Построение спецификаций программных интерфейсов в открытой системе покомпонентной верификации ядра Linux[1]

Новиков Е.М.
novikov@ispras.ru

Аннотация. Одним из наиболее перспективных методов поиска ошибок в программах в настоящее время является статическая верификация. Для успешного применения существующих инструментов к ядру операционной системы Linux приходится проводить верификацию покомпонентно. При этом инструментам необходимо предоставлять модель окружения, отражающую реальное окружение компонентов достаточно точно. Разработка полной модели окружения для компонентов ядра Linux является очень трудоемкой задачей, поскольку программных интерфейсов в ядре очень много и они не являются стабильными. В статье предлагается новый подход к построению спецификаций программных интерфейсов, который позволяет достаточно эффективно применять инструменты статической верификации для проверки выполнения правил использования программных интерфейсов в условиях неполноты модели окружения.

Ключевые слова: ядро Linux; компонент ядра; драйвер устройств; правило использования программных интерфейсов; спецификация; модель окружения; статическая верификация; аспектно-ориентированное программирование; язык программирования Си.

1. Введение

На основе изменений, сделанных в стабильных версиях ядра операционной системы (ОС) Linux за год разработки, был проведен анализ ошибок в компонентах ядра [1]. Результаты данного анализа показали, что среди ошибок, которые не связаны с нарушениями спецификаций устройств, сетевых протоколов, алгоритмов выделения памяти и т.п., около половины всех ошибок в компонентах происходят вследствие нарушения правил

[1] Работа поддержана ФЦП "Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007-2013 годы" (контракт N 11.519.11.4006).

использования программных интерфейсов. Для поиска подобных ошибок в настоящее время преимущественно применяются такие подходы, как экспертиза кода [2-4] и динамическое тестирование [5-8]. При использовании данных методов проверки осуществляются с учетом достаточно точных знаний о реальном окружении компонентов ядра ОС Linux, что позволяет выявлять ошибки с высоким уровнем достоверности.

Иначе дело обстоит с различными подходами статического анализа кода. Данные подходы продемонстрировали существенный прогресс в области поиска ошибок за несколько последних десятилетий. В настоящее время на практике, в том числе при поиске ошибок в ядре ОС Linux, очень широко применяются инструменты легковесного статического анализа кода [9-13]. Поскольку данные инструменты в первую очередь нацелены на небольшие время работы и процент ложных сообщений об ошибках, они либо не проводят межпроцедурный анализ, либо делают это достаточно ограниченным образом. Поэтому, даже несмотря на то, что инструментам при анализе доступен весь исходный код ядра ОС Linux, их знания об окружении компонентов ядра являются достаточно поверхностными. Последнее приводит к тому, что инструменты принципиально пропускают ошибки и делают ложные срабатывания.

Наряду с методами легковесного статического анализа активно развивались подходы к статической верификации, нацеленные на обнаружение всех возможных ошибок искомого вида [14-22]. Ввиду текущих ограничений инструментов статической верификации на размер и сложность анализируемого кода, при их применении к ядру ОС Linux приходится ставить задачу верификации покомпонентно. При этом для того, чтобы число ложных срабатываний не было очень большим, для верифицируемого компонента необходимо достаточно точно моделировать его окружение. Модель окружения должна строиться таким образом, чтобы имитировать реальное взаимодействие компонента с так называемой *сердцевиной ядра*, которое заключается в следующем:

- программные интерфейсы компонентов ядра (*функции-обработчики*) регистрируются в сердцевине ядра при вызове стандартной функции инициализации компонентов, а затем вызываются из сердцевины в ответ на системные вызовы со стороны пользовательских приложений и на прерывания со стороны соответствующей аппаратуры;
- программные интерфейсы сердцевины ядра ОС Linux предоставляются компонентам посредством заголовочных файлов аналогично стандартным Си-библиотекам.

Для того, чтобы применить различные инструменты статической верификации для проверки правил использования программных интерфейсов в исходном коде компонентов ядра ОС Linux была разработана открытая система верификации Linux Driver Verification (LDV) [23-25]. Перед проведением непосредственно верификации LDV готовит исходный код компонентов.

Сначала на основе файлов сборки ядра извлекается информация о составе и опциях компиляции и компоновки компонентов. Затем для каждого компонента в отдельности генерируется модель окружения на основе конфигурации, которая описывает несколько типов компонентов и предоставляет шаблоны для остальных. На заключительном этапе подготовки осуществляется постановка задачи верификации для проверки некоторого правила использования программных интерфейсов сердцевины ядра.

Правила использования программных интерфейсов сердцевины ядра в открытой системе верификации LDV задаются с помощью спецификаций. Спецификации включают в себя модельное состояние (набор глобальных переменных) и модельные реализации программных интерфейсов (набор функций). Модельное состояние представляет собой отображение реального состояния ядра ОС Linux. Модельные реализации программных интерфейсов выполняют изменения модельного состояния в соответствии с семантикой моделируемых интерфейсов, а также проверки, которые требуется для правил. При необходимости в спецификации задается привязка точек использования программных интерфейсов сердцевины ядра в компонентах к модельным реализациям интерфейсов с помощью конструкций аспектно-ориентированного программирования [26, 27].

Практическое применение открытой системы верификации LDV для поиска ошибок нескольких видов в коде компонентов ядра ОС Linux с помощью инструментов статической верификации BLAST и CPAchecker позволило обнаружить несколько десятков критических ошибок. Также в ходе экспериментов было выявлено, что из-за неполноты модели окружения (с точки зрения вызовов функций-обработчиков компонентов и программных интерфейсов сердцевины ядра) у инструментов статической верификации достаточно часто происходили ложные срабатывания. Анализ ложных срабатываний инструментов статической верификации является по сути ручным. Данный анализ требует значительных трудозатрат по времени и привлечения дорогостоящих экспертов ядра ОС Linux. Также стоит заметить, что в случае ложных срабатываний инструменты статической верификации не могут автоматически ни обнаружить в коде другие ошибки искомого вида, ни доказать, что их нет.

До сих пор основные усилия для решения данной проблемы были направлены на улучшение полноты модели окружения компонентов. Например, для применения системы верификации SDV [21] разработчики драйверов ОС Microsoft Windows должны предварительно вручную проаннотировать их функции-обработчики. Поскольку программные интерфейсы ядра данной ОС являются стабильными, разработчики SDV сумели описать достаточно хорошую модель для них. Разработка полной модели окружения для компонентов ядра ОС Linux является очень трудоемким процессом ввиду существенно большего многообразия программных интерфейсов ядра, а также их непрерывного изменения по мере развития ядра. Существующие системы верификации компонентов ядра ОС Linux не предложили

автоматизированных решений. В системе верификации Avinux требуется вручную задавать последовательности вызовов функций-обработчиков [22]. В системе верификации DDVerify необходимо полностью описывать модель окружения [20]. Разработчики данной системы построили модель для нескольких типов драйверов устройств, но большие трудозатраты не позволили им развивать данное направление.

Для того чтобы эффективно применять инструменты статической верификации для проверки выполнения правил использования программных интерфейсов в исходном коде компонентов ядра ОС Linux в статье предложен новый метод построения спецификаций программных интерфейсов, который описан в разделе 2. Раздел 3 посвящен инструментарию, реализующему предложенный подход. В разделе 4 представлены результаты практического применения разработанного инструментария. На основании данных результатов делаются выводы о сильных и слабых сторонах предложенного подхода, а также об области его применения. В заключении подводятся итоги работы и рассматриваются направления дальнейшего развития.

