На правах рукописи

Курмангалеев Шамиль Фаимович

Автоматизация переноса Си/Си++-приложений на новые платформы

Специальность 05.13.11 –

математическое и программное обеспечение

вычислительных машин, комплексов и компьютерных сетей

Автореферат

диссертации на соискание ученой степени

кандидата физико-математических наук

Москва

2013

Работа выполнена в Федеральном государственном бюджетном учреждении науки Институте системного программирования РАН.

Научный руководитель: Аветисян Арутюн Ишханович, доктор физико-математических наук, доцент

Официальные оппоненты: Галатенко Владимир Антонович, доктор физико-математических наук, старший научный сотрудник, заведующий сектором автоматизации программирования Федерального государственного бюджетного учреждения науки Научно-исследовательского института системных исследований Российской академии наук

Волконский Владимир Юрьевич кандидат технических наук, старший научный сотрудник открытого акционерного общества "Институт электронных управляющих машин им. И.С. Брука"

Ведущая организация: Федеральное государственное бюджетное учреждение науки Институт прикладной математики им. М.В. Келдыша Российской академии наук

Защита диссертации состоится 31 октября 2013 г. в 15 часов на заседании диссертационного совета Д 002.087.01 при Федеральном государственном бюджетном учреждении науки Институте системного программирования Российской академии наук по адресу: 109004, Москва, ул. Александра Солженицына, д. 25.

С диссертацией можно ознакомиться в библиотеке Федерального государственного бюджетного учреждения науки Института системного программирования Российской академии наук.

Автореферат разослан "___" _____ 2013 г. / Прохоров С.П./

Учёный секретарь
диссертационного совета,
кандидат физ.-мат. наук

Актуальность

На сегодняшний день всё более популярной становится схема распространения программного обеспечения (ПО) через Интернет-магазины приложений. Самыми известными такими магазинами являются Apple App Store (более 900 000 приложений), Google Play (более 800 000 приложений), Microsoft Store (более 115 000 приложений).

Процесс распространения ПО состоит в следующем: разработчик передаёт программный продукт владельцу магазина приложений, приложение размещается в интернет-магазине и становится доступным пользователю. Чтобы программный продукт оставался конкурентоспособным в течение длительного времени, разработчику необходимо обеспечивать функционирование приложения на большинстве программно-аппаратных платформ, поддерживаемых данным магазином приложений, добавляя поддержку новых платформ, в том числе оптимизировать программный продукт под новые версии ОС, модификации аппаратуры и др. Эта проблема особенно актуальна в связи со стремительным развитием мобильных платформ (обновление каждые несколько месяцев).

Как правило, эта проблема решается следующим образом: приложение или компилируется и оптимизируется разработчиком повторно для каждой новой платформы, с последующим добавлением в магазин новой версии бинарного кода, или реализуется с использованием динамических языков. Так для разработки приложений, не требующих контроля над доступными ресурсами, достаточно применения HTML 5 в совокупности с JavaScript. В данном случае разработчику не нужно реализовывать поддержку самих платформ, однако требуется обеспечивать совместимость с наиболее популярными браузерами, используемыми на этих платформах.

При применении JavaVM приложение распространяется в промежуточном представлении. Переносимость достигается ценой потери производительности из-за использования дополнительного слоя абстракции, скрывающего реальное оборудование от исполняемой программы. В этом случае для обеспечения приемлемого уровня производительности дополнительно применяются динамические оптимизации на целевой архитектуре во время исполнения приложения, в том числе оптимизации, учитывающие профиль исполнения программы. Примером использования такого подхода для мобильных платформ может служить Java со специальной версией виртуальной машины Dalvik VM для Android, или C# в сочетании с виртуальной машиной, разрабатываемой в рамках проекта Mono для мобильной игровой платформы PS Vita.

Однако, использование динамически компилируемых языков не всегда обеспечивает приемлемые характеристики приложений, особенно в контексте минимизации энергопотребления. Разработка на традиционных языках (например, Си/Си++) позволяет учитывать особенности платформ, используя машинно-зависимые оптимизации (распределения регистров, планирования и конвейеризации кода, векторизации), выигрыш от применения которых может

достигать нескольких десятков процентов. По этой причине производитель предоставляет Native SDK, позволяющий использовать такие языки. Вместе с тем, разработка приложений на традиционных языках и их распространение через магазины приложений требуют существенных дополнительных накладных расходов разработчиков на перенос приложений, в связи с необходимостью поддержки большого количества различных платформ. Это серьезно ограничивает количество приложений, разрабатываемых на традиционных языках.

В настоящее время активно ведутся работы по автоматизации переноса приложений на языках Си/Си++ на различные платформы. Эти работы базируются на идее компилирования Си/Си++-приложений в промежуточное представление и их распространение по аналогии с приложениями на динамических языках. Наиболее интересные результаты получены компанией Google, которая разрабатывает проект Portable Native Client, имеющий целью обеспечить запуск единой версии программы на архитектурах ARM и x86. Эта разработка рассчитана на небольшие приложения, работающие в браузере Chrome, и имеет ряд ограничений, в том числе ухудшение показателей производительности.

Кроме того, такой подход ставит особенно остро проблему защиты приложений от обратной инженерии, которая актуальна ввиду конкуренции разработчиков приложений и высокого уровня пиратства.

На сегодняшний день не существует инфраструктуры разработки и распространения приложений на языках Си/Си++, удовлетворяющей следующим требованиям:

1. Автоматизация переноса Си/Си++-приложений на новые платформы с сохранением показателей производительности;
2. Сохранение процесса разработки ПО, принятого у пользователя;
3. Защита ключевых участков кода от обратной инженерии.

Таким образом, тема развития методов автоматизации переноса Си/Си++ приложений на новые платформы, которой посвящена диссертация, является актуальной.

Целью диссертационной работы является исследование и разработка методов статического и динамического анализа программ и соответствующей компиляторной инфраструктуры, обеспечивающих возможность разрабатывать и распространять Си/Си++-приложения, удовлетворяя сформулированным требованиям 1 – 3.

Научная новизна.

Научной новизной обладают следующие результаты:

1. Метод двухфазной компиляции Си/Си++-приложений, позволяющий разделить фазу генерации машинно-независимого внутреннего представления и фазу генерации кода для целевой платформы, на которой обеспечивается автоматический учет особенностей конкретной аппаратуры и окружения.

2. Методы машинно-независимой оптимизации, учитывающие данные профиля программы: открытая вставка функций, клонирование базовых блоков, вынос редко исполняемого кода в функции, спекулятивная девиртуализация функций языка Си++, формирование суперблоков. Методы машинно-зависимой оптимизации, учитывающие специфику конкретной платформы: автоматическая генерация команд предвыборки, модификация алгоритма распределения регистров.

3. Методы запутывания программного кода: перенос локальных переменных в глобальную область видимости, шифрование строк, вставка фиктивных циклов, преобразование «диспетчер», переплетение функций, сокрытие вызовов функций, генерация несводимых участков в графе потока управления, клонирование функций, разбиение констант.

