Архитектура Linux Driver Verification

Мутилин В.С., Новиков Е.М., Страх А.В., Хорошилов А.В., Швед П.Е.
mutilin@ispras.ru, joker@ispras.ru, strakh@ispras.ru, khoroshilov@ispras.ru, shved@ispras.ru

Аннотация. В настоящей статье исследуются требования к построению архитектуры открытой системы верификации, которая могла бы предоставить площадку для экспериментов с различными методами статического анализа кода на реальном программном обеспечении и в то же время являлась полноценной системой верификации, готовой к индустриальному применению. По результатам обсуждения требований предлагается архитектура такой системы верификации и детально рассматриваются ее компоненты. В заключении описывается имеющийся опыт работы с предложенной архитектурой на практике и предлагаются пути дальнейшего развития.

Ключевые слова: статический анализ кода; правила корректности; верификация драйверов устройств; моделирование окружения драйвера; аспектно-ориентированное программирование; верификатор достижимости; визуализация трассы ошибки.

1. Введение

Статический анализ кода позволяет проверять выполнимость определенных свойств программ на основе некоторого представления их исходного кода без необходимости реального выполнения программ. Основными преимуществами данного подхода являются то, что, во-первых, статический анализ не требует подготовки специального тестового окружения и тестовых данных и может осуществляться сразу после написания исходного кода программы; во-вторых, статический анализ позволяет рассмотреть сразу все пути выполнения программы, в том числе, редко встречающиеся и сложно воспроизводимые при динамическом тестировании.

Однако практическое применение статического анализа кода имеет некоторые ограничения. Наиболее существенным ограничением при использовании статического анализа кода является время проведения анализа. Дело в том, что современные программы являются очень большими и сложными. В свою очередь это приводит к тому, что выполнение полного статического анализа реальных приложений за разумное время практически невозможно, также как, впрочем, и проведение любого вида исчерпывающего тестирования. Поэтому при практическом применении статического анализа используются различные методы упрощения анализируемых моделей программ и эвристики, которые

позволяют получить результат за приемлемое время за счет снижения качества статического анализа. Ключевыми характеристиками качества анализа являются число ложных предупреждений и число пропущенных ошибок искомого вида. В зависимости от степени упрощения и целевого времени работы в рамках статического анализа кода можно условно выделить легковесные и тяжеловесные подходы.

Легковесные подходы нацелены на то, чтобы получать результаты быстро, сравнимо по порядку величины со временем компиляции анализируемого приложения. Для достижения такой высокой скорости данные подходы обычно используют анализ графа потока данных в сопровождении с множеством различных эвристик, что в конечном итоге отрицательно сказывается на качестве анализа, как в плане количества пропущенных ошибок, так и в плане количества ложных предупреждений. Причем часто уменьшение числа ложных предупреждений имеет даже больший приоритет, чем обнаружение всех ошибок, так как опыт использования инструментов статического анализа показывает, что при большом проценте ложных предупреждений общая эффективность их применения значительно падает. Несмотря на это, на сегодняшний день легковесные подходы развиты достаточно хорошо. Существует большое количество различных инструментов, которые их реализуют и широко применяются в индустриальной разработке программ. К наиболее успешным коммерческим инструментам относятся Coverity [1] и Klocwork Insight [2], академическим — Svace [3], [4], [5], Saturn [6], FindBugs [7], Splint [8] и др.

При использовании тяжеловесных подходов ограничению по времени работы придается существенно меньшее значение, хотя, тем не менее, время проверки должно оставаться в разумных пределах. Это позволяет использовать значительно меньше эвристик при интерпретации исходного кода программ и, соответственно, применять более качественные методы статического анализа кода, что в свою очередь приводит как к уменьшению числа ложных срабатываний, так и к увеличению числа обнаруживаемых ошибок. Однако на сегодняшний день тяжеловесные подходы мало используются при анализе реальных приложений. Существует большое количество академических проектов, которые предлагают различные реализации тяжеловесных подходов, например, SLAM [9], BLAST [10], CPAchecker [11], CBMC [12], ARMC [13] и др. Но в индустрии тяжеловесные методы нашли свое применение только в проекте Microsoft SDV [14]. Этот проект следует выделить особо, так как он предоставляет полноценный набор инструментов, позволяющих проводить тяжеловесный статический анализ кода драйверов операционной системы (ОС) Microsoft Windows. Предлагаемые инструменты используются в процессе сертификации драйверов и включены в состав Microsoft Windows Driver Developer Kit, начиная с 2006 года. Проект Microsoft SDV наглядно демонстрирует возможность применения тяжеловесного подхода для верификации реальных программ. Однако, Microsoft SDV является, во-первых, узкоспециализированным, так как по сути нацелен на

применение только для драйверов ОС Microsoft Windows, а во-вторых, закрытым, что не позволяет ни расширять область его применения, ни использовать для экспериментов в области алгоритмов статического анализа.

Применение других существующих тяжеловесных инструментов статического анализа на практике носит фрагментарный характер. По сути, не существует площадки, на которой можно было бы сравнить характеристики различных инструментов анализа на исходном коде программ, активно используемых в промышленности. В проекте *Linux Driver Verification* [15], [16], [17] сделана попытка построить такую площадку для инструментов (в первую очередь, тяжеловесных), предназначенных для анализа программ на языке Си, на примере драйверов устройств ОС Linux. В настоящей статье исследуются требования и предлагается архитектура такой площадки, которая одновременно должна являться полноценной системой верификации, готовой к индустриальному применению.

Настоящая статья построена следующим образом. В разделе 2 сформулированы требования к открытой системе верификации драйверов устройств ОС Linux и проведен анализ существующих инструментов статического анализа драйверов для ОС Linux и Microsoft Windows. Раздел 3 содержит подробное описание предложенной архитектуры системы верификации *Linux Driver Verification* и ее компонентов. В разделе 4 описан накопленный опыт использования системы верификации, в том числе с точки зрения предложенных архитектурных решений. В заключении подведен итог и представлены направления дальнейшего развития проекта.

2. Требования к открытой системе верификации и существующие решения

Одной из основных целей проекта *Linux Driver Verification* — является построение открытой площадки для экспериментов с различными методами статического анализа кода, в первую очередь, тяжеловесными, при верификации реальных программ. Для достижения этой цели система верификации должна предоставлять удобные средства для интеграции новых инструментов и сравнительного анализа их работы с различными настройками. В качестве целевых программ в проекте рассматриваются драйвера ОС Linux, тем не менее, архитектура инструментария должна предусматривать возможность последующего расширения и на другие приложения.

С точки зрения применения статического анализа кода драйверы ядра ОС Linux являются весьма привлекательной целью по следующим причинам:

- Драйверов устройств ОС Linux достаточно много и скорость их появления только возрастает с непрерывно растущей популярностью ОС Linux.

- Большинство драйверов публикуются вместе с исходным кодом, который является необходимым для статического анализа.
- Корректность драйверов является важной составляющей безопасности систем [18], так как драйверы работают с тем же уровнем привилегий, что и остальное ядро.
- Исходный код драйверов относительно простой, а потому может быть проанализирован с высоким уровнем качества.
- Драйверы достаточно небольшие по размеру, а потому можно предположить, что время проверки одного драйвера будет сравнительно невелико.