2. Метод построения спецификаций программных интерфейсов в условиях неполноты модели окружения

Рассмотрим два примера, которые демонстрируют, как неполнота модели окружения приводит к ложным срабатываниям у инструмента статической верификации BLAST. Для данных примеров проверялась спецификация правила, которое накладывает следующие ограничения на корректное использование в одном потоке функций захвата (*mutex_lock*) и освобождения (*mutex_unlock*) мьютексов[2]:

- запрещается повторно захватывать уже захваченный мьютекс;
- запрещается повторно освобождать уже освобожденный мьютекс;
- запрещается освобождать незахваченный мьютекс;
- запрещается оставлять захваченные мьютексы после завершения работы компонента.

[2] Мьютекс – это примитив синхронизации, который позволяет обеспечить взаимоисключающий доступ к разделяемым данным.

```
  * struct mmc_host *mmc;
  * sdmmc_request(mmc, mpq);
676   struct realtek_pci_sdmmc *host = mmc_priv(mmc);
677   struct rtsx_pcr *pcr = host->pcr;
687   mutex_lock(&pcr->pcr_mutex);
697   mutex_lock(&host->host_mutex);
```

Рисунок 1. *Код драйвера PCI-Express SD/MMC карт drivers/mmc/host/rtsx_pci_sdmmc.c («*» помечена модель окружения, сгенерированная LDV).*

2.1. Пример ложного срабатывания вследствие неполноты модели вызовов функций-обработчиков

В ядро ОС Linux версии 3.8 был добавлен драйвер PCI-Express SD/MMC карт, который состоит из одного Си-файла *drivers/mmc/host/rtsx_pci_sdmmc.c*[3]. Для данного драйвера система верификации LDV генерирует модель вызовов функций-обработчиков неполным образом. На Рис. 1 наглядным образом продемонстрирован один из допустимых данной моделью путей в коде драйвера (опущены несущественные детали). При вызове функции-обработчика *sdmmc_request* ей в качестве параметра передается **mmc** – указатель на структуру *mmc_host*. В строке 676 с помощью функции *mmc_priv* по указателю **mmc** получается поле **host** данной структуры. В строке 677 получается поле **pcr** структуры *realtek_pci_sdmmc*, на которую указывает **host**. Затем в строках 687 и 697 захватываются мьютексы **pcr_mutex** из структуры *rtsx_pcr*, на которую указывает **pcr**, и **host_mutex** из структуры *realtek_pci_sdmmc*, на которую указывает **host**.

На данном пути у инструмента статической верификации BLAST происходит ложное срабатывание. Причиной этого является то, что инструмент считает, что мьютексы **pcr_mutex** и **host_mutex**, получаемые на основе указателя **mmc**, могут совпадать (правильнее говорить, что данные идентификаторы могут обозначать одну и ту же область памяти, то есть могут совпадать адреса **pcr_mutex** и **host_mutex**), поскольку на момент вызова функции-обработчика *sdmmc_request* память под структуру *mmc_host*, на которую указывает **mmc**, не была выделена и инициализирована. Выделение и инициализация памяти

[3] http://lxr.free-electrons.com/source/drivers/mmc/host/rtsx_pci_sdmmc.c?v=3.8.

происходит в функции-обработчике *rtsx_pci_sdmmc_drv_probe*, которая также вызывается имеющейся моделью окружения, но неупорядоченно с *sdmmc_request*.

```
  * struct spi_device *spi;
  * lis3l02dq_probe(spi);
681   struct lis3l02dq_state *st;
682   struct iio_dev *indio_dev;
684   indio_dev = iio_device_alloc(sizeof *st);
689   st = iio_priv(indio_dev);
690   spi_set_drvdata(spi, indio_dev);
695   mutex_init(&st->buf_lock);
  * lis3l02dq_remove(spi);
787   struct iio_dev *indio_dev = spi_get_drvdata(spi);
793   lis3l02dq_stop_device(indio_dev);
765     mutex_lock(&indio_dev->mlock);
766     ...lis3l02dq_spi_write_reg_8(indio_dev, ...);
 89       struct lis3l02dq_state *st = iio_priv(indio_dev);
 91       mutex_lock(&st->buf_lock);
```

Рисунок 2. *Код драйвера линейного акселерометра drivers/staging/iio/accel/lis3l02dq_core.c («*» помечена модель окружения, сгенерированная LDV; волнистой линией подчеркнуты функции, реализации и модели которых не предоставляются для анализа).*

2.2. Пример ложного срабатывания вследствие неполноты модели программных интерфейсов сердцевины ядра

Для драйвера линейного акселерометра, состоящего из одного Си-файла *drivers/staging/iio/accel/lis3l02dq_core.c*[4], система верификации LDV генерирует полным образом ту часть модели окружения, которая имитирует вызовы функций-обработчиков. На Рис. 2 показано, что модель окружения вызывает функцию-обработчик *lis3l02dq_probe*, передавая ей в качестве параметра **spi** – указатель на структуру *spi_device*. В данной функции **indio_dev** присваивается указатель на структуру *iio_dev*, который возвращается функцией *iio_device_alloc* (строка 684). После этого из

[4] http://lxr.free-electrons.com/source/drivers/staging/iio/accel/lis3l02dq_core.c?v=3.8.

indio_dev с помощью функции *iio_priv* получается поле **st** данной структуры (строка 689). Далее посредством функции *spi_set_drvdata* **indio_dev** связывается с **spi** (строка 690), а затем инициализируется мьютекс **buf_lock** из структуры *lis3l02dq_state*, на которую указывает **st** (строка 695).

При вызове функции-обработчика *lis3l02dq_remove*, которой в качестве параметра передается тот же **spi**, что и *lis3l02dq_probe*, в строке 787 из данного **spi** с помощью функции *spi_get_drvdata* получается указатель на структуру *iio_dev* **indio_dev**. Данный **indio_dev** передается в качестве параметра функции *lis3l02dq_stop_device* (строка 793). В строке 765 захватывается мьютекс **mlock** из структуры *iio_dev*, на которую указывает **indio_dev**. Затем **indio_dev** передается функции *lis3l02dq_spi_write_reg_8* в качестве параметра. В этой функции из **indio_dev** с помощью функции *iio_priv* получается поле структуры *iio_dev* **st** (строка 89) и захватывается мьютекс **buf_lock** из структуры *lis3l02dq_state*, на которую указывает **st** (строка 91).

Ложное срабатывание у инструмента статической верификации BLAST обусловлено тем, что инструмент считает, что мьютексы **mlock** и **buf_lock**, получаемые на основе указателя **spi**, могут совпадать, поскольку при анализе BLAST не предоставляются ни реализации, ни модели для функций сердцевины ядра *iio_device_alloc*, *spi_set_drvdata* и *spi_get_drvdata*. В функции *iio_device_alloc* выделяется память под структуру *iio_dev* и, в частности, инициализируется мьютекс **mlock** из данной структуры. Функции *spi_set_drvdata* и *spi_get_drvdata* соответственно помещают и извлекают указатель на структуру *iio_dev* **indio_dev** из структуры *spi_device*, на которую указывает **spi**.

Аналогичные проблемы происходили в ходе практического применения системы верификации LDV при проверке большинства спецификаций правил использования программных интерфейсов сердцевины ядра ОС Linux, поскольку неполная модель окружения не позволяла различать определенные объекты[5]. Например, это правила, связанные с выделением и освобождением ресурсов, с использованием различных механизмов синхронизации, с инициализацией объектов перед их использованием и т.д.

[5] В данной статье под объектами понимаются не классические объекты из объектно-ориентированного программирования, а те объекты, которые интересны с точки зрения специфицируемых программных интерфейсов. Например, для функций захвата и освобождения мьютексов такими объектами являются мьютексы. Различными считаются объекты, занимающие разные области памяти или, иными словами, представимые различными указателями.