Практическая значимость. Разработанная компиляторная инфраструктура позволяет создать «облачное хранилище» приложений нового поколения, обеспечивающее, с одной стороны, переносимость программ в рамках семейства ARM, а с другой стороны, высокую степень защищенности приложений за счет использования методов запутывания. Разработанные программные средства и методы используются в научно-исследовательских и коммерческих организациях. Предложенное изменение в алгоритме распределения регистров одобрено сообществом разработчиков LLVM для включения в основную ветвь разработки в апреле 2012. Разработанный метод распространения приложений реализован и внедрен в рамках мобильной промышленной платформы Tizen.

Апробация работы и публикации. По теме диссертации опубликованы 4 работы в изданиях из перечня ВАК. Список работ приведен в конце автореферата. Основные результаты также представлены в докладах на следующих конференциях:

- 55-ая научная конференция МФТИ. 19-25 ноября 2012 г.Долгопрудный.
- «РусКрипто’2013», 27 — 30 марта 2013 года, Московская область, Солнечногорский р-н, ГК «Солнечный Park Hotel & SPA»

Структура и объем работы. Диссертация состоит из введения, 5 глав, заключения. Работа изложена на 102 страницах. Список источников насчитывает 73 наименования. Диссертация содержит 8 таблиц и 28 рисунков.

Краткое содержание работы

Во введении обсуждаются проблемы, возникающие при распространении программ, написанных на языках Си/Си++**,** формулируются цели и задачи работы, обосновывается ее актуальность, обсуждаются вопросы практического применения разработанных методов и инструментов, производится краткий обзор работы.

Глава 1 посвящена обзору работ, имеющих непосредственное отношение к теме диссертации. В разделе 1.1 рассматриваются существующие подходы к решению задачи распространения ПО на различных платформах.

Предлагаемые подходы могут использоваться для распространения программ, но обладают существенными ограничениями:

1. Использование дополнительного уровня абстракции понижает быстродействие, увеличивает потребление ресурсов системы. В силу ограничений, налагаемых на код пользовательских программ в целях сохранения переносимости и обеспечения безопасности, затрудняется или становится невозможной прямая работа с системными библиотеками, что также ведет к понижению быстродействия.
2. Рассмотренные методы не предполагают сохранения существующего процесса разработки ПО, кроме того, объединение в одном файле бинарного кода для разных платформ на практике не позволяет учесть особенности конкретной программно-аппаратной платформы, поскольку версии кода генерируются в рамках всей архитектуры, и обязаны функционировать на всем семействе процессоров.

Важным требованием для распространения ПО является предоставление средств защиты от обратной инженерии. Раздел 1.2 посвящен обзору методов запутывания программ, формулируются правила, которым необходимо следовать при запутывании программы для повышения качества защиты от обратного проектирования. Ниже сформулированы требования к запутывающим преобразованиям:

1. Маскирующее преобразование должно затрагивать и поток управления, и поток данных запутываемой программы;
2. Стойкость преобразования должна основываться на алгоритмически сложных задачах, например, требовать от атакующего применения анализа указателей для точного восстановления потоков данных защищенной программы;
3. При разработке преобразования нужно учитывать особенности работы средств анализа. Например, для автоматических декомпиляторов следует насытить граф потока управления несводимыми участками.

В главе 2 рассматривается предлагаемый метод двухфазной компиляции. В разделе 2.1 описывается схема функционирования предлагаемого решения**,** обеспечивающего распространение программ, написанных на языках высокого уровня Си/Си++, в промежуточном представлении компиляторной инфраструктуры LLVM. Такой подход позволяет организовать как статическую, так и динамическую компиляцию программ, а также оптимизации с учетом профиля выполнения программы. Кроме того, предложенный подход позволяет производить запутывание программы без использования ее исходного кода, при этом располагая информацией (потоки данных, граф потока управления и т.д.) о программе, доступной компилятору.

В предлагаемой реализации метода двухфазной компиляции на первом этапе приложение компилируется на машинах разработчиков специальным

набором компиляторных инструментов на базе LLVM, при этом выполняются лишь машинно-независимые оптимизации. Результат компиляции сохраняется в файлах с биткодом LLVM, дополнительно автоматически генерируется информация об устройстве программного пакета и о схеме его инсталляции. На втором этапе программа оптимизируется на машине пользователя, возможно, с учетом его поведения и особенностей его вычислительной системы. Поддерживается несколько режимов работы: а) автоматическая генерация кода бинарной программы, оптимизированной под конкретную архитектуру, и ее развертывание с помощью сохраненной на первом этапе информации; б) динамическая оптимизация программы во время её работы с учетом собранного профиля пользователя; в) оптимизация программы с учетом профиля пользователя во время простоя системы (idle-time optimization), для экономии ресурсов.

