика решения задач аэроупругости для лопасти ветроустановки с использованием СПО. Труды ИСП РАН, том 29, вып. 6, 2017 г., стр. 253-270. DOI: 10.15514/ISPRAS-2017-29(6)-16

1. Введение

В связи с развитием ветроэнергетики в РФ, проектированием новых ветроэлектрических установок (ВЭУ), ветропарков и их эксплуатацией в различных климатических условиях на обширной территории РФ возникает потребность в решении ряда прикладных задач. К одной из них можно отнести оценку динамических и прочностных характеристик лопастей ВЭУ с учетом ветровой нагрузки. Наиболее приспособленными для решения таких задач являются универсальные коммерческие программные пакеты. Но статистика показывает, что различные крупные и средние предприятия, постепенно стали осознавать преимущества, которые предоставляет свободное программное обеспечение (СПО): снижение затрат на программные комплексы, открытость исходного кода, а также защита от возможных санкций иностранных правообладателей. Поэтому поиск, модификация, усовершенствование существующих и создание новых программных средств с открытым исходным кодом является актуальной задачей на сегодняшний день.

В настоящее время СПО уже успешно используется для расчета конструкций в области энергетики. В частности, можно отметить пакет Code_Aster, используемый крупнейшей энергетической компанией Франции EDF.

Важным преимуществом свободного программного обеспечения является возможность создания на базе нескольких пакетов нового программного комплекса для решения сложных задач мультифизики. К таким задачам традиционно относятся задачи аэроупругости.

Основной целью работы является отладка методики решения сопряженных задач взаимодействия деформируемой конструкции и потока среды на основе двух открытых пакетов: OpenFOAM и Code_Aster.

Возможны различные постановки задачи аэроупругости, основанные на упрощениях и допущениях [1,2]. Исходя из анализа литературы [2-5], можно выделить два варианта связанной задачи FSI:

- задачи статической аэроупругости: совместное рассмотрение уравнений теории упругости и уравнений для стационарного течения жидкости и газа;

- задачи динамической аэроупругости: совместное рассмотрение уравнений теории упругости и уравнений для нестационарного течения жидкости и газа.

В [6] выделяется несколько подходов к решению задачи FSI:

- one-way coupling: раздельное последовательное интегрирование уравнение двух подсистем на каждом шаге интегрирования, без учета перемещения тела в потоке;
- two-way coupling: согласованное интегрирование уравнений двух подсистем на каждом шаге интегрирования, с учетом перемещения тела в потоке.

Основываясь на характере действующей на конструкцию лопасти ВЭУ нагрузки, величине влияния деформации тела на поток и представленной выше классификации было выбрано три варианта методики расчета:

- квазистатическая постановка – расчет напряженно-деформированного состояния (НДС) лопасти при статическом нагружении полем давлений, полученным в результате стационарного аэродинамического расчета недеформированной конструкции.
- динамическая постановка – расчет вынужденных колебаний лопасти при нагружении полем давлений, полученным в результате нестационарного аэродинамического расчета.
- сопряженная квазистатическая постановка – расчет НДС лопасти при статическом нагружении полем давлений, полученным в результате стационарного аэродинамического расчета с учетом влияния деформации лопасти на параметры течения.

2. Описание методик

2.1. Квазистатическая постановка

Простейшую последовательную методику решения задачи аэроупругости, блок-схема которой показана на рис. 1, допустимо использовать в случае, когда режим обтекания лопасти можно считать безотрывным и стационарным, а влиянием упругих перемещений конструкции на характер течения и распределением аэродинамических нагрузок по лопасти можно пренебречь. Также упрощающим предположением является допущение о компенсации сил инерции диссипативными силами что приводит задачу определения НДС к статической. Методика состоит из следующих этапов:

- проведение стационарного аэродинамического расчета методом контрольного объема в пакете OpenFOAM с использованием решателя simpleFOAM. В результате определяются поля скоростей и давлений;

- перенос результатов аэродинамического расчета из пакета OpenFOAM в пакет Code_Aster;
- определение НДС конструкции методом конечных элементов в Code_Aster.

Здесь ключевым для совместного использования пакетов является второй этап, в ходе которого осуществляется проецирование поля давления, полученного в аэродинамическом расчете с контрольно-объемной (КО) сетки на конечно-элементную (КЭ), для дальнейшего использования в качестве нагрузки при расчете НДС.

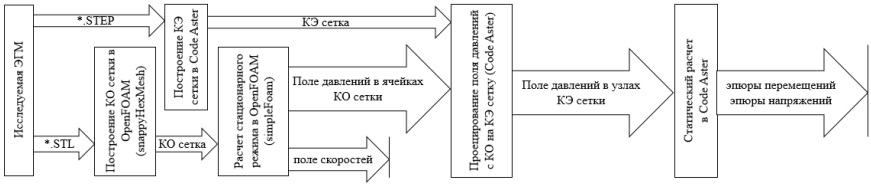


Рис. 1. Алгоритм расчета при квазистатической постановке задачи
Fig. 1. Simulation algorithm at quasistatic problem definition

Преобразование данных из формата *.foam, используемого в OpenFOAM, в формат *.med, используемый в качестве входного формата для Code_Aster осуществлялось посредством модуля визуализации Paraview, встроенного в препостпроцессор Salome-Meca. Для проецирования значений давлений с одной сетки на другую применялся оператор Code_Aster `PROJ_CHAMP`, в котором использовался метод коллокаций. В этом методе определяется положение каждого узла новой сетки относительно элементов проецируемой сетки. При нахождении узла внутри элемента значение в нем вычисляется с помощью функций формы соответствующего элемента. Если узел не лежит в элементе, то значение в нем определяется как значение в точке, лежащей в ближайшем элементе. Задание максимального расстояния между узлом и этой точкой позволяет интерполировать значения с заданной точностью.

Ниже показана часть командного файла Code_Aster, в которой содержатся операторы, которые считывают сетку в формате *.med и проецируют значения на заданную группу КЭ сетки, содержащую элементы, лежащие на поверхности тела. Полученное поле давлений переводится в тип данных, необходимый для приложения его в качестве нагрузки.

```
field = LIRE_CHAMP     (
                       TYPE_CHAM = 'NOEU_PRES_R',
                       MAILLAGE = fluid,
                       NOM_MED = 'p',
                       NOM_CMP_MED = '',
                       NOM_CMP='PRES',
                       UNITE = 21
                       )
```

```
proj = PROJ_CHAMP      (
                       METHODE = 'AUTO',
                       MAILLAGE_1 = fluid,
                       MAILLAGE_2 = mesh,
                       CHAM_GD = field,
                       VIS_A_VIS = _F (       TOUT_1 = 'OUI',
                                     GROUP_MA_2 = 'shell'
                                     )
                       )
pres = CREA_RESU       (
                       OPERATION = 'AFFE',
                       TYPE_RESU = 'EVOL_CHAR',
                       NOM_CHAM = 'PRES',
                       AFFE= _F (CHAM_GD = proj, INST = 0)
                       )
load = AFFE_CHAR_MECA (MODELE = model, EVOL_CHAR = pres)
```

Недостатком описанной методики является исключение из рассмотрения влияние сил инерции, что не позволяет учитывать существенные динамические перемещения, деформации и напряжения, возникающие при переходных режимах движения и колебаниях конструкции.

2.2. Динамическая постановка

В случае, когда необходимо учитывать динамические нагрузки на лопасть, вызванные силами инерции и пульсациями аэродинамических нагрузок, однако влиянием изменения формы лопасти на характер обтекания можно пренебречь, допустимо использовать методику расчета вынужденных колебаний конструкции, блок-схема которой показана на рис. 2. Для использования методики требуется, чтобы частоты изменения аэродинамических нагрузок и собственные частоты колебаний лопасти были сильно отстроены друг от друга. Методика состоит из следующих этапов:

- проведение аэродинамического нестационарного расчета в пакете OpenFOAM с использованием решателя pimpleFOAM и определение полей давлений и скоростей как функций от времени;
- перенос полей давлений для каждого момента времени с КО на КЭ сетку и определение нестационарных внешних нагрузок в Code_Aster;
- решение в Code_Aster задачи динамики конструкции под действием полученной нагрузки.

Рис. 2. Алгоритм расчета при динамической постановке задачи
Fig. 2. Simulation algorithm at dynamic problem definition

Ниже показан пример скрипта, исполняемого в препостроцессоре Salome-Meca, автоматически преобразующего результаты расчета поля давлений из формата OpenFOAM в формат *.med для некоторого шага (`n=5`) по времени.

```
n=5
from paraview import simple as sp
reader = sp.OpenDataFile('/my_directory/field.foam')
sp.SetActiveSource(reader)
sp.Show(reader)
view = sp.GetActiveView()
tsteps = reader.TimestepValues
view.ViewTime = tsteps[n]
sp.Render()
writer = sp.CreateWriter('/my_directory/field.med', reader)
writer.UpdatePipeline()
```

Данная методика не позволяет моделировать режимы автоколебаний.

2.3. Квазистатическая постановка с учётом влияния деформации тела на поток

В случае, когда можно пренебречь динамикой движения лопасти, считать режим обтекания лопасти безотрывным и стационарным, однако требуется учесть влияние упругих перемещений лопасти на условия ее обтекания необходимо определить новое равновесное положение обтекаемой деформированной конструкции в потоке. Для этого применяется более сложная методика расчета, в которой используется итерационный цикл, показанный на рис. 3. Одна итерация поиска нового положения равновесия состоит из следующих этапов:

- проведение стационарного аэродинамического расчета методом контрольного объема в пакете OpenFOAM с использованием решателя simpleFOAM. В результате определяются поля скоростей и давлений;
- перенос результатов аэродинамического расчета из пакета OpenFOAM в Code_Aster;
- определение НДС конструкции методом конечных элементов в Code_Aster;
- передача поля перемещений поверхности конструкции из Code_Aster в OpenFOAM для перестроения или деформации сетки.

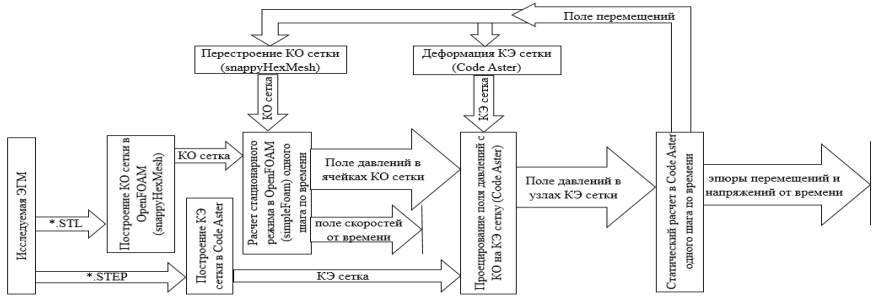


Рис.3. Алгоритм расчета при квазистатической постановке задачи с учетом влиянии деформации тела на поток
Fig. 3. Simulation algorithm at quasistatic problem definition taking into account body deformation influence on a stream

Для корректности расчета необходимо, чтобы на каждой последующей итерации поле давлений проецировалось на узлы деформированной от предыдущей нагрузки КЭ сетки. С этой целью на каждой итерации перед проведением процедуры проецирования поля давлений КЭ сетка сдвигалась согласно, рассчитанным перемещениям, после проецирования сетка возвращалась в исходное положение и проводился статический расчет. Для смещения сетки согласно рассчитанным перемещениям могут быть использованы операторы Code_Aster, приведенные ниже.

```
depl = CREA_CHAMP      (
                       TYPE_CHAM = 'NOEU_DEPL_R',
                       OPERATION = 'EXTR',
                       RESULTAT = stat,
                       NOM_CHAM = 'DEPL',
                       NUME_ORDRE=1
                       )
mesh = MODI_MAILLAGE   (
                       reuse = mesh,
                       MAILLAGE = mesh,
                       DEFORME = _F (OPTION = 'TRAN', DEPL = depl)
                       )
```

3. Применение методик для расчета лопасти ВЭУ

3.1. Исследуемая модель

В качестве исследуемой модели (рис. 4) выбрана лопасть для ветроустановки мощностью 5 МВт [7, 8]. Длина лопасти равна 61,5 метра. Лопасть имеет переменное сечение, состоящее из 6 различных профилей. Площадь поперечного сечения уменьшается от корневого сечения к вершине с максимальным соотношением равным 74,2. Так же лопасть имеет начальную закрутку с плавным увеличением углов установки сечений от вершины к

корню. Максимальный угол закрутки составляет ≈ 13,3 градусов. Конструкция лопасти представляет собой оболочку, подкрепленную по всей длине ребрами жесткости.

На практике для изготовления лопастей ВЭУ используются композиционные материалы, к ним относятся стеклоуглеткани, комбинированные армирующие материалы, сотовые плиты и т.п., расчет прочности конструкций из таких материалов является достаточно сложной задачей и является одним из направлений дальнейшего исследования.

Поскольку в данной работе основное внимание уделялось методике объединения пакетов для решения задачи аэроупругости, в расчетах рассматривалась упрощенная модель конструкции, в которой был использован изотропный материал со следующими механическими характеристиками: модуль упругости E=0,68*10^{11} Па, плотность p=2700 кг/м^3, коэффициент Пуассона n=0,3. Толщина стенок составила d=1,5 мм.

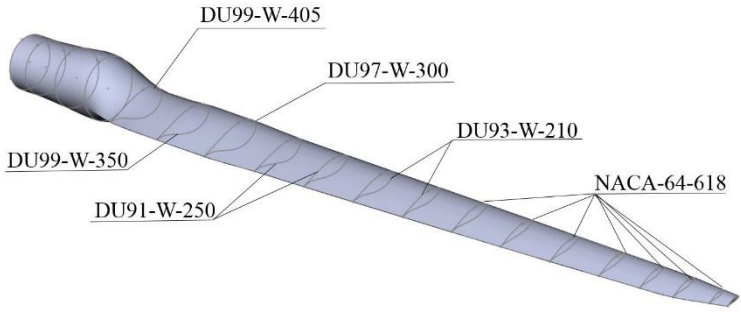


Рис. 4. Исследуемая лопасть ВЭУ
Fig. 4. Research blade of the wind turbine

3.2. Расчёт лопасти при квазистатической постановке

Расчет обтекания лопасти в пакете OpenFOAM проводился на неструктурированной гексаэдральной сетке из 400 тысяч элементов с 4 уровнями сгущения сетки.

В качестве граничных условий на лопасти заданы условия жесткой непроницаемой стенки (для давления – zeroGradient, для скорости – noSlip), а для боковых граней расчетной области заданы условия свободного потока (для давления – freestreamPressure, для скорости - freestream со значением 12 м/с). Число Рейнольдса, рассчитанное по характерному размеру тела D (наибольшая из хорд профилей) составило:

$$Re = \frac{\rho * |U| * D}{\eta} = \frac{|U| * D}{\nu} = \frac{12\ \text{м/с} * 5.5\ \text{м}}{15.1 * 10^{-6}\ \text{м}^2/\text{с}} = 4.37 * 10^6$$

Вычисления проводятся с помощью стационарного решателя simpleFOAM для несжимаемой вязкой среды. Параметры среды соответствуют параметрам

воздуха при 20 ^{0}C (плотность 1,204 кг/м^3 и кинематическая вязкость 1,51*10^{-5} м^2/с). Для моделирования турбулентного течения использовалась модель турбулентности Spalart-Allmaras. Турбулентная вязкость в свободном потоке равняется 0,14 м^2/с, на поверхности лопасти 0. Все физические величины в расчетной области определялись в центре расчетной ячейки. Аппроксимация слагаемых в исходных уравнениях была выполнена со вторым порядком точности по времени и пространству. Уравнения для связи скорости и давления решались с помощью итерационного алгоритма SIMPLE [9].

Полученные поля давлений и скоростей среды представлены на рис. 5 и 6. Из рисунков видно, что, когда поток сталкивается с лопастью, он тормозится и изменяет направление движения, обтекая её. При этом около одной поверхности лопасти возникает область с повышенным давлением воздуха, а около другой поверхности - с пониженным. Величина разницы давлений составляет ≈ 190 Па. Из-за разности давлений на лопасть начинает действовать аэродинамическая сила. График итерационной сходимости расчета представлен на рис.7, график сеточной сходимости проекций аэродинамической силы, действующей на лопасть представлен на рис. 8.

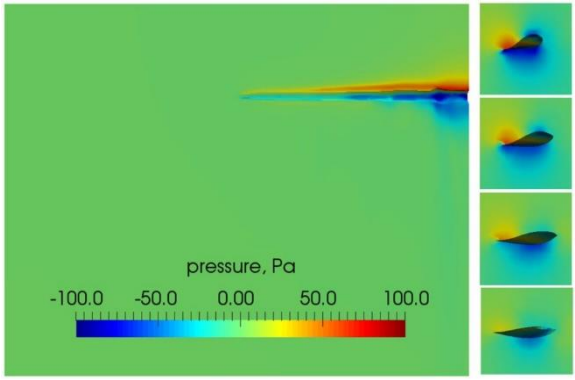


Рис. 5. Полученное поле давлений при стационарном расчете
Сечения (сверху вниз) x = 12 м, x = 24 м, x = 36 м, x = 48 м
Fig. 5. Pressure field at steady-state solution
Sections (from top to down) x = 12 m, x = 24 m, x = 36 m, x = 48 m

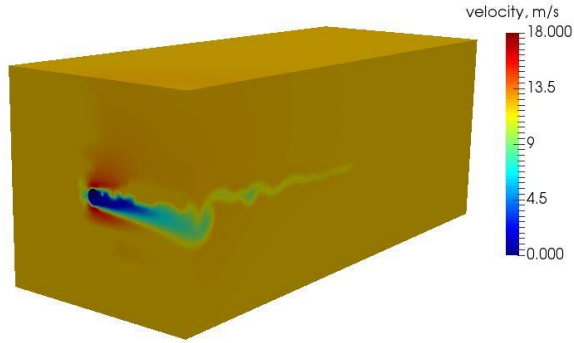


Рис. 6. Полученное поле скоростей при стационарном расчете
Fig. 6. Velocity field at steady state-solution

Распределение значения давления на гранях контрольно-объемной сетки показаны на рис. 9.

Для определения НДС лопасти использовалась конечно-элементная модель, состоящая из треугольных оболочечных элементов первого порядка. Размер сетки составил 7714 элементов (из них 1774 приходятся на ребра жесткости). Внешний вид данной сетки показан на рис. 10.