2.3. Подход к построению спецификаций программных интерфейсов

С целью уменьшения числа ложных срабатываний инструментов статической верификации, возникающих при проверке выполнения требований спецификаций для правил использования программных интерфейсов в исходном коде компонентов ядра ОС Linux в условиях неполноты модели окружения, в этой статье предложен новый подход к построению спецификаций программных интерфейсов. В основу данного подхода была положена идея различать объекты с помощью анализа тех выражений, в которых они участвуют. Для этого было предложено определить функцию Ψ, которая должна на вход принимать выражение, в котором может использоваться явным или неявным образом некоторый объект интересный с точки зрения специфицируемых программных интерфейсов. На выходе Ψ должна возвращать информацию о том, какому объекту или объектам соответствует рассматриваемое выражение. Для всех выявленных с помощью данной функции объектов в подходе было предложено строить спецификацию таким образом, чтобы хранить информацию об их текущем состоянии в уникальных переменных модельного состояния, которые должны изменяться и проверяться независимо друг от друга в модельных реализациях программных интерфейсов.

Функция Ψ должна удовлетворять следующим условиям:

1. Для двух выражений, которым соответствует один и тот же объект, Ψ должна возвращать одно и то же значение. Данное условие необходимо для того, чтобы гарантировать надежность метода или, иными словами, обнаружение всех возможных ошибок искомого вида.
2. Для двух выражений, которым соответствуют разные объекты, Ψ должна вернуть разные значения. Это условие говорит о нацеленности данной функции на различение объектов и, соответственно, на уменьшение числа ложных срабатываний.

Легко предложить такую функцию, которая удовлетворяет только первому условию, – это функция Ψ', которая для любого выражения возвращает одно и то же значение. Забегая вперед, можно отметить, что результаты экспериментов с Ψ', представленные в подразделе 4.4, показали большое число ложных срабатываний. Также легко предложить функцию Ψ'', которая удовлетворяет только второму условию, – это функция, которая для любого входа всегда возвращает разные значения.

Функция Ψ, которая удовлетворяет одновременно двум условиям в полной мере, должна учитывать, например, алиасы и арифметику с указателями. Для этого Ψ должна уметь решать задачу, которая отчасти сопоставима с задачей инструментов статической верификации. Это означает, что в условиях неполноты модели окружения реализация функции Ψ неизбежно столкнется с

теми же проблемами, что и статическая верификация (см. подразделы 2.1 и 2.2).

Более простое определение функции Ψ можно предложить только при условии выполнения определенных правил кодирования в проекте. Например, если в проекте не используются операции преобразования типов, затрагивающие рассматриваемые объекты, а также другие операции, которые могут привести к совпадению объектов с разными именами, то в качестве Ψ можно рассмотреть функцию, которая возвращает имена объектов для соответствующих выражений[6]. С помощью такой функций для примера из подраздела 2.1 можно выявить мьютексы **pcr_mutex** и **host_mutex**, для которых в подходе предложено использовать различное модельное состояние, например, переменные **ldv_mutex_pcr_mutex** и **ldv_mutex_host_mutex** соответственно. Для мьютексов **mlock** и **buf_lock** (подраздел 2.2) – **ldv_mutex_mlock** и **ldv_mutex_buf_lock** соответственно. Благодаря этому в дальнейшем при проведении верификации становится возможным избежать ложных срабатываний, которые происходили ранее из-за того, что неполная модель окружения не позволяла различить данные мьютексы.

```
 1 struct A {
 2   int x;
 3 } global;
 4 struct B {
 5   int y;
 6 };
 7 int main()
 8 {
 9   struct B *local1;
10   struct A *local2 = (struct A *)malloc(sizeof(struct A));
11   int *z;
12   /* Преобразование типа для адреса глобальной переменной. */
13   local1 = (struct B *)&global;
14   assert(&global.x == &local1->y); /* Совпадают. */
15   /* Преобразование типа для адресов из кучи. */
16   local1 = (struct B *)local2;
17   assert(&local2->x == &local1->y); /* Совпадают. */
18   /* Использование алиасов. */
19   z = &global.x;
20   assert(&global.x == z); /* Совпадают. */
21   return 0;
22 }
```

Рисунок 3. Пример программы, в которой не выполняются правила кодирования, позволяющие различать объекты по их именам.

[6] Стоит отметить, что при рассматриваемых условиях данная функция полностью отвечает только первому условию, поскольку интересные с точки зрения специфицируемых программных интерфейсов объекты могут иметь одинаковые имена, но при этом быть вложены в разные структуры (см. подраздел 4.3).

В общем случае рассмотренные правила кодирования могут не соблюдаться. На Рис. 3 приведены три примера, для которых при использовании данного подхода к построению спецификаций программных интерфейсов будут различаться объекты с различными именами, **x**, **y** и **z**, притом, что адреса **x** и **y** совпадают с **z** (строки 14, 17 и 20). Подобное происходит при преобразованиях типа для адреса глобальной переменной (строка 13) и для адреса из кучи (строка 16), а также при использовании алиасов (строка 19). Для компонентов ядра ОС Linux рассмотренные правила кодирования выполняются с высокой степенью достоверности, поэтому подход применим для покомпонентной верификации ядра.

3. Инструментарий для построения спецификаций программных интерфейсов

Первоначально предложенный подход к построению спецификаций программных интерфейсов был реализован в инструменте статической верификации BLAST. В данной реализации функции Ψ передавались на вход те выражения, которые являлись фактическими аргументами специфицируемых программных интерфейсов сердцевины ядра ОС Linux (см. примеры передачи мьютексов функции *mutex_lock* в подразделах 2.1 и 2.2). С целью указания информации о данных интерфейсах инструменту BLAST на вход подавалась дополнительная конфигурация. На основе фактических аргументов специфицируемых программных интерфейсов функция Ψ вычисляла имена интересных объектов. В дальнейшем в данной статье результат работы устроенной таким образом функции Ψ будет называться *подписью аргумента* (по аналогии с повседневной жизнью, где подпись практически однозначно идентифицирует человека).

Для создания соответствующих выявленным объектам переменных модельного состояния и модельных реализаций программных интерфейсов, которые изменяют и проверяют данные переменные независимо друг от друга, имена этих переменных и модельных реализаций программных интерфейсов помечались в спецификации с помощью специального суффикса. Для всех уникальных вычисленных подписей аргументов спецификация дублировалась, причем суффикс заменялся на соответствующую подпись аргумента, что гарантировало уникальность переменных модельного состояния и независимость их изменения и проверки для различных объектов.

Применение данной реализации на практике позволило существенно сократить число ложных срабатываний, благодаря чему повысилась эффективность выявления ошибок. При этом первоначальная реализация подхода обладала несколькими существенными недостатками. Во-первых, она не поддерживала привязку точек использования программных интерфейсов сердцевины ядра в компонентах к модельным реализациям интерфейсов с помощью конструкций аспектно-ориентированного программирования, что

требовалось для большинства спецификаций правил. Во-вторых, в реализации были ошибки, из-за чего подписи аргументов не всегда вычислялись корректно. В-третьих, что являлось наиболее существенным, реализация подхода на основе инструмента BLAST не позволяла воспользоваться ей при применении других инструментов статической верификации.

Для преодоления указанных недостатков была разработана новая реализация подхода. В этой реализации было предложено вычислять подписи аргументов специфицируемых программных интерфейсов с помощью запросов по исходному коду компонентов ядра ОС Linux [28]. Для создания уникальных переменных модельного состояния и модельных реализаций программных интерфейсов, которые изменяют и проверяют данные переменные независимо друг от друга, было предложено задавать спецификации в виде шаблонов [29,30].