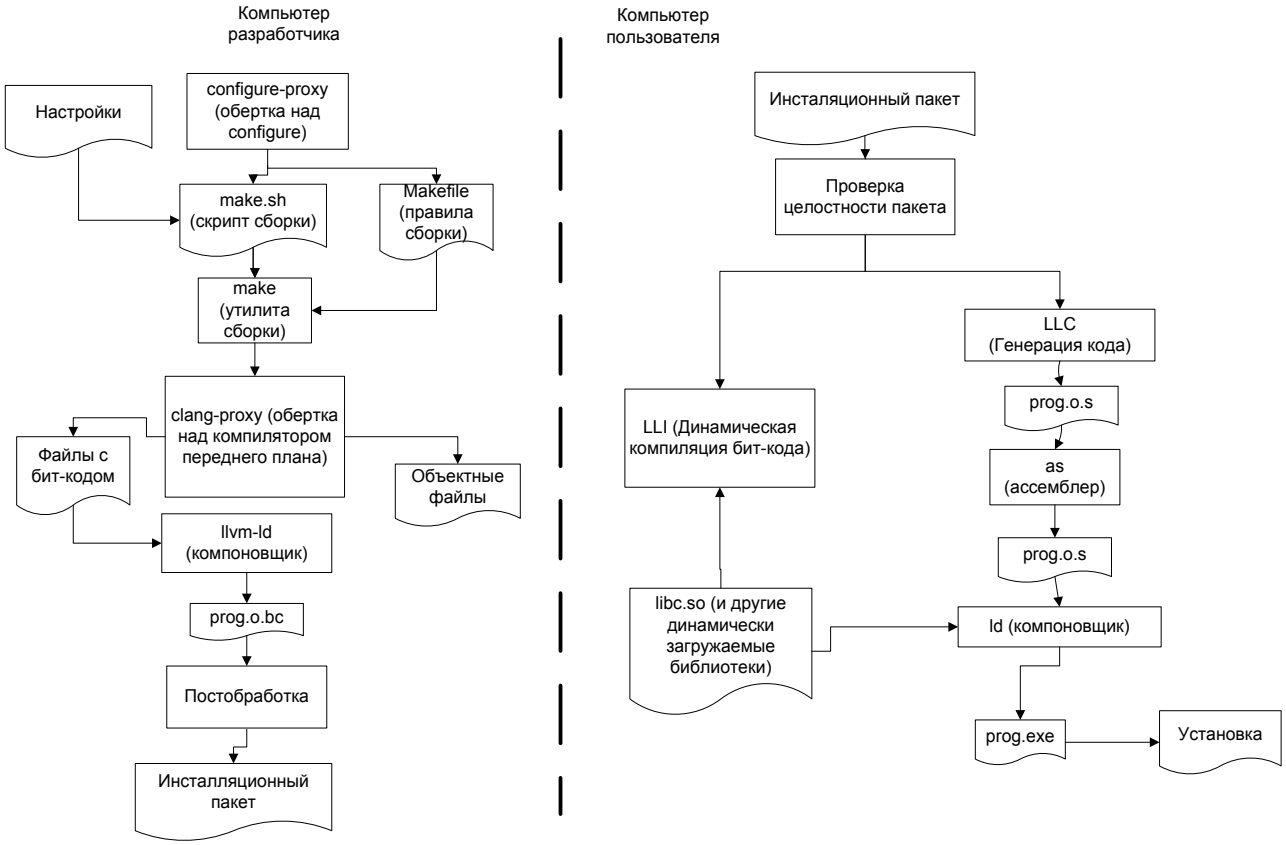

Рис. 1. Схема двухфазной компиляции

Схема работы системы двухфазной компиляции для программ, сборка которых основана на использовании утилит configure и make, представлена на рисунке 1. Указанные на рисунке дополнительные инструменты были разработаны и реализованы из-за того, что в LLVM не предусмотрены средства прозрачного, автоматического получения биткода с учетом зависимостей между модулями, а также отсутствует поддержка динамического связывания модулей с биткодом.

В разделе 2.2 описываются изменения, внесенные в компилятор переднего плана и компоновщик, позволяют отследить зависимости между отдельными

модулями программы. После окончания компиляции программы с помощью скриптов пост-обработки, создается инсталляционный пакет на основе сгенерированных зависимостей. Инсталляционный пакет содержит файлы с биткодом, файлы, помеченные как зависимости на этапе постобработки, и скрипты компиляции и установки.

Дополнительно была проведена оптимизация компонент LLVM для их более быстрой работы и потребления меньшего объема памяти, что существенно для встраиваемых архитектур. Во время генерации кода для программ, на платформе ARM, было достигнуто сокращение использования памяти на 1.6-10.9% и времени компиляции на 10-20%.

В главе 3 рассматриваются предложенные методы оптимизации. Раздел 3.1 содержит описание машинно-независимых оптимизации, учитывающие данные профиля программы: открытая вставка функций, клонирование базовых блоков, вынос редкоисполняемого кода в функции, спекулятивная девиртуализация функций языка Си++, формирование суперблоков. В разделе 3.2 описываются методы машинно-зависимой оптимизации, учитывающие специфику конкретной платформы: автоматическая генерация команд предвыборки, модификация алгоритма распределения регистров.

Открытая вставка функций – оптимизирующее преобразование компилятора, вставляющее код функции на место его вызова в тело вызывающей функции.

Была рассмотрена реализация в компиляторе GNU GCC. Эта оптимизация использует данные о характере исполнения программы для решения вопроса о вставке функции: относительная частота вызова функций задается параметром frequency в пределах от 0 до 1, количество вызовов задается параметром calls, а потенциальный рост общего количества инструкций задается параметром growth. Таким образом, решение о том, вставить ли малую функцию вычисляется:

growth < 0 → growth

growth >=0 → (calls/growth или growth/frequency) – в зависимости от того, включен глобальный или локальный профиль соответственно.

Вес функции вычисляется по формуле:

$$Func_{Weight} = Num_{Instr} \times Penalty_{Instr} - Num_{Args} \times Penalty_{Alloca} - Num_{ConstInstr} \times Penalty_{Constant}, \quad (1)$$

где

$Penalty_{Instr}$=2 – штраф за каждую инструкцию в функции

$Penalty_{Alloca}$=2 – штраф за локальную переменную

$Penalty_{Constant}$=2 – штраф за константу

Вес функции с учетом информации из профиля вычисляется следующим образом:

$$NewWeight = (NumCallFromProfileInfo)/Function_{Weight} \quad (2)$$

После присвоения весов, функции сортируются по возрастанию веса и встраиваются до тех пор, пока суммарный вес не превысит некоторого значения (по умолчанию 1000).

Доказано **утверждение 1:** Выигрыш от встраивания функции рассчитывается по формуле:

$$Profit_{inline} = \frac{CallCost_{called}}{FullCost_{Caller}} * 100\%, \quad (3)$$

где

$CallCost_{called}$- стоимость вызова кандидата на встраивание

$\mathrm{FullCost_{Caller}}$ – стоимость выполнения вызывающей функции до модификации

На тестах SQLite, Expedite, Cray и Coremark оптимизация дала прирост скорости в ~2%.

Клонирование блоков - удвоение часто исполняемых базовых блоков графа потока управления, имеющих более одного исходящего ребра и более одного входящего. Суть алгоритма показана на рисунке 2.

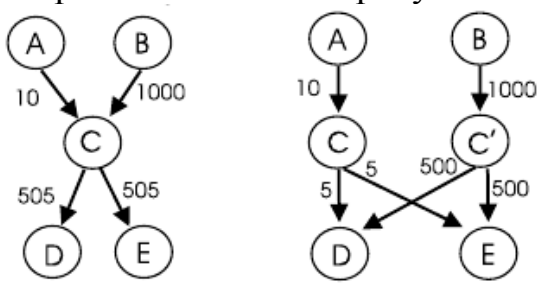

Рис. 2 Клонирование соединенных базовых блоков: слева - до разбиения, справа - после

Данное преобразование не является оптимизирующим само по себе, но позволяет оптимизациям, основанным на анализе потока данных, - таким, как удаление загрузок, удаление границ массивов, замещение на стеке и пр., - работать более эффективно.

Клонирование блоков, позволяет повысить вероятность проведения различных оптимизаций по каждому пути.

N- количество взаимоисключающих оптимизаций, для каждого из путей

P_i- вероятность проведения оптимизации на i- том пути

$P_{i,clone}$ - вероятность проведения оптимизации на i- том пути, после разбиения

P_0- вероятность отсутствия возможности оптимизаций на выбранном пути

тогда суммарная вероятность проведения любой из оптимизаций:

$$\sum P_i = 1 - P_0 \quad (4)$$

В случае создания эквивалентного множества путей, вероятность оптимизации по i-тому пути:

$$P_{i,clone} = 1 - P_0 \quad (5)$$

Видно, что $P_{i,clone} > P_i$

Вынос "холодных" участков кода в отдельные функции. Для оптимизации предлагается рассматривать функции, которые исполняются наибольшее число раз (”горячие”). Если функцию условно можно разделить на две части: большой редко исполняемый участок кода, относительно малый "горячий" участок. Из таких функций предлагается выносить ”холодную” часть функции в отдельную новую функцию, уменьшая при этом размер

рассматриваемой функции и расстояния в памяти между часто исполняемыми участками кода.

Для выделения ”горячих” функций используется одномерная кластеризация по весу из профиля с помощью алгоритма k-средних при $k = 3$. Функции разделяются на 3 класса: ”горячие”, ”средние” и ”холодные”.

Начальными центрами масс каждого класса выбраны максимальный, средний и минимальный веса функций соответственно.