В то же время ОС Linux имеет ряд особенностей, которые должны учитываться при разработке системы верификации. Ядро ОС Linux является одним из самых динамично развивающихся проектов в мире. В среднем, начиная с ядра версии 2.6.30, которое было выпущено в середине 2009 года, каждый день добавляется порядка 9000 новых строк кода, удаляется порядка 4500 и модифицируется порядка 2000 строк кода [19]. Драйверы устройств занимают наибольшую часть (до 70%) ядра, кроме того именно в исходном коде драйверов содержится наибольшее число (более 85%) различных ошибок, приводящих к некорректной работе всей ОС, зависаниям и падениям [20], [21]. Обеспечивать надежность драйверов Linux вручную, даже несмотря на большое количество разработчиков (более 1000 человек на сегодняшний день [19]), весьма затруднительно ввиду огромного количества достаточно сложного исходного кода (более 13 млн. строк кода [19]), который должен удовлетворять достаточно большому числу разнообразных правил корректности, начиная от общих правил, которым должны подчиняться все программы на Си (ядро и драйверы ОС Linux разрабатываются на языке программирования Си) и заканчивая специфичными правилами, которые говорят о том, как драйверы должны использовать интерфейс ядра. При этом важным отличием Linux от многих других ОС является то, что интерфейс ядра с драйверами постоянно расширяется и не является стабильным, поэтому со временем появляются новые правила, а старые частично модифицируются. Как следствие система верификации драйверов ОС Linux должны быть готова развиваться одновременно с развитием ядра и предоставлять удобные возможности для настройки существующих и добавления новых правил корректности.

Второй целью проекта *Linux Driver Verification* является разработка системы верификации, готовой к индустриальному применению, что требует минимизации участия человека в настройке инструментов и максимально удобный интерфейс для их использования. С точки зрения минимизации участия человека, в первую очередь, необходимо автоматизировать извлечение информации о составе драйвера и настройках его компиляции из уже имеющихся данных, предназначенных для сборки драйвера. При этом

важно учитывать, что с одной стороны у драйверов есть много различных зависимостей, а с другой — что для эффективного применения статического анализа количество строк кода должно быть не очень большим.

Вторая потребность в автоматизации связана с отсутствием традиционной точки входа (иными словами, функции *main*) у драйверов устройств. Для большинства тяжеловесных подходов статического анализа кода наличие точки входа является необходимым условием в виду того, что они исследуют пути выполнения в программе, начиная от данной точки. Поэтому для проведения верификации драйверов требуется генерация модельного окружения драйвера в виде искусственной точки входа, где должны вызываться функции обработчики драйверов (например, функция инициализации драйвера, чтения с устройства и т.п.) на манер того, как это делается при реальном взаимодействии драйверов, ядра и оборудования.

Удобство использования инструментов включает в себя удобство запуска верификации и удобство анализа ее результатов. Причем последнее является наиболее значимым, так как анализ выявленных ошибок может занимать немалое время и требовать привлечения высококвалифицированных, а значит и дорогостоящих, специалистов.

Рассмотрим существующие проекты по верификации драйверов, основывающиеся на тяжеловесных методах статического анализа кода, в первую очередь, с точки зрения сформулированных требований к построению открытой системы верификации, подходящей для индустриального использования.

Наиболее полноценным образом подход реализуется в уже упоминавшемся ранее проекте Microsoft SDV [14]. Данный проект предоставляет широкие возможности по верификации драйверов ОС Microsoft Windows и используется как для анализа драйверов, входящих в состав ОС Windows, так и в существующей программе сертификации драйверов сторонних разработчиков. К особенностям подхода относятся следующие:

- Для создания окружения от пользователя требуется вручную аннотировать исходный код драйверов, указав в нем роли каждой из функций обработчиков.
- Проверяемые правила корректности формализуются с помощью языка SLIC [22], в котором связь с исходным кодом драйвера задается с помощью аспектно-ориентированных конструкций, перехватывающих вызовы функций ядра. В настоящее время уже выделен набор из примерно 200 правил, а в исследовательской версии была добавлена возможность добавления новых правил. Следует отметить, что в отличие от ядра Linux в ядре Microsoft Windows интерфейс меняется гораздо реже, поэтому проблема подстраивания под изменения ядра для Microsoft SDV не столь актуальна.

- Известно, что в собственных исследовательских целях разработчики Microsoft SDV могут подключать два статических анализатора кода SLAM и Yogi [23]. Подключение верификаторов сторонними разработчиками не предусмотрено.
- По результатам проведения анализа генерируются статистические данные о запуске и детальные трассы ошибок, для анализа которых разработаны специализированные графические инструменты.

Существует также несколько инструментов, использующих тяжеловесные методы статического анализа кода для верификации драйверов ядра ОС Linux: Avinux [24], разработка университета города Тюбинген (Германия) и DDVerify [25], разработка университета Карнеги-Меллон (США).

Особенности инструмента Avinux таковы:

- Получение исходного кода драйвера для последующей верификации происходит на основе встраивания в процесс сборки ядра путем модификации файлов, описывающих сборку. Однако, Avinux предоставляет возможность автоматической работы только с единичными препроцессированными файлами. Поэтому, например, верификация драйверов, состоящих из нескольких файлов возможна только вручную, так как информация о зависимостях теряется.
- Так же, как и в Microsoft SDV, для создания окружения драйвера требуется вручную задать функции инициализации, выхода и обработчиков. При этом код инициализации состояния входных параметров генерируется автоматически.
- Для задания правил используются аспектно-ориентированные конструкции похожие на конструкции SLIC.
- Инструмент интегрирован с единственным статическим верификатором CBMC [12].

Инструмент DDVerify, который также предназначен для верификации драйверов ядра ОС Linux, обладает следующими характерными особенностями:

- Информацию об исходном коде драйверов для проверки DDVerify получает, используя собственные файлы сборки без учета файлов сборки ядра. Поэтому инструмент не учитывает специфику компиляции ядра в полной мере.
- Для создания окружения используется модель ядра для некоторых типов драйверов. Разработчиками были написаны модели для трех типов, а всего их несколько десятков.
- Правила корректности задаются как часть модели ядра. Код ограничений, накладываемых правилом, задается вместе с кодом, описывающим семантику функции. В данном подходе можно проверять только те функции, которые привязываются к драйверу с

помощью линковки. Поэтому для использования DDVerify требуется существенно изменять заголовочные файлы ядра.

- Инструмент позволяет подключать два инструмента статического анализа кода: CBMC [12] и SATABS [26].

Требования	Microsoft SDV	Avinux	DDVerify
Поддержка интеграции новых инструментов верификации	–	–	–
Переиспользование информации о параметрах сборки	+	Только для одного файла	–
Генерация модели окружения	По аннотации	Ручная	Для 3-х типов драйверов
Сопровождаемость в условиях непрерывного развитию ядра	Не требуется	+	–
Поддержка добавления новых правил корректности	+	+	±
Автоматизация анализа результатов	+	–	–

Табл. 1. Сравнение существующих инструментов верификации драйверо.