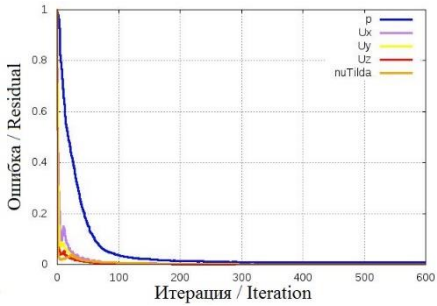


Рис. 7. Сходимости невязки давления, проекций скорости и турбулентной вязкости
Fig. 7. Pressure, speed and turbulent viscosity residuals

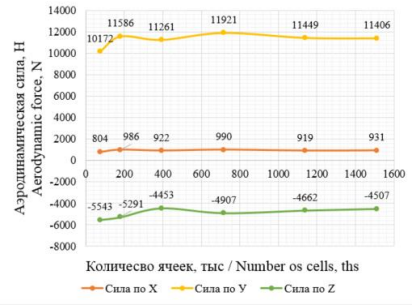
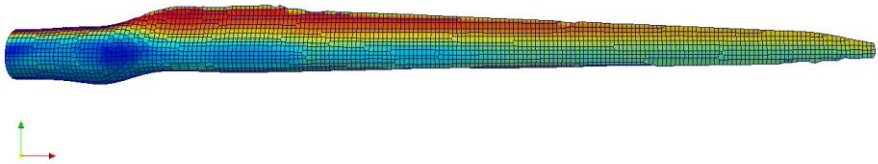


Рис. 8. Графики сходимости по сетке значений проекций аэродинамической силы, действующей на лопасть
Fig. 8. Aerodynamic force mesh convergence



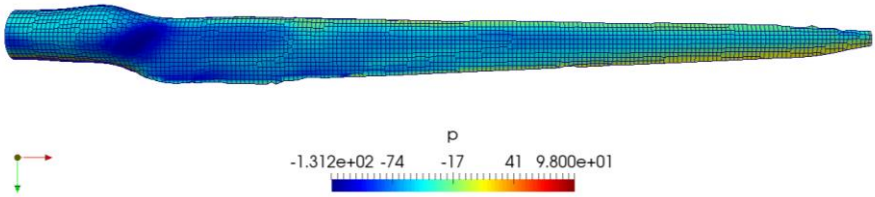


Рис. 9. Распределение аэродинамического давления в КО у поверхности лопасти
Fig. 9. Aerodynamic pressure in the CV at the blade surface

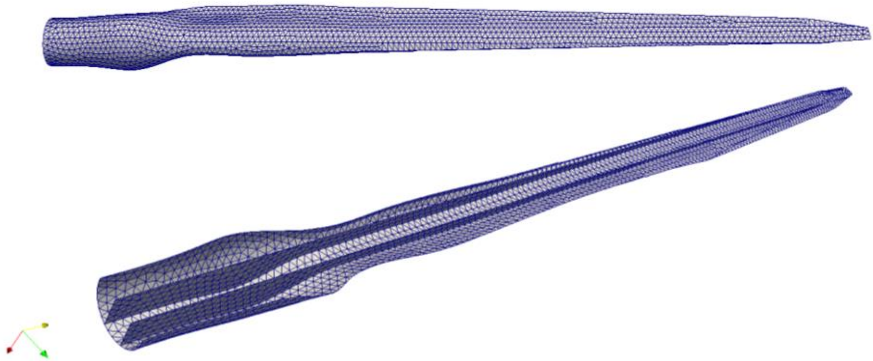


Рис. 10. Конечно-элементная сетка лопасти
Fig. 10. Finite element mesh of the blade

Результат применения операций проецирования давления показан на рис.11. Следует отметить, что в данной модельной задаче рассматривалась только аэродинамическая нагрузка, а силы другой природы (центробежные, собственный вес лопасти) не учитывались.

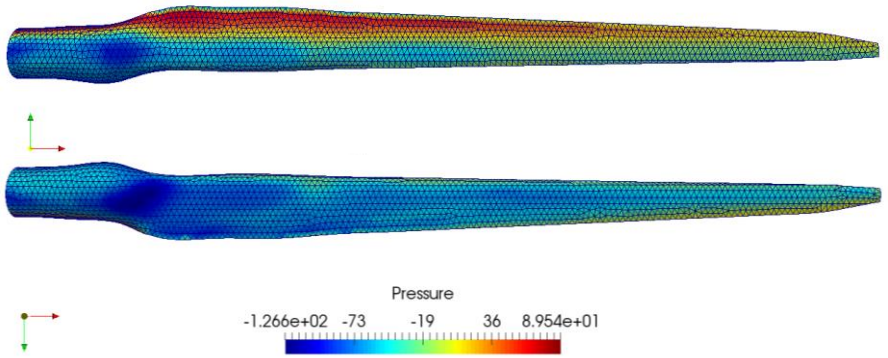


Рис. 11. Поле давлений, интерполированное на КЭ сетку
Fig. 11. Pressure field interpolated to the FE mesh

Полученное поле давлений было использовано в качестве внешней нагрузки при определении НДС лопасти, закрепленной по левому торцу. Результаты расчета значений перемещений узлов и напряжений показаны на рис. 12, 13.

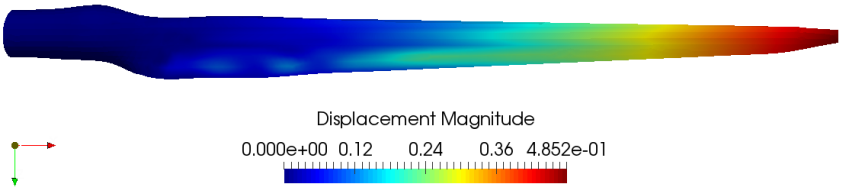


Рис. 12. Эпюра перемещений
Fig. 12. Plot of the displacement

По результатам расчета можно сделать вывод, что при заданной скорости потока 12 м/с максимальные перемещения на конце лопасти составляют около 0.49 метра, а в конструкции лопасти из изотропного материала не возникает напряжений, приводящих к разрушению.

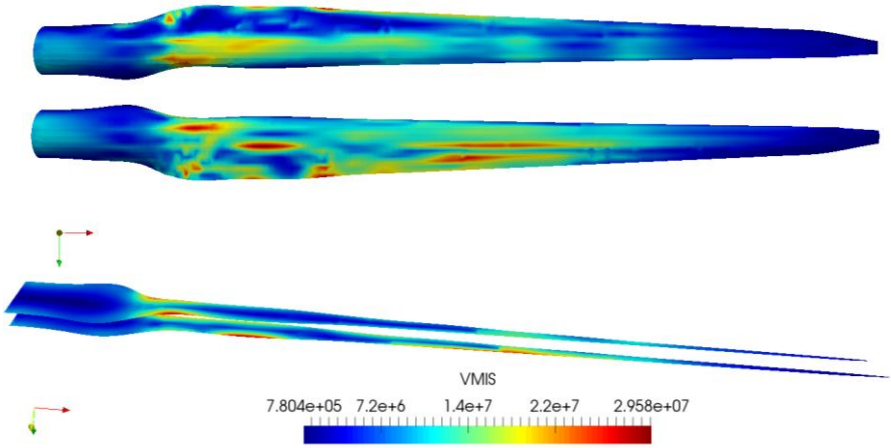


Рис. 13. Эпюра напряжений (эквивалентные по Мизесу)
Fig. 13. Plot of the stress (Von Mises)

3.3. Расчёт лопасти в динамической постановке

При решении нестационарной задачи обтекания в OpenFOAM вычисления проводились с помощью нестационарного решателя pimpleFOAM для несжимаемой вязкой среды. В качестве начальных полей давления, скорости и турбулентной вязкости использовались результаты стационарного расчета. Шаг по времени выбирался автоматически из условия для значения числа

Куранта $Co_{max} < 0.9$. Характерная величина шага по времени – 0.005 секунды. Время расчета переходного режима длительностью 10 секунд составило 2 часа на 1 ядре (Intel(R) Xeon(R) CPU X5670, 2.93GHz). Для проведения расчетов был использован открытый облачный сервис UNIHUB от ИСП РАН [10, 11].

В качестве примера для двух точек, лежащих вблизи поверхности лопасти, на рис. 14 приведены графики зависимости давлений от времени.

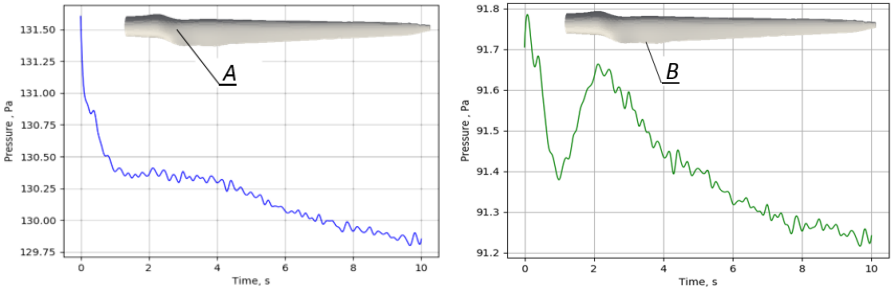


Рис. 14. Графики давлений для двух точек (A и B) вблизи поверхности лопасти
Fig. 14. Pressure plots for two points (A and B) near the surface of the blade

Полученные в результате аэродинамического расчета поля давлений для каждого момента времени были спроецированы на КЭ сетку, и для каждого момента времени был составлен вектор внешней нагрузки. Динамический расчет переходного режима колебаний лопасти был проведен для той же модели, что и статический. При этом было введено демпфирование по Рэлею с коэффициентами $\alpha=\beta=0.05$. В качестве контролируемого параметра была выбрана Y-компонента перемещений точки, лежащей на свободном конце лопасти. Для этой точки был построен график перемещений на интервале времени 0...6 секунды. Шаг интегрирования был выбран 0.1 секунды. На рис. 15 показано сравнение переходных режимов для двух случаев. В первом случае (синий) внешняя нагрузка была нестационарной (менялась на каждом шаге интегрирования), а во втором случае (красный) нагрузка была постоянной, внезапно приложенной в начальный момент времени.

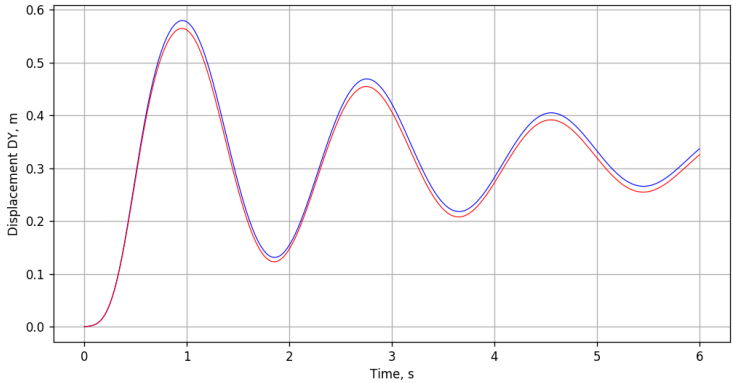


Рис. 15. Перемещения точки конца лопасти
Fig. 15. Displacement of the tip of the blade

Как видно из графика на рис. 15, в рассмотренном случае учет нестационарности аэродинамической нагрузки поля давлений практически не оказывает значительного влияния на параметры переходного процесса. Объяснить это можно тем, что собственные частоты лопасти (низшая собственная частота 0.56 Гц) значительно ниже частот изменения внешней нагрузки (3.0...4.0 Гц). Максимальное перемещение конца лопасти в переходном режиме составило около 0.58 метра (что на 18% больше статического перемещения в предыдущем расчете), в то время как перемещение в новом статическом положении равновесия, к которому стремиться переходный режим составляет около 0.32 метра, что на 30% меньше чем перемещение в расчете по первой методике.

В дальнейшем расчет может быть проведен с помощью вихреразрешающего моделирования, с использованием метода крупных вихрей, для получения мгновенных составляющих значений скорости и давления [12].

3.4. Расчёт лопасти при квазистатической постановке с учётом влияния деформации конструкции на поток

Для той же самой модели согласно блок-схеме алгоритма, показанной на рис. 3, было проведено 5 итераций (внутренних циклов). Эпюры перемещений лопасти для некоторых итераций даны на рис. 16. График перемещений свободного конца лопасти показан на рис. 17. Видно, что учет влияния деформаций лопасти существенно влияет на параметры ее обтекания, поскольку в данном случае максимальное перемещение составило около 0.53 метра, что на 10% больше, чем в квазистатическом расчете по первой методике.

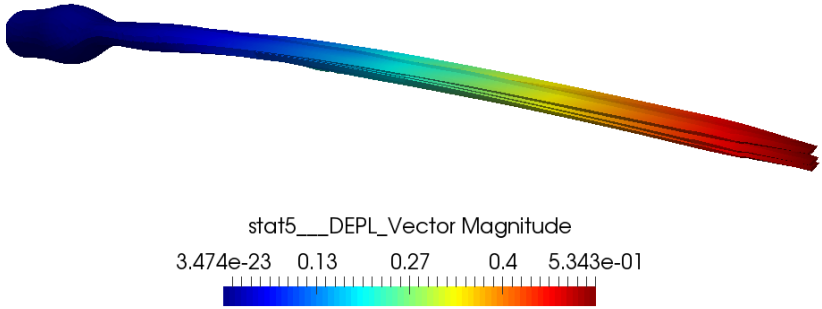


Рис. 16. Поле перемещения лопасти на каждой итерации
Fig. 16. Blade displacement on each iteration

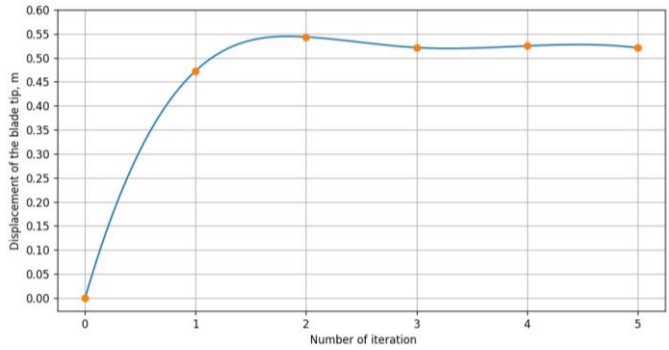


Рис. 17. Перемещения точки конца лопасти
Fig. 17. Displacement of the tip of the blade

4. Выводы

Работа позволяет сделать вывод о том, что пакеты свободного программного обеспечения OpenFOAM и Code_Aster в сочетании с препостпроцессором Salome-Meca имеют все необходимые средства для создания на их базе единого программного комплекса по решению задач статической аэроупругости и исследованию малых вынужденных колебаний лопасти ВЭУ в потоке вязкой среды.

Предложенные и протестированные на модельной задаче три простых методики связи двух пакетов, основанные на переносе результатов между расчетными сетками, показывают устойчивость счета и адекватность получаемых результатов. Однако величины прогибов, полученные по

различным методикам, существенно различаются, что требует проведения дальнейшей верификации.

В дальнейшем представленные методики могут быть использованы для решения более сложных задач верификации методик по известным экспериментальным данным. Так же полученные результаты являются базой для реализации метода решения полностью связанных задач аэроупругости на базе свободного программного обеспечения.

Благодарности и ссылки на Гранты

Работа выполнена при финансовой поддержке РФФИ (грант № 17-07-01391).

Список литературы

[1]. Бисплингхофф Р.Л., Эшли Х., Халфмэн Р.Л. Аэроупругость. М. Изд-во Иностр. лит. 1958, 800 с.

[2]. Горшков А.Г., Морозов В.И., Пономарев А.Т., Шклярчук Ф.Н. Аэрогидроупругость конструкций. М.: Физматлит, 2000, 592 стр.

[3]. Tukovic Z., Jasak H., Updated Lagrangian finite volume solver for large deformation dynamic response of elastic body. Transaction of FAMENA XXX (1), 2007, pp. 599–608.

[4]. Kotsur O., Scheglov G., Leyland P. Verification *of* modelling *of* fluid structure interaction *(*FSI*)* problems based *on* experimental research *of* bluff body oscillations *in* fluids. In Proceedings: 29th Congress of the International Council of the Aeronautical Sciences, 2014.

[5]. Sekutkovski B., Kostic I., Simonovic A., Cardiff P., Jazarevic V. Three-dimensional fluid-structure interaction simulation with a hybrid RANS-LES turbulence model for applications in transonic flow domain. Aerospace Science and Technology, vol. 49, 2016, pp. 1-16.

[6]. Benra F.-K., Dohmen H. J., Pei J., Schuster S., Wan B. A Comparison of One-Way and Two-Way Coupling Methods for Numerical Analysis of Fluid-Structure Interactions. Journal of Applied Mathematics. Article ID 853560, 2011, p. 16, doi:10.1155/2011/853560

[7]. Resor B. R. Definition of a 5MW/61.5m Wind Turbine Blade Reference Model. SANDIA REPORT SAND2013-2569 Unlimited Release, Printed April 2013, pp.1-53.

[8]. Jonkman J., Butterfield S., Musial W., Scott G. Definition of a 5-MW Reference Wind Turbine for Offshore System Development, National Renewable Energy Laboratory. US, Colorado, Technical Report NREL/TP-500-38060, February 2009, p. 75.

[9]. Weller H.G., Tabor G., Jasak H., Fureby C. A tensorial approach to computational continuum mechanics using object oriented techniques, Computers in Physics, vol.12, № 6, 1998, pp. 620-631.

[10]. Крапошин М.В., Самоваров О.И., Стрижак С.В. Особенности реализации Web-лаборатории механики сплошной среды на базе технологической платформы программы “Университетский кластер”. Труды международной суперкомпьютерной конференции, 2011.

[11]. UniHUB: сайт. Режим доступа: http://www.unicluster.ru/unihub.html (дата обращения: 13.11.2017).

[12]. Strijhak S., Redondo J.M., Tellez J. Multifractal analysis of a wake for a single wind turbine. Topical Problems of Fluid Mechanics 2017: Proceedings, 2017. pp. 275-284.

The method of solving aeroelasticity problems for wind blade using open source software

[1, 2] *P.S. Lukashin <ski1l@mail.ru>*
[1, 2] *V.G. Melnikova <vg-melnikova@yandex.ru>*
[2] *S.V. Strijhak <strijhak@yandex.ru>*
[1] *G.A. Shcheglov <shcheglov_ga@bmstu.ru>*
[1] *Bauman Moscow State Technical University,*
5/1, 2-nd Baumanskaya st., Moscow. 105005, Russia
[2] *Ivannikov Institute for System Programming of the RAS,*
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. Due to the development of Wind Energy and construction of new wind farms in Russian Federation there is a need for the solution of application-oriented problems and development of effective methods for calculation of wind turbine's elements. One of the directions for computational continuous mechanics is connected with problems in aeroelasticity (fluid-structure interaction). The possibility of solving one of the problem in aeroelasticity using a complex program approach on the basis of open source software OpenFOAM and Code_Aster is shown in this article. On the example of the blade for wind turbine, 61.5 meters long, the techniques of solving problem for a static and dynamic aeroelasticity in which calculation of flow of the blade with a subsonic air flow is done in OpenFOAM library (solvers simpleFOAM and pimpleFOAM) are considered. The calculation of the intense deformed status of the blade is done in Code_Aster code. The flowcharts for three different approaches for solving problems of aeroelasticity, examples of scripts and command files for data transfer between two codes in the course of calculation are provided in article. The control-volume mesh consisting their hexahedral elements, the total number is about 400000 elements, for calculation of flow around the blade is constructed in OpenFOAM library, the finite-element mesh consisting of triangular shell elements of first order, the total number is 7714, for calculation of the intense deformed status is constructed in Salome-Meca code. The results of calculation are provided in the form of fields for pressure and velocities; graphics for residuals of pressure, velocity, turbulent viscosity; projections of aerodynamic force from time; diagrams of displacement and stress; the values of pressure for two points for the surfaces and displacement of the tip of the blade from time. The calculations are run using resources of UniHUB web-laboratory ISPRAS.

Keywords: aeroelasticity; wind blade; solver, Code_Aster; OpenFOAM; finite-volume method, finite-element method, mesh, turbulence model, static, dynamic, frequency, displacement, stress, calculations, web-laboratory.