Для выделения ”холодных” ребер внутри ”горячих” функций рассматриваются условные переходы: считаем ребро холодным, если оно имеет вес в 100 и более раз меньший другого ребра данного условного перехода. После выделения "холодного" ребра, мы должны выделить максимальный набор базовых блоков, в которые может попасть поток управления только при прохождении по выделенному на предыдущем шаге ребру. Выбираются все блоки, над которыми доминирует блок, в который входит выделенное ребро. Для случая с множественными выходами, мы создаем специальный блок, который будет единственной точкой выхода из функции. В созданном блоке автоматически выбирается нужный код возврата. Фи-узлы корректируются разделением на две части, одна остается во внешнем коде, другая вставляется в последний блок выносимого кода, обрабатывая соответствующие переходы. Вынос редкоисполняемого кода в отдельную функцию, повышает эффективность программы, за счет более эффективного использования кэша процессора.

На тестах SQLite, Expedite, Cray и Coremark оптимизация дала средний прирост скорости в 0,8%. При использовании ее вместе с оптимизацией открытой вставки функций получен средний прирост в ~3%. Размер исполняемого файла увеличивается на 1- 7%, в зависимости от приложения.

Спекулятивная девиртуализация. Для объектно-ориентированных языков программирования решение о вставке виртуальных функций является проблемой, для решения которой недостаточно знать количество ее вызовов. В программах, написанных на языке Си++, могут быть два типа виртуальных вызовов функций: классические вызовы по указателю на функцию и вызовы виртуальных методов классов. Когда компилятор встречает такие вызовы, он не может определить, какая функция будет вызываться. Чтобы понять, какая функция будет вызвана, необходимо произвести дополнительный анализ. Этот анализ включает в себя: сравнение сигнатур функций, анализ иерархии наследования классов и анализ типов существующих в точке вызова. Сравнение сигнатур отсекает «неподходящие» по возвращаемому значению и параметрам функции. Анализ иерархии наследования выявляет классы, для которых существует реализация виртуального метода. Анализ существующих в точке вызова объектов рассматривает, объекты каких классов были созданы и еще не уничтожены в момент вызова функции.

Помимо этого была добавлена возможность инструментирования вызовов виртуальных функций сохранением информации о количестве вызовов конкретной виртуальной функции. Таким образом, используя профиль, мы

можем определить наиболее вероятного кандидата на девиртуализацию.

Реализованный алгоритм сочетает в себе вышеописанные методы. После проведения девиртуализации и принятия решения о вставке функции, если оказывается, что кандидат на вставку всего один, – вставляется он. Если кандидатов несколько - производится спекулятивная девиртуализация: по данным профилирования мы можем сказать, какая реализация виртуальной функции исполнялась наиболее часто, и вставляем инструкцию “if”, в теле которой производится вставка «горячей» функции, а в ветке “else” произведется вызов альтернативной, «холодной» версии функции.

Доказано **утверждение 2:** Преобразование девиртуализации, сохраняет корректность программы.

Поскольку биткод LLVM не содержит высокоуровневой информации, анализ иерархии должен производиться в компиляторе переднего плана, с сохранением результатов в метаинформации биткода. Таким образом, понадобилось дополнительно реализовать экспорт метаинформации в компиляторе переднего плана Clang. Во время тестирования был отмечен прирост производительности в ~5% при использовании только статического анализа и до ~7% с использованием спекулятивного алгоритма. Тестирование производилось на программе Clucene совместно с оптимизацией вставки функций. Алгоритм успешно проходит синтетические тесты для тестирования девиртуализации предложенные сообществом GCC.

Формирование суперблоков. Классические оптимизации используют статические методы анализа, такие как анализ времени жизни переменных или анализ достигающих определений, для обеспечения корректности преобразований кода. Эти методы не различают часто и редко исполняемые пути. Однако, часто бывают случаи, когда значение портится на редко исполняемом пути, который существует, например, для обработки событий. В результате невозможно применить оптимизации к часто исполняемым путям, пока редко исполняемые пути не будут исключены из анализа. Это требует точной оценки поведения программы во время исполнения.

Введем структуру данных, называемую суперблоком, для отображения частоисполняемых путей. Суперблок – линейная последовательность базовых блоков, в которую можно попасть только через первый блок последовательности. Поток управления может покинуть суперблок из любого блока последовательности. Когда суперблок начинает исполняться, вероятно, что все его базовые блоки будут исполнены. Рассмотрим взвешенный граф потока управления, который получился после сбора профиля от отработавшей программы.

Весом ребра является вероятность перехода от одного блока к другому. Для формирования суперблоков мы последовательно находим новую трассу, т.е. последовательность базовых блоков, исполненную чаще других, и копируем хвост трассы для каждого блока, через который можно покинуть трассу. Поиск длится до тех пор, пока он возможен (суперблоки из одного базового блока не имеют смысла). Пример работы алгоритма представлен на рисунке 3.

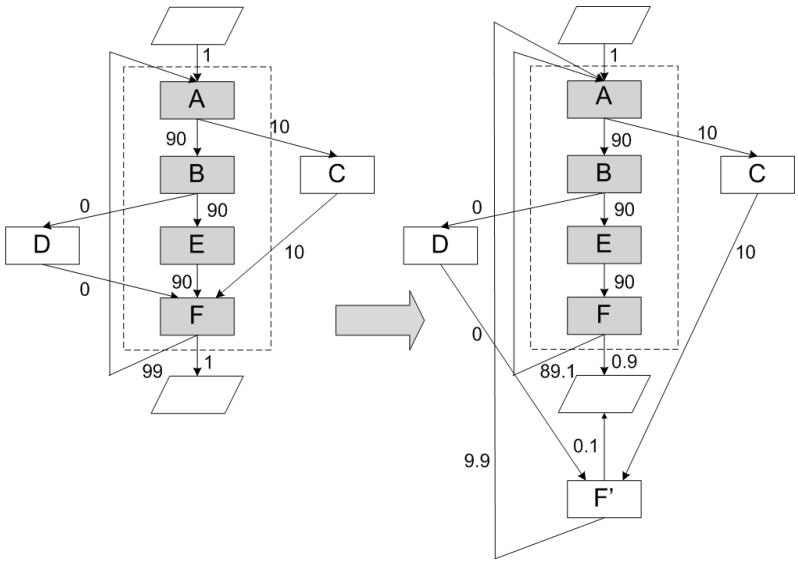

Рис. 3. Пример выделения суперблока

Использование команд предвыборки при обработке массивов в цикле повышает эффективность использования кэша процессора во время последовательной загрузки данных.

Процессоры архитектуры ARM серии Cortex-A9 имеют встроенный автоматический механизм предвыборки данных, который загружает данные в кэш, учитывая промахи кэша, массив загружается в кэш после нескольких итераций и промахов кэша. Такое поведение неоптимально и может быть исправлено с помощью команды предвыборки “PLD”, которая указывает процессору, что вскоре будут использованы данные, на которые указывает команда, так что их желательно загрузить в кэш, если их там еще нет.

Если количество итераций цикла не является константой и вычисляется во время выполнения программы, то оно определяется из собранного профиля выполнения программы, если он доступен.

