ПРОГРАММИРОВАНИЕ, 2015, № 1, с. 44–67

БЕЗОПАСНОСТЬ И ЗАЩИТА

УДК 681.3.06

КОНФИГУРИРУЕМАЯ СИСТЕМА СТАТИЧЕСКОЙ ВЕРИФИКАЦИИ МОДУЛЕЙ ЯДРА ОПЕРАЦИОННЫХ СИСТЕМ *

© 2015 г. И.С. Захаров, М.У. Мандрыкин, В.С. Мутилин, Е.М. Новиков, А.К. Петренко, А.В. Хорошилов

Институт системного программирования РАН

109004 Москва, ул. А. Солженицына, 25

E-mail: {ilja.zakharov, mandrykin, mutilin, novikov, petrenko, khoroshilov}@ispras.ru

Поступила в редакцию 12.09.2014

Ядро операционной системы (ОС) представляет собой критичную в отношении надежности и производительности программную систему. Качество ядра современных ОС уже находится на достаточно высоком уровне. Иначе обстоит дело с модулями ядра, например, драйверами устройств, которые по ряду причин имеют существенно более низкий уровень качества. Одними из наиболее критичных и распространенных ошибок в модулях ядра являются нарушения правил корректного использования программного интерфейса ядра ОС. Выявить все такие нарушения в модулях или доказать их корректность потенциально можно с помощью инструментов статической верификации, которым для проведения анализа необходимо предоставить контрактные спецификации, описывающие формальным образом обязательства ядра и модулей по отношению друг к другу. В статье рассматриваются существующие методы и системы статической верификации модулей ядра различных ОС. Предлагается новый метод статической верификации модулей ядра ОС Linux, который позволяет конфигурировать процесс проверки на каждом из его этапов. Показывается, каким образом данный метод может быть адаптирован для проверки компонентов ядра других ОС. Описывается архитектура конфигурируемой системы статической верификации модулей ядра ОС Linux, реализующей предложенный метод, и демонстрируются результаты ее практического применения. В заключении рассматриваются направления дальнейшего развития.

1. ВВЕДЕНИЕ

Надежность и производительность ядра операционной системы (ОС) являются важными характеристиками его качества, поскольку ядро лежит в основе ОС и на результаты его работы во многом полагаются все пользовательские приложения. В ядре большинства современных ОС реализуется только основная функциональность, например, планирование процессорного времени, управление памятью и межпроцессным взаимодействием. Благодаря этому ядро, с одной стороны, является достаточно небольшим по объему, а с другой стороны, представляет собой программную систему, высокое качество которой формировалось в течение длительного периода времени в разнообразных сценариях использования.

В большинстве ОС набор функций ядра можно расширить путем динамической загрузки модулей. Типичным примером модуля ядра служит драйвер устройства, необходимость в котором может возникнуть при подключении данного устройства. Также в виде модулей часто реализуют файловые системы, сетевые протоколы, аудиокодеки и т.д. Объем исходного кода модулей, поставляемых вместе с ядром ОС, может существенно превышать объем последнего, например, для ядра ОС Linux примерно

*Исследование проводилось при финансовой поддержке Минобрнауки РФ (уникальный идентификатор проекта — RFMEFI60414X0051).

в 8 раз. Кроме того, многие разработчики по разным причинам не предоставляют доступ к исходному коду некоторых из своих модулей, а распространяют их в виде бинарного кода, что существенно затрудняет применение некоторых подходов обеспечения качества. Не все модули ядра могут использоваться активно, например, если это драйверы специфичных устройств. Все это приводит к тому, что модули ядра различных ОС имеют существенно меньший уровень качества по сравнению с самим ядром. Это подтверждается исследованиями, которые показали, что в тех модулях, чей исходный код доступен для анализа, содержится примерно в 7 раз больше ошибок, чем в ядре ОС [1–3].