DOI: 10.15514/ISPRAS-2017-29(6)-16

For citation: Lukashin P.S., Melnikova V.G., Strijhak S.V., Shcheglov G.A. The method of solving aeroelasticity problems for wind blade using open source software. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 6, 2017, pp. 253-270 (in Russian). DOI: 10.15514/ISPRAS-2017-29(6)-16

References

[1]. Bisplinghoff R.L., Ashley H., Halfman R.L. Aeroelasticity. Dover Publications, Inc., Mineola, N.Y., 1996, 800 p.

[2]. Gorshkov A.G., Morosov V.I., Ponomarev A.T., Schklyaruk F.N. Aeroelasticity of structures. M.: Fizmatlit, 2000, 592 p. (in Russian).

[3]. Tukovic Z., Jasak H., Updated Lagrangian finite volume solver for large deformation dynamic response of elastic body. Transaction of FAMENA XXX (1), 2007, pp. 599–608.

[4]. Kotsur O., Scheglov G., Leyland P. Verification *of* modelling *of* fluid structure interaction *(*FSI*)* problems based *on* experimental research *of* bluff body oscillations *in* fluids. In Proceedings: 29th Congress of the International Council of the Aeronautical Sciences, 2014.

[5]. Sekutkovski B., Kostic I., Simonovic A., Cardiff P., Jazarevic V. Three-dimensional fluid-structure interaction simulation with a hybrid RANS-LES turbulence model for applications in transonic flow domain. Aerospace Science and Technology, vol. 49, 2016, pp. 1-16.

[6]. Benra F.-K., Dohmen H. J., Pei J., Schuster S., Wan B. A Comparison of One-Way and Two-Way Coupling Methods for Numerical Analysis of Fluid-Structure Interactions. Journal of Applied Mathematics. Article ID 853560, 2011, p. 16, doi:10.1155/2011/853560

[7]. Resor B. R. Definition of a 5MW/61.5m Wind Turbine Blade Reference Model. SANDIA REPORT SAND2013-2569 Unlimited Release, Printed April 2013, pp.1-53.

[8]. Jonkman J., Butterfield S., Musial W., Scott G. Definition of a 5-MW Reference Wind Turbine for Offshore System Development, National Renewable Energy Laboratory. US, Colorado, Technical Report NREL/TP-500-38060, February 2009, p. 75.

[9]. Weller H.G., Tabor G., Jasak H., Fureby C. A tensorial approach to computational continuum mechanics using object oriented techniques, Computers in Physics, vol.12, № 6, 1998, pp. 620-631.

[10]. Kraposhin M.V., Samovarov O.I., Strijhak S.V. The features of design Web-laboratory of computational continuum mechanics on the basis of the technological platform in scope of the program "University Cluster". The Proceedings of the International Supercomputer Conference, 2011 (in Russian).

[11]. UniHUB. Available at: http://www.unicluster.ru/unihub.html (Accessed 13 November 2017).

[12]. Strijhak S., Redondo J.M., Tellez J. Multifractal analysis of a wake for a single wind turbine. Topical Problems of Fluid Mechanics 2017: Proceedings, 2017. pp. 275-284.

Software package to calculate the aerodynamic characteristics of aircrafts

V.N. Koterov <vkoterov@yandex.ru>
V.M. Krivtsov <vlkrivtsov@yandex.ru>
V.I. Zubov <vladimir.zubov@mail.ru>
Dorodnicyn Computing Centre FRC CSC RAS,
40, Vavilov st., Moscow, 119333, Russia.
Moscow Institute of Physics and Technology (State University),
Institutskii per. 9, Dolgoprudnyi, Moscow oblast, 141700, Russia

Abstract. The software package to calculate parameters of three-dimensional steady and unsteady gas flows in complex devices is presented. The mathematical flow model used in package is based on the Reynolds-averaged Navier Stokes equations for a two-component equilibrium turbulent medium and a two-parameter semiempirical turbulence model. A numerical implementation of the software package to modelling three complex flows in modern devices is described.

Keywords: software package; mathematical simulation; gas flows; turbulence; exhaust jet; Navier Stokes equations; aircraft nozzle.

DOI: 10.15514/ISPRAS-2017-29(6)-17

For citation: Koterov V.N, Krivtsov V.M., Zubov V.I. Software package to calculate the aerodynamic characteristics of aircrafts. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 6, 2017, pp. 271-288. DOI: 10.15514/ISPRAS-2017-29(6)-17

1. Introduction

The computation of characteristics of steady gas flows around prospective flight vehicles is a complicated task combining external and internal aerodynamics problems, which can be addressed with state-of-the-art computers. Formally, an aircraft can be represented as a streamlined body equipped with an internal gasdynamic nozzle. The ambient gas flow interacts heavily with the nozzle flow. Additionally, it should be noted that the flow characteristics in an extended region surrounding the nozzle are not known beforehand. The difficulties arising in the numerical simulation of a flow of this type are as follows:

- the flow is essentially three-dimensional and turbulent and, in the general case, can be of mixed type (sub- and supersonic);

- the computational domain has to be large, which is required for the flow characteristics near the aircraft to be reliably determined;
- relatively narrow boundary layers developing on the aircraft surface have to be resolved;
- the moving medium is multicomponent. In the general case, this is a mixture of fuel combustion products and the ambient air flowing past the aircraft;
- additionally, some difficulties are usually associated with the representation of numerically computed three-dimensional gasdynamic fields in a convenient form.

The goal of this work is to demonstrate developed mathematical model of the flow, numerical algorithm and a software package in order to estimate the influence exerted on the aircraft characteristics by user-specified input parameters.

2. Mathematical formulation of the problem

The moving gas was simulated as an equilibrium two-component mixture of ambient air (fraction 1) and fossil fuel combustion products (fraction 2). The flow of an equilibrium gas mixture was described by the three-dimensional unsteady Reynolds-averaged compressible Navier-Stokes equations closed with a model of eddy viscosity and thermal conductivity (RANS model) (see, for example, [1, 2]).

2.1 Basic Equations

In Cartesian coordinates, these equations can be written as

$$\frac{\partial \rho \mathbf{V}}{\partial t} + \nabla \cdot \left(\rho \mathbf{\Lambda} - \mathbf{\Pi}\right) + \nabla p = 0, \quad \frac{\partial}{\partial t}\left(\rho H - p\right) + \nabla \cdot \left(\rho H \mathbf{V} - \mathbf{\Pi V} - k_\Sigma \nabla T\right) = 0,$$

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \left(\rho \mathbf{V}\right) = 0, \quad \frac{\partial \rho_a}{\partial t} + \nabla \cdot \left(\rho_a \mathbf{V} - D_\Sigma \nabla \rho_a\right) = 0, \quad \gamma = \frac{\rho_a}{\rho},$$

$$\mathbf{\Lambda} = \left\| u_i u_j \right\|^T, \quad \mathbf{\Pi} = \mu_\Sigma \left\| -\frac{2}{3}\delta_{ij} \nabla \cdot \mathbf{V} + \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right\|^T, \quad H = h + \frac{1}{2}\sum_{i=1}^{3} u_i^2,$$

$$p = R_0 \rho T \left(\frac{1-\gamma}{m_1} + \frac{\gamma}{m_2} \right), \quad h = \int_{T_0}^{T} C_p(T')\, dT' + h_0, \quad C_p = (1-\gamma) C_{p1} + \gamma C_{p2},$$

$$\mu_\Sigma = \mu + \mu_T, \quad k_\Sigma = C_p \left(\frac{\mu}{\mathrm{Pr}} + \frac{\mu_T}{\mathrm{Pr}_T} \right), \quad D_\Sigma = \frac{1}{\rho}\left(\frac{\mu}{\mathrm{Sc}} + \frac{\mu_T}{\mathrm{Sc}_T} \right).$$

Here, t is time; $x_1 = x$, $x_2 = y$ and $x_3 = z$ are the Cartesian coordinates; $\mathbf{V}=\| u_i \|^T$ is the flow velocity with the components u_i, (i = 1, 2 ,3); δ_{ij} - Kronecker Delta; ρ, p, T, h,

and C_p are the medium's density, pressure, temperature, specific enthalpy, and heat capacity at constant pressure, respectively; R_0 is the universal gas constant; C_{p1}, m_1, C_{p2}, and m_2 are the respective heat capacities and molar masses of the combustion gases and air; ρ_a is the reduced air density in the mixture; γ is the mass fraction of the air in the mixture; μ_Σ and k_Σ are the effective viscosity and heat conductivity of the medium; μ and μ_T are the molecular and eddy viscosities of the mixture; Pr and Pr_T are the laminar and turbulent Prandtl numbers; D_Σ is the effective diffusivity; and Sc and Sc_T are the laminar and turbulent Schmidt numbers.

The eddy viscosity and heat conductivity were computed using Coacley's q–ω model [3]. In Cartesian coordinates, the equations of the q–ω model can be written as

$$\frac{\partial \rho q}{\partial t} + \nabla \cdot \left[\rho q \mathbf{V} - (\mu + \mu_T)\nabla q\right] = \frac{\rho q}{2\omega}\left(C_\mu f S - \frac{2}{3}\omega \nabla \cdot \mathbf{V} - \omega^2\right),$$

$$\frac{\partial \rho \omega}{\partial t} + \nabla \cdot \left[\rho \omega \mathbf{V} - (\mu + 1.3\mu_T)\nabla \omega\right] = \rho\left[C_1\left(C_\mu S - \frac{2}{3}\omega \nabla \cdot \mathbf{V}\right) - C_2\omega^2\right],$$

$$\mu_T = C_\mu f(n)\rho\frac{q^2}{\omega}, \quad S = \sum_{i,j=1}^{3}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right)\frac{\partial u_i}{\partial x_j}, \quad f(n) = 1 - e^{-0.0065\frac{\rho q n}{\mu}},$$

$$C_1 = 0.045 + 0.405 f(n), \quad C_2 = 0.92, \quad C_\mu = 0.09.$$

Here, q and ω are the "pseudovelocity" and "pseudovorticity"; $f(n)$ is the near-wall function introduced to correctly describe the flow parameters in the laminar sublayer developing at the walls; n is the normal distance from a given point to the nearest surface; S is a dissipative function.

2.2 Boundary Conditions

The conditions on the boundaries of the flow region must take into account the following data:

- The state of the unperturbed air flow far away from the aircraft (at the outer boundary of the computational domain and in the outlet cross section);
- The heat transfer conditions on the solid walls;
- The state of the flow in the inlet cross section of the nozzle (if present).

In the case of supersonic gas flows, the flow state away from the aircraft was specified by three parameters: the free-stream velocity $\mathbf{V}_\infty$, the static pressure p_∞, and the static temperature T_∞.

The boundary conditions on the solid walls were set as follows:

$$u_i = 0, \quad (i=1,2,3), \qquad T = T_w \quad (\text{or} \quad \partial T/\partial n = 0),$$
$$q = 0, \quad \partial\omega/\partial n = 0,$$

where T_w is a given wall temperature and n is the normal to the wall.

3. Computational Algorithm

The numerical solution to the boundary value problem stated above was obtained using finite differences.

3.1 Numerical Grid

The first step in the finite-difference implementation was the generation of a uniform three-dimensional mesh in the computational domain. This problem was an important inherent element of the numerical algorithm and was comparable in complexity with the other algorithmic parts. The accuracy of the results was determined to a large extent by the quality of the mesh used.

The grid algorithm generates three-dimensional boundary-conforming structured meshes with hexahedral cells. Degenerate (pentahedral) cells could be used on the axis of symmetry of the domain (if any).

There are numerous commercial and open-access grid generators for constructing boundary-conforming hexahedral structured meshes (see [4-6]). As a rule, algorithms for generating such grids involve two principal stages:

- The construction of a parametric representation of the surfaces bounding the computational domain (including grid node arrangement on these surfaces);
- The generation of a three-dimensional structured mesh inside the domain.

In domains of complex geometry, these generators can sometimes produce degenerate meshes with self-intersecting cells. It should also be kept in mind that, even in the case of nondegenerate meshes, a fairly accurate solution for internal and external turbulent gas flows can be obtained if the equations and the boundary conditions are well approximated, which imposes rather severe restrictions on the quality of these meshes.

The term "quality of a numerical grid" cannot be defined rigorously. In this case, by such we mean the following collection of intuitive and, generally speaking, quite incompatible requirements:

- The mapping to a parametric parallelepiped used in structured mesh generation must be continuously differentiable and its Jacobian must not vanish.
- To resolve the boundary layers, the mesh must be refined toward the solid walls. Moreover, the transverse (to the flow direction) mesh size near these surfaces can be 10-4 and less of the characteristic length of the problem.
- The grid lines near the walls must be nearly orthogonal to them. Wherever possible, the mesh cells must be similar to parallelepipeds.
- The numerical grid must be quasi-uniform in the sense that, as the number

of grid nodes tends to infinity, the difference between the characteristic lengths of neighboring cells tends to zero faster than the lengths themselves. Anyway, the ratio of the characteristic lengths of neighboring cells must not be too large.

The numerical results presented below were obtained on grids generated by the algorithms described in [6].

3.2 Conservative Approximation of Spatial Operators

The approximations of the spatial operators used in the algorithm are described as applied to the Navier-Stokes equations given in Section 2. These approximations are underlain by the following identities, which are often used as definitions of the corresponding operators:

$$\operatorname{div}\mathbf{g} = -\lim_{\Omega\to 0}\frac{\oint \mathbf{g}\cdot\mathbf{n}\,d\Gamma}{|\Omega|}, \quad \operatorname{grad} p = -\lim_{\Omega\to 0}\frac{\oint p\mathbf{n}\,d\Gamma}{|\Omega|},$$

where **n** is the inward normal to the boundary Γ of the domain **Ω,** |Ω| is the volume of **Ω,** and **g** and *p* are arbitrary vector and scalar fields. Note also that such approximations lead to skew-symmetric matrices approximating first spatial derivatives.

To explain what was said above, we consider the case of flat structured meshes consisting of quadrilaterals. Cells in structured meshes are numbered by indices. Each face is adjacent to two cells, whose indices differ in one position. The faces are numbered by half-integer indices: $(i-1/2,\ j)$, $(i+1/2,$ j), $(i,\ j-1/2)$, and (i, j+1/2). Assume that all the sought quantities **g** and *f* are associated with cells (in the smooth case, with barycenters, i.e., the centers of mass of cells).

To use previous formulas, the sought quantities have to be extended to faces. To recover the values φ of the function φ on faces, we introduce two extension procedures, M_+ and $M\_$. The input data are the values of φ on both sides of a face. For the face between the cells (i, j) and (i, j+1), the procedures are defined as

$$\varphi^{+}_{i,j+1/2} = M_{+}^{i,j+1/2}(\varphi_{i,j},\varphi_{i,j+1}) = \varphi_{i,j}, \quad \varphi^{-}_{i,j+1/2} = M_{-}^{i,j+1/2}(\varphi_{i,j},\varphi_{i,j+1}) = \varphi_{i,j+1}.$$

For the face between the cells (i, j) and $(i+1, j)$, they are defined in a similar manner. Note that these extension procedures ensure only the first order of accuracy.

For simplicity, we consider the approximation of the derivative

$$\frac{\partial f}{\partial x} = -\lim_{\Omega\to 0}\frac{\oint f n_x\,d\Gamma}{|\Omega|},$$

where n_x is the projection of the inward normal onto the *x* axis.

To use the above formula, we need to know the values of f on the faces of the current cell. Assume that, on a structured quadrilateral mesh, the face extensions are produced by the procedure M_+. Then the above derivative is approximated as

$$f^+_{x,i,j} = -\frac{(f^+Sn_x)_{i-1/2,j} + (f^+Sn_x)_{i+1/2,j} + (f^+Sn_x)_{i,j-1/2} + (f^+Sn_x)_{i,j+1/2}}{|\Omega_{i,j}|},$$

where S is the edge length and $|\Omega_{i,j}|$ is the cell area. For each cell (i, j), we have

$$(Sn_x)_{i-1/2,j} + (Sn_x)_{i,j-1/2} + (Sn_x)_{i+1/2,j} + (Sn_x)_{i,j+1/2} = 0.$$

Consider the scalar product of the derivative f^+_x and an arbitrary compactly supported function φ:

$$(f^+_x, \varphi) = \sum_{i,j} f^+_{x,i,j}\varphi_{i,j}|\Omega_{i,j}|.$$

In this sum, we group the terms involving $f_{i,j}$. There are four of them: two correspond to the cell (i, j) and a single term corresponds to each of the cells $(i+1, j)$ and $(i, j+1)$; specifically,

$$(((Sn_x)_{i+1/2,j} + (Sn_x)_{i,j+1/2})\varphi_{i,j} - (Sn_x)_{i+1/2,j}\varphi_{i+1,j} - (Sn_x)_{i,j+1/2}\varphi_{i,j+1})f_{i,j}.$$

In the terms corresponding to the cell (i, j), the sum of edge lengths multiplied by the corresponding components of the normals is replaced using the above equality. As a result, we obtain the relation

$$(f^+_x, \varphi) = -(f, \varphi^-_x),$$

where the derivative φ^-_x of φ is given by the same formula as for f^+_x but with φ extended to the faces with the help of the procedure M_-.

The first y-derivatives f_y of the function f can be approximated in a similar fashion.

Using the basis vectors $\mathbf{e}_x$ and $\mathbf{e}_y$, we can construct the operations $\mathrm{grad}^{\pm}f$, $\mathrm{div}^{\pm}\mathbf{g}$, and $\mathrm{curl}^{\pm}\mathbf{g}$.

To determine the order of accuracy of the proposed formulas, we consider a uniform rectangular mesh with edge lengths S_i and S_j. The formula for computing the derivative $f^+_{x,i,j}$ involves $f_{i-1,j}$ and $f_{i,j-1}$, which can be represented as

$$f_{i-1,j} = f_{i,j} - \frac{\partial f_{i,j}}{\partial S_i}S_i + 0.5\frac{\partial^2 f_{i,j}}{\partial S_i^2}S_i^2 + o(S_i^2), \quad f_{i,j-1} = f_{i,j} - \frac{\partial f_{i,j}}{\partial S_j}S_j + 0.5\frac{\partial^2 f_{i,j}}{\partial S_j^2}S_j^2 + o(S_j^2).$$

After substituting these representations into the formula for $f^+_{x,i,j}$, it is easy to see that the truncation error in the approximation of $\partial f/\partial x$ by f^+_x is first order.

Note that the leading terms of the truncation errors in f_x^+ and f_x^- are identical in absolute value but opposite in sign. Therefore, the derivative $\partial f/\partial x$ is approximated with second order accuracy by the combination $0.5(f_{x,i,j}^+ + f_{x,i,j}^-)$ in a cell and by $0.5(f_{x,i,j}^+ + f_{x,i+1,j}^-)$ and $0.5(f_{x,i,j}^+ + f_{x,i,j+1}^-)$ on the corresponding edge. This idea was used in [7].

An alternative approach to an increase in the order of accuracy is to extrapolate or interpolate function values from the cells adjacent to the given face.

Next, the values on cell boundaries can be obtained using exact (iterative) or approximate Riemann solvers (see [8]). This approach was implemented in a software code.