Если же и профиль программы недоступен, то количество итераций цикла оценивается на основе вероятностной оценки ветвления программы. Подход заключается в оценке количества переходов по каждому ребру на основе эвристических признаков и статистических данных. После применения эвристических оценок каждому ребру графа потока управления соответствует вероятность перехода по этому ребру. Используя эту информацию, можно получить ожидаемую частоту выполнения каждого базового блока.

Предвыборка данных должна осуществляться заранее, до их непосредственного использования. На процессоре ARM Cortex-A9 требуется выполнение около 200 тактов после команды предвыборки, чтобы данные были

загружены в кэш. Тогда момент вставки инструкции упреждающей загрузки можно рассчитать как отношение задержки загрузки данных в кэш после выполнения команды предвыборки к количеству команд в цикле. Задержка может составлять довольно большое количество машинных циклов.

Введем обозначения:

T_{loop} - количество машинных циклов на одну итерацию без учета команды предвыборки, допуская, что необходимые данные находятся в кэше.

t_{use} - количество машинных циклов от начала итерации до первого использования данных в цикле.

t_{pref} - количество машинных циклов на выполнение команды предвыборки.

T_{pref} - задержка загрузки данных в кэш после выполнения команды предвыборки

Первое использование i-го блока данных происходит на такте $t_{use} + i*T_{loop}$ без предвыборки и на такте $t_{use} + i *(T_{loop} + t_{pref})$ с предвыборкой.

Таким образом, чтобы данные находились в кэше к моменту использования, необходимо выполнить команду предвыборки на $t_{pref} + T_{pref}$ тактов ранее, или $\left\lfloor \frac{T_{pref}}{T_{loop}+t_{pref}} \right\rfloor$ итераций ранее и $T_{pref} \bmod (T_{loop} + t_{pref})$ тактов ранее.

Для того, чтобы команды предвыборки не выполнялись слишком часто и не указывали на участки памяти, которые уже загружены в кэш, используется развертывание циклов. Цикл развертывается столько раз, чтобы загружаемые данные за один проход развернутого цикла полностью заполняли одну строку кэша. Например, если размер строки кэша 32 байта (как на процессоре ARM Cortex-A9), а размер загружаемых каждую итерацию данных равен 4 байта, то цикл стоит развернуть 8 раз (32 / 4), и вставить команду предвыборки лишь в первую итерацию.

Тестирование на наборе тестов SPEC CPU 2000 показало, что прирост производительности составляет ~0.9%. На тестах SQLite, Expedite, Cray и Coremark прирост производительности составил 0,5 до 5 %, средний прирост составляет ~2.5%

Модификация алгоритма распределения регистров

Архитектура ARM поддерживает команды, осуществляющие множественную загрузку/сохранение по последовательным адресам - LDM/STM. Инфраструктура LLVM учитывает эту особенность архитектуры в оптимизирующем проходе "ARM load / store optimization pass", который осуществляет свертку последовательных операций (LDR/STR) в одну или несколько команд множественной загрузки/сохранения. Копирование структур в LLVM осуществляется посредством вызова функции memcpy, вызов которой в целях оптимизации заменяется серией команд, осуществляющих загрузку/сохранение. Но алгоритм распределения регистров не учитывает возможность такой оптимизации, поэтому регистры распределяются не в порядке строгого возрастания номеров. Алгоритм распределения регистров был модифицирован таким образом, чтобы обеспечить выбор следующего

свободного регистра с учетом его номера, обеспечивая последовательное возрастание номера используемого регистра, что повысило качество работы оптимизации "ARM load / store optimization pass" и привело к росту быстродействия генерируемого кода. На одну команду LDR/STR с 32 битным операндом процессор тратит 1 такт, на команду LDM/STM: $latency_m(x) = max(\frac{x}{2}, 2)$, где $x -$ число регистров. Пусть сгенерировано N последовательных команд загрузки/сохранения, тогда выигрыш от генерации команд множественной загрузки/сохранения оценивается по следующей формуле:

$$Profit = \frac{latency_m(N)}{N*2}, \quad (6)$$

Видно, что свертка имеет смысл при x>2.

В главе 4 рассматриваются проблемы связанные переносимостью биткода LLVM. Поскольку учет особенностей архитектуры в языке Си производится на ранних этапах компиляции, получаемое промежуточное представление оказывается машинно-зависимым. Основные причины, препятствующие компиляции биткода на архитектуре, отличной от той на которой он был получен, – это учет размеров типов данных и выравнивание, и различные соглашения о вызовах. Таким образом, можно говорить о портировании с потерями производительности между архитектурами со схожими размерами и выравниванием данных, такими как x86 и ARM.

В том числе, были разработаны соглашения о представлении типов данных на различных архитектурах, специальная версия оператора sizeof, работающая во время второй фазы компиляции, поддержка различных соглашений о вызовах функций.

В разделе 4.1 описываются выявленные проблемы переносимости биткода LLVM между архитектурами x86 и ARM:

- Несовместимость ABI рассматриваемых архитектур
  - Различный размер типов данных
  - Различное выравнивание элементов структур
- Различный результат работы оператора «sizeof», при компиляции для различных архитектур
- Несовместимый встраиваемый ассемблер

В разделе 4.2 рассматриваются методы решения проблем, связанных с переносимостью биткода LLVM. Соглашения ABI различаются между архитектурами x86 и ARM. Переносимости промежуточного представления LLVM препятствуют различия размеров типов данных и выравниваний элементов структур, согласно ABI архитектуры.

Из всех фундаментальных типов данных на архитектурах x86 и ARM размер отличается только у типа чисел с плавающей запятой двойной точности «long double».

Для решения данной проблемы в Clang было разработано переносимое ABI на основе архитектуры x86 (gnuportable), в котором 80-битные числа с

плавающей запятой заменяются на 64-битные, выравнивание типов устанавливается таким образом, чтобы не происходила генерация фиктивных полей необходимых для выравнивания. Из-за уменьшения размера чисел с плавающей запятой теряется точность вычислений в программе, однако она становится переносимой.

Рассмотрим следующую структуру данных, описанную на языке Си:

```
struct TrickyStruct {
  int a;
  double b;
  int c;
};
```

Данная структура данных будет расположена в памяти на архитектурах x86 и ARM по-разному – так, как показано на рисункке 4. Цветом показано выравнивание для элемента TrickyStruct::b.

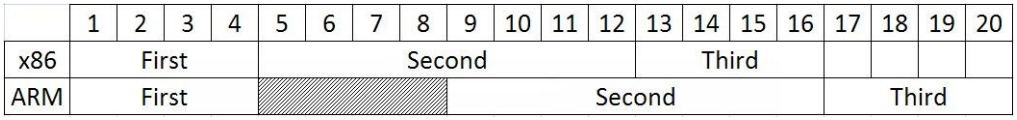

Рис.4. Расположение структуры TrickyStruct в памяти на x86 и ARM

При вызове системной функции, например fstat() или printf(), ожидается, что передаваемые структуры будут выровнены согласно ABI архитектуры. Поэтому для обеспечения переносимости промежуточного представления было решено выравнивать структуры данных согласно ABI до вызова внешней функции и выравнивать возвращаемые результаты так, как они были выровнены в программе.

Стандартный способ для решения данной проблемы состоит в использовании библиотеки libffi, которая обеспечивает прозрачную адаптацию вызовов функций для необходимого ABI.