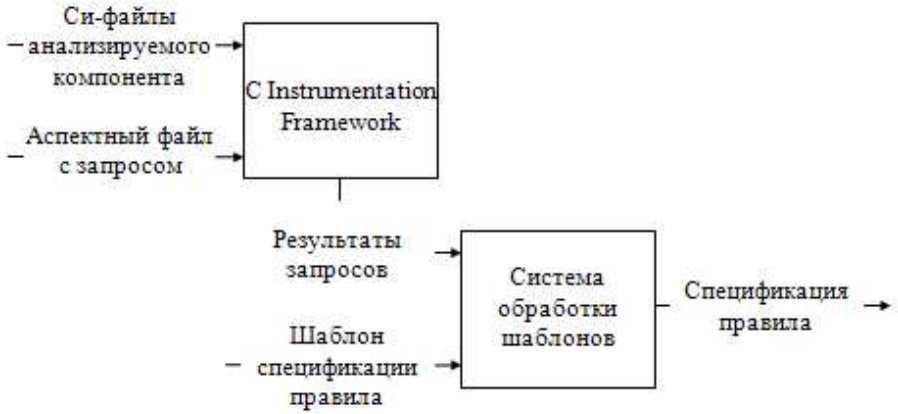

Рисунок 4. Архитектура инструментария для построения спецификаций правил использования программных интерфейсов.

Схема работы нового инструментария, реализующего предложенный метод, показана на Рис. 4. На первом шаге аспектный файл с запросом по исходному коду применяется к Си-файлам анализируемого компонента ядра ОС Linux с помощью C Instrumentation Framework (CIF) [26-28]. Затем получившиеся результаты запросов объединяются (например, при генерации подписей аргументов эта процедура заключается в исключении дубликатов) и используются для построения итоговой спецификации правила на основе шаблона спецификации. На практике аспектный файл с запросом также строится на основе шаблона спецификации правила, что позволяет избежать ненужного дублирования кода.

Новый инструментарий был реализован в компонентах системы верификации LDV:

- Был разработан новый компонент LDV, который выполняет запросы по исходному коду компонентов ядра ОС Linux с помощью CIF, обрабатывает полученные результаты и готовит на их основе и на основе шаблона финальную спецификацию правила.
- Был доработан существующий компонент Rule Instumentor, который на основе заданной спецификации осуществляет инструментирование исходного кода анализируемого компонента ядра ОС Linux с помощью CIF. Благодаря этому Rule Instrumentor стал получать информацию об аспектном файле с запросом, шаблоне спецификации правила и различных дополнительных настройках на основе базы данных спецификаций правил, поддерживаемых LDV, а также вызывать новый компонент соответствующим образом.

4. Практическое применение разработанного инструментария

В данном разделе будут использоваться следующие обозначения:

- *32_** – спецификации правила, описывающего корректное использование функций захвата и освобождения мьютексов;
- *39_** – спецификации правила, описывающего корректное использование функций захвата и освобождения спин блокировок;
- *32_1a* – спецификация, которая использует подход аспектно-ориентированное программирование для описания привязок модельных реализаций программных интерфейсов к использованию интерфейсов в компонентах ядра ОС Linux;
- *32_7* – спецификация, которая использует первоначальную реализацию подхода, встроенную в инструмент статической верификации BLAST;
- *32_7a* – расширенная версия спецификации *32_1a* в виде шаблона;
- *39_7* – аналог *32_7* для спин блокировок;
- *39_7a* – аналог *32_7a* для спин блокировок;
- *SIMPLE_ID* – механизм генерации подписей аргументов, который принимает во внимание только имя объекта (использовался по умолчанию для спецификаций 32_7a и 39_7a, а также единственный механизм, поддерживаемый для 32_7 и 39_7);
- *COMPLEX_ID* – механизм генерации подписей аргументов, который помимо имени объекта дополнительно использует имя структурного типа, если объект является полем переменной или параметра соответствующего структурного типа или указателя на него;
- *Safe* – вердикт верификации, который говорит о том, что в анализируемой программе нет нарушений проверяемой спецификации;

- *Unsafe* – вердикт верификации, который говорит о том, что в анализируемой программе обнаружено нарушение проверяемой спецификации, которое может быть либо ошибкой, либо ложным срабатыванием;
- *Unknown* – вердикт верификации, который говорит о том, что инструмент статической верификации не смог вынести вердикт *Safe* или *Unsafe* по той или иной причине (например, нехватка памяти или ошибка в парсере инструмента).

Далее с целью демонстрации эффективности предложенного подхода и его реализации в подразделах 4.1-6 приведен анализ результатов проверки спецификаций 32_* и 39_* для компонентов ядра ОС Linux с помощью инструмента статической верификации BLAST. Также подход был успешно использован для ряда других спецификаций, а новая реализация подхода позволила использовать для проверки инструмент статической верификации CPAchecker, но полученные результаты не приводятся в данной статье для краткости. В подразделе 4.7 демонстрируется альтернативный вариант использования подхода к построению спецификаций программных интерфейсов, а в подразделе 4.8 делается обобщение области применения предложенного подхода.

4.1. Сравнение спецификаций 39_7 и 39_7a

Сравнение результатов верификации для спецификаций 39_7 и 39_7a на всех драйверах устройств ядра Linux версии 3.5 с ограничением по памяти 6 Гб показало 324 перехода на 3630 запусков (около 9%):

- 114 Unknown → Safe. Анализ вердиктов Unknown демонстрирует следующее распределение по проблемам: около 80% из-за ошибок в BLAST, по 10% из-за нехватки памяти и времени.
- 40 Unsafe → Safe. Все вердикты Unsafe были обусловлены ошибками реализации подхода в инструменте BLAST.
- 23 Unsafe → Unknown. 19 Unsafe были обусловлены ошибками реализации подхода в инструменте BLAST. Для 4 оставшихся Unsafe соответствующие вердикты Unknown произошли из-за ошибок в BLAST.
- 145 Safe → Unknown. Анализ вердиктов Unknown демонстрирует следующее распределение по проблемам: около 70% из-за ошибок в CIF, по 10% из-за ошибок в CIL (парсер BLAST) и из-за нехватки памяти, 7% и 3% из-за ошибок в BLAST и генераторе модели окружения соответственно.
- 1 Safe → Unsafe. Unsafe не находился раньше по непонятной причине. Найденный Unsafe является ложным срабатываниям, т.к. разные спин блокировки сначала передаются в одну и ту же функцию (*diva_os_enter_spin_lock*) и только потом захватываются с помощью

одного из специфицируемых программных интерфейсов (при этом объекты называются одинаково).

Данное сравнение продемонстрировало, что новая реализация подхода работает в целом лучше, в частности, с ее помощью были преодолены проблемы реализации подхода в инструменте BLAST. Все негативные переходы происходили из-за ошибок в инструментах CIF, BLAST, CIL и генераторе модели окружения.

4.2. Сравнение спецификаций 32_7 и 32_7a

При проведении экспериментов для спецификации 32_7 была сделана специальная настройка системы верификации LDV для того, чтобы уменьшить число проблем, вызванных ошибками реализации подхода в инструменте статической верификации BLAST (использовать данную настройку на постоянной основе нецелесообразно). Также на тот момент были исправлены большинство ошибок в CIF.