3.3 Approximation of Viscous Operators

The second-order operators related to viscosity are approximated by applying the variation principle. Specifically, the viscous operators in momentum equations (see Section 2) can be obtained by varying a special functional with a symmetric nonnegative quadratic part

$$J(\mathbf{V}) = \int_\Omega \left[I(\mathbf{V}) - 2(\mathbf{V}\cdot\Theta) \right] dxdydz$$

with respect to the components u, v and w of the velocity $\mathbf{V}$; here,

$$I(\mathbf{V}) = \mu\left[\frac{4}{3}(u_x^2 + v_y^2 + w_z^2 - u_x v_y - v_y w_z - u_x w_z) + \right.$$
$$\left. + (u_y + v_x)^2 + (u_z + w_x)^2 + (v_z + w_y)^2 \right]$$

is a dissipation function, $\boldsymbol{\Theta}$ is the vector $\boldsymbol{\Theta}$=(div τ_x, div τ_y, div τ_z), and $\boldsymbol{\tau}$ is the viscous stress tensor. The first variation of functional $I(\mathbf{V})$ is computed in the class of functions satisfying given boundary conditions.

In each mesh cell c, the components of the velocity gradient are determined as described for the pressure gradient. Then the approximate value $J_h(\mathbf{V})$ of functional $I(\mathbf{V})$ is calculated as

$$J_h(\mathbf{V}) = \sum_{c\in\Omega} \left\{ 0.5\left[\tilde{I}^+(\mathbf{V}(c)) + \tilde{I}^-(\mathbf{V}(c)) \right] - 2(\mathbf{V}\cdot\Theta) \right\} |\Omega_c|,$$

where the sum is taken over all cells of the computational domain, $|\Omega_c|$ is the volume of the c-th cell, and $\tilde{I}^\pm$ are approximations of $I(\mathbf{V})$ with first derivatives determined by the procedures M_+ and $M\_$, respectively. Note that

$$I(\mathbf{V}) \le 2\mu(u_x^2 + u_y^2 + u_z^2 + v_x^2 + v_y^2 + v_z^2 + w_x^2 + w_y^2 + w_z^2).$$

Relying on this inequality, a simple algorithm can be constructed in which the viscous terms in the considered equations are taken into account in the form of correction terms at the next iteration (see [9]).

3.4 Implicit Scheme and Iterative Algorithm

Governing Equations are numerically solved using a two-level fully implicit difference scheme of the form

$$\frac{\boldsymbol{\varphi}^{k}-\boldsymbol{\varphi}^{k-1}}{\Delta t}+A^{-1}H(\boldsymbol{\varphi}^{k})=0,$$

where $\boldsymbol{\varphi}$ is a grid vector function involving all unknown functions, k is the time level index, Δt is the time step, H is an operator containing first and second difference derivatives with respect to spatial variables, and A^{-1} is an operator improving the convergence of the iterations (in the simplest case, $A = E$, where E is the identity operator). The norm of $H(\boldsymbol{\varphi}^{k})$ is a measure indicating the proximity of the solution to a steady state.

We use conservative (flux) approximations of the spatial operators. Conservative variables are associated with mesh cells, while flux variables, with cell boundaries. The resulting system of nonlinear algebraic equations is solved using the following simple iterative procedure at each grid node:

$$\boldsymbol{\varphi}^{k}_{(s+1)}=\boldsymbol{\varphi}^{k}_{(s)}-\xi\mathbf{G}\left(\boldsymbol{\varphi}^{k}_{(s)}\right),\quad \mathbf{G}\left(\boldsymbol{\varphi}^{k}_{(s)}\right)=\boldsymbol{\varphi}^{k}_{(s)}-\boldsymbol{\varphi}^{k-1}+\Delta tA^{-1}H\left(\boldsymbol{\varphi}^{k}_{(s)}\right),\quad \boldsymbol{\varphi}^{k}_{(0)}=\boldsymbol{\varphi}^{k-1}.$$

Here, s is the iteration number and $\xi > 0$ is an iteration parameter.

Note that one step of this iterative procedure with $\xi=1$ is equivalent to computation based on an explicit scheme.

At every time step, the iterations are continued until $\left\|\mathbf{G}\left(\boldsymbol{\varphi}^{n}_{(s)}\right)\right\|/\left\|\mathbf{G}\left(\boldsymbol{\varphi}^{n}_{(0)}\right)\right\|<\varepsilon,$, where ε is the prescribed accuracy of these inner iterations. It turns out that the iterative error introduced at the k-th time step is not accumulated at the subsequent steps but decays. Accordingly, in the case of nonstationary problems, we can use $\varepsilon \sim 0.1$, which ensures the stability of the computation. In the case of a linear operator H and $A=E$, necessary conditions for the convergence of iterative process were obtained in [7].

Consider the general case in more detail. Let the discrete approximation of the spatial derivatives be represented as

$$H(\boldsymbol{\varphi}^{k})=T(\boldsymbol{\varphi}^{k})+V(\boldsymbol{\varphi}^{k}).$$

Here, the first term approximates the first derivatives (convective terms of the equations) and the second term approximates the viscous terms. Both terms are nonlinear functions of their arguments. To analyze them, we consider their linear approximation of the form

$$H'\boldsymbol{\varphi}^k = T'\boldsymbol{\varphi}^k + V'\boldsymbol{\varphi}^k,$$

where the coefficients of the corresponding matrices T' and V' are calculated using the values of the unknowns from the preceding iteration step.

Let the operator A be given by

$$A = E + \frac{1}{d}\tilde{V}', \quad d = \|T'\|,$$

where $\|V'\| \le \|\tilde{V}'\|$ and the matrix A is easily invertible (see [13]). Then we have the estimate

$$\|A^{-1}H'\| \le \|A^{-1}\|\|H'\| = (1 + \frac{1}{d}\|\tilde{V}'\|)^{-1}\|H'\| \le \|T'\|.$$

This estimate is used to analyze the convergence of the iterative process mentioned. Indeed, the matrix T' approximates the first derivatives of the unknown quantities. It is determined by the approximation of only the convective terms. As a result, the norm of the operator $A^{-1}H'$ to be inverted can be substantially reduced in the case of fine grids (see also [10]).

4. Numerical Results

Several interesting problems were numerically solved by applying the algorithm described above and developed software package. Some of the results are presented below.

4.1 Gas Flows around Aircraft with Allowance for the Flow/Exhaust Jet Interaction

Below are the numerical results obtained for the complex gas flow around a prospective flight vehicle. An external view of this vehicle is shown in Fig. 1.

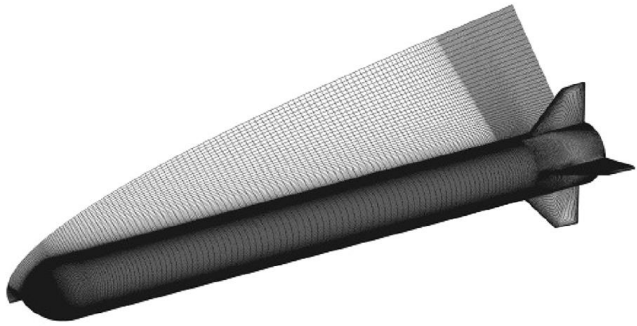


Fig. 1. External view of vehicle.

The vehicle is a blunt cylinder of radius R=0.3 m with a spherical cap of radius R/10 (the distance along the axis of symmetry from the cylinder to the outer boundary of the sphere is 1.9 R). The cylinder and the cap are adjusted by a spline surface of revolution around of the cylinder's axis of symmetry. The nozzle radius is 80% of the radius of the vehicle rear end. The tail of the vehicle is equipped with rudders.

It is assumed that fuel combustion products are exhausted from the nozzle of the vehicle. The exhaust jet has a high temperature. In the general case, the physical characteristics of the jet differ noticeably from those of the ambient air. The task is to determine the flow parameters near the vehicle and some of its integral characteristics.

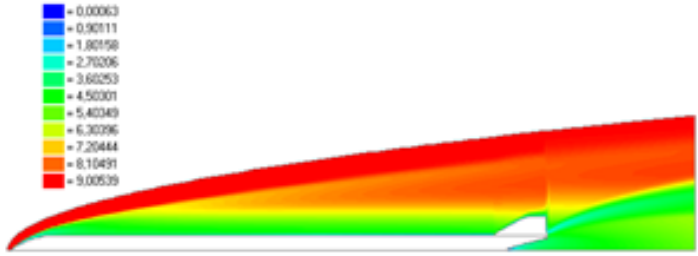


Fig. 2. Mach number distribution.

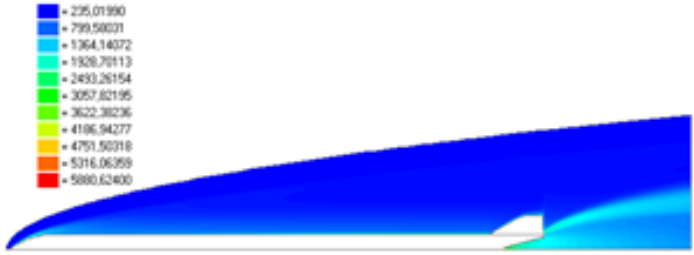


Fig. 3. Temperature distribution.

Fig. 4. Fraction distribution.

Fig. 2 shows the Mach number distributions in a longitudinal cross section of the computational domain containing a rudder. As expected, the Mach number decreases near the body surface, the rudders, and in the gas mixing zone. Inspection of the figure reveals an external shock wave determined by the overall vehicle shape, an internal shock wave arising due to the interaction of the outer airflow with the combustion products, and a rarefaction wave caused by the airflow past the convex part of the body. Figs. 3 and 4 present the temperature and fraction distributions near a rudder. The temperature near body surface, the rudders, and in the mixing zone increases and becomes close to that of the combustion products exhausted from the nozzle. The fraction distribution near a rudder shows that the combustion products nearly do not propagate upstream toward the rudders.

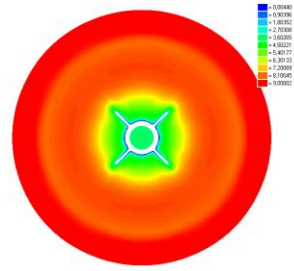


Fig. 5. Mach number distribution.

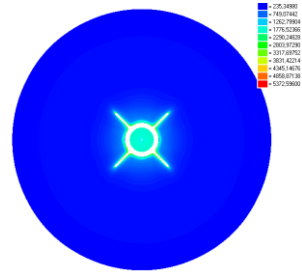


Fig. 6. Temperature distribution.

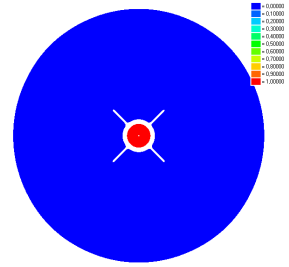


Fig. 7. Fraction distribution.

Figs. 5-7 illustrate the distributions in a transverse cross section of the computational domain near the rear end. As in the case of longitudinal cross sections, the Mach number near the body surface and the rudders is observed to decrease. The temperature near the body and the rudders increases and becomes close to that of the combustion products exhausted from the nozzle. The fraction distribution shows that the combustion products do not propagate upstream toward the rudders.

An analysis of the results suggests that complex flows of a mixture of widely different gases can be fairly well described by applying the algorithm developed. This algorithm well reproduces external and internal shock waves, rarefaction waves, and gas mixing zones. A priori information on the features of a flow (obtained, for example, via computations on a coarse grid) can be used to improve the quality of the computed flow by the refining mesh in areas of strong variations in the flow parameters.

4.2 Three-Dimensional Turbulent Gas Flows in Complex Nozzle Systems

The computation of gas flow parameters in nozzles is a classical numerical problem in internal aerodynamics. Another classical problem is that of external aerodynamics consisting in the computation of gas flows past various bodies. In some cases, problems combining the features of flows of both types have to be considered in practice.

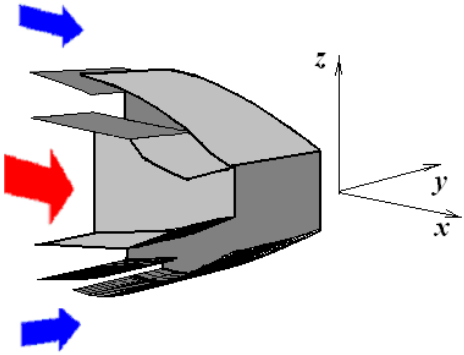


Fig. 8. Ejector nozzle.

An example of such problems combining internal and external aerodynamics is the computation of the steady gas flow in a no axisymmetric supersonic ejector nozzle system. Fig. 8 displays an example of a nozzle system of this kind. The system consists of an internal gasdynamic (primary) nozzle and a surrounding ejector contour. Such a primary nozzle is placed, for example, at the exit of a jet combustor with the combustion gases being exhausted through it. The ejector contour is a

system of air inlets designed so that the air stream from the surrounding space mixes with the hot combustion gas flow at the exit from the primary nozzle.

The gas flows in the different regions of the ejector nozzle strongly interact with each other. It should be mentioned that the flow characteristics in the rather long external-flow region is not known in advance.

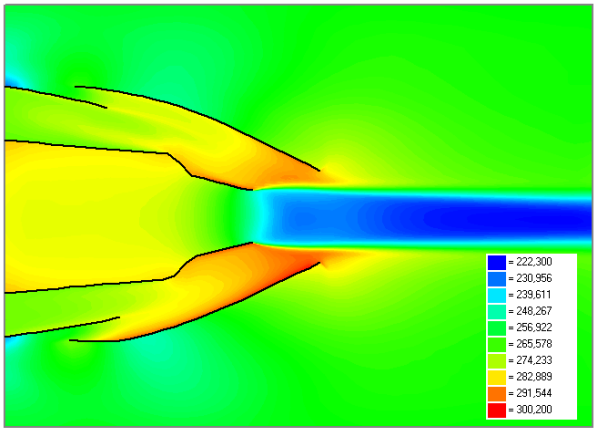


Fig. 9. Temperature distribution in the middle section.

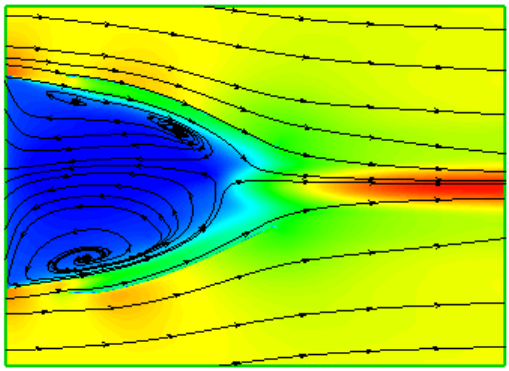


Fig. 10. Streamlines in a vertical plane through the interwall spacing.

Some results of test computations imitating a cold flow are presented. In this case, air is drawn in the primary nozzle and air flows past the system. Figs. 9-11 display some of the computed gasdynamic fields. Fig. 9 shows the distribution of the static temperature (^{0}K) in the middle section *x0y* of the nozzle. The behavior of some streamlines for gas flows is shown in Figs. 10, 11.

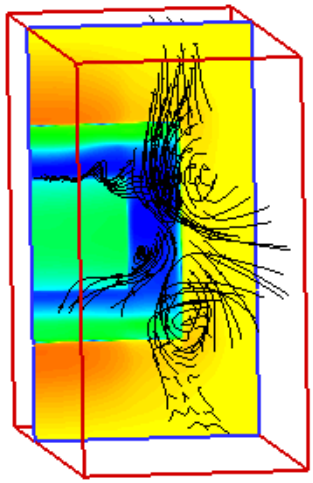


Fig. 11. Streamlines for the flow behind the nozzle exit..

The figures demonstrate a complex flow pattern, various vortex regions inside the nozzle (Fig. 10), and a vortex sheet forming behind the nozzle (Fig. 11). For more details of this work and comparison with results of other authors see [11].

4.3 Simulation of gas flow in cooled axial turbines

Numerical simulations of the working flow in gas turbines are the complex problem of internal aerodynamics. The flow under consideration occurs in the regions of complex shapes, representing the channels between rotor and stator blades moving relative to each other. In addition, from surfaces of the turbine's blades can usually be blowing the cooling air.

Software package based on algorithm described above are considered all flows within all channels between the blades of each blade row (stators and rotors) as the same, that the process of generation of long wave axial disturbances is ignored. The governing equations of gas flow within the interblade channels of stators are written with the use of a fixed system of coordinates and within the interblade channels of rotors, these equations are written with the use of a rotating system of coordinates. The numerical solution for the fixed and rotating regions are joint at the expense of requirements based on the continuity of the mass, momentum end energy fluxes. The calculation of these fluxes are carried out with the use of axial average values. This axial average allows to avoid the high frequency pulsation of the flow and to reach its average stationary state with the use of a numerical iterative procedure.

Some results of computations are shown on Fis. 12-14.

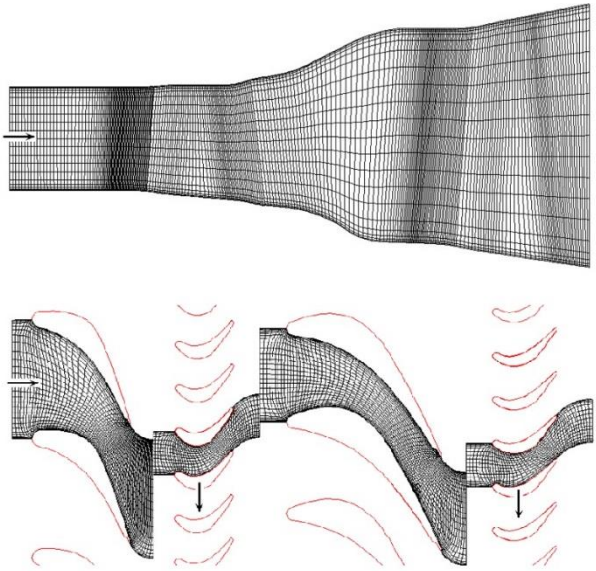


Fig. 12. Numerical grid in middle radial and axial plans.

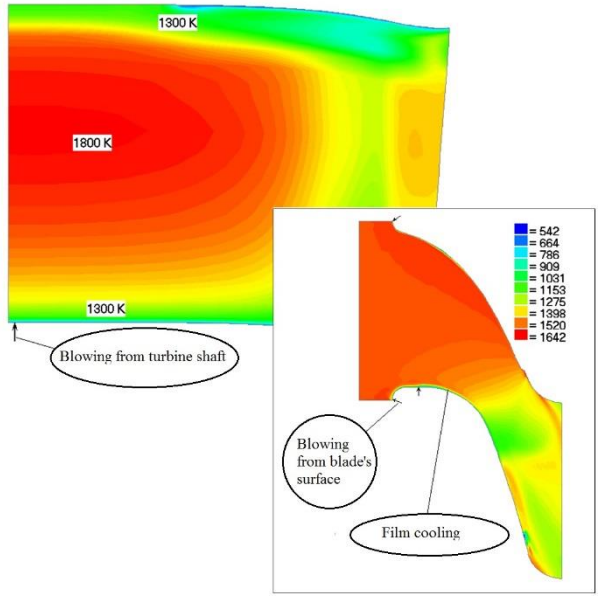


Fig. 13. Calculated temperature in first stage stator of turbine in middle radial and axial plans.