Использование библиотеки libffi возможно и из исходного кода на Си/Си++, но ее использование довольно громоздко и неудобно для разработчика. Был реализован компиляторный проход, создающий функцию-обертку для каждого вызова внешней функции, которая выравнивает элементы передаваемых и возвращаемых структур нужным образом до и после вызова внешней функции. Генерация обертки происходит полностью автоматически.

Существует проблема с использованием libffi в случае, если передаваемая структура содержит указатель на другую структуру (например, связный список, дерево и т. п.). В вызываемой функции может происходить обращение к данным по этому указателю, и ожидается, что они тоже будут выровнены согласно ABI системы, которое может отличаться от ABI используемого программой.

Другая проблема переносимости – использование оператора «sizeof», результат которого вычисляется во время компиляции и используется в биткоде как константное выражение. Для обеспечения переносимости необходима

версия оператора, вычисляемая во время генерации машинного кода для целевой платформы.

Была реализована поддержка специальной функции «@llvm.size.of». Вызов данной функции в некоторых случаях может быть использован вместо оператора «sizeof». Данная функция заменяется константой во время генерации машинного кода для целевой архитектуры. Для использования вручную на уровне исходного кода был реализован оператор «__sizeof_portable».

Использование переносимой версии оператора накладывает ограничения на код. Например, оператор нельзя использовать для:

1. Расчета размера структур данных содержащих поля, зависимые от результата оператора «sizeof»
2. Вычисления длины массивов, размер которых зависит от результата оператора «sizeof»
3. Инстанцирования шаблонов C++

В языках Си/Си++ есть возможность использовать ассемблерные вставки. Использование встраиваемого ассемблера препятствует переносимости программы между различными процессорными архитектурами. Для обеспечения переносимости использование встраиваемых ассемблерных вставок на данный момент запрещено.

В результате использования описанных выше приемов, удалось получить переносимый биткод LLVM, для приложения SQLite, а также некоторых тестов из набора Aburto. Полученный биткод корректно компилируется в машинный код для архитектур ARM и x86.

Замедление работы тестируемых приложений составило 5-10% (максимум в 10% был, достигнут на приложении SQLite). Замедление обусловлено дополнительными накладными расходами из-за использования функций-оберток над вызовами внешних функций.

В главе 5 описываются методы, обеспечивающие запутывание ключевых участков кода. В настоящее время актуальна задача защиты программ, как от статического, так и от динамического анализа кода. Доступность качественных средств анализа кода и большой выбор подключаемых модулей, в автоматическом режиме обходящих многие приемы противодействия анализу, понижают планку требований к квалификации аналитика, что ведет к повышению требований к защите программ. Необходимо использовать либо методы противодействия анализу неизвестные широкому кругу лиц, либо использовать трудоемкие для анализа преобразования. Оптимальным выбором, позволяющим реализовать стойкие варианты запутывания программ, является создание обфусцирующего компилятора на базе одной из существующих компиляторных инфраструктур. С одной стороны, это позволит производить запутывание программы, имея полную информацию о ней на всех этапах компиляции, а с другой, позволит сосредоточиться на разработке защиты, а не на создании требуемой инфраструктуры.

В разделе 5.1 рассматриваются критерии эффективности методов обфускации. Часто используются оценки, основанные на метриках размера и

семантической сложности программ, применяемые при разработке ПО. Такие метрики носят эвристический характер, и дают оценку того, как изменяется сложность программы после запутывания для понимания ее человеком. В работе используются метрики: размера процедуры, сложности циклической структуры, сложности потока данных.

В разделе 5.2 описываются предлагаемые методы запутывания кода. Все разработанные преобразования представляют собой отдельные компиляторные проходы, запускаемые поочередно, после окончания работы оптимизирующих проходов. Преобразования производятся во время обработки промежуточного представления LLVM на машинно-независимом уровне, что, с одной стороны, позволяет получать запутанное промежуточное представление, которое в дальнейшем можно преобразовать в код на языке Си с помощью стандартных инструментов LLVM. С другой стороны, такой подход обеспечивает поддержку нескольких архитектур при условии совпадения порядка байтов и минимального различия в ABI.

Перенос локальных переменных в глобальную область видимости с последующим их использованием разных функциях производится с целью затруднить точный анализ потоков данных в программе.

В общем случае нельзя изменять значения переменных, вынесенных из других функций в произвольном месте программы, так как это может привести к неправильному выполнению компилируемой программы. Поэтому строится граф вызовов для всех функций в модуле, затем для каждой функции вычисляется множество переменных, модификация которых не нарушит работоспособность программы. Такими переменными будут переменные, вынесенные из функций, расположенных на разных путях в дереве вызовов. После формирования множеств подходящих переменных осуществляется добавление мусорного кода, использующего для вычислений «безопасные» переменные. Также найденные переменные используются в предикатах. Функции, передаваемые по адресу в другие функции, не обрабатываются, так как они могут использоваться в многопоточном коде, и обращение к одной глобальной переменной может вызвать сбой в работе программы.

При восстановлении алгоритма работы программы используется построение слайсов программы. Выполняется отбор тех операторов программы, выполнение которых влияет на выходные данные или на выполнение которых повлияли входные данные. Во время статического анализа для переменных, расположенных в глобальной области памяти, требуется проводить межпроцедурный анализ.

Доказано **утверждение 3:** Сложность построения слайсов программы увеличивается как отношение сложности проведения межпроцедурного слайсинга к сложности проведения внутрипроцедурного.

Во время статического анализа строковые константы, хранящиеся в открытом виде, могут дать аналитику дополнительную информацию о функционировании программы. **Шифрование строк** выполняется следующим образом: вначале все константные строки, кроме тех, что содержатся в

агрегатных типах (массивы, контейнеры из стандартной библиотеки), шифруются, в модуль добавляются шифрующая и дешифрующая функции. Перед каждым использованием той или иной строки вставляется вызов функции дешифратора, а после – шифрующей функции. Это справедливо для строк, для которых не выполняются операции с указателями. Если же такие операции имеют место, то для корректной работы запутывающего алгоритма необходим анализ указателей. В таких случаях обратного шифрования строки не производится. Шифрование строк после использования требуется для того, что бы во время работы программы все строки не находились в памяти расшифрованными. Шифрование строк производится с помощью операции XOR со случайным ключом.

Фиктивный цикл – цикл, в котором никогда не происходит более одной итерации. В коде запутываемой программы происходит поиск участков кода, по структуре напоминающих одну итерацию цикла. Далее в начало участка или в его конец (в зависимости от разновидности вставляемого цикла) вставляется базовый блок с условным переходом в противоположный конец выделенного участка кода. Условный переход содержит в себе непрозрачный предикат, который и маскирует лишь одно исполнение цикла. В качестве подходящего участка кода рассматривается участок с одним входом и выходом.

Маскирующее преобразование «диспетчер» применяется для сокрытия графа потока управления программы от средст статического анализа. Данное преобразование состоит в том, что базовым блокам присваиваются номера. В начало функции вставляют блок «диспетчер» - аналог switch в языке C. В конец каждого блока дописывается код, устанавливающий номер следующего блока для выполнения и передающий управление на блок-диспетчер. Диспетчер же на основе переданной ему информации принимает решение, куда дальше передать управление.