Сравнение результатов верификации спецификаций 32_7 и 32_7a на всех драйверах устройств ядра Linux версии 3.5 с ограничением по памяти 6 Гб показало 236 переходов на 3630 запусков (около 7%):

- 134 Unknown → Safe. Анализ вердиктов Unknown демонстрирует следующее распределение по проблемам: около 90% из-за ошибок в BLAST, 10% из-за нехватки памяти.
- 6 Unsafe → Unknown. Четыре Unknown были обусловлены ошибками в BLAST, два были вызваны нехваткой времени.
- 46 Safe → Unknown. Анализ вердиктов Unknown демонстрирует следующее распределение по проблемам: 40% из-за ошибок в CIF, 20% из-за ошибок в CIL, 20% из-за нехватки памяти, по 10% из-за ошибок в BLAST и генераторе модели окружения.

Данное сравнение продемонстрировало, что новая реализация подхода позволяет выявить существенно больше Safe вердиктов при незначительном уменьшении числа Unsafe вердиктов. Все негативные переходы происходили из-за ошибок в инструментах BLAST, CIF, CIL и генераторе модели окружения.

4.3. Сравнение механизмов генерации подписей аргументов SIMPLE_ID с COMPLEX_ID

Сравнение результатов верификации, полученных для спецификации 32_7a с SIMPLE_ID и COMPLEX_ID на всех драйверах устройств ядра Linux версии 3.5 с ограничением по памяти 6 Гб, показало 4 перехода:

- 3 Unsafe → Safe. Все вердикты Unsafe были обусловлены тем, что захватывались мьютексы с одинаковыми именами из разных структур.
- 1 Unsafe → Unknown. Вердикт Unsafe был обусловлен тем, что захватывались мьютексы с одинаковыми именами из разных структур. Вердикт Unknown был вызван нехваткой памяти – в данном случае это лучше, чем получить ложное срабатывание.

Сравнение результатов верификации, полученных для спецификации 39_7a с SIMPLE_ID и COMPLEX_ID на всех драйверах устройств ядра Linux версии 3.5 с ограничением по памяти 6 Гб, показало 10 переходов:

- 7 Unsafe → Safe. Все вердикты Unsafe были обусловлены тем, что захватывались спин блокировки с одинаковыми именами из разных структур.
- 2 Unknown → Safe. Оба вердикта Unknown были связаны с ошибками в инструменте BLAST. Для 39_7 вердикт для одного из данных драйверов был Unsafe, так как захватывались мьютексы с одинаковыми именами из разных структур; для второго – тоже ошибка в инструменте BLAST.
- 1 Unknown → Unsafe. Вердикт Unknown был связан с ошибкой в инструменте BLAST. Для 39_7 вердикт для данного драйвера также был Unsafe.

Таким образом, в ряде случаев COMPLEX_ID позволяет получить корректные вердикты вместо ложных срабатываний, получаемых для SIMPLE_ID из-за того, что захватываются объекты с одинаковыми именами из разных структур. Значит, механизм генерации подписей аргументов COMPLEX_ID целесообразно использовать по умолчанию.

4.4. Сравнение спецификаций 32_1a и 32_7a

Сравнение спецификаций 32_1a и 32_7a (с использованием механизма генерации подписей COMPLEX_ID) на всех компонентах ядра Linux версии 3.8-rc1 с ограничением по памяти 15 Гб показало 188 переходов на 5372 запуска (около 3%):

- 115 Unsafe → Safe. Переходы обусловлены тем, что подход позволил различить мьютексы с разными именами из разных структур.
- 47 Unsafe → Unknown. Переходы обусловлены тем, что подход позволил различить мьютексы с разными именами из разных структур.
- 14 Unknown → Safe. Анализ вердиктов Unknown демонстрирует следующее распределение по проблемам: примерно по 40% из-за ошибок в BLAST и из-за нехватки времени, 20% из-за нехватки памяти.

- 1 Unknown → Unsafe. Вердикт Unknown был из-за ошибки в инструменте BLAST. Найденный Unsafe оказался истинной ошибкой[7].
- 2 Safe → Unknown. 1 Unknown возник из-за нехватки памяти, 1 - из-за ошибки в BLAST.

Таким образом, предложенные подход и его реализация позволяют сократить число ложных срабатываний, возникших вследствие неполноты модели окружения, на 162 (что составляет около 73% от общего числа Unsafe), а в ряде случаев даже помогают преодолеть некоторые ошибки в других инструментах.

4.5. Оценка границы применимости предложенного подхода

Для определения границы применимости предложенного подхода к построению спецификаций в условиях неполноты модели окружения рассмотрим причины оставшихся 59 Unsafe, полученных для спецификации 32_7a с COMPLEX_ID на всех драйверах устройств ядра Linux версии 3.8-rc1 с ограничением по памяти 15 Гб:

```
  * struct usb_interface *intf;
  * synusb_pre_reset(intf);
493   struct synusb *synusb = usb_get_intfdata(intf);
494   struct input_dev *input_dev = synusb->input;
496   mutex_lock(&input_dev->mutex);
  * synusb_reset_resume(intf)
520   return synusb_resume(intf);
475     struct synusb *synusb = usb_get_intfdata(intf);
476     struct input_dev *input_dev = synusb->input;
479     mutex_lock(&input_dev->mutex);
```

Рисунок 5. Код драйвера USB устройств Synaptics drivers/input/mouse/synaptics_usb.c («» помечена модель окружения, сгенерированная LDV).*

- 23 Unsafe остались из-за неполноты и некорректности модели окружения. Например, для драйвера USB устройств Synaptics, который состоит из одного Си-файла *drivers/input/mouse/synaptics_usb.c*[8], система верификации LDV

[7] https://lkml.org/lkml/2013/3/29/313.
[8] http://lxr.free-electrons.com/source/drivers/input/mouse/synaptics_usb.c?v=3.8.

сгенерировала модель окружения таким образом, что предложенный подход не позволил предотвратить ложное срабатывание. На Рис. 5 показано, что модель окружения вызывает функцию-обработчик *synusb_pre_reset*, передавая ей в качестве параметра **intf**. В данной функции из **intf** с помощью функции *usb_get_intfdata* получается **synusb** (строка 493), а из **synusb** получается его поле **input_dev** (строка 494). Затем в строке 496 захватывается мьютекс **mutex** из **input_dev**. Затем вместо функции-обработчика *synusb_post_reset*, которая освобождает данный мьютекс, модель окружения вызывает *synusb_reset_resume*, которая пытается повторно захватить его. Для решения подобных проблем сложно предложить что-нибудь, кроме уточнения конфигурации, на основе которых генерируется модель окружения.

- 16 Unsafe связаны с неточным анализом указателей в инструменте статической верификации BLAST. Например, для файловой системы NCP, которая состоит из нескольких Си-файлов, в том числе *fs/ncpfs/dir.c*, *fs/ncpfs/ncplib_kernel.c* и *fs/ncpfs/sock.c*[9], BLAST не отслеживает изменение поля структурной переменной, передаваемой в качестве параметра вызываемых функций (Рис. 6). В строке 866 захватывается мьютекс **mutex** из **server**, после чего в строке 869 полю **lock** из **server** присваивается 1. Позже, в строке 874, инструмент считает это поле равным нулю и не выполняет освобождение мьютекса **mutex** из **server**. В итоге это приводит к ложному срабатыванию. Ложные срабатывания из-за неточного анализа указателей в инструменте статической верификации BLAST не обусловлены ни спецификацией правила, ни моделью окружения, поэтому они явным образом не влияют на границу применимости предложенного подхода.