Высокая производительность модулей ядра ОС обеспечивается за счет того, что большинство из них работают в том же адресном пространстве и с тем же уровнем привилегий, что и ядро. Из-за этого ошибки в модулях могут привести к ненадежной работе и снижению производительности ядра, а также и ОС в целом. Анализ изменений в модулях ядра ОС Linux за год разработки показал, что нарушения правил корректного использования программного интерфейса ядра в модулях являются источником около половины ошибок, которые не связаны с нарушениями спецификаций аппаратного обеспечения, сетевых протоколов, аудиокодеков и т.д. [4]. Для других ОС авторам недоступна подобная статистика, но существующие направления исследований также демонстрируют достаточно высокий интерес к ошибкам данного типа [5].

1.1. Подходы к обнаружению нарушений правил корректного использования программного интерфейса ядра ОС в модулях

Выявлять нарушения правил корректного использования программного интерфейса ядра ОС в модулях можно различными способами. На практике наиболее часто используются экспертиза кода и тестирование. Благодаря использованию данных подходов удается обнаружить и исправить достаточно большое количество ошибок в модулях. Однако они не позволяют выявить все возможные ошибки [6]. Тщательная экспертиза кода требует больших трудозатрат, а потому она в полной мере проводится только для ядра ОС, но не для такой большой, сложной и динамично развивающейся программной системы, как модули ядра. Тестирование обычно проводится автоматизированным образом, что позволяет сократить трудозатраты по сравнению с экспертизой кода. Однако данный подход требует подготовки тестового окружения, что, например, в случае драйверов устройств бывает достаточно затруднительно. Кроме того, с помощью тестирования можно тщательно проверить только небольшие программные системы.

В последнее время основной тенденцией в проверке программных систем является применение методов и инструментов статического анализа кода, т.е. анализа кода без его реального выполнения (как правило, анализируется исходный код программных систем). На практике преимущественно применяются так называемые легковесные подходы, которые благодаря использованию ряда эвристик позволяют проверять код больших программных систем за время, сравнимое по порядку со временем компиляции. Негативными последствиями использования эвристик является то, что инструменты, которые реализуют данные подходы статического анализа кода, с одной стороны, пропускают ошибки, а с другой стороны, выдают большое количество ложных сообщений об ошибках. Эти инструменты преимущественно применяются для поиска нарушений общих правил безопасного программирования, таких как разыменования нулевых указателей, выход за границу массива и т.д. [7, 8]. Некоторые инструменты были использованы для проверки правил корректного использования программного интерфейса ядра ОС Linux в модулях [9].

Выявить все ошибки искомого вида в программных системах или доказать их корректность потенциально можно с помощью тщательного статического анализа – *статической верификации*. Современные инструменты статической верификации, которые реализуют, например, метод уточнения абстракции по контрпримерам [10], уже позволяют доказывать выполнимость специфицированных свойств для средних по размеру программных систем за приемлемое время. В частности, с помощью

данных инструментов можно верифицировать отдельные компоненты ядра ОС, такие как модули ядра (для ядра ОС Linux размер большинства модулей составляет несколько тысяч строк кода).

Сами по себе инструменты статической верификации не способны искать нарушения правил корректного использования программного интерфейса ядра ОС в модулях – они позволяют решать *задачу достижимости*. Как правило, инструменты позволяют определить возможность достижимости некоторого оператора, помеченного заданной меткой, от указанной точки входа. Поэтому необходимо некоторым образом свести задачу обнаружения нарушений проверяемых правил к задачам достижимости. Для этого требуется разработать *спецификации правил корректного использования программного интерфейса ядра ОС*, которые устанавливают соответствие между нарушениями правил и достижимостью оператора, помеченного заданной меткой. Кроме того, исследования показали, что для того, чтобы получить приемлемые результаты статической верификации модулей ядра (найти ошибки искомого вида при умеренном количестве ложных сообщений об ошибках), инструментам требуется достаточно точная *модель окружения*, которая должна описывать те же сценарии взаимодействия ядра и модулей, которые возможны при их работе в реальном окружении [11].