Для каждого блока делается до 5 копий, которые так же добавляются в диспетчер. Помимо этого, производится усреднение размера базовых блоков. Для сокрытия переменной диспетчеризации предпринято следующее: Значение переменной диспетчеризации вычисляется по формуле I = X1 XOR Z; а следующее значение Z по формуле $Z_{след}$ = X2 XOR $Z_{текущее}$; Z, X1 и X2 выбираются случайным образом для блока, предшествующего диспетчеру, и X2 генерируется случайным образом для каждого блока исходной функции во время его обработки. В каждом блоке выбирается одна переменная подходящего типа, с которой посредством операции "исключающего ИЛИ" (XOR) происходит сцепление переменной диспетчеризации. Такое преобразование затруднит автоматическое выделение переменной диспетчеризации, так как в ее вычисление будут вовлечены живые переменные, вычисляемые в программе.

Классический подход к **переплетению функций** обладает малой стойкостью. Он предполагает объединение сигнатур функций и наличие параметра, по которому происходит диспетчеризация. Восстановить исходный код переплетенных таким образом функций не составляет особого труда.

Предложена модификация упомянутого алгоритма: помимо диспетчеризующего условия, переплетаемые функции должны иметь точки пересечения потоков управления и потоков данных, чтобы применение алгоритма обратного слайсинга не позволяло найти единственную точку, в которой производится выбор рабочей функции. Ключевое изменение оригинального алгоритма состоит в следующем:

В новой функции, полученной на основе переплетения двух функций, произвольно выбираются два базовых блока (один блок из первой функции, второй блок из второй), для которых производится преобразование зацепления дуг. В генерируемом общем базовом блоке производятся вычисления с глобальными переменными. Для затруднения анализа потоков данных эти переменные используются для вычислений в других функциях модуля. Таким образом, у двух переплетенных функций всегда будут общие вычисления. Результат вычислений используется в качестве возвращаемого значения, а также сохраняется в глобальную переменную, что не позволит исключить внедренные вычисления как мертвый код.

Сокрытие вызов функций применяется для маскировки вызовов функций, поскольку знание имени вызываемой функции облегчает восстановление алгоритма работы программы. Для маскируемого вызова создается функция-переходник, внутри которой содержится несколько вызовов функций. Внутри переходника вызов нужной функции диспетчеризуется по значению трудного предиката. Реализовано два варианта преобразования: только для вызовов внешних функций и для вызовов всех функций.

Для каждого вызова функции производится его замена на вызов функции-переходника. Чтобы избежать чрезмерной вложенности вызовов, переходники на переходники не создаются. Для выбора функций, которые будут размещены внутри переходника, была введена мера "близости функций". Значение меры - коэффициент Жаккарда для множеств типов аргументов двух функций:

$$\Gamma(f_1, f_2) = \frac{|TypeArgs(f_1) \cap TypeArgs(f_2)|}{|TypeArgs(f_1) \cup TypeArgs(f_2)|} \quad (7)$$

Половина функций в переходнике выбираются, как самые "похожие" по введенной мере, и другая половина - "непохожие".

Аргументы, передаваемые в функцию-переходник, запутываются с помощью битовой операции "исключающее ИЛИ" (XOR). Ко всем операциям применяется операция XOR, обозначим результат операции за S. Затем к каждому аргументу применяется операция XOR с S, и в таком виде аргумент передается в функцию. Также для распутывания передается само значение S. Внутри функции-переходника происходит распутывание аргументов. Затем вычисляется непрозрачный предикат, по результату которого происходит диспетчеризация вызова функций. Диспетчеризация вызовов функций производится с помощью большого switch блока. Каждое значение в нем сгенерировано случайным образом и соответствует какой-либо функции.

Последний аргумент функции-переходника служит для определения функции, которая будет вызвана. Этот аргумент передается в непрозрачный предикат. Так, как значение предиката известно на этапе компиляции, то можно подобрать такое значение аргумента, которое соответствует нужной функции.

Генерация несводимых участков в графе потока управления применяется для затруднения работы автоматических декомпиляторов**.** Колберг[1] описывает алгоритм, который приводит граф потока управления к несводимому. Для каждого цикла добавляется «фиктивное» ребро из заголовка цикла в его тело. Добавление такого ребра осуществляется с помощью вставки непрозрачных предикатов. Предложена модификация упомянутого алгоритма: для всех циклов функции добавляются «фиктивные» ребра из одного цикла в другой.

Недостаток такой трансформации состоит в том, что она эффективно запутывает только код функций, содержащих несколько циклов. Поэтому дополнительно производится следующая трансформация: для множества блоков функции выбирается N блоков и между ними случайно добавляются ребра. Фиктивные переходы защищаются непрозрачными предикатами.

Часто в коде в явном виде встречаются константы, характерные для определенных алгоритмов, например константа 0x67452301 для MD5. Поиск констант позволяет определить используемый алгоритм, что упрощает анализ программы. Для противодействия предложен алгоритм **разбиения констант**. Разбиваются только константы большие единицы. Для разбиения случайным образом выбирается число меньшее исходного, которое будет выступать в качестве первого слагаемого, второе слагаемое получается автоматически.

Клонирование функций - для каждого использования функции внутри программного модуля, производится создание своего экземпляра вызываемой функции. Такое преобразование увеличивает размер кода программы и время, требуемое для его автоматического анализа. Также, будучи применено совместно с другими запутывающими преобразованиями, зависящими от генератора случайных чисел, функции утратят полную идентичность, что повысит сложность анализа, так как потребуется проанализировать каждую копию.

В качестве вспомогательного преобразования используется **формирование непрозрачных предикатов.**

Определение: Предикатом является базовый блок или несколько базовых блоков, имеющих один общий терминальный базовый блок. Терминальный базовый блок предиката заканчивается инструкцией условного перехода, которая всегда передает управление только по одной ветке. Причем, основываясь на информации, доступной на этапе компиляции, известно, по какому пути произойдет переход.

---

[1] Christian Collberg , Clark Thomborson , Douglas Low. A Taxonomy of Obfuscating Transformations. Technical report 148, Department of Computer Science, The University of Auckland, New Zeland, July 1997.

Разработано API для автоматической генерации предикатов. В запутывающих преобразованиях используются три типа предикатов:

1. Выражения, которые могут быть как истинны, так и ложны в зависимости от выбранных параметров, например проверка истинности диофантова уравнения $x^2 - n * y^2 = 1$. Если параметр n не является точным квадратом, то это уравнение Пелля. При вставке этого предиката случайным образом выбирается, будет ли он всегда иметь истинное значение либо ложное.

2. Выражения, которые всегда истинны. Например, уравнение:

$(x^3 - x)\ mod\ 3 = 0$. Значение переменной x для вычисления значения предиката выбирается случайным образом среди целочисленных глобальных переменных. Если таких глобальных переменных нет, то для вычисления предиката используется случайная целочисленная константа.

3. Выражения, которые всегда ложны. Например, целочисленное уравнение: $7 * y^2 - 1 = x^2$. Это выражение всегда ложно. Значения переменных x и y выбираются также, как и в предыдущем случае.