[9] http://lxr.free-electrons.com/source/fs/ncpfs/dir.c?v=3.8, http://lxr.free-electrons.com/source/fs/ncpfs/ncplib_kernel.c?v=3.8, http://lxr.free-electrons.com/source/fs/ncpfs/sock.c?v=3.8.


```
  * ncp_readdir(...);
551   ncp_read_volume_list(...);
706     ncp_get_volume_info_with_number(...);
209       ncp_init_request_s(server, 44);
 96         ncp_lock_server(server);
866           mutex_lock(&server->mutex);
869           server->lock = 1;
234       ncp_unlock_server(server);
874         if (!server->lock) {
875           printk(KERN_WARNING ...);
876           return;
877         }
```

Рисунок 6. Код файловой системы NCP fs/ncpfs/dir.c, fs/ncpfs/ncplib_kernel.c и fs/ncpfs/sock.c («» помечена модель окружения, сгенерированная LDV).*

```
116 static struct mousedev *mousedev_mix;
  * mousedev_init();
1092   mousedev_mix = mousedev_create(...);
  * mousedev_connect(...);
970    struct mousedev *mousedev;
973    mousedev = mousedev_create(...);
977    ... mixdev_add_device(mousedev);
931      ...mutex_lock_interruptible(&mousedev_mix->mutex);
936      ...mousedev_open_device(mousedev);
427        ... mutex_lock_interruptible(&mousedev->mutex);
```

Рисунок 7. Код драйвера мыши IntelliMouse Explorer PS/2 drivers/input/mousedev.c («» помечена модель окружения, сгенерированная LDV).*

- 13 Unsafe остались из-за того, что разные объекты имели одинаковые имена и получались из переменных и параметров с одним и тем же структурным типом. Например, для драйвера мыши IntelliMouse Explorer PS/2, который состоит из одного Си-файла *drivers/input/mousedev.c*[10], в строках 931 и 427 захватываются мьютекс **mutex** из **mousedev_mix** и **mutex** из **mousedev** соответственно,

[10] http://lxr.free-electrons.com/source/drivers/input/mousedev.c?v=3.8.

причем и **mousedev_mix**, и **mousedev** являются указателями на один и тот же структурный тип **struct mousedev** (Рис. 7). В данном случае **mousedev_mix** и **mousedev** указывают на разные объекты (они создаются независимым друг от друга образом в строках 1092 и 973), но предложенный подход не позволяет различить соответствующие мьютексы. В строке 538, которая не попала в рассматриваемый путь **mousedev** присваивается **mousedev_mix**. Поэтому захват в одном потоке мьютексов **mutex** из **mousedev_mix** и **mousedev**, которые указывает на одну и ту же область памяти, мог бы привести к зависанию системы. Данные Unsafe вердикты говорят о том, что предположение об отсутствии использовании алиасов для интересных объектов не всегда выполняется. Но поскольку это не может привести к пропуску ошибок и незначительно увеличивает число ложных срабатываний, этим можно пренебречь.

- 7 Unsafe являются реальными ошибками. Для большей части данных ошибок были сделаны патчи, которые были одобрены разработчиками ядра ОС Linux[11]. Достаточно много ошибок, связанных с некорректным захватом и освобождением мьютексов были обнаружены с помощью системы верификации LDV для более старых версий ядра.

Таким образом, использование предложенного подхода не позволило предотвратить ложные срабатывания в 52 случаях (около 23% от числа всех Unsafe), что является достаточно хорошим показателем на фоне преодоленных 162 ложных срабатываний (около 73% от общего числа Unsafe). Для устранения большинства оставшихся ложных срабатываний необходимо дорабатывать модель окружения и инструменты статической верификации.

4.6. Оценка эффективности работы инструментария

Оценка накладных расходов, возникших при применении разработанного инструментария для построения спецификаций программных интерфейсов, была произведена на основании общего времени работы системы верификации LDV на ряде запусков в различных конфигурациях. Значение имеет именно такая комплексная оценка времени, поскольку сравнивать время подготовки исходного кода к верификации нецелесообразно по нескольким причинам. Во-первых, использование разработанного инструментария требует строго больше времени, чем его неиспользование. Тоже справедливо в отношении времени работы алгоритма генерации подписей COMPLEX_ID по сравнению с SIMPLE_ID. Во-вторых, точно неизвестно, сколько по времени

[11] https://lkml.org/lkml/2013/2/19/468, https://lkml.org/lkml/2012/12/21/319, https://lkml.org/lkml/2013/3/29/313.

работает первоначальная реализация подхода, встроенная в инструмент статической верификации BLAST. Параметры конфигураций, а также результирующее время работы приведены в Табл. 1.

№	Версия ядра ОС Linux	Компоненты ядра	Спецификация правила	Ограничение по памяти (Гб)	Алгоритм генерации подписей	Общее время работы LDV (чч:мм)
1	3.5	drivers/media	32_7	6	SIMPLE_ID	2:37
2	3.5	drivers/media	32_7a	6	SIMPLE_ID	2:44
3	3.5	drivers	39_7	6	SIMPLE_ID	17:58
4	3.5	drivers	39_7a	6	SIMPLE_ID	16:51
5	3.5	drivers	39_7a	6	COMPLEX_ID	17:08
6	3.8-rc1	все	32_1a	15	-	34:16
7	3.8-rc1	все	32_7a	15	COMPLEX_ID	39:23

Таблица 1. *Общее время работы системы верификации LDV для различных конфигураций.*

По данной таблице видно следующее:

- На небольшом подмножестве драйверов устройств (*drivers/media*) для проверки спецификации 32_7a требуется незначительно больше времени, чем для 32_7 (номера запусков 1 и 2). Зато на всех драйверах устройств для проверки спецификации 39_7a требуется примерно на 1 час меньше времени, чем для 39_7 (номера запусков 3 и 4).
- На всех драйверах устройств для проверки спецификации 39_7a при использовании алгоритма генерации подписей COMPLEX_ID требуется незначительно больше времени, чем для SIMPLE_ID (номера запусков 4 и 5).
- На всех компонентах ядра ОС Linux для проверки спецификации 32_7a требуется на 5 с небольшим часов больше, чем для 32_1a (номера запусков 6 и 7). Данное увеличение затрат по времени происходит вследствие того, что время для вынесения вердикта Unsafe достаточно сильно меньше, чем время, требуемое на доказательство Safe. Для 32_7a находится более чем на 160 Unsafe меньше, чем для 32_1a (раздел 4.4).

4.7. Применение предложенного подхода при разработке спецификаций для новых правил

Помимо рассмотренной ранее возможности применения предложенного подхода к построению спецификаций программных интерфейсов в условиях неполноты модели окружения, метод позволил разработать спецификации для трех новых правил, которые требовали инициализировать объекты ядра ОС Linux до их использования. Дело в том, что зачастую инициализация объектов

в компонентах ядра ОС Linux осуществляется в глобальной области видимости, например, в драйвере *drivers/char/virtio_console.c*[12] в строке 84 с помощью макрофункции *DEFINE_SPINLOCK(pdrvdata_lock)* объявляется и инициализируется спин блокировка, а в строке 85 с помощью макрофункции *DECLARE_COMPLETION(early_console_added)* объявляется и инициализируется *completion*. Выполнить инструментирование данных конструкций таким же образом, как, например, вызовы функций, нельзя, поскольку в глобальной области видимости нельзя использовать неконстантные выражения.

С помощью предложенного подхода стало возможным построить спецификацию данных правил. Например, для рассматриваемого драйвера на основе его исходного кода сначала извлекается информация о том, какие спин блокировку (**pdrvdata_lock**) и *completion* (**early_console_added**) он объявляет и инициализирует, а затем эти данные используются для получения итоговой спецификации на основе шаблона.

4.8. Обобщение области применения предложенного подхода

Изначально подход был нацелен на преодоление ложных срабатываний у инструментов статической верификации, возникающих из-за неполноты модели окружения. Как это было показано ранее в данном разделе, с этой задачей подход справился достаточно хорошо. В предыдущем подразделе был продемонстрирован альтернативный вариант использования подхода для разработки спецификаций правил использования программных интерфейсов сердцевины ядра ОС Linux.