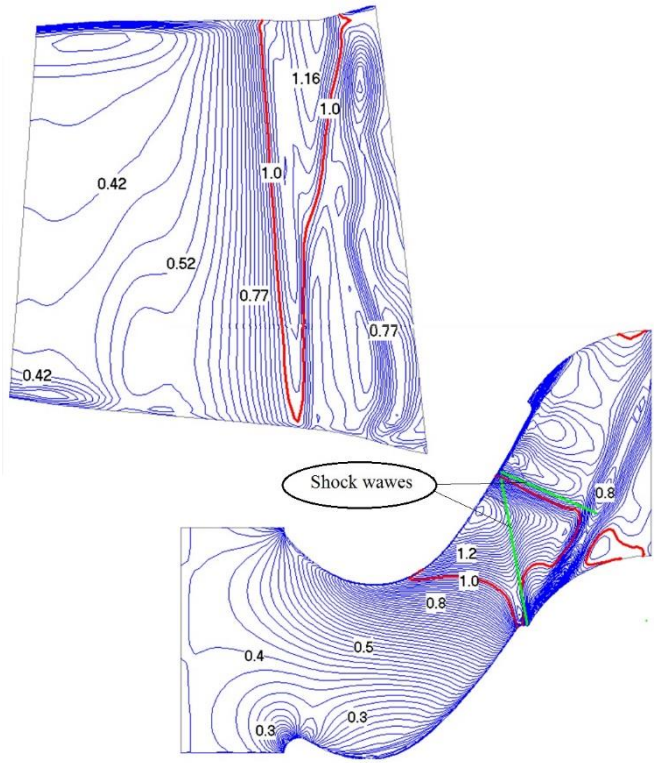


Fig. 14. Calculated Mach number in first stage rotor of turbine in middle radial and axial plans.

5. Conclusion

The submitted results show the ability to use the software package for solving wide range of stationary and non-stationary aerodynamic problems

References

[1]. Nigmatulin R.I. Dynamics of Multiphase Media. Part 1. *Nauka, Moscow*, 1987 (in Russian)

[2]. Lapin Yu.V. and Strelets M.Kh. Internal Flows of Gas Mixtures. *Nauka, Moscow*, 1989 (in Russian).

[3]. Coacley T.J., Turbulence Modeling Methods for the Compressible Navier-Stokes Equations. *AIAA Paper*, no. 83-1693, 1983.

[4]. Koterov V.N. Three-dimensional grid generation in multistage axial turbines based on the variational barrier method. *Comput. Math. Math. Phys*., vol. 45, no. 8, pp. 1325-1333, 2005.
[5]. Official home of The Open Source Computational Fluid Dynamics (CFD) Toolbox. Available at: http://www.openfoam.com/
[6]. Garanzha V.A., Kudryavtseva L.N., and Utyzhnikov S.V. Untangling and optimization of spatial meshes. *J. Comput. Appl. Math*., no. 269, pp. 24-41, 2014.
[7]. Zubov V. I., Inyakin V. A., Koterov V. N., and Krivtsov V. M. Numerical simulation of three-dimensional turbulent gas flows in complex nozzle systems. *Comput. Math. Math. Phys*. vol. 45, no. 10, pp. 1802-1814, 2005.
[8]. Rodionov A.V. Monotonic scheme of the second order of approximation for the continuous calculation of nonequilibrium flows. *USSR Comput. Math. Math. Phys*. vol. 27, no. 2, pp. 175-180, 1987.
[9]. Krivtsov V.M. On a numerical scheme for solving the Navier—Stokes equations. *USSR Comput. Math. Math. Phys*., vol. 26, no. 3, pp. 172-178, 1986.
[10]. Saad Y. Itertive methods for sparse linear systems. *2nd edition with corrections. — SIAM, 2003.*
[11]. Zabarko D.A., Zubov V.I., Kotenev V.P., Krivtsov V.M., Polezhaev Yu.A. Numerical Simulation of Gas Flows around Aircraft with Allowance for the Flow/Exhaust Jet Interaction. *Comput. Math. Math. Phys*., vol. 55, no. 4, pp. 677-689, 2015.

Программный пакет для расчета аэродинамических характеристик летательных аппаратов

В.Н. Котеров <vkoterov@yandex.ru>
В.М. Кривцов <vlkrivtsov@yandex.ru>
В.И. Зубов <vladimir.zubov@mail.ru>
Вычислительный центр им. А.А. Дородницына РАН
Федерального исследовательского центра «Информатика и управление» РАН,
119333, Россия, Москва, ул. Вавилова, 40
Московский физико-технический институт,
141700, Россия, Московская область, г. Долгопрудный, Институтский пер., 9

Аннотация. Представлен программный пакет для расчета параметров трехмерных стационарных и нестационарных потоков газа в сложных устройствах. Модель математического потока, используемая в пакете, основана на усредненных по Рейнольдсу уравнениях Навье-Стокса для двухкомпонентной равновесной турбулентной среды и двухпараметрической полуэмпирической модели турбулентности. Описана численная реализация программного пакета для моделирования трех сложных потоков в современных устройствах.

Ключевые слова: программный пакет; математическое моделирование; потоки газа; турбулентность; реактивная струя; уравнения Навье-Стокса; летательный аппарат

DOI: 10.15514/ISPRAS-2017-29(6)-17

Для цитирования: Котеров В.Н., Кривцов В.М., Зубов В.И. Программный пакет для расчета аэродинамических характеристик летательных аппаратов. Труды ИСП РАН, том 29, вып. 6, 2017 г., стр. 271-288 (на английском языке). DOI: 10.15514/ISPRAS-2017-29(6)-17

Список литературы

[1]. Нигматулин *Р. И.* Динамика многофазных сред. Часть I. М.: Наука, 1987, 464 стр.
[2]. Лапин Ю.В. Стрелец М.Х. Внутренние течения газовых смесей. М.: Наука, 1989, 368 стр.
[3]. Coacley T.J., Turbulence Modeling Methods for the Compressible Navier-Stokes Equations. *AIAA Paper*, no. 83-1693, 1983.
[4]. Koterov V.N. Three-dimensional grid generation in multistage axial turbines based on the variational barrier method. *Comput. Math. Math. Phys*., vol. 45, no. 8, pp. 1325-1333, 2005.
[5]. Official home of The Open Source Computational Fluid Dynamics (CFD) Toolbox. http://www.openfoam.com/
[6]. Garanzha V.A., Kudryavtseva L.N., and Utyzhnikov S.V. Untangling and optimization of spatial meshes. *J. Comput. Appl. Math*., no. 269, pp. 24-41, 2014.
[7]. Zubov V. I., Inyakin V. A., Koterov V. N., and Krivtsov V. M. Numerical simulation of three-dimensional turbulent gas flows in complex nozzle systems. *Comput. Math. Math. Phys*. vol. 45, no. 10, pp. 1802-1814, 2005.
[8]. Rodionov A.V. Monotonic scheme of the second order of approximation for the continuous calculation of nonequilibrium flows. *USSR Comput. Math. Math. Phys*. vol. 27, no. 2, pp. 175-180, 1987.
[9]. Krivtsov V.M. On a numerical scheme for solving the Navier—Stokes equations. *USSR Comput. Math. Math. Phys*., vol. 26, no. 3, pp. 172-178, 1986.
[10]. Saad Y. Itertive methods for sparse linear systems. *2nd edition with corrections. — SIAM, 2003.*
[11]. Zabarko D.A., Zubov V.I., Kotenev V.P., Krivtsov V.M., Polezhaev Yu.A. Numerical Simulation of Gas Flows around Aircraft with Allowance for the Flow/Exhaust Jet Interaction. *Comput. Math. Math. Phys*., vol. 55, no. 4, pp. 677-689, 2015.

Image processing technique for hydraulic application

[1] *J. Tellez-Alvarez <Jackson.david.tellez@upc.edu>*
[1] *M. Gomez <manuel.gomez@upc.edu>*
[2] *B. Russo <brusso@unizar.es>*
[1] *Institute FLUMEN, Department of Civil and Environmental Engineering*
Technical University of Catalonia BarcelonaTech, Barcelona, Spain.
[2] *Group of Hydraulics and Environmental Engineering*
Technical College of La Almunia (University of Zaragoza)
Zaragoza, Spain.

Abstract. In this paper, we present techniques Surface Flow Image Velocimetry (SFIV) that is a practical extension of Particle Image Velocimetry method (PIV). In particular, this technique is to evaluate the behavior on the surface flow for complex flow. SFIV allows to measure complex surface velocity fields for engineering applications. The objective of this work was the application of the SFIV technique to determine the velocity fields and their hydraulic efficiency of grated inlet, making a comparison between the programs able to correlate images like Digiflow or PIVlab program.

Keywords: Digiflow; PIVlab; image processing; grated inlet; MATLAB; Particle Image Processing (PIV); Surface Flow Image Velocimetry (SFIV).

DOI: 10.15514/ISPRAS-2017-29(6)-18

For citation: Tellez-Alvarez J., Gomez M., Russo B. Image processing technique for hydraulic application. ISP RAN/Proc. ISP RAS, vol. 29, issue 6, 2017. pp. 289-298 (in English). DOI: 10.15514/ISPRAS-2017-29(6)-18

1. Introduction

In general, the particle image velocimetry (PIV) method is a classic velocity meter through video cameras, where velocity fields can be obtained by following the definition of velocity u:

$$u(x,t) = \frac{\Delta x(x,t)}{\Delta t} \quad (1)$$

where Δx is the displacement of a marker, located at x at time t, over a short time interval Δt separating observations of the marker images [1].

Surface Flow Image Velocimetry (SFIV) similar to the Particle Image Velocimetry (PIV) or laser anemometry, that may be used without the need of adding tracer particles to the flow, but following the perturbations of the surface flow as shapes of the vortices or wave flows as defined in [5], for these applications in surface flows local capillary waves, vortices or wakes as in the case of convective or shock induced flows, the vortex cores or the vortex filaments are needed to be able to follow in detail the lagrangian aspects of the flow.

Very high-resolution cameras with high frame speeds are necessary, together with large computer capacity and the implementation of programs coupled to advanced image processing for fluid mechanics, such as Imacal, DigImage and DigiFlow [4, 8].

Technique called Surface Flow Image Velocimetry (SFIV) was developed to obtain the velocity field from the image processing of the surface flow in the vicinity of the grated inlet [6, 7].

The equipment for obtaining these images consists of a camera of high resolution and speed accompanied to a large computer where the image analysis was carried out through the application of the advanced image-processing program for fluid mechanics called Digiflow, developed by the Department of Applied Mathematics and Physical Theory, University of Cambridge and Dalziel Research Partners [2].

In addition, Digiflow is an open software developed for the community of research focus in the area of fluids mechanics and turbulence applications.

In order to follow the same line of research and compare the results with Digiflow, we also used a free code for image processing called PIVlab developed by the University of Groningen in the Netherlands. [10, 11], written in the mathematical language MATLAB. Considering that MATLAB language is a private code, but the company Matworks provide a special permission as open software for research community and student, for this reason, now a day is the more common program used in the universities for mathematics application and programming.

2. Experimental Set-Up

The experiment consists of fixed laboratory facilities such as grated inlet platform, 2 reflectors and a grate of study. Mobile installations as the high-resolution camera and speed connected to a large computer.

The platform has dimensions of 5.5 m long and 4 m wide, with a working area of 1.5 m long and 3 meters wide in the vicinity of the grated inlet. The platform is able to modify its longitudinal slope between 0% to 10% and the transverse slope of 0% to 4% [5]. In addition, the maximum approach flow for the experiments is 200 l/s, as shown in Figure1.

In order to obtain high-resolution images of the order of 1280 x 1024 pixels with a capture rate of 150 frames per second, it was decided to place the artificial lighting with two reflectors of power of 500 and 1000 Watt located strategically to provide the best possible illumination in the area of study (Figure 1). In addition, we painted

the gray color the platform, which allowed us to see better the surface flow and avoid reflections. Finally, marks were made around the grate used as reference in the image.

Fig. 1. Area of study

The platform has dimensions of 5.5 m long and 4 m wide, with a working area of 1.5 m long and 3 meters wide in the vicinity of the grated inlet. The platform is able to modify its longitudinal slope between 0% to 10% and the transverse slope of 0% to 4% [5]. In addition, the maximum approach flow for the experiments is 200 l/s, as shown in fig. 2.

In order to obtain high-resolution images of the order of 1280 x 1024 pixels with a capture rate of 150 frames per second, it was decided to place the artificial lighting with two reflectors of power of 500 and 1000 Watt located strategically to provide the best possible illumination in the area of study (fig. 1). In addition, we painted the gray color the platform, which allowed us to see better the surface flow and

avoid reflections. Finally, marks were made around the grate used as reference in the image.

3. Surface Flow Image Velocimetry

The Surface Flow Image Velocimetry (SFIV) technique allows the measurement of complex surface flows to obtain the field of velocities in engineering. This methodology has been developed using the physical model of grated inlet located in the Hydraulic Laboratory of the Unicersitat Politècnica of Catalonia BarcelonaTech.

The Digiflow algorithm is a method of the synthetic schlieren type, which is a novel technique that gives us a qualitative visualization of the fluctuations of the densities of the frames, and has origin in the classical schlieren method and the moire fringe technique. It also allows us to get good visualization in large study domains [4].

The synthetic schlieren method has different approaches to measure flows within the Digiflow program, such as line refractometry, dot-tracking refractometry, and pattern matching refractometry [3]. The final method is based on PIV techniques and was the method of correlation and image analysis proposed in the SFIV technique with Digiflow.

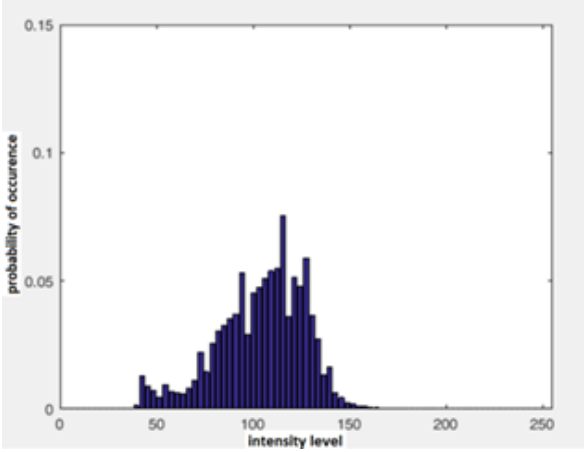


Fig. 2. Visualization of histogram. Longitudinal slope 0%. Transversal slope 0%. Flowrate 200 l/s

PIVlab is a free access PIV program developed in the MATLAB calculation platform, which allows us to have easier access to this tool. We must take into account that for the image processing using PIVlab, we used the same videos that were used for the analysis with the program Digiflow, only that, in this case, we must take into account that PIVlab does not accept video formats only sequence of fixed images, for this reason, we developed an additional code in MATLAB capable

of decomposing videos into frames, which allowed us to introduce the sequence of images in the PIVlab.

In MATLAB it is possible to generate the histograms of the digital videos, allowing to represent the frequency of intensity of each pixel and the form of the histogram. This is possible through a function within the tools developed in MATLAB, in this case the utility is "imhist".

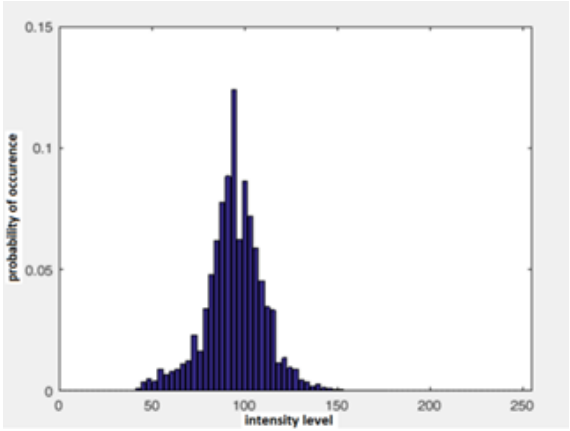


Fig. 3. Visualization of histogram. Longitudinal slope 10%. Transversal slope 4%. Flowrate 200 l/s.

The intensity scale is in the range of 0 to 256 grayscale. The objective of the visualization of the histograms was to see the difference in form and values for different geometric combinations and a same flow of approximation.

In fig. 2 and fig. 3, a comparison of the histograms was made when the platform is completely flat and with a longitudinal slope of 10% and transverse slope of 4%, and these graphs show the significant difference between the intensity values for different videos, which indicates that when we do not have slopes the intensity histogram is more widespread, while in extreme conditions of slope the intensities are accumulating around the mean value of 100. It is evident that this affects the recognition of flow patterns, being one of the reasons why the calculation algorithm must be powerful enough to be able to reproduce the velocity fields for different geometric combinations and approximation flowrates on the platform.

4. Results

After performing several tests with different PIV algorithms, it was finally possible to obtain the velocity fields for 24 different geometries and 5 approximation flows, in order to compare with the Digiflow code. The visualization of the video captured, and the flow approach around the grated inlet is how in the fig. 4. The velocity

fields in the vicinity of the grated inlet are shown in fig. 5 with PIVlab code and fig. 6 using Digiflow program.

Moreover, both algorithm show a good approximation of field velocities around the grated inlet, but the different is on the grated inlet, because Digiflow present better approximation in this area, but also it is not enough, because the holes of the grate represent a behavior more tri-dimensional.

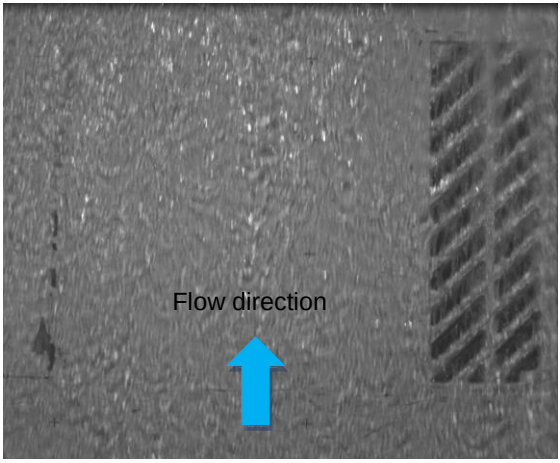


Fig. 4. Visualization of flow approach. Flowrate 200 l/s. Longitudinal slope 10%. Transversal slope 4%.

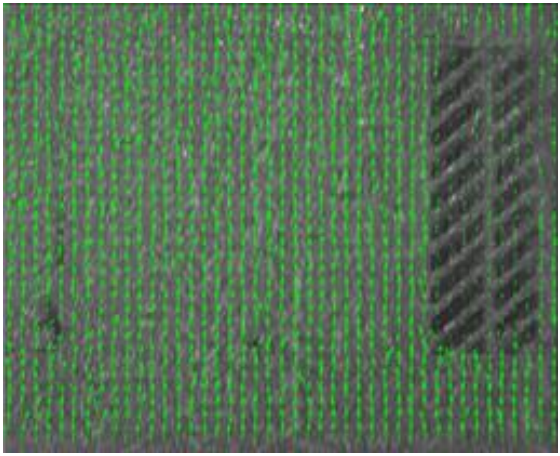


Fig. 5. Field velocity. Flowrate 200 l/s. Longitudinal slope 10%. Transversal slope 4%. PIVlab code

4. Discussion

Considering that both analysis with different programs have different algorithms, but follow a procedure similar to particle velocimetry (PIV), in both cases the SFIV technique was applied for surface flow, without adding particles as tracers and validating the results with experimental data for the measurement of flowrate collected by the grated inlet. The proposed methodology can become a useful tool to understand the hydraulic behavior of the flow in the surroundings of a catchment of grated inlet where traditional measurement equipment has serious problems and limitations.