В разделе 5.3 производится оценка понижения быстродействия и увеличения потребляемой памяти, приводятся результаты применения средств статического и динамического анализа к программе, запутанной с помощью предлагаемых методов. В практических целях был произведен замер замедления и потребления памяти на тестах из пакета OpenSSL 1.0.1. и посчитаны метрики сложности.

Таблица 1. Параметры замедления и увеличения потребления памяти

Метод	Замедление программы	Увеличение потребления памяти
Клонирование функций	1,20	1,10
Переплетение функций	1,20	1,10
Шифрование строк	5,00	1,05
Вставка фиктивных циклов	1,20	1,10
Разбиение констант	1,20	1,05
Сокрытие вывозов внешних функций	5,00	1,50
Сокрытие вывозов всех функций	8,00	1,70
Диспетчер	5,50	2,50
Генерация несводимых участков в графе потока управления	1,20	1,05
Перенос локальных переменных в глобальную область видимости	1,20	1,05

Таблица 2. Метрики сложности

Метод	Размер кода	Сложность циклической структуры	Сложность потока данных	Уровень вложенности условий	Комплексная метрика
До запутывания	383	93	19	1903	-
Перенос локальных переменных в глобальную область видимости	383	127	19	1903	300
Разбиение констант	464	471	29	1903	310
Шифрование строк	453	311	13	2901	146
Вставка фиктивных циклов	819	289	39	2018	358
Диспетчер	10205	699	398	3490	817
Переплетение функций	416	297	20	1820	300
Сокрытие вывозов внешних функций	4521	481	87	2105	601
Сокрытие вывозов всех функций	4903	607	93	2410	617
Генерация несводимых участков в графе потока управления	987	640	40	616	313
Клонирование функций	608	301	26	2019	491

Для работы алгоритмов статического слайсинга требуется построить граф зависимостей системы (SDG- system dependence graph, система в данном контексте это программа полностью), сложность его построения оценивается как $O(TCS*CS^2+NP*V^2)$. NP- количество процедур в программе, V- максимальное количество выражений и предикатов в процедуре, TCS-общее количество вызов процедур в программе, CS- максимальное количество вызова процедур в одной из функций программы.

Доказано **утверждение 4:** сложность статического анализа программы после запутывания возрастает полиномиально от количества добавляемых при запутывании выражений и вызовов процедур.

Совместное применение нескольких опций позволит увеличить сложность пропорционально произведению увеличения сложностей каждого преобразования в отдельности. Для примерной оценки сложности анализа был проведен эксперимент. К программе Sqlite были применены преобразования: переплетение функций, перенос локальных переменных в глобальную область видимости, преобразование «Диспетчер», сокрытие вызовов функций. Размер кода приложения увеличился с 2.9 МБ до 15 Мб. Потребление памяти дизассемблером Ida Pro возросло в ~10 раз, время анализа по сравнению с оригинальным кодом возросло примерно в 10 раз. Кроме того, некоторые версии Ida Pro оказались неспособны закончить анализ, поскольку во время работы возникает исключение в одной из библиотек, и программа аварийно

завершает работу. Также было произведено исследование с помощью инструмента комбинированного анализа Trex. Полученные результаты свидетельствуют о том, что обеспечиваемый уровень защиты сравним с уровнем, обеспечиваемым коммерческими разработками.

Заключение содержит выводы и направления дальнейших исследований по проблемам, рассмотренным в работе.

В диссертации:

1. Проведен анализ существующих методов распространения ПО;
2. Предложен метод двухфазной компиляции программ, на языках Си/Си++, обеспечивающий распространение программ в промежуточном представлении LLVM;
3. Предложены методы оптимизации программ с учетом информации о профиле выполнения программы, методы машинно-зависимой оптимизации для платформы ARM, методы запутывания программ;
4. Исследованы проблемы, возникающие при переносе промежуточного представления LLVM между архитектурами x86 и ARM;
5. Предложены методы, позволяющие обеспечить переносимость между указанными архитектурами;
6. Предложены методы запутывания программ, обеспечивающие повышение устойчивости программы к методам обратной инженерии;

На основе предложенных методов разработана и реализована компиляторная инфраструктура, обеспечивающая возможность распространения Си/Си++ приложений, в промежуточном представлении LLVM для Linux-платформ на базе архитектур x86 и ARM с сохранением производительности. Перед внедрением произведена экспериментальная проверка устойчивости предлагаемых методов запутывания с помощью среды комбинированного анализа Trex. Разработанный метод распространения приложений реализован и внедрен в рамках мобильной промышленной платформы Tizen.

Среди направлений дальнейших работ по данной тематике можно выделить наиболее важные:

1. Разработка методов машинно-ориентированной оптимизации, позволяющих осуществлять эффективный перенос приложений для широкого круга платформ.
2. Разработка методов запутывания программ, позволяющих повысить устойчивость к динамическому анализу.
3. Разработка методов, обеспечивающих адаптивную оптимизацию приложений, написанных на языках Си/Си++, во время динамической компиляции.

Основные результаты:

1. Разработан метод двухфазной компиляции Си/Си++-приложений, позволяющий разделить фазу генерации машинно-независимого внутреннего представления и фазу генерации кода для целевой платформы, на которой обеспечивается автоматический учет особенностей конкретной аппаратуры и окружения.
2. Разработаны методы машинно-независимой оптимизации, учитывающие данные профиля программы: открытая вставка функций, клонирование базовых блоков, вынос редко исполняемого кода в функции, спекулятивная девиртуализация функций языка Си++, формирование суперблоков. Разработаны методы машинно-зависимой оптимизации, учитывающие специфику конкретной платформы: автоматическая генерация команд предвыборки, модификация алгоритма распределения регистров.
3. Разработаны методы запутывания программного кода: перенос локальных переменных в глобальную область видимости, шифрование строк, вставка фиктивных циклов, преобразование «диспетчер», переплетение функций, сокрытие вызовов функций, генерация несводимых участков в графе потока управления, клонирование функций, разбиение констант.
4. На основе предложенных методов разработана и реализована компиляторная инфраструктура, обеспечивающая возможность распространения Си/Си++ приложений в промежуточном представлении LLVM для Linux-платформ на базе архитектур x86 и ARM с сохранением производительности приложений.

Список опубликованных статей по теме диссертации:

1. А. И. Аветисян, К. Ю. Долгорукова, Ш. Ф. Курмангалеев. Динамическое профилирование программы для системы LLVM. // Труды Института системного программирования РАН, том 21, 2011, с. 71-82.

2. Ш. Ф. Курмангалеев, В. П. Корчагин, Р. А. Матевосян. Описание подхода к разработке обфусцирующего компилятора. // Труды Института системного программирования РАН, том 23, 2012, с. 67-76.

3. Ш. Ф. Курмангалеев, В. П. Корчагин, В. В. Савченко, С. С. Саргсян. Построение обфусцирующего компилятора на основе инфраструктуры LLVM. // Труды Института системного программирования РАН, том 23, 2012, с. 77-92.

4. Курмангалеев Ш. Ф. Методы оптимизации C/C++- приложений распространяемых в биткоде LLVM с учетом специфики оборудования. // Труды Института системного программирования РАН, том 24, 2013, с. 127-144.