Таким образом, ни один из рассмотренных инструментов не позволяет полностью достигнуть заданных целей. Инструмент Microsoft SDV является закрытым и предназначен только для верификации драйверов ОС Microsoft Windows. Инструменты Avinux и DDVerify не подходят для широкомасштабного использования. Avinux требует описания функций обработчиков драйвера и не поддерживает драйверы, состоящие из нескольких файлов. DDVerify требует серьезной переработки собственного процесса сборки, заголовочных файлов и модели ядра для каждой новой версии ядра, что заметно усложняет сопровождение инструмента в условиях большого количества изменений в ядре ОС Linux. В случае обнаружения ошибки и Avinux, и DDVerify выдают пользователю текстовый вывод инструмента верификации, в то время как SDV обладает специализированными инструментами визуализации пути в программе, приводящей к обнаруженному нарушению правила корректности. Ни один из рассмотренных инструментов не поддерживает интеграцию сторонних инструментов верификации. В табл. 1 приведена сравнительная информация по рассмотренным инструментам с точки зрения выявленных требований к открытой системе верификации.

3. Архитектура системы верификации Linux Driver Verification

Архитектура *Linux Driver Verification* разрабатывалась для достижения описанных ранее целей: предоставить инфраструктуру для проверки драйверов ядра ОС Linux с высокой степенью автоматизации и возможность подключения различных инструментов, реализующих тяжеловесные подходы статического анализа кода.

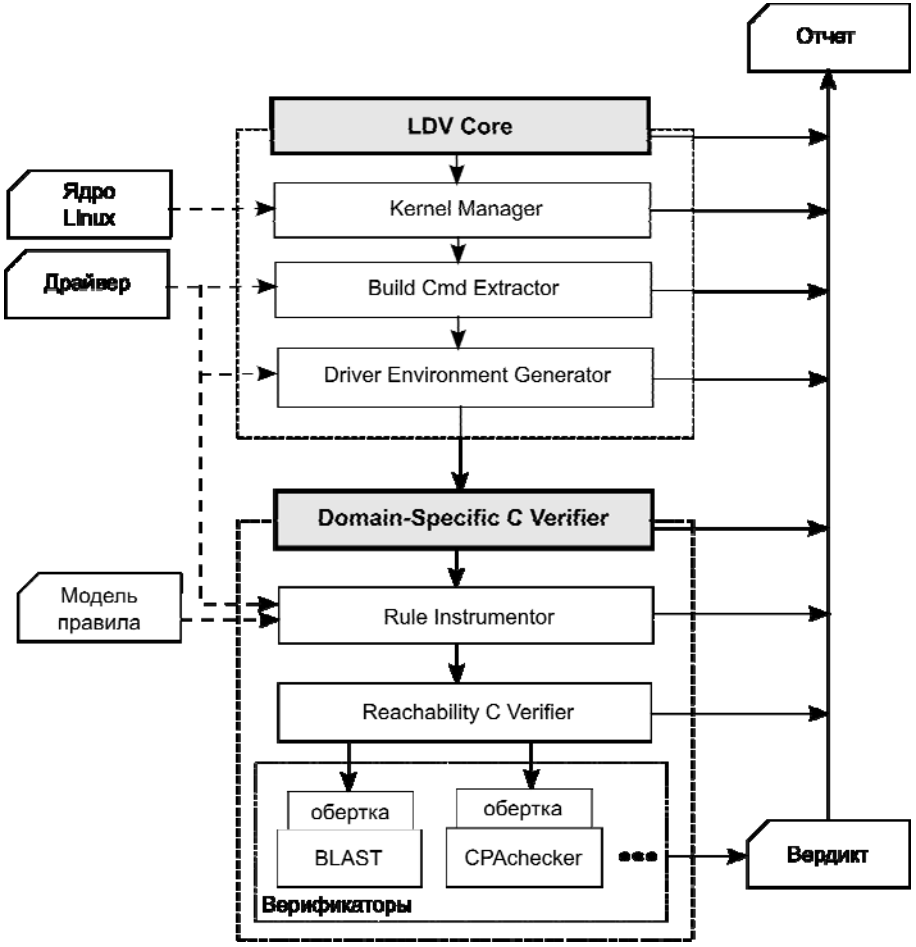

Рис. 1. Компоненты архитектуры Linux Driver Verification

Схематично архитектура изображена на рис. 1. Компоненты архитектуры изображены в центре. Стрелки указывают порядок, в котором соответствующие компоненты обрабатывают входные данные. Слева

изображены входные данные, которые предоставляются пользователем, однако, некоторые из них частично подготовлены разработчиками *Linux Driver Verification* (например, модели некоторых правил). Справа показан порядок формирования отчета по результатам верификации.

В общих чертах, процесс верификации драйверов происходит следующим образом. Запуск инструментария происходит через компонент *LDV-Core*. Данный компонент в начале вызывает компонент *Kernel Manager*, который на основе архива с исходным кодом ядра Linux создает на диске копию этого ядра со специально модифицированной подсистемой сборки (при последующих запусках эта копия переиспользуется).

Далее *LDV-Core* запускает процесс компиляции драйверов (внешних для данного ядра или входящих в его состав), и одновременно с этим процессом на основе модификаций сборки *Build Cmd Extractor* читает поток команд компиляции и линковки, выделяя из них те, которые относятся к верифицируемым драйверам.

Затем *LDV-Core* запускает *Driver Environment Generator*, который автоматически создает для каждого типа драйвера одну или несколько моделей окружения, добавляя код моделей к соответствующим файлам драйвера.

Domain Specific C Verifier уже не имеет представления о том, что поданный ему на вход код является драйверами ядра ОС Linux и может быть использован в неизменном виде для верификации произвольных программ на языке программирования Си. Данный компонент принимает абстрактную программу на языке программирования Си и передает ее компоненту *Rule Instrumentor*, чтобы тот встроил в нее код проверок указанных правил корректности.

Затем *Domain Specific C Verifier* передает модифицированный код конкретному инструменту верификации. Для этого он использует специальные обертки, реализации интерфейса взаимодействия конкретных верификаторов и инфраструктуры *Linux Driver Verification*. Вообще говоря, такие обертки предоставляются пользователем, но несколько из них входят непосредственно в состав инструментария. Обертки, получив окончательный список опций для вызова верификатора, непосредственно вызывают его.

На сегодняшний день в проекте *Linux Driver Verification* используется только один вид инструментов верификации, так называемые *верификаторы достижимости*, то есть инструменты, предназначенные для выявления нарушений правил корректности, выраженных в виде достижимости ошибочной точки в программе. В дальнейшем планируется реализовать поддержку и других классов верификаторов, например, решающих задачу завершаемости, и это не потребует внесения изменений в архитектуру системы верификации. Тем не менее, в рамках настоящей статьи мы будем рассматривать только верификаторы достижимости.