Fig. 6. Field velocity. Longitudinal slope 10%.
Trasnversal slope 4%. Digiflow program.

As for the difference of the calculation programs, we have noticed that the Digiflow presents us with more advanced options for the analysis and processing of images, since it is a specialized program for visualization of complex flows in fluids mechanics, and it allows us complete stream of videos. In contrast, PIVlab, it has the disadvantage that it is only possible to work with sequence of frames so we need to create a tool capable of decomposing the videos into frames, but at the same time, it is a public access tool written in a common calculation language like is the MATLAB.

4. Conclusion

The Surface Flow Image Velocimetry (SFIV) technique is very useful for determining the fields of surface velocities and the structure of the flow through perturbations of the fluids in surfaces like forms or waves; we must say that works better in a supercritical regime. The proposed methodology of the SFIV technique is

able to obtain velocity fields in the vicinity of the grated inlet from high resolution images.

The results show that the velocity range is between 0.5 m / s and 2.45 m / s, for the 24 geometric combinations studied, with an approximate flow rate of 100 l/s, so we can say that the velocity values calculated through of the Manning and Izzard equation compared to the surface velocities obtained through Digiflow and PIVlab are quite similar.

The SFIV technique, through high resolution video capture and speed, presents certain advantages against traditional instruments of mechanical measures such as Acoustic Doppler Velocimetry (ADV) or EMG Electromagnetic that are able to capture the speed in a single point, while through images it is possible to have a field velocity in the area of focus of the camera, which allows us to better study the behavior of the fluid and thus obtain larger measurement points.

References

[1]. Adrian R.J. (1991) Particle-Imaging Techniques for Experimental Fluid Mechanics. Annual Review of Fluid Mechanics, 23(1), pp.261–304.

[2]. Dalziel R. P. (2012). DigiFlow User Guide. Cambridge, England: Department of Applied Mathematics and Theoretical Physics (DAMTP) - University of Cambridge.

[3]. Dalziel S.B., Hughes G.O. & Sutherland, B.R. (1998) Synthetic Schlieren. Paper 062. Proceedings of the 8th international Symposium on Flow Visualization, In: Carlomagno; Grant (ed), Tennessee, United States.

[4]. Dalziel, S.D. and Redondo J.M. (2007) New visualization and self-similar analysis in experimental turbulence studies. Models, Experiments and Computation in Turbulence. CIMNE, Barcelona, Spain.

[5]. Gómez M., 2008. Curso de Hidrología Urbana. Segunda Ed. Universitat Politècnica de Cataluña. Barcelona, España.

[6]. Tellez J., Gómez M., Russo M., Redondo J.M. 2016a. “Metodología para la obtención del campo de velocidad en la proximidad de las rejas de alcantarillado”. XXVII Congreso latinoamericano de hidráulica. 28 al 30 de septiembre, 2016. Lima, Perú.

[7]. Tellez J., Gómez M., Russo M., Redondo J.M. 2016b. “Surface Flow Image Velocimetry (SFIV) for Hydraulics Applications” In 18th International Symposium on the Application of Laser and Imaging Techniques to Fluid Mechanics. Julio 4-7. Lisboa, Portugal.

[8]. Tellez J., Russo B., Gómez M. 2015. “Técnica para la obtención del campo de velocidad del flujo superficial en proximidad de rejas de alcantarillado”. IV Jornadas de Ingeniería del Agua. Octubre 21-22. Córdoba, España.

[9]. Thielicke W., Stamhuis E. J. (2014). PIVlab - Time-Resolved Digital Particle Image Velocimetry Tool for MATLAB (version: 1.41).

[10]. Thielicke W. & Stamhuis E.J. (2014b). PIVlab – Towards User-friendly, Affordable and Accurate Digital Particle Image Velocimetry in MATLAB. Journal of Open Research Software. 2(1), p.e30. DOI: http://doi.org/10.5334/jors.bl

Технология обработки изображений для гидравлических приложений

[1] *Дж. Теллез-Альварез <Jackson.david.tellez@upc.edu>*
[1] *М. Гомез <manuel.gomez@upc.edu>*
[2] *Б. Руссо <brusso@unizar.es>*
[1] *Испания, Барселона, Политехнический университет Каталонии,*
Научно-исследовательский институт FLUMEN,
Департамент строительства и охраны окружающей среды
[2] *Испания, Сарагоса, Университет Сарагосы*
Технический колледж Ла-Альмунии
Группа гидравлики и охраны окружающей среды

Аннотация. В этой статье мы представляем методы Surface Flow Image Velocimetry (SFIV), являюшиеся практическим расширением метода Velocimetry Image Particle Image (PIV). В частности, этот метод позволяет оценивать поведение поверхностного потока для комплексного течения. SFIVпозволяет измерять сложные поля поверхностной скорости для инженерных приложений. Целью этой работы было применение метода SFIV для определения полей скоростей и гидравлической эффективности решетчатого водоприемника. Сравнивались результаты программ Digiflow и PIVlab, способных коррелировать изображения.

Ключевые слова: Digiflow; PIVlab; обработка изображений; MATLAB; Particle Image Processing (PIV); Surface Flow Image Velocimetry (SFIV)

DOI: 10.15514/ISPRAS-2017-29(6)-18

For citation: Теллез-Альварез Дж., Гомез М., Руссо Б. Технология обработки изображений для гидравлических приложений. Труды ИСП РАН, том 29, вып. 6, 2017 г., стр. 289-298 (на английском языке). DOI: 10.15514/ISPRAS-2017-29(6)-18

Список литературы

[1]. Adrian R.J. (1991) Particle-Imaging Techniques for Experimental Fluid Mechanics. Annual Review of Fluid Mechanics, 23(1), pp.261–304.
[2]. Dalziel R. P. (2012). DigiFlow User Guide. Cambridge, England: Department of Applied Mathematics and Theoretical Physics (DAMTP) - University of Cambridge.
[3]. Dalziel S.B., Hughes G.O. & Sutherland, B.R. (1998) Synthetic Schlieren. Paper 062. Proceedings of the 8th international Symposium on Flow Visualization, In: Carlomagno; Grant (ed), Tennessee, United States.
[4]. Dalziel, S.D. and Redondo J.M. (2007) New visualization and self-similar analysis in experimental turbulence studies. Models, Experiments and Computation in Turbulence. CIMNE, Barcelona, Spain.
[5]. Gómez M., 2008. Curso de Hidrología Urbana. Segunda Ed. Universitat Politècnica de Cataluña. Barcelona, España.

[6]. Tellez J., Gómez M., Russo M., Redondo J.M. 2016a. "Metodología para la obtención del campo de velocidad en la proximidad de las rejas de alcantarillado". XXVII Congreso latinoamericano de hidráulica. 28 al 30 de septiembre, 2016. Lima, Perú.

[7]. Tellez J., Gómez M., Russo M., Redondo J.M. 2016b. "Surface Flow Image Velocimetry (SFIV) for Hydraulics Applications" In 18th International Symposium on the Application of Laser and Imaging Techniques to Fluid Mechanics. Julio 4-7. Lisboa, Portugal.

[8]. Tellez J., Russo B., Gómez M. 2015. "Técnica para la obtención del campo de velocidad del flujo superficial en proximidad de rejas de alcantarillado". IV Jornadas de Ingeniería del Agua. Octubre 21-22. Córdoba, España.

[9]. Thielicke W., Stamhuis E. J. (2014). PIVlab - Time-Resolved Digital Particle Image Velocimetry Tool for MATLAB (version: 1.41).

[10]. Thielicke W. & Stamhuis E.J. (2014b). PIVlab – Towards User-friendly, Affordable and Accurate Digital Particle Image Velocimetry in MATLAB. Journal of Open Research Software. 2(1), p.e30. DOI: http://doi.org/10.5334/jors.bl

Numerical Simulation of High-Speed Non-equilibrium Flow with Applied Magnetic Field

[1, 2]*A.I. Ryakhovskiy <alexey.i.ryakhovskiy@mail.ioffe.ru>*
[1, 2]*A.A. Schmidt< alexander.schmidt@mail.ioffe.ru>*
[2]*V.I. Antonov< antonovvi@mail.ru >*
[1]*Ioffe Institute*
Polytechnicheskaya st., 26, Saint-Petersburg, 194021, Russian Federation
[2]*Peter the Great St.Petersburg Polytechnic University*
Polytechnicheskaya st., 29, Saint-Petersburg 195251, Russian Federation

Abstract. The paper describes the development and testing processes of a modification of an existing solver for hypersonic reacting flow within the OpenFOAM numerical simulation framework. The modification is suited to simulate the interaction between the flow and the constant applied magnetic field. The purpose of the development is to create a simulation tool for the study of the concept of magnetohydrodynamical flow control and its possible technological applications. Resulting application utilizes Navier-Stokes-Fourier system of equations supplemented with appropriate auxiliary models for the accurate assessment of process-specific additional terms. Solver testing has been carried out using the cases that highlight solvers capabilities to model MHD high-speed flow in different regimes.

Keywords: numerical simulation; fluid flow control; magnetohydrodynamics; aerodynamics; shock waves

DOI: 10.15514/ISPRAS-2017-29(6)-19

For citation: Ryakhovskiy A.I., Schmidt A.A., Antonov V.I. Numerical Simulation of High-Speed Non-equilibrium Flow with Applied Magnetic Field. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 6, 2017, pp. 299-310. DOI: 10.15514/ISPRAS-2017-29(6)-17

1. Introduction

In the recent years there has been a resurgence of interest towards plasma-assisted technology in high-speed flight. Prospective applications of this technology include aerobraking, communication blackout mitigation, aerodynamics heating reduction, wave drag and turbulence cancelation and scramjet engine intake control. The principal idea behind most of these concepts is magnetohydrodynamic (MHD)

control i.e. affecting the gas flow incoming towards an aircraft with an applied magnetic field. The potential of MHD flow control system has been studied and demonstrated on numerous occasions [1][2][3], however it's prospective effectiveness remains an open question. With the new challenges facing aerospace industry in the upcoming decade comes a need for more advanced numerical simulation tools.

Near space is the region of the atmosphere between controlled commercial airspace and the low earth orbit (LEO) [4]. This is the region where most of the aircraft with a potential to benefit from MHD flow control system operate. The difficulty of modeling the MHD interaction in hypersonic flow within the near space region is that the rarefaction of the gas can be high enough that the CFD model can become non-applicable in some cases, yet DSMC-type (Direct Simulation Monte Carlo) can be too computationally prohibitive. The issue has to be addressed in developing a comprehensive simulation tool for the high-speed magnetohydrodynamic flow.

There are several simulation packages for hypersonic reacting flow, such as DPLR, LAURA, LeMANS, US3D and others. All of them, however, can only be distributed to and used by United States citizens or US government contractors. The challenge posed by this restriction can be answered by developing an open-source code for the use in high-speed non-equilibrium flow research and multiphysics problems associated with it.

2. Development Basis

We are using OpenFOAM CFD toolbox as a framework for out development because of its' flexibility, robust class structure for numerical simulation and a broad community of contributions. In previous works there have been attempts to use standard toolbox solvers to simulate magnetic field effect on the supersonic flow around the object [5]. However, OpenFOAM's basic kit lacks a dedicated solver for high-speed reacting flow. Due to this face we are basing our work on a solver, developed in the University of Strathclyde, Scotland [6] called *hy2Foam*. It combines the standard kit solvers *rhoCentralFoam* (density-based compressible flow solver based on central-upwind schemes of Kurganov-Tadmor) and *reactingFoam* (a solver for combustion with chemical reactions).

3. Mathematical Model

hy2Foam employs a Navier-Stokes-Fourier system of equations for reacting flow. To include a magnetic field interaction into the model the terms corresponding to Lorentz force and Joule heating have been added to the right hand sides of momentum and energy equation respectively. An important assumption needs to be made to allow for a reduction of the number of equations necessary to calculate the MHD terms. Magnetic Reynolds number for the studied flow regimes is generally low, which allows us to disregard self-induced magnetic field [7]. This means that the only magnetic field present is the applied one, which eliminated the need to

solve the Maxwell system of equations. The flux-divergence form of the resulting system is presented below.

$$\frac{\partial \boldsymbol{\mathcal{U}}}{\partial t} + \frac{\partial (\boldsymbol{\mathcal{F}}_{inv} + \boldsymbol{\mathcal{F}}_{visc})}{\partial x_i} = \boldsymbol{\mathcal{W}};$$

$$\boldsymbol{\mathcal{U}} = \begin{pmatrix} \rho \\ \rho_s \\ \rho u \\ \rho v \\ \rho w \\ \varepsilon_{ve,m} \\ \varepsilon \end{pmatrix}, \boldsymbol{\mathcal{F}}_{inv} = \begin{pmatrix} \rho u_i \\ \rho_s u_i \\ \rho u_i u + \delta_{i1} p \\ \rho u_i v + \delta_{i2} p \\ \rho u_i w + \delta_{i2} p \\ \varepsilon_{ve,m} u_i \\ (\varepsilon + p) u_i \end{pmatrix}, \boldsymbol{\mathcal{F}}_{visc} = \begin{pmatrix} 0 \\ -\mathcal{J}_s \\ \tau_{i1} \\ \tau_{i2} \\ \tau_{i3} \\ \varphi_{v,visc} \\ \varphi_{visc} \end{pmatrix}, \boldsymbol{\mathcal{W}} = \begin{pmatrix} 0 \\ \dot{\omega}_s \\ f_{i1} \\ f_{i2} \\ f_{i3} \\ \dot{\omega}_{v,m} \\ Q_m \end{pmatrix}.$$

$\boldsymbol{\mathcal{F}}_{inv}$and $\boldsymbol{\mathcal{F}}_{visc}$ correspond to inviscid and viscous components of the flux. A more detailed explanation of each term can be found in the original paper.

The magnetic terms are calculated using generalized Ohm's law. We need several of additional models for different associated physical phenomena to complete model. Among those are Landau-Teller model for trans-vibrational relaxation and Millikan-White model for V-T relaxation times with Park's correction. Thermal diffusivities of different species are calculated using Eucken's formula. Mixture quantities are recovered using Gupta's mixing rule. Species' viscosity follows Blottner's formula. Diffusion and heat conduction are modeled by Fick's and Fourier's law respectively. Total pressure is recovered from partial pressures by Dalton's law [6]. Chemical reactions and their rated are taken from the paper by Park [7].

A. Conductivity models

Electrical conductivity of the flow plays a crucial role in estimating the intensity of the it's interaction with magnetic field. An electric conductivity model interface has been added to the *hy2Foam*, so that different conductivity model can be implemented and tested for use in high-speed flow problems. Five different electric conductivity models have so far been implemented alongside a dummy constant conductivity model. Two of them are semi-analytical Spitzer and Harm [8] and Chapman and Colwig [9] models. Another two models, developed originally by Bush [10] and Raizer [11] are semi-empirical. The fifth model was taken from the paper by Bityurin and Bocharov [1] and is based on Chapman-Enskog theory of multicomponent gas mixture. This last model is the most advanced and utilizes the gas mixture data that is available from solving Navier-Stokes-Fourier equations.

Table 1. Electric conductivity models

Boltzmann	**Spitzer-Harm**	**Chapmann-Cowling**	**Bush**	**Raizer**
$\frac{e^2 n_e \tau_e}{m_e}$	$\frac{c_{1,S} T^{\frac{3}{2}}}{ln\left(c_{2,S}\frac{T^{\frac{3}{2}}}{\sqrt{n_e}}\right)}$	$c\frac{\alpha}{Q\sqrt{T}}$	$\sigma_0\left(\frac{T}{T_0}\right)^n$	$c_{1,R}e^{-\frac{T_1}{T}}$

4. Test Results

The developed code has been tested on a case from the book by Surjikov[12]. The case presents a hypersonic flow of molecular nitrogen around a cylinder with a 0.015 meters radius. Freestream conditions are presented in the table below.

Table 2. Surjikov case freestream conditions

Quantity	**Value**	**Measurement Units**
Pressure	1300	Pa
Temperature	3030	K
Density	0.0014	kg/m^3
Velocity	10300	m/s

To test the capabilities of our MHD modification we have added a magnetic field to be generated by a coil located inside the cylinder. The distribution of magnetic flux intensity generated by it is shown in a Fig. 1.

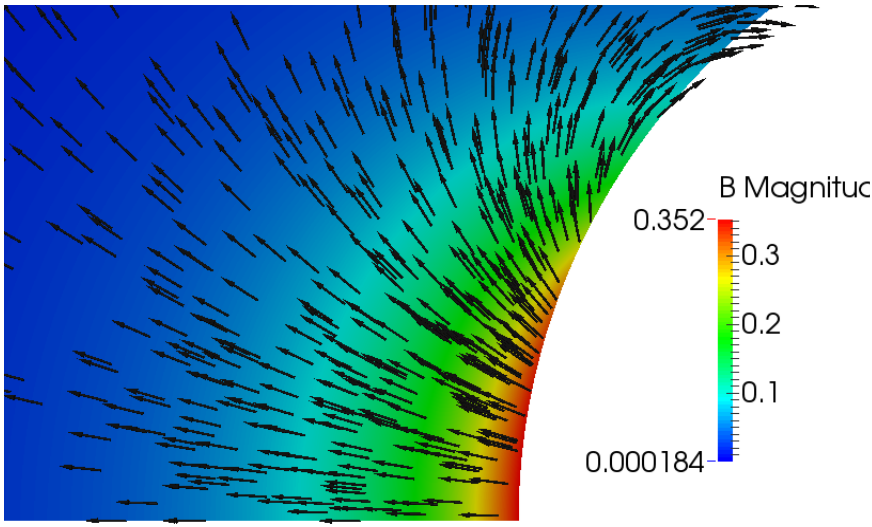


Fig. 1. Geometry of the case 1 and the distribution of magnetic flux density

Investigating this case with our MHD hypersonic solver serves 2 purposes. First, we can validate *hy2Foam* and its' auxiliary libraries on a well-known problem for which there are experimental data to use for comparison. Second, we can see how the modifications that we have added perform, when MHD interaction is introduced to the established case. Additionally, the numerical solution of the case with magnetic field interaction can give us insights on the potential of MHD control in the considered conditions.

The results for non-MHD case show good agreement with both experimental and numerical results from [10]. Apparent discrepancies can be attributed to the difference in choice of boundary conditions for temperature on the object (constant value in the source, zero gradient in our calculations) and using a different model for chemical reactions and their rates. The solution obtained for the case with magnetic interaction shows that without assisted ionization the effect of the applied magnetic field is negligible, albeit visible. Figures below show a clear increase of the shock standoff distance, the magnitude of which differs depending on model used to calculate electrical conductivity.