Таким образом, инструментам статической верификации для поиска нарушений правил корректного использования программного интерфейса ядра ОС в модулях необходимо предоставить *контрактные спецификации*, которые формальным образом описывают обязательства ядра и модулей по отношению друг к другу. Со стороны ядра контрактные спецификации должны задавать множество возможных сценариев взаимодействий с модулями корректным и полным образом, а также предоставлять модель программного интерфейса ядра, используемого модулями. Со стороны модулей контрактные спецификации должны задавать, какие обращения модулей к ядру являются корректными, а какие – нет.

В настоящее время инструменты статической верификации активно развиваются в университетах и научно-исследовательских институтах по всему миру. С каждым годом список инструментов пополняется [12–14]. Инструменты статической верификации реализуют различные подходы, что позволяет применять их для доказательства выполнимости различных специфицированных свойств. Среди этих инструментов нет однозначного лидера, так как они используют разные техники и нацелены на разные классы ошибок. Поэтому в долгосрочной перспективе важно иметь возможность использовать различные инструменты статической верификации.

В данной статье будут рассматриваться только те инструменты статической верификации, которые позволяют проверять программные системы на языке Си, поскольку модули ядра большинства ОС разрабатываются на языке Си.

1.2. Особенности процесса разработки ядра различных ОС

При разработке контрактных спецификаций и проведении статической верификации модулей необходимо учитывать особенности процесса разработки ядра различных ОС.

Ядро ОС Microsoft Windows разрабатывается централизованно. При этом выделяются большие ресурсы на развитие и применение новых технологий, в частности, для обеспечения качества посредством статической верификации. Большое количество модулей ядра ОС Microsoft Windows разрабатываются и поддерживаются исключительно производителями соответствующего аппаратного обеспечения. Это в том числе означает, что только разработчики имеют доступ к исходному коду модулей, необходимому для проведения статической верификации. В связи с этим часто сами разработчики выполняют статическую верификацию, анализируют получающиеся результаты и при необходимости могут провести доработку контрактных спецификаций. Программный интерфейс между модулями и ядром ОС Microsoft Windows является стабильным[1], что позволяет использовать одни и те же контрактные спецификации для разных версий ядра.

[1] Документация по Windows Driver Frameworks: http://msdn.microsoft.com/en-us/Library/Windows/Hardware/ff557565%28v=vs.85%29.aspx.

Ядро ОС Linux поставляется вместе с большим количеством модулей (около 4 тыс. в последних версиях) в виде исходного кода. В подготовке всех новых версий ядра принимают участие более 1000 разработчиков из более 200 организаций, рассредоточенных по всему миру [15]. При этом разработчики сами не пишут контрактные спецификации. Программный интерфейс между модулями и ядром ОС Linux не является стабильным[2] – это осложняет задачу разработки и поддержки контрактных спецификаций, поскольку при выходе новых версий ядра может потребоваться их переработка.

Особенности разработки ядра других ОС не рассматриваются в рамках данной статьи, поскольку авторам неизвестно об опыте применения инструментов статической верификации для них.

1.3. Существующие системы статической верификации модулей ядра различных ОС

Для того чтобы автоматизировать процесс статической верификации модулей ядра различных ОС, разрабатываются *системы статической верификации*. На сегодняшний день до уровня промышленного использования доросла единственная система статической верификации Static Driver Verifier (SDV) [16], которая была разработана в компании Microsoft. Данная система позволяет проверять модули ядра ОС Microsoft Windows различных версий на предмет выполнения правил корректного использования программного интерфейса ядра с помощью инструмента статической верификации SLAM [17].