Инструмент верификации выносит вердикт, который может принимать одно из трех значений: SAFE, UNSAFE и UNKNOWN. Вердикт SAFE означает, что инструмент убедился в отсутствии нарушений проверяемого правила корректности при условии выполнения предположений, специфичных для данного инструмента. Вердикт UNSAFE говорит об обнаружении нарушения правила и сопровождается более детальной информацией о проблеме. В случае верификаторов достижимости, такой информацией является путь выполнения программы, который приводит к ошибочному состоянию. Вердикт UNKNOWN означает, что инструмент по тем или иным причинам не смог найти однозначного ответа на поставленный вопрос.

После этого полученный вердикт проходит все описанные выше стадии в обратном порядке. По ходу этого процесса вердикт дополняется отчетами о работе других компонентов, после чего формируется финальный отчет о проверке всего задания.

Компоненты между собой общаются с помощью потока команд. Изначально он является некоторым представлением команд сборки (компиляции и линковки), но, по мере обработки, модифицируется каждым компонентом. Компоненты могут изменять опции препроцессора, дописывать мета-информацию и даже подменять пути к файлам теми, по которым они располагают модифицированные файлы.

Все компоненты имеют определенный интерфейс и могут быть, при необходимости, заменены и/или модифицированы. Например, это может быть сделано при частичном изменении постановки задачи (см. *LDV-Git*).

Далее приведено подробное описание компонентов и интерфейсов *Linux Driver Verification*.

3.1. LDV-Core

На вход LDV-core поступает список драйверов для верификации, список ядер и список идентификаторов моделей правил. Данный компонент последовательно вызывает Kernel Manager, Build Cmd Extractor и Driver Environment Generator, описанные ниже.

3.2. Kernel Manager

В начале *LDV-core* для каждого из ядер вызывает *Kernel Manager*. Данный компонент в свою очередь создает переиспользуемую копию ядра, поданного ему на вход, одновременно дополняя процесс сборки соответствующего ядра вспомогательными командами. Последнее нужно с целью последующего сохранения информации о выполнении команд компиляции. В дальнейшем подготовленные ядра могут либо предоставлять свои внутренние драйвера для верификации, либо быть ядрами, с которыми собираются какие-либо внешние, не включенные в состав ядра, драйвера.

3.3. Build Cmd Extractor

После подготовки ядер *LDV-core* вызывает *Build Cmd Extractor*. Задача *Build Cmd Extractor* — выделить из процесса сборки ядра поток команд *cmdstream* и необходимые файлы с зависимостями для последующей верификации. Для внешних драйверов *Build Cmd Extractor* определяет наличие *Makefile* и *Kbuild* файлов, которые необходимы для сборки драйвера, и наличие соответствующих целей сборки.

Подкомпонент *Command Stream Divider* разделяет полученный после сборки *cmdstream* на множество небольших фрагментов (небольших файлов *cmdstream),* в каждом из которых собирается только один модуль. Этот подкомпонент особенно важен для анализа драйверов, включенных в ядро, поскольку подсистема сборки ядра компилирует их все вместе, не делая такого разделения.

3.4. Driver Environment Generator

Драйвер ядра ОС Linux состоит из одного или нескольких модулей. Эти модули не исполняются непосредственно как программы, а взаимодействует с ядром через специальный интерфейс.

Каждый модуль определяет две специальные функции: *module_init* и *module_exit*. Функция *module_init* вызывается при загрузке драйвера в ядро. В задачи функции входит инициализация состояния драйвера, а также регистрация специальных функций-обработчиков, предоставляющих интерфейс для работы с устройством. При наступлении определенных событий ядро вызывает функции-обработчики драйвера, которые обращаются непосредственно к физическому устройству для выполнения требуемых действий. Кроме того драйвер может регистрировать обработчики таймеров, прерываний, очередей ожидания и другие, которые также вызываются ядром при наступлении определенных событий.

Задача *Driver Environment Generator* — смоделировать окружение драйвера, включающее загрузку и выгрузку драйвера, а также смоделировать вызовы ядром ОС функций-обработчиков драйвера. Сгенерированная модель окружения выводится на языке Си в форме функции *main*, в которой реализованы вызовы функций драйвера. В качестве примера, для следующего драйвера (исходный код приведен частично):

```
static struct file_operations fops = {
     .open = sample_open,
};
static int sample_open(struct inode *inode, struct file
                                                 *file);
static int __init sample_init(void) { … }
static void __exit sample_exit(void) { … }
module_init(sample_init);
module_exit(sample_exit);
```


компонент *Driver Environment Generator* генерирует следующую модель окружения (выделены вызовы функций драйвера):

```
#ifdef LDV_MAIN0_plain_sorted_withcheck
void ldv_main0_plain_sorted_withcheck(void) {
     struct inode * var_sample_open_0_p0;
     struct file * var_sample_open_0_p1;
     static int res_sample_open_0;
     if(sample_init() != 0)
          goto ldv_final;
     res_sample_open_0=sample_open(
          var_sample_open_0_p0, var_sample_open_0_p1);
     if(res_sample_open_0)
          goto ldv_module_exit;
ldv_module_exit:
     sample_exit();
ldv_final:
     check_final_state();
}
#endif
```

В данной модели продемонстрировано, как в начале объявляются переменные для использования в качестве аргументов функций (важно, чтобы получившийся код был корректной программой на Си), а затем происходит вызов инициализирующей функции *sample_init*. В том случае, если произойдет ошибка, дальнейшая работа модели бессмысленна, а потому в случае ошибки пропускаются вызовы остальных функций драйвера. В случае успеха инициализации, вызываются функции из полей стандартной структуры *fops* в некоторой заранее определенной последовательности, задающейся в *Driver Environment Generator*.

На данный момент *Driver Environment Generator* позволяет генерировать следующие модели окружения:

1. Фиксированная последовательность вызовов функций-обработчиков.
2. Произвольная последовательность вызовов функций-обработчиков ограниченной длины.
3. Произвольная последовательность вызовов функций-обработчиков неограниченной длины.

3.5. Domain Specific C Verifier

Проблемно ориентированный верификатор Си (*Domain Specific C Verifier*) предоставляет интерфейс верификации Си программ, не зависящий от способа описания моделей правил и от используемого верификатора. На вход ему поступает поток команд *cmdstream* и список идентификаторов моделей правил для проверки. Для каждого идентификатора модели *Domain Specific C Verifier*

вызывает *Rule Instrumentor*, который преобразует *cmdstream* в задание верификатору (подробнее о данном компоненте будет сказано в соответствующем разделе). Взаимодействие с верификатором происходит через специальную обертку *Reachibility C Verifier*, специфичную для каждого верификатора достижимости. Таким образом, обеспечивается достаточно простая взаимозаменяемость верификаторов.

Отметим, что информация, специфичная для ядер Linux, полностью скрывается в описании модели правила, поэтому компонент *Domain Specific C Verifier* (как впрочем и все нижележащие компоненты), вообще говоря, может быть применен и для других предметных областей в том случае, если предоставлены соответствующие описания моделей правил корректности и поток команд. Для примера, уже на сегодняшний день в базе моделей правил описано правило проверяющее, что в программе не нарушается ни один *assert*. Для проверки этого правила *Domain Specific C Verifier* может быть запущен без *LDV-core* и *Driver Environment Generator* на исходном коде любой программы на языке Си.