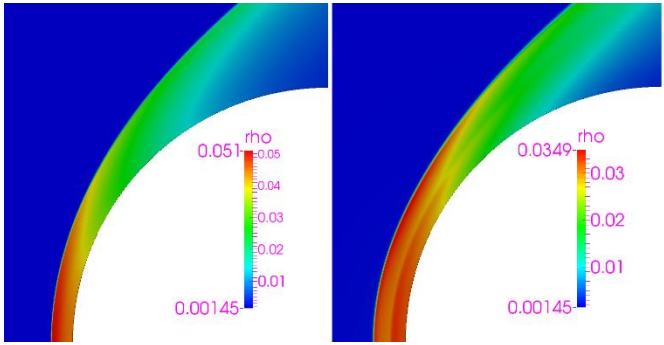


Fig. 2. Bow shocks compared for cases 1 without (left) and with (right) MHD interaction.

The difference in the effect for different conductivity models can be better illustrated using the case with freestream conditions characteristic for atmospheric entry.

Table 3. Re-entry case freestream conditions

Quantity	Value	Measurement Units
Pressure	33	Pa
Temperature	200	K
Density	0.001	kg/m^3
Velocity	5411	m/s

The simulation results indicate that for these conditions the size and the parameter distribution within stagnation layer will vary significantly depending on the used electric conductivity model. The shock standoff distance can differ by as much as 2.5 times.

Results show that using both Spitzer-Harm and Raizer model can greatly exaggerate the MHD interaction effect. Bush model, on the other hand, considerably underestimates it compared to the other models. The remaining models provide similar results and can both be used reliably for this type of conditions.

Our third case serves to illustrate one of the applications of the concept. It is a model of a scramjet with a design Mach number of 10. The freestream Mach number, however, is much lower, which results in a “shock spillage”. This phenomenon is known to adversely affect the flight characteristics of the aircraft. The proposed solution is to use the magnetic field to preserve the position of oblique shock during the changes of flight regime. Similarly to our first test case we need to artificially increase the flow’s electric conductivity to achieve the necessary

intensity of the effect. For this purpose, we have used constant conductivity model. The location of inductor that generates the field is shown in the bottom picture of Fig. 6.

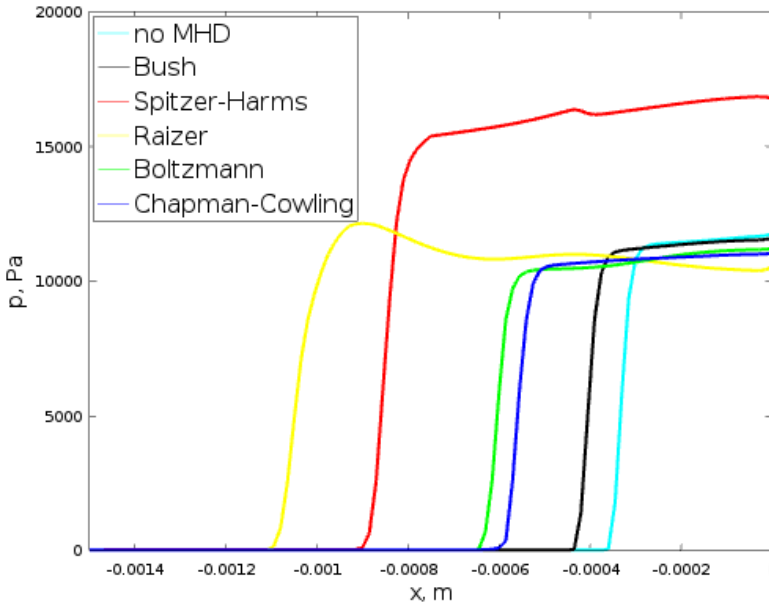


Fig. 3. Pressure distibution on the stagnation line, calculated using different conductivity models

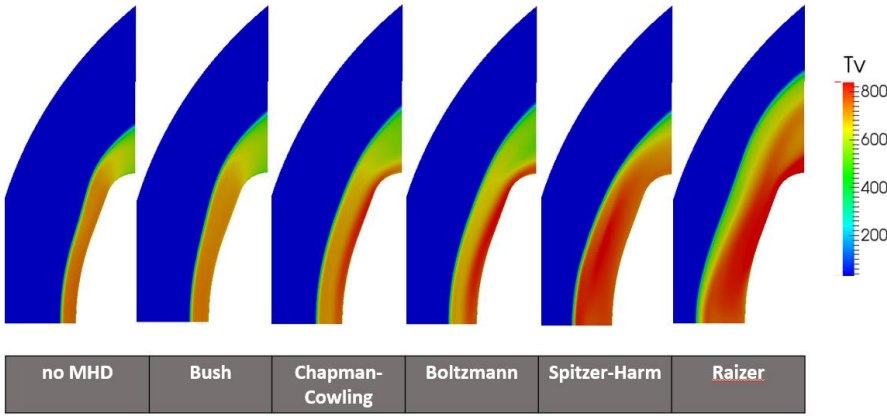


Fig. 4. Vibrational temperature distribution calculated using different electric conductivity models.

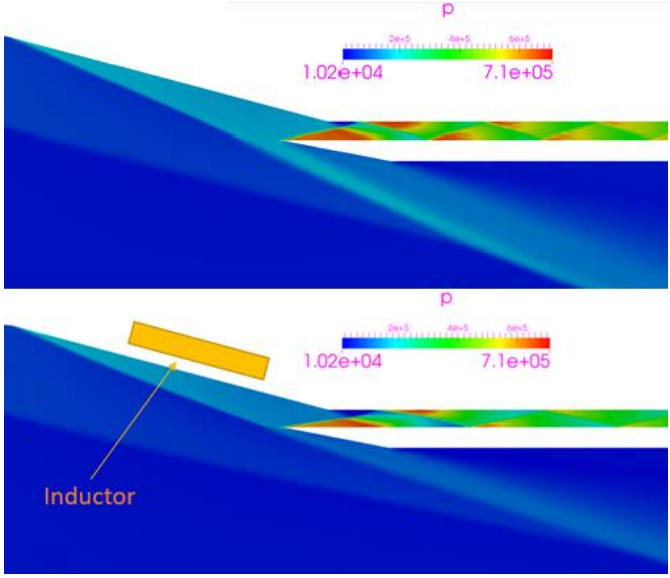


Fig. 5. Pressure distribution for scramjet case with and without MHD control effect

5. Conclusion

We have demonstrated that even at this stage of development the code can be used as a simulation tool by the researchers studying the interaction between the applied magnetic field and high-speed weakly ionized gas flow. Being developed within OpenFOAM framework the solver and it's auxiliary libraries can be easily modified and supplemented by anyone familiar with OpenFOAM structure, like it had been done by the authors of this paper with the solver's "parent" code hy2Foam.

Table 4. Scramjet case freestream conditions

Quantity	Value	Measurement Units
Pressure	26436.3	Pa
Temperature	223.150	K
Density	0.412707	kg/m^3
Velocity	1800	m/s

The results of simulations provide an example of how the electric conductivity models can be tested for use in modeling of MHD hypersonic flow control with the developed software. They also showcase the potential effect of MHD flow control system on the shockwave configuration and the applications this effect might have in hypersonic aircraft design.

Further development of the code is going to include additional models for the electric conductivity flow and the processes related to the assisted ionization of the flow. It may also be necessary to develop a hybrid Direct Simulation Monte-Carlo – CFD code to adequately simulate MHD control of the flow around atmospheric entry vehicles, which is another potential application of this technology.

References

[1]. V. Bityurin, A. Bocharov, and J. Lineberry. MHD flow control in hypersonic flight. AIAA Paper, 3225, 2005.

[2]. T. Fujino, T. Yoshino, and M. Ishikawa. Numerical Analysis of Reentry Trajectory Coupled with Magnetohydrodynamics Flow Control. Journal of Spacecraft and Rockets, 45(5), 911, 2008.

[3]. N. Bisek, I. Boyd, and J. Poggie. Numerical study of plasma-assisted aerodynamic control for hypersonic vehicles. Journal of Spacecraft and Rockets, 46(3), 568, 2009.

[4]. E. Allen. The Case for Near Space. Aerospace America, February 2006,

[5]. A. Ryakhovskiy, and A. Schmidt. MHD supersonic flow control: OpenFOAM simulation. Trudy ISP RAN / Proc. ISP RAS, vol, 28, issue 1, 2016, pp. 197-206. DOI: 10.15514/ISPRAS-2016-28(1)-11

[6]. V. Casseau, R. Palharini, T. Scanlon, and R. Brown. A two-temperature open-source CFD model for hypersonic reacting flows, part one: zero-dimensional analysis. Aerospace, vol. 3(4), 2016.

[7]. I. Sarris, G. Zikos, A. Grecos, A. P., and N. Vlachos. On the limits of validity of the low magnetic Reynolds number approximation in MHD natural-convection heat transfer. Numerical Heat Transfer. Part B: Fundamentals, 50(2), pp. 157-180, 2006.

[8]. L. Spitzer and R. Härm. Transport phenomena in a completely ionized gas. Physical Review, 89(5), 977, 1953.

[9]. S. Chapman. and T. Cowling. The mathematical theory of non-uniform gases: an account of the kinetic theory of viscosity, thermal conduction and diffusion in gases. Cambridge university press, 1970.

[10]. Bush, W. B. Magnetohydrodynamic - Hypersonic Flow Past a Blunt Body. Journal of Aero/Space Sciences, Vol. 25, No. 11, November 1958, pp. 685–690.

[11]. Raizer, Y. P., Gas Discharge Physics, Springer-Verlag, 1991

[12]. S. Surzhikov. A computational study of aerothermodynamics of hypersonic flow around dull bodies using the example of analysis of experimental data. M.: IPMech, 2011, 74 p. (in Russian)

Численное исследование высокоскоростного неравновесного течения с приложенным магнитным полем

[1, 2]А.И. Ряховский <alexey.i.ryakhovskiy@mail.ioffe.ru>
[1, 2]А.А. Шмидт <alexander.schmidt@mail.ioffe.ru>
[2]В.И. Антонов<antonovvi@mail.ru>
[1]ФТИ им. А.Ф. Иоффе
Политехническая ул., 26, Санкт-Петербург, 194021, Российская Федерация
[2]Санкт-Петербургский Политехнический Университет Петра Великого
Политехническая ул., 29, Санкт-Петербург, 195251, Российская Федерация

Аннотация. Статья описывает разработку и тестирование модификации решателя для гиперзвукового реагирующего течения в среде численного моделирования OpenFOAM. Модификация создается для моделирования взаимодействия между течением и приложенным постоянным магнитным полем. Цель разработки – создать численный инструментарий для исследования концепции магнитогидродинамического управления потоком и его возможных применений. Создаваемое приложение использует математическую модель на основе уравнений Навье-Стокса, дополненных необходимыми вспомогательными моделями для описания сопряженных процессов. Тестирования решателя проводилось на задачах, демонстрирующих основные возможности созданного приложения в моделировании высокоскоростных МГД-течений различных режимов. Тестовые примеры представляют собой задачи обтекания двумерных плоско и цилиндрически симметричных тел, имеющих форму, характерную для аппаратов, для которых известны потенциальные способы применения МГД управления. Исследовалось влияние выбора модели электропроводимости на результаты численного моделирования. Сравнение результатов показало зависимость важности выбора модели электропроводимости от разреженности рассматриваемого газового потока.

Ключевые слова: численное моделирование; управление потоком; магнитогазодинамика; неравновесные течения; ударные волны

DOI: 10.15514/ISPRAS-2017-29(6)-19

Для цитирования: Ряховский А.И., Шмидт А.А., Антонов В.И. Численное исследование высокоскоростного неравновесного течения с приложенным магнитным полем. Труды ИСП РАН, том 29, вып. 6, 2017 г., стр. 299-310 (на английском языке). DOI: 10.15514/ISPRAS-2017-29(6)-19

Список литературы

[1]. V. Bityurin, A. Bocharov, and J. Lineberry. MHD flow control in hypersonic flight. AIAA Paper, 3225, 2005.

[2]. T. Fujino, T. Yoshino, and M. Ishikawa. Numerical Analysis of Reentry Trajectory Coupled with Magnetohydrodynamics Flow Control. Journal of Spacecraft and Rockets, 45(5), 911, 2008.
[3]. N. Bisek, I. Boyd, and J. Poggie. Numerical study of plasma-assisted aerodynamic control for hypersonic vehicles. Journal of Spacecraft and Rockets, 46(3), 568, 2009.
[4]. E. Allen. The Case for Near Space. Aerospace America, February 2006,
[5]. A. Ryakhovskiy, and A. Schmidt. MHD supersonic flow control: OpenFOAM simulation. Trudy ISP RAN / Proc. ISP RAS, vol, 28, issue 1, 2016, pp. 197-206. DOI: 10.15514/ISPRAS-2016-28(1)-11
[6]. V. Casseau, R. Palharini, T. Scanlon, and R. Brown. A two-temperature open-source CFD model for hypersonic reacting flows, part one: zero-dimensional analysis. Aerospace, vol. 3(4), 2016.
[7]. I. Sarris, G. Zikos, A. Grecos, A. P., and N. Vlachos. On the limits of validity of the low magnetic Reynolds number approximation in MHD natural-convection heat transfer. Numerical Heat Transfer. Part B: Fundamentals, 50(2), pp. 157-180, 2006.
[8]. L. Spitzer and R. Härm. Transport phenomena in a completely ionized gas. Physical Review, 89(5), 977, 1953.
[9]. S. Chapman. and T. Cowling. The mathematical theory of non-uniform gases: an account of the kinetic theory of viscosity, thermal conduction and diffusion in gases. Cambridge university press, 1970.
[10]. Bush, W. B. Magnetohydrodynamic - Hypersonic Flow Past a Blunt Body. Journal of Aero/Space Sciences, Vol. 25, No. 11, November 1958, pp. 685–690.
[11]. Raizer, Y. P., Gas Discharge Physics, Springer-Verlag, 1991
[12]. С. Суржиков. Расчетное исследование аэротермодинамики гиперзвукового обтекания затупленных тел на примере анализа экспериментальных данных. М.: ИПМ им. АЮ Ишлинского РАН, 2011, 84 стр.

Three-dimensional numerical analysis of a dam-break using OpenFOAM

[1, 2] *Esteban Sánchez-Cordero <esteban.sanchezc@ucuenca.edu.ec>*
[2] *Manuel Gómez <manuel.gomez@upc.edu>*
[2] *Ernest Bladé <ernest.blade@upc.edu>*
[1] *Department of Civil Engineering, Faculty of Engineering, University of Cuenca, Cuenca, Ecuador*
[2] *Department of Civil and Environmental Engineering, FLUMEN Research Institute, Polytechnic University of Catalonia, Barcelona, Spain*

Abstract. This paper presents a 3D numerical analysis of flow field patterns in a dam break laboratory scale by applying the numerical code based on Finite Volume Method (FVM), OpenFOAM. In the numerical model the turbulence is treated with RANS methodology and the VOF (Volume of Fluid) method is used to capture the free surface of the water. The numerical results of the code are assessed against experimental data. Water depth and pressure measures are used to validate the numerical model. The results demonstrate that the 3D numerical code satisfactorily reproduce the temporal variation of these variables.

Keywords: dam break; 3D; RANS; k-ε (RNG); OpenFOAM

DOI: 10.15514/ISPRAS-2017-29(6)-20

For citation: Sánchez-Cordero E., Gómez M., Bladé E. Three-dimensional numerical analysis of a dam-break using OpenFOAM. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 6, 2017. pp. 311-320. DOI: 10.15514/ISPRAS-2017-29(6)-20

1. Introduction

A dam is an engineering structure constructed across a valley or natural depression to create a water storage reservoir. The fast-moving flood wave caused by a dam failure can result in the loss of human lives, great amount of property damage, and have a severe environmental impact. Therefore, significant efforts have been carried out over the last years to obtain satisfactory mathematical numerical solutions for this problem. Due to advances in computational power and the associated reduction in computational time, three-dimensional (3D) numerical models based on Navier Stokes equations have become a feasible tool to analyze the flow pattern in those days.

Analytical studies of the dam break for a horizontal channel were performed by Dressler [1]. Several numerical studies based on 2D approaches have been validated against experimental data sets as demonstrated in [2] and [3]. Two dimensional numerical models assume negligible vertical velocities and accelerations which results in a hydrostatic pressure distribution. However, when an abrupt failure of a dam happens, in which initially a high free surface gradients occurs, the hydrostatic pressure assumption is no longer valid. Three dimensional numerical models have been used to solve the structure of the flow in these areas.

This document presents a 3D numerical analyze of a dam break (laboratory scale) using the numerical code based on the finite volume method (FVM) – OpenFOAM. Turbulence is treated using Reynolds-averaged Navier Stokes equations (RANS) k-ε (RNG) approach, and the volume of fluid (VOF) method is used to simulate the air-water interface. The numerical results of the code are assessed against experimental data obtained by Kleefsman et al [4]. Water depth and pressure measures are used to validate the model. The results demonstrate that the 3D numerical code satisfactorily reproduce the temporal variation of these variables.

2. Experimental set-up model

Fig. 1 shows the schematic and the positions of the measured laboratory quantities performed by Kleefsman et al [4]. The dimensions of the tank are 3.22x1x1 m. The right part of the tank can be filled up with water up a height of 0.55 m using a sluice gate. Then the sluice gate opens abruptly and the water in the tank empties.

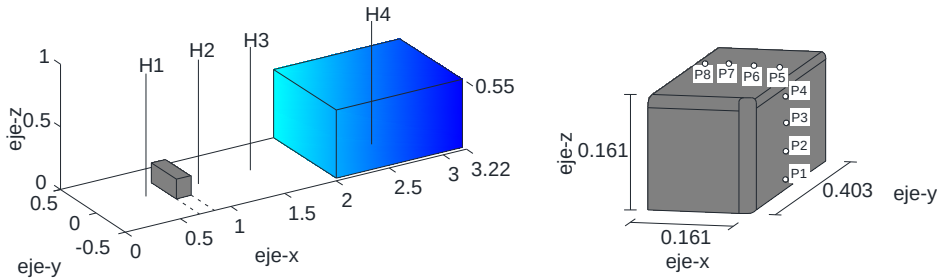


Fig. 1. Measurements positions for water heights and pressure (adapted from Kleefsman et al [4])

3. Numerical Model

3.1 Fluid Flow model

The governing equations for mass and momentum for the fluid flow can be expressed as [5]:

$$\nabla \cdot u = 0 \qquad (1)$$

$$\frac{\partial \rho u}{\partial t} + \nabla \cdot (\rho u\, u) - \nabla \cdot \big((\mu + \mu_t)S\big) = -\nabla p + \rho g + \sigma K \frac{\nabla \alpha}{|\nabla \alpha|} \qquad (2)$$

where u is velocity vector field, p is the pressure field, μ_t is the turbulent eddy viscosity, S strain tensor ($S = 1/2\,(\nabla u + \nabla u^t)$, σ surface tension, K surface curvature, and α volume fraction function (between 0-1).

3.2 Free surface model

Volume of Fluid Method (VOF) is used for the analysis of free surface flow. A volume fraction indicator α is used to determine the fluid contained at each mesh element. To calculate α a new transport equation is introduced.

$$\frac{\partial \alpha}{\partial t} + \nabla(\alpha \cdot u) + \nabla \cdot (\alpha(1-\alpha)u_r) = 0 \qquad (3)$$

OpenFOAM imposes the third term of equation (3) called phase compression; where, $u_r = u_l - u_g$. The density ρ and viscosity μ in the domain are given by:

$$\rho = \alpha\rho_l + (1-\alpha)\rho_g \qquad (4)$$

$$\mu = \alpha\mu_l + (1-\alpha)\mu_g \qquad (5)$$

where *l* and *g* denotes the different fluids (water and air).