Область применения предложенного подхода можно обобщить следующим образом. Посредством аспектно-ориентированного программирования возможно выполнять инструментирование исходного кода компонента ядра с целью привязки точек использования программных интерфейсов сердцевины ядра в компонентах к модельным реализациям интерфейсов [26, 27]. Предложенный подход, который во многом опирается на аспектно-ориентированное программирование, расширяет его возможности. Он позволяет выполнять инструментирование некоторых точек программы на основе информации, полученной для других, связанных с данными некоторым образом точек. Это открывает новые возможности использования аспектно-ориентированного программирования, в частности для спецификации программных интерфейсов.

[12] http://lxr.free-electrons.com/source/drivers/char/virtio_console.c?v=3.8.

5. Заключение

В статье был предложен подход к построению спецификаций программных интерфейсов, нацеленный на уменьшение числа ложных срабатываний инструментов статической верификации, которые возникают при проверке выполнения требований спецификаций для правил использования программных интерфейсов в исходном коде компонентов ядра ОС Linux в условиях неполноты модели окружения. Была разработана новая реализация данного подхода на основе C Instrumentation Framework, которая показала в целом более хорошие результаты, чем его первоначальная реализация на основе инструмента статической верификации BLAST. Новый инструментарий позволил использовать для проверки спецификаций инструмент статической верификации CPAchecker, а также применять подход для спецификаций, для которых требуется привязка модельных реализаций программных интерфейсов к местам использования интерфейсов в коде компонентов на основе конструкций аспектно-ориентированного программирования.

Результаты практического применения новой реализации данного подхода в составе системы верификации Linux Driver Verification продемонстрировали, что подход справился с данной задачей достаточно успешно: он позволил сократить число ложных срабатываний более чем на 73%. В статье было показано, что для преодоления оставшихся ложных срабатываний необходимо дорабатывать модель окружения и инструменты статической верификации.

Помимо ядра ОС Linux предложенный подход к построению спецификаций программных интерфейсов в условиях неполноты модели окружения может быть применен при покомпонентной верификации в других проектах, в которых соблюдаются определенные правила кодирования. Более того, подход расширяет существующие возможности аспектно-ориентированного программирования, поскольку он позволяет выполнять инструментирование некоторых точек программы на основе информации, полученной для других, связанных с данными некоторым образом точек. В статье приводятся примеры, как с помощью новых возможностей были разработаны спецификации для нескольких правил использования программных интерфейсов.

Новый инструментарий был реализован в компонентах Linux Driver Verification. Для удобства использования в будущем планируется перенести соответствующую функциональность в C Instrumentation Framework и расширить его интерфейс. Также планируется расширить возможности реализации по выявлению объектов, интересных с точки зрения правил использования программных интерфейсов.

Литература

[1]. В.С. Мутилин, Е.М. Новиков, А.В. Хорошилов. Анализ типовых ошибок в драйверах операционной системы Linux. Труды Института системного программирования РАН, т. 22, стр. 349-374, 2012. DOI: 10.15514/ISPRAS-2012-22-19.

[2]. M.E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. IBM Systems Journal, 15(3), 182-211, 1976.

[3]. B. Boehm, V. Basili. Software Defect Reduction Top 10 List. IEEE Computer, 34(1), 135-137, January, 2001.

[4]. E.S. Raymond. The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. O'Reilly, Sebastopol, CA, USA, 2001.

[5]. P. Larson. Testing Linux with the Linux Test Project. Proceedings of the Ottawa Linux Symposium, Ottawa, Ontario, Canada, June 26–29, 2002.

[6]. Novell System Test Kit for Linux. https://www.suse.com/partners/ihv/pdf/SystemTestKit-7.1-Linux-10_22_12.pdf.

[7]. Oracle Linux Tests. https://oss.oracle.com/projects/olt/dist/documentation/OLT_TestCoverage.pdf.

[8]. А.В. Цыварев, В.А. Мартиросян. Тестирование драйверов файловых систем в ОС Linux. Труды Института системного программирования РАН, т. 23, стр. 413-426, 2012. DOI: 10.15514/ISPRAS-2012-23-24.

[9]. D. Engler, B. Chelf, A. Chou, S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. Proceedings of the 4th conference on Symposium on Operating System Design & Implementation, San Diego, California, pp.1-16, October 22-25, 2000.

[10]. Coverity Scan: 2011 Open Source Integrity Report. http://www.coverity.com/library/pdf/coverity-scan-2011-open-source-integrity-report.pdf.

[11]. H. Stuart. Hunting bugs with Coccinelle. Masters Thesis, University of Copenhagen, August, 2008.

[12]. Sparse - a Semantic Parser for C. https://sparse.wiki.kernel.org/index.php/Main_Page.

[13]. Dan Carpenter. Killing Bugs in C with Smatch. Linux Plumbers Conference, Santa Rosa, California, September 7-9, 2011.

[14]. P. Shved, M. Mandrykin, V. Mutilin. Predicate Analysis with Blast 2.7. Proceedings of TACAS, vol. 7214, pp. 525–527, 2012.

[15]. S. Löwe, P. Wendler. CPAchecker with Adjustable Predicate Analysis. Proceedings of TACAS, vol. 7214, pp. 528–530, 2012.

[16]. C. Sinz, F. Merz, S. Falke. LLBMC: A Bounded Model Checker for LLVM's Intermediate Representation. Proceedings of TACAS, vol. 7214, pp. 542–544, 2012.

[17]. D. Beyer. Competition on software verification. Proceedings of TACAS, vol. 7214, pp. 504–524, 2012.

[18]. Results of the 2013 2nd International Competition on Software Verification. http://sv-comp.sosy-lab.org/2013/results/index.php.

[19]. D. Engler, M. Musuvathi. Static analysis versus model checking for bug finding. In VMCAI, pp. 191-210, 2004.

[20]. T. Witkowski, N. Blanc, D. Kroening, G. Weissenbacher. Model checking concurrent Linux device drivers. In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pp. 501-504, 2007.

[21]. T. Ball, E. Bounimova, V. Levin, R. Kumar, J. Lichtenberg. The Static Driver Verifier Research Platform. CAV 2010, 2010.

[22]. H. Post, W. Küchlin. Integration of static analysis for Linux device driver verification. The 6th Intl. Conf. on Integrated Formal Methods, IFM 2007, 2007.

[23]. В.С. Мутилин, Е.М. Новиков, А.В. Страх, А.В. Хорошилов, П.Е. Швед. Архитектура Linux Driver Verification. Труды Института системного программирования РАН, т. 20, стр. 163-187, 2011.

[24]. A. Khoroshilov, V. Mutilin, E. Novikov, P. Shved, A. Strakh. Towards an Open Framework for C Verification Tools Benchmarking. Proceedings of the Eighth International Andrei Ershov Memorial Conference «Perspectives of Systems Informatics» (PSI 2011), pp. 82-91, 2011.

[25]. Открытая система верификации Linux Driver Verification. http://forge.ispras.ru/projects/ldv.

[26]. E. Novikov. One Approach to Aspect-Oriented Programming Implementation for the C programming language. roceedings of the 5th Spring/Summer Young Researchers' Colloquium on Software Engineering, Yekaterinburg, pp. 74-81, 12-13 May, 2011.

[27]. Реализация аспектно-ориентированного программирования для языка Си C Instrumentation Framework. http://forge.ispras.ru/projects/cif.

[28]. Е.М. Новиков, А.В. Хорошилов. Использование аспектно-ориентированного программирования для выполнения запросов по исходному коду программ. Труды Института системного программирования РАН, т. 23, стр. 371-386, 2012. DOI: 10.15514/ISPRAS-2012-23-21.