3.6. Rule Instrumentor

Rule Instrumentor — компонент, основное назначение которого — связывать формализованные в виде моделей правила корректности с исходным кодом проверяемой программы для последующей верификации с помощью некоторого инструмента статического анализа кода.

Следует обратить внимание на два важных момента. Во-первых, в настоящий момент в рамках проекта *Linux Driver Verification* на основе различных источников для драйверов уже выделено достаточно много правил (более 70), а в будущем будут постоянно появляться новые правила [17]. Обычно правило корректности описывает, каким образом драйверам следует использовать интерфейс ядра Linux для того, чтобы они работали корректно. Это описание в конечном итоге сводится к тому, как необходимо использовать некоторые специфичные конструкции языка Си, например, вызов функции блокировки или выделения памяти. Принимая во внимание еще тот факт, что процесс верификации должен быть очень хорошо автоматизирован, получается, что необходимо иметь возможности для достаточно легкой формализации правил корректности, а также последующей связи формализованного представления с исходным кодом программ. Близкая по виду задача решается в аспектно-ориентированном программировании (АОП). Также стоит отметить, что практически все инструменты, которые используют реализации тяжеловесных подходов статического анализа кода, в той или иной степени применяют АОП для записи правил.

Rule Instrumentor на вход поступает идентификатор модели и командный файл *cmdstream*. Информация о моделях хранится в базе данных моделей. Используя идентификатор, компонент получает описание необходимой модели, которое, по сути, состоит из путей к аспектным файлам и информации

для верификатора. Аспектные файлы пишутся на языке, подобном аспектно-ориентированному расширению языка программирования Си.

Для выполнения всего процесса инструментирования в целом в настоящее время используется набор инструментов LLVM [27]. Непосредственно для препроцессирования, разбора файлов с исходным кодом и одновременно выполняемого инструментирования на основе аспектных файлов используется LLVM GCC Front End, ввиду того, что GCC является основным для сборки ядра под ОС Linux.

В качестве примера далее приведено упрощенное формализованное описание правила корректности для функции блокировки и общая схема инструментирования исходного кода.

- «*Аспект 1*» – модельные состояние и функции:

```
int islocked = UNLOCKED;   // Модельное состояние.
void own_mutex_lock() {    // Модельная функция.
  if (islocked == LOCKED) // Проверка правила.
    ERROR: abort();        // Ошибочная точка.
  islocked = LOCKED;       // Моделирование поведения
                           // исходной функции.
}
```

- «*Аспект 2*» – связь конструкций исходного кода с «*аспектом 1*»:

```
before:call($mutex_lock(..)) // Перед вызовом mutex_lock
{
  own_mutex_lock();        // вызвать модельную функцию.
}
```

- Исходный код (drivers/pcmcia/cs.c):

```
…
// Вызов функции extern void mutex_lock(struct mutex *).
mutex_lock(&socket_mutex);
…
```

- Инструментированный исходный код:

```
…
// Вызов вспомогательной функции.
ldv_mutex_lock(&socket_mutex);
…
// Определение вспомогательной функции.
void ldv_mutex_lock(struct mutex *arg) {
  own_mutex_lock(); // Вызов модельной функции.
  mutex_lock(arg);  // Вызов функции mutex_lock.
}
```

3.7. Reachibility C Verifier

Компонент Reachibility C Verifier решает задачу преобразования задачи верификации из внутреннего представления в виде потока команд cmdstream в представление конкретного верификатора. Подающийся на вход поток команд уже содержит файлы для проверки со сгенерированными точками входа и ошибочными метками. Для каждого верификатора, который используется в LDV, пользователь должен написать обёртку, которая содержит:

1. Описание способа вызова верификатора.
2. Указание необходимости препроцессирования, упрощения, линковки входных файлов.
3. Интерпретацию ошибочных точек и точек входа и перевод этой информации на язык, понятный верификатору.
4. Интерпретацию специальных настроек верификатора.
5. Интерпретатор результатов анализа.

В настоящее время для целей тестирования и демонстрации возможностей реализованы 2 обертки соответственно для верификаторов достижимости BLAST и CPAchecker.

3.8. Пользовательский интерфейс Linux Driver Verification

3.8.1. LDV-manager

LDV-manager предоставляет наиболее высокоуровневый интерфейс использования всего инструментария из командной строки. Данный компонент позволяет проверить некоторый набор драйверов (внутренних или внешних) для некоторого набора ядер по одному или нескольким правилам корректности. На выходе, в случае успешной работы, получается архив, содержащий результаты анализа, включая трассы ошибок и необходимые для их визуализации файлы с исходным кодом. Далее данный архив может быть загружен в базу данных и использован для анализа, например, с помощью сервера статистики или LDV-online, которые будут рассмотрены ниже.

3.8.2. LDV-online

LDV-online предоставляет веб-интерфейс для проведения верификации драйверов и последующего анализа получаемых результатов по некоторому набору ядер и правил. *LDV-online* нацелен на разработчиков драйверов, которые в наименьшей степени заинтересованы во внутренностях проводимого процесса верификации. Поэтому данный компонент в наибольшей степени упрощает процедуру запуска верификации и анализа результатов.

Для верификации драйвера с помощью LDV-online достаточно загрузить архив с драйвером с помощью веб-интерфейса на специальный сервер, а затем дождаться результатов. По мере появления промежуточных результатов они поступают пользователю, и он может сразу же приступить к их анализу. Представление результатов для их анализа тоже предлагается в упрощённом виде; трассы ошибок и текстовые описания правил, тем не менее, будут предложены пользователю.

Для верификации драйвера с помощью *LDV-online* в настоящее время, по сути, нужно только загрузить архив с драйвером с помощью веб-интерфейса, а затем дождаться результатов. Набор ядер Linux и правил является предопределенным. По мере появления промежуточных результатов они поступают пользователю, и он может сразу же приступить к их анализу.

LDV-online поддерживает авторизацию и хранение истории запусков. История запусков авторизованных пользователей показывается только им, а неавторизованных видна всем.

3.8.3. LDV-Git

LDV-Git — инструмент для непрерывного отслеживания изменений в драйверах ядра Linux, которое хранится в репозитории под управлением системы контроля версий Git. Данная система выбрана ввиду своей популярности при разработке ядра ОС Linux. Компонент LDV-Git, на основе истории изменений, вносимых в ядро Linux, формулирует и запускает задания для других компонентов Linux Driver Verification. Он автоматически определяет, какие драйвера следует перепроверить, а для каких результат верификации не изменится. LDV-Git проделывает полную проверку всех драйверов ядра один раз, а после внесения изменений в ядро запускает анализ только для тех драйверов, поведение которых могло измениться. Для изменившихся драйверов LDV-Git создает поток команд cmdstream, который сразу же передается генератору моделей окружения, а затем — Domain Specific C Verifier.

Определение драйверов для перепроверки делается на основе исходного кода и истории изменений, получаемой от Git; возможность этого обусловлена тем, что дальнейшая верификация также будет осуществляться по исходному коду драйверов.