3.3 Turbulence model

In the RANS equations the instantaneous variables of flow are decomposed into their time-averaged and fluctuating quantities. In this analysis k-ε (RNG) turbulence model is used due to provides an improved performance for types of flows that include flows with separation zones [6]. It is a two equation model which provides independent transport equations for both the turbulence length scale and the turbulent kinetic energy.

3.3 Initial and Boundary Conditions

In the numerical configuration of the model, the sides surrounding the experiment and the bottom are defined as wall. The top of the experimental box atmospheric pressure prevails. At the beginning of the simulation an initial water height is established, which is the initial water volume of the experiment.

3.4 Model validation – Grid convergence

In this subsection, three grid resolution values are evaluated for the grid convergence. The domain is discretized using a structured mesh made up of hexahedral elements. The mesh sizes to be analyzed are 2, 1.5 and 1 cm. The water depth variable is chosen as the analysis due to the reliability of the measurement of its values. In order to quantify the numerical assessment quadratic mean value R^2 is used. Fig. 2 shows how the statistical value R^2 increases when the mesh size decreases in the four measurement points.

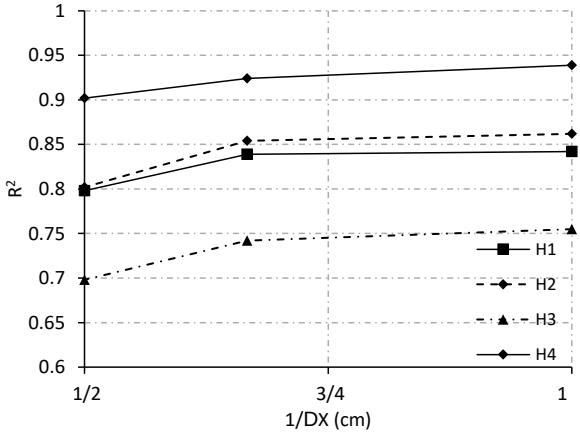


Fig. 2. Grid mesh analysis

3.5 Numerical Simulation

In this study, after the mesh analysis a grid of 1 cm is used. The grid cells has been used with some narrowing towards the bottom and the walls of the tank. An Explicit 2nd order limited scheme for the convection term, Explicit second order scheme for the diffusion term, and first order Euler scheme for the transient term are used. The simulation is continued for 7 s with an automatically adapted time step using maximum CFL-numbers around 0.50.

4. Results and Discussion

In order to analyze the capabilities of the numerical model in the reproduction of the flow variables in a dam break, a comparison between RANS numerical simulation and experimental data was quantified by the quadratic mean value R^2.

4.1 Water Depth

The evolution in time of water depth (H1, H2, H3, and H4) are shown in Fig. 3. A qualitative evaluation of the results shows that the 3D model configuration is able to reproduce satisfactorily the variability in time of the water depth in the four points of study. Additionally, a qualitative evaluation of the results shows that the 3D numerical model explains the variability in time of water depth (Fig. 4). The best numerical data occurs at the point denominated H4 (Fig. 4-d), this point is located inside the water tank formed at the beginning of the experiment. On the other hand, the worse numerical data occurs in the point denominated H3 (Fig. 4-c).

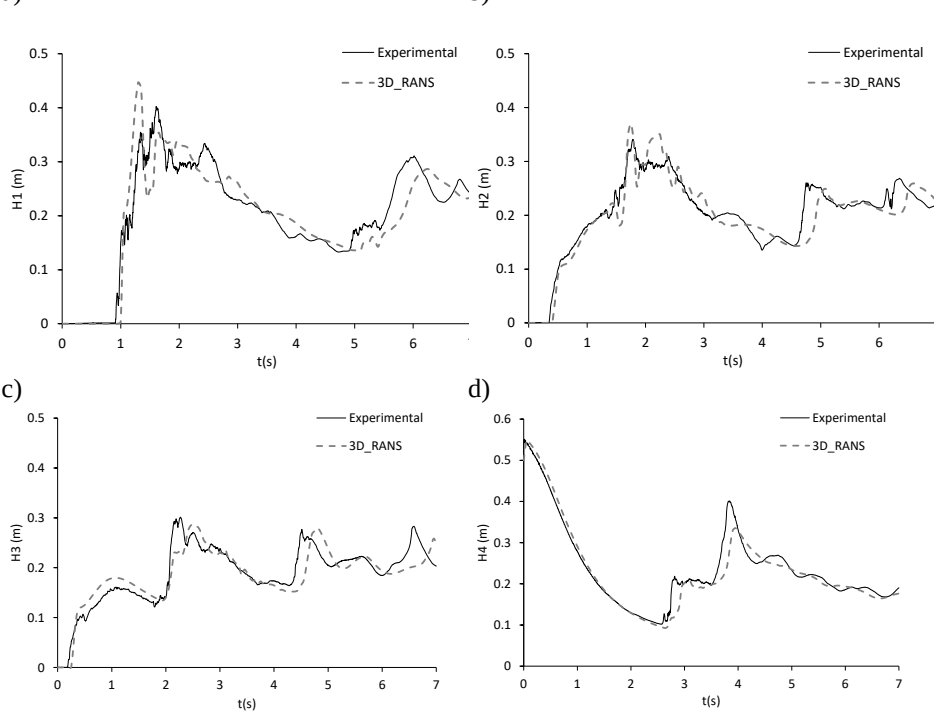


Fig. 3. Comparison of simulated and measured water depth in time
a) H1, b) H2, c) H3, and d) H4

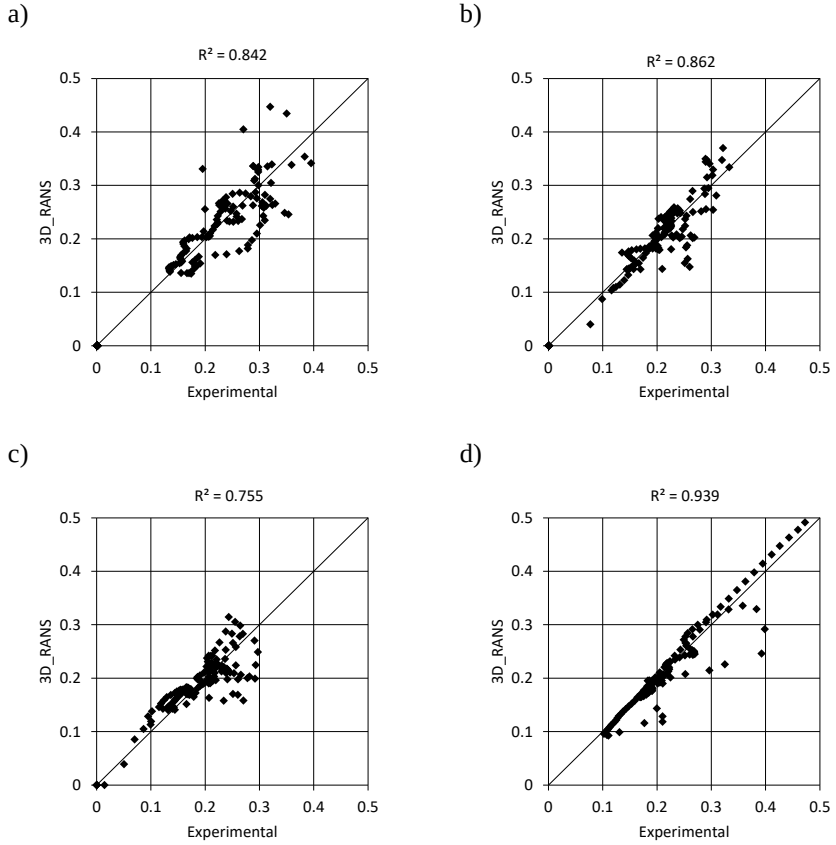


Fig. 4. Comparison of simulated and measured water depth (m)
a) H1, b) H2, c) H3, and d) H4

4.2 Pressure

Fig. 5 shows the variation in time of the pressure variable measured at different points of the obstacle. The points P1, P3, P5, and P7 are analyzed. A qualitative analysis shows a better capture of the temporal variation of the points in the frontal face of the obstacle (P1, P3) than those that are in the upper part of the obstacle (P5, P7). Numerical model results shows a time lag of peak value in the points P1 and P3. Quantitative analysis allows for qualitative assertion (Figure 6). Thereby, R^2 is 0.747 and 0.606 for pressure P1 and P3 respectively, whilst P5 and P7 are 0.595 and 0.576.

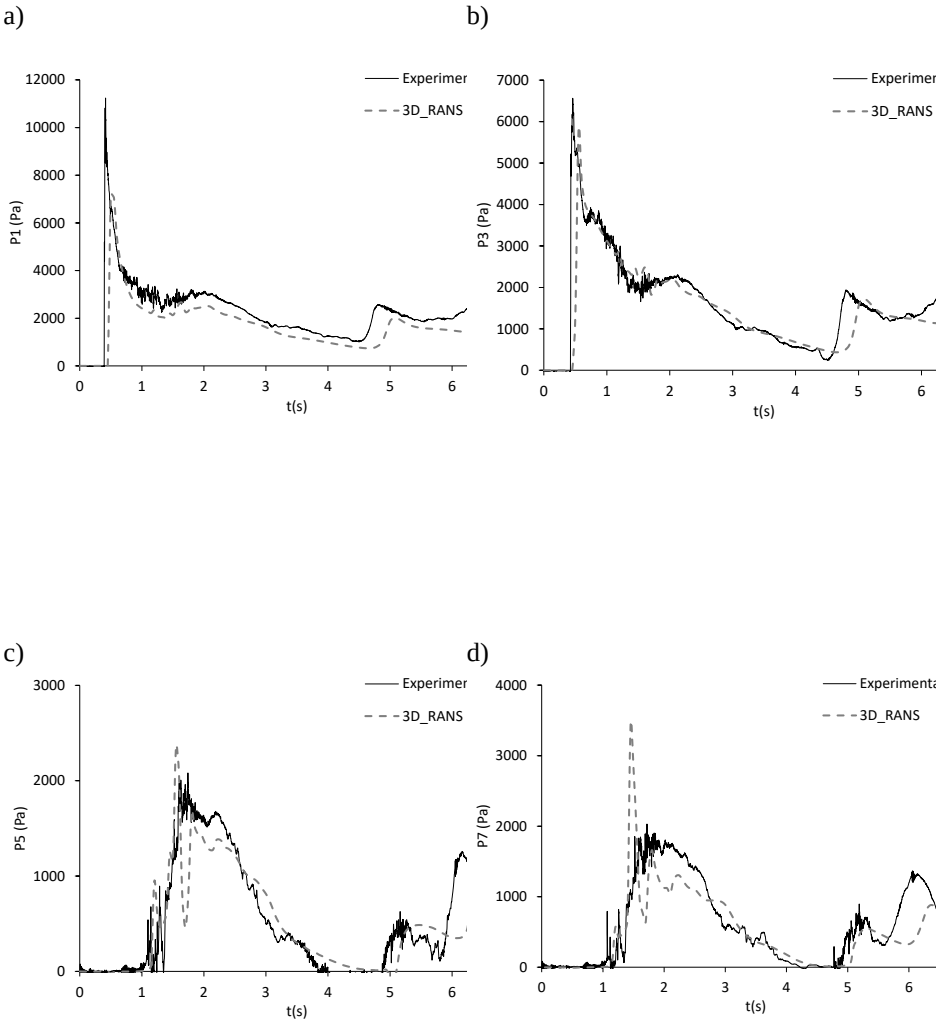


Fig. 5. Comparison of simulated and measured pressure in time
a) P1, b) P3, c) P5, and d) P7

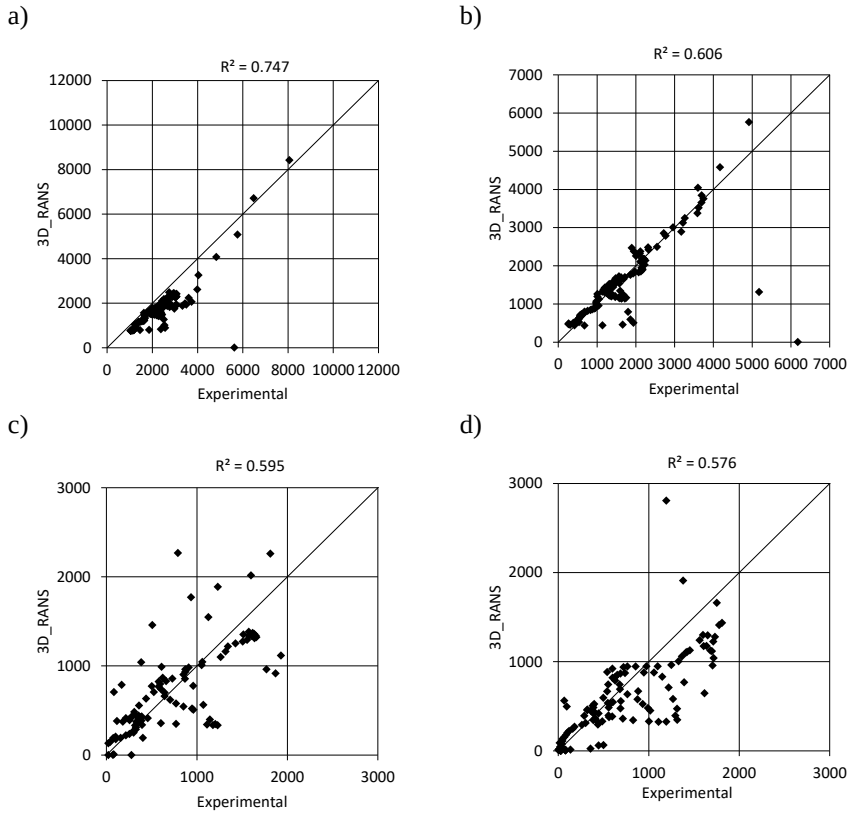


Fig. 6. Comparison of simulated and measured pressure (Pa)
a) P1, b) P3, c) P5, and d) P7

5. Conclusions

This study investigates the applicability of OpenFOAM code for the generation of flow field variables - water depth and pressure- in a dam-break laboratory scale using the RANS approach. The results demonstrate that the 3D numerical model configuration with RANS k-ε (RNG) approach can provide reliable results of the flow field in a dam-break case. Water depth values are reproduced better than pressure values by the 3D numerical model. Although the matching between the numerical solution and the physical experiment is quite promising, the application of the 3D numerical model for field-scale simulation would be computationally expensive.

Acknowledgments

This work was made possible largely because the financial support given by Ecuadorian Government's Secretaria Nacional de Educacion Superior, Ciencia y Tecnologia (SENESCYT) through a PhD grant for the first author.

References

[1]. R. F. Dressler, “Hydraulic Resistance Effect Upon the Dam-Break Functions*,” J. Res. Natl. Bur. Stand. (1934)., vol. 49, no. 3, 1952.
[2]. L. Fraccarollo and E. F. Toro, “Experimental and numerical assessment of the shallow water model for two-dimensional dam-break type problems,” J. Hydraul. Res., vol. 33, no. 6, pp. 843–864, Nov. 1995.
[3]. S. Soares-Frazão and Y. Zech, “Experimental study of dam-break flow against an isolated obstacle,” J. Hydraul. Res., vol. 45, no. sup1, pp. 27–36, Dec. 2007.
[4]. K. M. T. Kleefsman, G. Fekken, A. E. P. Veldman, B. Iwanowski, and B. Buchner, “A Volume-of-Fluid based simulation method for wave impact problems,” J. Comput. Phys., vol. 206, no. 1, pp. 363–393, Jun. 2005.
[5]. X. Liu and M. H. García, “Three-Dimensional Numerical Model with Free Water Surface and Mesh Deformation for Local Sediment Scour,” J. Waterw. Port, Coastal, Ocean Eng., vol. 134, no. 4, pp. 203–217, 2008.
[6]. A. S. Ramamurthy, S. S. Han, and P. M. Biron, “Three-Dimensional Simulation Parameters for 90° Open Channel Bend Flows,” J. Comput. Civ. Eng., vol. 27, no. 3, pp. 282–291, 2013.

Трехмерный численный анализ прорыва плотины с использованием OpenFOAM

[1, 2] *Эстебан Санчес-Кордеро <esteban.sanchezc@ucuenca.edu.ec>*
[2] *Мануэль Гомез <manuel.gomez@upc.edu>*
[2] *Эрнест Блейд <ernest.blade@upc.edu>*
[1] *Эквадор, г. Куэнка, Университет Куэнки,*
Инженерный факультет, Департамент строительства
[2] *Испания, Барселона, Политехнический университет Каталонии,*
Научно-исследовательский институт FLUMEN,
Департамент строительства и охраны окружающей среды

Аннотация. В статье представлен трехмерный численный анализ структуры поля течения при прорыве плотины в масштабах лаборатории путем. Численные вычисления основывались на методе конечных объемов (Finite Volume Method, FVM), OpenFOAM. В численной модели турбулентность обрабатывается с применением методологии RANS, а метод жидких объёмов (VOF, Volume of Fluid) используется для отслеживания свободной поверхности воды. Результаты вычислений оцениваются на основе экспериментальных данных. Для валидации численной модели используются измерения глубины и давления воды. Результаты показывают, что численные

вычисления удовлетворительно воспроизводят изменения этих переменных во времени.

Ключевые слова: прорыв платины; 3D; RANS; k-ε (RNG); OpenFOAM

DOI: 10.15514/ISPRAS-2017-29(6)-20

For citation: Санчес-Кордеро Э., Гомез М., Блейд Э. Трехмерный численный анализ прорыва плотины с использованием OpenFOAM. Труды ИСП РАН, том 29, вып. 6, 2017 г., стр. 311-320 (на английском языке). DOI: 10.15514/ISPRAS-2017-29(6)-20

Список литературы

[1]. R. F. Dressler, “Hydraulic Resistance Effect Upon the Dam-Break Functions*,” J. Res. Natl. Bur. Stand. (1934)., vol. 49, no. 3, 1952.

[2]. L. Fraccarollo and E. F. Toro, “Experimental and numerical assessment of the shallow water model for two-dimensional dam-break type problems,” J. Hydraul. Res., vol. 33, no. 6, pp. 843–864, Nov. 1995.

[3]. S. Soares-Frazão and Y. Zech, “Experimental study of dam-break flow against an isolated obstacle,” J. Hydraul. Res., vol. 45, no. sup1, pp. 27–36, Dec. 2007.

[4]. K. M. T. Kleefsman, G. Fekken, A. E. P. Veldman, B. Iwanowski, and B. Buchner, “A Volume-of-Fluid based simulation method for wave impact problems,” J. Comput. Phys., vol. 206, no. 1, pp. 363–393, Jun. 2005.

[5]. X. Liu and M. H. García, “Three-Dimensional Numerical Model with Free Water Surface and Mesh Deformation for Local Sediment Scour,” J. Waterw. Port, Coastal, Ocean Eng., vol. 134, no. 4, pp. 203–217, 2008.

[6]. A. S. Ramamurthy, S. S. Han, and P. M. Biron, “Three-Dimensional Simulation Parameters for 90° Open Channel Bend Flows,” J. Comput. Civ. Eng., vol. 27, no. 3, pp. 282–291, 2013.