[29]. D. Chamberlain, D. Cross, A. Wardley. Perl Template Toolkit. O'Reilly Media, pp. 592, December, 2003.

[30]. N. Gunton. Creating Modular Web Pages With EmbPerl. March 13, 2001, http://www.perl.com/pub/2001/03/embperl.html.

Building Programming Interface Specifications in the Open System of Componentwise Verification of the Linux Kernel

Novikov E.M.
novikov@ispras.ru
ISP RAS, Moscow, Russia

Abstract. Nowadays static verification is one of the most promising methods for finding bugs in programs. To apply successfully existing tools for the Linux kernel one needs to perform componentwise verification. Such verification needs an environment model that reflects a real environment of components rather accurately. Development of the complete environment model for Linux kernel components is a very time-consuming task since there are too many programming interfaces in the kernel and they are not stable. The given paper suggests a new approach for building programming interface specifications. This approach allows one to apply static verification tools efficiently (with small number of false alarms but without missed bugs) for checking rules of programming interfaces usage under conditions of the incomplete environment model, if component developers obey a specific coding style. Two implementations of the suggested approach were developed. The first one extended the BLAST static verification tool while the second one utilized C Instrumentation Framework – an aspect-oriented programming implementation for the C programming language. The latter allowed to use various static verification tools, like BLAST, CPAchecker, CBMC, etc., for checking specifications. In practice that implementation helped to reduce the number of false alarms on more than 70% for some rules of programming interfaces usage. The paper shows that to avoid left false alarms one needs to develop more precise environment models and to use more accurate static verification tools. Besides the Linux kernel the suggested approach can be applied for componentwise verification in projects where developers obey the specific coding style.

Keywords: Linux kernel; kernel component; device driver; rule of programming interfaces usage; specification; environment model; static verification; aspect-oriented programming; C programming language.

References

[1]. Mutilin V.S., Novikov E.M., Khoroshilov A.V. Analiz tipovykh oshibok v drajverakh operatsionnoj sistemy Linux [Analysis of typical faults in Linux operating system drivers]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 22, pp. 349-374, 2012. DOI: 10.15514/ISPRAS-2012-22-19. (in Russian).

[2]. Fagan M.E. Design and Code Inspections to Reduce Errors in Program Development. IBM Systems Journal, vol. 38, issue 2/3, pp. 258-287, 1999. doi: 10.1147/sj.382.0258

[3]. Boehm B., Basili V. Software Defect Reduction Top 10 List. Computer, vol. 34, issue 1, pp. 135-137, 2001. doi: 10.1109/2.962984

[4]. Raymond E.S. The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. O'Reilly Media, 1999.

[5]. Larson P. Testing Linux with the Linux Test Project. In Proc. Ottawa Linux Symposium, 2002.

[6]. Novell System Test Kit for Linux. https://www.suse.com/partners/ihv/pdf/SystemTestKit-7.1-Linux-10_22_12.pdf.

[7]. Oracle Linux Tests. https://oss.oracle.com/projects/olt/dist/documentation/OLT_TestCoverage.pdf.

[8]. TSyvarev A.V., Martirosyan V.A. Testirovanie drajverov fajlovykh sistem v OS Linux [Testing of Linux File System Drivers].Trudy ISP RAN [The Proceedings of ISP RAS], vol. 23, pp. 413-426, 2012. DOI: 10.15514/ISPRAS-2012-23-24. (in Russian).

[9]. Engler D., Chelf B., Chou A., Hallem S. Checking system rules using system-specific, programmer-written compiler extensions. In Proc. 4th conference on Symposium on Operating System Design & Implementation (OSDI), vol. 4, pp. 1-16, 2000.

[10]. Coverity Scan: 2011 Open Source Integrity Report. http://www.coverity.com/library/pdf/coverity-scan-2011-open-source-integrity-report.pdf.

[11]. Stuart H. Hunting bugs with Coccinelle. University of Copenhagen, Masters Thesis, 2008.

[12]. Sparse - a Semantic Parser for C. https://sparse.wiki.kernel.org/index.php/Main_Page.

[13]. Shved P., Mandrykin M., Mutilin V. Predicate Analysis with Blast 2.7. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7214, pp. 525–527, 2012. doi: 10.1007/978-3-642-28756-5_39

[14]. Shved P., Mandrykin M., Mutilin V. Predicate Analysis with Blast 2.7. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7214, pp. 525–527, 2012. doi: 10.1007/978-3-642-28756-5_39

[15]. Löwe S., Wendler P. CPAchecker with Adjustable Predicate Analysis. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7214, pp. 528–530, 2012. doi: 10.1007/978-3-642-28756-5_40

[16]. Sinz C., Merz F., Falke S. LLBMC: A Bounded Model Checker for LLVM's Intermediate Representation. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7214, pp. 542–544, 2012. doi: 10.1007/978-3-642-28756-5_44

[17]. Beyer D. Competition on Software Verification. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7214, pp. 504-524, 2012. doi: 10.1007/978-3-642-28756-5_38

[18]. Results of the 2013 2nd International Competition on Software Verification. http://sv-comp.sosy-lab.org/2013/results/index.php.

[19]. Engler D., Musuvathi M. Static analysis versus model checking for bug finding. In Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI), LNCS, vol. 2937, pp. 191-210, 2004. doi: 10.1007/978-3-540-24622-0_17

[20]. Witkowski T., Blanc N., Kroening D., Weissenbacher G. Model checking concurrent Linux device drivers. In Proc. 22nd IEEE/ACM international conference on Automated Software Engineering (ASE), pp. 501-504, 2007. doi: 10.1145/1321631.1321719

[21]. Ball T., Bounimova E., Levin V., Kumar R., Lichtenberg J. The Static Driver Verifier Research Platform. In Proc. Computer Aided Verification (CAV), LNCS, vol. 6174, pp. 119–122, 2010. 10.1007/978-3-642-14295-6_11

[22]. Post H., Küchlin W. Integrated static analysis for Linux device driver verification. In Proc. Integrated Formal Methods (IFM), LNCS, vol. 4591, pp. 518-537, 2007. doi: 10.1007/978-3-540-73210-5_27

[23]. Mutilin V.S., Novikov E.M., Strakh A.V., Khoroshilov A.V., Shved P.E. Arkhitektura Linux Driver Verification [Linux Driver Verification Architecture]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 20, pp. 163-187, 2011 (in Russian).
[24]. Khoroshilov A., Mutilin V., Novikov E., Shved P., Strakh A. Towards an Open Framework for C Verification Tools Benchmarking. In Proc. Perspectives of Systems Informatics (PSI), LNCS, vol 7162, pp. 82-91, 2012. doi: 10.1007/978-3-642-29709-0_17
[25]. Open verification system Linux Driver Verification. http://linuxtesting.ru/ldv.
[26]. Novikov E. One Approach to Aspect-Oriented Programming Implementation for the C programming language. In Proc. 5th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE), pp. 74-81, 2011.
[27]. C Instrumentation Framework: aspect-oriented programming implementation for the C programming language. http://forge.ispras.ru/projects/cif.
[28]. Novikov E.M., Khoroshilov A.V. Ispol'zovanie aspektno-orientirovannogo programmirovaniya dlya vypolneniya zaprosov po iskhodnomu kodu programm [Using Aspect-Oriented Programming for Querying Source Code]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 23, pp. 371-386, 2012. DOI: 10.15514/ISPRAS-2012-23-21. (in Russian).
[29]. Chamberlain D., Cross D., Wardley A. Perl Template Toolkit. O'Reilly Media, 2003.
[30]. Gunton N. Creating Modular Web Pages With EmbPerl. 2001, http://www.perl.com/pub/2001/03/embperl.html.