LDV-Git предоставляет удобный интерфейс для пользователя, без его участия обрабатывая и сохраняя данные о предыдущих проверках. Основным способом использования *LDV-Git* предполагается периодическая проверка какой-либо ветви разработки ядра, например, при интенсивном внесении изменений в драйверы беспроводных устройств. В начале разработки ветви *LDV-Git* позволяет пометить некоторую ревизию как базисную; после этого влиять на выбор драйверов для проверки будут только изменения, внесенные в последующих версиях.

LDV-Git является примером того, как можно заменить несколько компонентов *Linux Driver Verification*, придав инструментарию не предусмотренную в нем изначально функциональность.

3.8.4. Сервер статистики

Сервер статистики – это компонент, который предоставляет интерфейс для статистического анализа результатов и их изменений, происходящих с течением времени. Целесообразность использования сервера статистики происходит, прежде всего, из необходимости быстрого анализа огромного количества данных и оценки динамики изменения данных по мере развития ядра, драйверов и компонентов описываемой архитектуры, в том числе, и внешних (например, инструментов статического анализа кода).

В основе составления статистики и сравнения результатов лежит упорядочивание и группировка данных по некоторому ключу, состоящему из одного или нескольких полей таких, как, например, имя ядра, идентификатор модели, имя драйвера и т.д. Это позволяет кластеризовать данные и существенно уменьшить размер их представления вплоть до характерных размеров экрана компьютера.

Для визуального представления данных сервер статистики использует веб-интерфейс. Благодаря использованию гиперссылок сервер статистики позволяет также анализировать полные списки, отвечающие некоторому набору ограничений таких, как ключ и, например, ошибки драйвера или некоторая проблема в работе компонента. В результате имеется как возможность быстрой оценки большого количества данных, так и их детального анализа в зависимости от текущих задач (например, оценка качества работы верификатора или исследование ошибок в компоненте Rule Instrumentor).

Linux Driver Verification может быть использован различной целевой аудиторией такой, как разработчики компонентов, разработчики ядра, разработчики верификаторов достижимости и т.д. Как правило, запросы к представлению статистики у этих групп отличаются, поэтому сервер статистики предлагает различные заранее подготовленные профили представления.

3.8.5. Визуализатор трассы ошибки

Когда инструмент верификации выдает вердикт UNSAFE, то есть находит в коде ошибку, её нужно показать пользователю. Причём пользователь не только должен понять, как эта ошибка проявляется, но и проверить, не является ли она ложным срабатыванием. Поэтому необходима визуализация трассы ошибки и связь её с контекстом — исходным кодом самого проверяемого драйвера.

Error trace

☑ Function bodies ☑ Blocks Others...

```
  - entry_point();
  {
113 + tmp___1 = my_init();
113 assert(tmp___1 == 0);
119 - rtmp0 = misc_open(var0…
    {
 22   + mutex_lock(&my_lock)
 23   - alock();
      {
 16    + mutex_lock(&my_lock)
      }
    }
  }
```

Source code

doublelock.c. common.c | model0032.c | fs.h | slub_def.h

```
 1 /* Check mutex lock/unlock. */
 2 #include <linux/kernel.h>
 3 #include <linux/module.h>
 4 #include <linux/mutex.h>
 5 #include <linux/major.h>
 6 #include <linux/fs.h>
 7
 8 static DEFINE_MUTEX(my_lock);
 9
10 static const struct file_operati…
11   .owner = THIS_MODULE,
12   .open  = misc_open
13 };
14
15 static void alock(void) {
16   mutex_lock(&my_lock);
17   mutex_unlock(&my_lock);
18 }
19
20 static int misc_open(struct inod…
21 {
22   mutex_lock(&my_lock);
23   alock();
24   mutex_unlock(&my_lock);
25   return 0;
26 }
27
28 static int __init my_init(void)
34 {
     …
41 }
42 static void __exit my_exit(void)
43 {
     …
45 }
47 module_init(my_init);
48 module_exit(my_exit);
```

Табл. 2. Визуализация трассы ошибки

Визуализатор трассы, таким образом, интерпретирует трассу, полученную от верификатора, преобразует её и связывает её с исходным кодом. Для представления результата используются средства разметки HTML и Javascript. Для различных конструкций трассы ошибки (например, вызов модельных функций, проверка условий и т.д.) используются различные стили и цвета.

Исходный код синтаксически подсвечивается. Та часть конструкций трассы ошибки, которая представляет небольшой интерес для ее анализа, автоматически скрывается.

Пример работы визуализатора трассы ошибки представлен в табл. 2 (представление значительно упрощено в виду ограниченности средств печати).

3.9. Тестовые наборы

По мере развития и постепенной стабилизации интерфейсов компонентов, возникла необходимость в различных регрессионных тестовых наборах. Основными поставщиками данных для тестовых наборов являются драйверы (как внешние, так и внутренние драйверы ядра). Еще один вариант использования тестовых наборов заключается в слежении за изменениями во времени работы различных компонентов, причем, как разрабатываемых, так и внешних. Здесь наибольший интерес представляют различные статические верификаторы (на практике, их работа занимает основную часть времени). В настоящий момент тестовые наборы уже используются для контроля времени работы компонентов, а также для сравнения различных верификаторов.

4. Апробация

Представленная архитектура *Linux Driver Verification* была реализована и для реализации была произведена апробация на ядрах Linux с версии 2.6.30 по 2.6.37. При этом использовались как драйверы, входящие в состав ядра и компилируемые как модули (порядка двух тысяч), так и внешние драйвера (несколько десятков). Для прогонов были выбраны девять формализованных правил.

Результаты показывают, что верификация достаточно большого количества драйверов возможна за приемлемое время. Так, например, общее время работы при проверке всех драйверов ядра 2.6.31.6, сконфигурированного для сборки всех модулей (*allmodconfig*), на одном правиле составляет около 17 часов. Зависимость среднего времени ожидания ответа от размера верифицируемого драйвера представлена на рис. 2. Среднее время считается по группам драйверов. В первую группу попадают драйверы размером до 2 тыс. строк кода, во вторую — с размером от 2 до 4 тыс. строк кода, в третью — от 4 до 6 тыс. и т.д. Для прогонов был установлен лимит по времени в 15 минут (900 секунд).

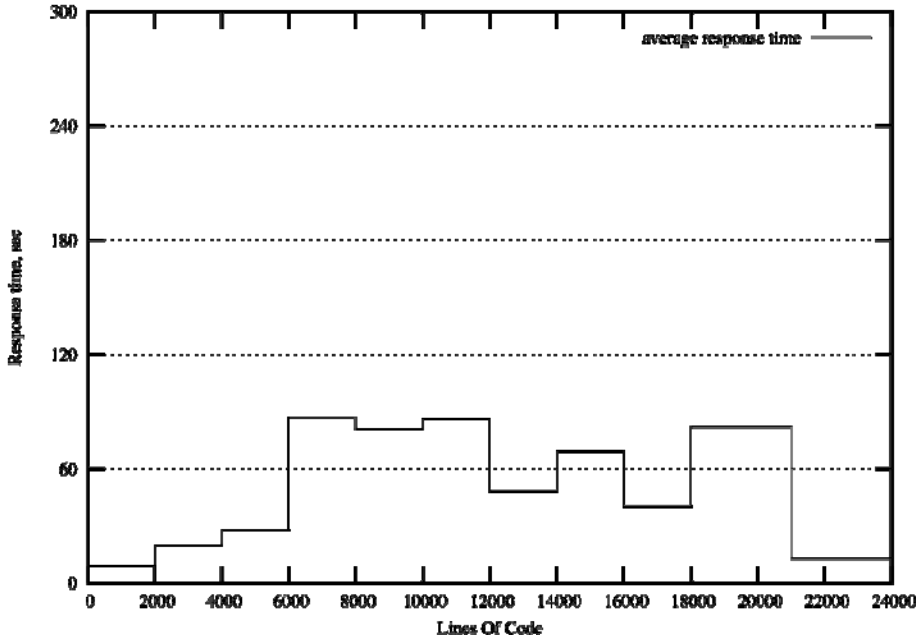

Рис. 2. Зависимость среднего времени ответа от размера исходного кода драйверов

Количество успешных результатов верификации составляет 86%, соответственно для 14% не удалось установить выполнено ли правило корректности из-за внутренних ошибок компонентов и наложенного ограничения по времени. Кроме того, прогоны показывают, что наиболее ресурсоемким компонентом является *Reachibility C Verifier*, на него приходится 90-95% процентов временных затрат.

Характерный размер драйвера, который успешно может быть проверен с помощью *Linux Driver Verification* можно оценить на основе графика, представленного на рис. 3. На нем показано распределение доли успешно завершившихся запусков по размеру верифицируемого драйвера в строках кода. Драйверы сгруппированы через каждые 2 тыс. строк кода; так же, как и на рис. 2. Сплошной линией показано общее количество драйверов, лежащих в данной группе. Пунктирная линия показывает долю (в процентах) успешно завершившихся запусков среди драйверов данной группы. Из этого графика видно, что для выбранного правила (проверка двойных блокировок) можно ожидать, что драйвер размером менее чем 12 тыс. строк кода, будет успешно проверен с не менее, чем 50% вероятностью. При этом ограничения на ресурсы составили 15 минут времени и 2 Гб памяти. Аномально высокие

показатели при размерах, превышающих 24 тыс. строк кода, обусловлены тем, что таких драйверов очень мало, а статическому верификатору было достаточно самой грубой абстракции для доказательства отсутствия достижимых ошибочных точек.

4.1. Найденные ошибки в драйверах

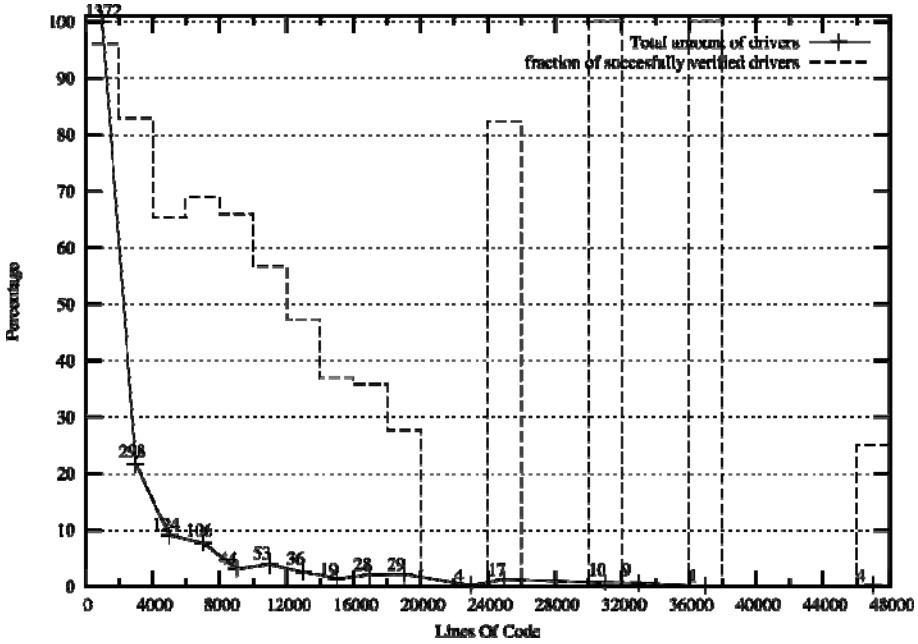

Рис. 3. Зависимость доли успешных результатов верификации от размера верифицируемого драйвера в строках кода

Самые простые из найденных ошибок - разыменование нулевых указателей, потенциальные возможности переполнения буфера и утечки памяти в драйверах fs/cifs/cifsencrypt.c и security/selinux/hooks.c. По последнему разработчикам было отправлено исправление (патч) и ошибка была исправлена. Некоторые найденные ошибки на практике встречаются редко и их трудно обнаруживать. Например, в драйвере drivers/media/video/cafe_ccic.c после неудачного запроса прерывания не освобождался mutex, а в драйвере drivers/media/video/hdpvr/hdpvr-core.c — mutex не освобождался после неудачной отсылки управляющего сообщения подсистемы USB. По этим двум драйверам были направлены исправления, которые были включены в последние версии ядра. Также были найдены ошибки, связанные с вызовами некоторых функций в запрещенном контексте (drivers/char/isicom.c и drivers/net/znet.c), двойная блокировка (drivers/hid/hidraw.c), разблокировка не

захваченного mutex (drivers/net/wireless/iwlwifi/iwl3945-base.c, drivers/input/input.c); неправильное использование функции module_put в drivers/mtd/mtd_blkdevs.c.

В общей сложности, из 25 выявленных нарушений, 21 нарушение было признано разработчиками ядра как ошибки, требующие исправления. Важно отметить то, что эти ошибки были обнаружены на достаточно малом наборе правил (5 правилах) при тестовых прогонах, в то время как потенциал использования *Linux Driver Verification* гораздо шире.

4.2. Гибкость и расширяемость

За время разработки ядер с версии 2.6.30 по 2.6.37 был изменен интерфейс ядра, используемый драйверами. Например, макрос *spin_lock* был заменен на *static inline* функцию, при этом его семантика не менялась. Кроме того, были добавлены специальные конструкции, поддерживаемые только новыми компиляторами *gcc*. Несмотря на эти изменения, трудозатраты на поддержание новых версий ядра в представленной архитектуре *Linux Driver Verification* оставались минимальными. Например, для того, чтобы учесть изменения в интерфейсе ядра, потребовалось добавить привязку к новому интерфейсу в связующий аспектный файл соответствующей модели.

Архитектура обладает высокой степенью расширяемости. Добавление новых правил осуществляется с помощью написания аспектных файлов на языке подобном аспектно-ориентированному расширению языка программирования Си. Использование аспектно-подобных описаний хорошо зарекомендовало себя в инструментах Microsoft SDV и Avinux. Наш опыт также свидетельствует о том, что это сравнительно простой интуитивный язык, с помощью, которого удобно поддерживать актуальность моделей для стремительно меняющегося ядра Linux.

Добавление верификаторов достижимости осуществляется заданием специальной обертки, описывающей связь соответствующего верификатора с компонентами *Linux Driver Verification*. На данный момент обертки были написаны для двух верификаторов достижимости BLAST и CPAchecker. В ближайшем будущем планируется расширение списка поддерживаемых верификаторов.

Коллектив разработчиков, которые также являются и авторами данной статьи, продолжает поддерживать и развивать архитектуру *Linux Driver Verification* по различным направлениям, а также приглашает желающих принять участие в данном процессе. Текущее состояние архитектуры можно увидеть на странице проекта [28]. Также с данной страницы возможно скачать полный набор компонентов, прочитать инструкции, установить требуемые компоненты и использовать их для решения своих задач.

5. Направления дальнейшего развития

Развитие архитектуры *Linux Driver Verification* в краткосрочной перспективе планируется по следующим направлениям:

1. Распараллеливание многих тяжеловесных подпроцессов позволит значительно повысить скорость верификации при более полном использовании современных вычислительных мощностей. Реализованная на сегодняшний день архитектура во многом способствуют проведению распараллеливания за счет аккуратной функциональной декомпозиции компонентов.
2. Расширение базы данных правил и увеличение числа формализованных правил с целью повышения практической ценности результатов работы системы верификации для разработчиков драйверов.
3. Дальнейшее исследование и улучшение используемых инструментов статического анализа кода для достижения большего уровня качества и увеличения скорости анализа. Помимо этого планируется проведение исследований возможностей других инструментов верификации.
4. Повышение конфигурируемости за счет автоматического анализа различных конфигураций ядра, в том числе, различных архитектур; автоматического подбора более подходящего окружения драйверов и статического анализатора с опциями для получения более точных результатов верификации за меньшее время.

6. Заключение

В статье представлена архитектура открытой системы верификации *Linux Driver Verification*, которая разработана с тем, чтобы предоставить площадку для экспериментов с различными методами статического анализа кода на реальном программном обеспечении и в то же время стать полноценной системой верификации, готовой к индустриальному применению в области верификации драйверов ОС Linux. Первоначальный опыт использования системы верификации *Linux Driver Verification* показывает, что предложенная архитектура позволяет решать поставленные задачи и, кроме того, является достаточно гибкой для безболезненной реализации заранее не предусмотренных вариантов использования. Тем не менее, еще предстоит сделать немало шагов для внедрения системы верификации в процессы разработки драйверов ОС Linux и для привлечения сторонних исследователей к экспериментам с новыми алгоритмами верификации на основе *Linux Driver Verification*, чтобы качество предложенной архитектуры получило объективную оценку, а цели проекта оказались достигнутыми.

Литература

[1] Инструмент *Coverity*. http://www.coverity.com/products/static-analysis.html.
[2] Инструмент *Klocwork Insight*. http://www.klocwork.com/products/insight/.
[3] В.С. Несов, О.Р. Маликов. Использование информации о линейных зависимостях для обнаружения уязвимостей в исходном коде программ. Труды Института системного программирования РАН, том 9, стр. 51-56, 2006.
[4] В.С. Несов, С.С. Гайсарян. Автоматическое обнаружение дефектов в исходном коде программ. Методы и технические средства обеспечения безопасности информации: Материалы XVII Общероссийской научно-технической конференции. СПб.: Изд-во Политехн. Ун-та, с. 107, 2008.
[5] V. Nesov. Automatically Finding Bugs in Open Source Programs. Proceedings of the Third International Workshop on Foundations and Techniques for Open Source Software Certification, Volume 20, pages 19-29, 2009.
[6] Инструмент *Saturn*. http://saturn.stanford.edu/.
[7] Инструмент *FindBugs*. http://findbugs.sourceforge.net/.
[8] Инструмент *Splint*. http://www.splint.org/.
[9] T. Ball, E. Bounimova, R. Kumar, V. Levin. SLAM2: Static Driver Verification with Under 4% False Alarms. FMCAD, 2010.
[10] D. Beyer, T. Henzinger, R. Jhala, R. Majumdar. The Software Model Checker Blast: Applications to Software Engineering. Int. Journal on Software Tools for Technology Transfer, 9(5-6):505-525, 2007.
[11] D. Beyer, M. Keremoglu. CPAchecker: A Tool for Configurable Software Verification. Technical report SFU-CS-2009-02, School of Computing Science (CMPT), Simon Fraser University (SFU), January 2009.
[12] E. Clarke, D. Kroening, F. Lerda. A Tool for Checking ANSI-C Programs. In Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, LNCS, Vol. 2988/168-176, 2004.
[13] Инструмент *ARMC*. http://www.mpi-sws.org/~rybal/armc/.
[14] T. Ball, E. Bounimova, V. Levin, R. Kumar, J. Lichtenberg. The Static Driver Verifier Research Platform. CAV 2010, 2010.
[15] A. Khoroshilov, V. Mutilin. Formal Methods for Open Source Components Certification. OpenCert 2008 2nd International Workshop on Foundations and Techniques for Open Source Software Certification 52-63 Milan, 2008.
[16] A. Khoroshilov, V. Mutilin, V. Shcherbina, O. Strikov, S. Vinogradov, V. Zakharov. How to Cook an Automated System for Linux Driver Verification. SYRCoSE'2008 2nd Spring Young Researchers' Colloquium on Software Engineering, Volume 2 11-14 St. Petersburg May 29-30, 2008.
[17] A. Khoroshilov, V. Mutilin, A. Petrenko, V. Zakharov. Establishing Linux Driver Verification Process. PSI 2009 Perspectives of System informatics 2009 LNCS, Vol. 5947/ 165-176, 2009.
[18] В.П.Иванников, А.К. Петренко. Задачи верификации ОС Linux в контексте ее использования в государственном секторе. Труды Института системного программирования РАН, том 10, стр. 9-14, 2006.
[19] G. Kroah-Hartman, J. Corbet, A. McPherson. Linux kernel development. http://www.linux-foundation.org/publications/linuxkerneldevelopment.php, 2008.
[20] A. Chou. An Empirical Study of Operating System Errors. Proc. 18th ACM Symp. Operating System Principles, ACM Press, 2001.

[21] M. Swift, B. Bershad, H. Levy. Improving the reliability of commodity operating systems. In: SOSP ’03: Proceedings of the nineteenth ACM symposium on Operating systems principles, New York, NY, USA, ACM 207–222, 2003.
[22] T. Ball, S. K. Rajamani. Slic: A specification language for interface checking of C. Technical Report MSR-TR-2001-21, Microsoft Research, 2001.
[23] N. Beckman, A. Nori, S. Rajamani, R. Simmons. Proofs from tests. In ISSTA’08: International Symposium on Software Testing and Analysis, pages 3–14, 2008.
[24] H. Post, W. Küchlin. Integration of static analysis for linux device driver verification. The 6th Intl. Conf. on Integrated Formal Methods, IFM 2007, 2007.
[25] 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 (ASE '07), pages 501-504, 2007.
[26] E. Clarke, D. Kroening, N. Sharygina, K. Yorav. SATABS: SAT-based Predicate Abstraction for ANSI-C. In Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems, LNCS, Vol. 3440/570-574, 2005.
[27] Набор инструментов *LLVM*. http://llvm.org/.
[28] Инструмент *Linux Driver Verifier*. http://forge.ispras.ru/projects/ldv.