Информацию о составе и опциях сборки модулей SDV получает автоматически на основе оригинальных скриптов, описывающих сборку. В данной системе статической верификации часть модели окружения, описывающая возможные сценарии взаимодействия ядра и модулей, генерируется автоматически на основе спецификаций, которые разработчики SDV задали для всех типов модулей, и аннотаций к модулям, которые пишутся вручную. Модель программного интерфейса ядра ОС Microsoft Windows, используемого модулями, входит в состав SDV. Спецификации правил корректного использования программного интерфейса ядра ОС пишутся на языке Specification Language for Interface Checking (SLIC) [18]. В настоящее время SDV поставляется с набором, включающим около 200 спецификаций правил. Исследовательская версия SDV [19] позволяет добавлять свои спецификации правил, а также использовать инструмент статической верификации Yogi. Большое внимание разработчики SDV уделили автоматизации запуска процесса статической верификации модулей ядра и поддержке анализа результатов статической верификации. Пользователям показываются как суммарные результаты статической верификации для анализируемых модулей по всем проверяемым правилам, так и визуализированные *трассы ошибок* (пути в исходном коде, на которых возможны нарушения проверяемых правил). По состоянию на 2010 год с помощью SDV было выявлено 270 ошибок в модулях ядра, входящих в поставку ОС Microsoft Windows.

Для модулей ядра ОС Linux были разработаны две системы статической верификации: DDVerify (Университет Карнеги-Меллон, Питтсбург, США) [20] и Avinux (Университет Эберхарда и Карла, Тюбинген, Германия) [21].

DDVerify использует собственные скрипты сборки для извлечения информации о составе и опциях сборки анализируемых модулей. Контрактные спецификации в этой системе статической верификации задаются полностью вручную на языке программирования Си. Разработчики DDVerify сделали модель окружения для четырех типов модулей, а также для обработчиков аппаратных прерываний, таймеров и т.д. Кроме того, непосредственно в коде модели окружения они задали восемь спецификаций правил корректного использования примитивов синхронизации и правил корректной инициализации переменных до их использования. DDVerify позволяет проверять модули с помощью двух инструментов статической верификации CBMC [22] и SATABS [23]. Для упрощения анализа трасс ошибок в состав системы статической верификации входит плагин для интегрированной среды разработки Eclipse.

[2] Программный интерфейс ядра ОС Linux для драйверов: http://www.kernel.org/doc/Documentation/stable_api_nonsense.txt.

Система статической верификации Avinux предоставляет возможность автоматизированным образом проверять единичные препроцессированные файлы модулей ядра, для чего в ней осуществляется встраивание в процесс сборки ядра путем модификации оригинальных скриптов, описывающих сборку. Модель окружения в Avinux задается практически полностью вручную (автоматически вызываются только экспортируемые функции модулей, для параметров которых генерируется код инициализации). Спецификации правил корректного использования программного интерфейса ядра ОС пишутся на расширении языка SLIC. В Avinux поддерживается проверка правил корректного использования примитивов синхронизации и правил корректной работы с памятью. Система статической верификации была интегрирована с единственным инструментом статической верификации CBMC. Avinux реализован в виде плагина для Eclipse, что позволяет запускать данную систему автоматизированным образом. Но пользователю выдаются трассы ошибок в формате CBMC, что существенно затрудняет их анализ.

Ни одна из рассмотренных систем статической верификации не учитывает особенности процесса разработки ядра ОС Linux в полной мере. Система SDV предназначена только для статической верификации модулей ядра ОС Microsoft Windows, программный интерфейс которого не изменяется с течением времени. DDVerify требует переработки собственного процесса сборки, заголовочных файлов ядра и контрактных спецификаций для каждой новой версии ядра. Кроме того, в данной системе статической верификации была реализована модель окружения только для четырех типов модулей из нескольких сотен. Avinux не поддерживает автоматическую проверку модулей, состоящих из нескольких файлов, требует задавать большую часть модели окружения вручную и также вручную следить за актуальностью спецификаций правил. Обе системы статической верификации модулей ядра ОС Linux не продемонстрировали значительных результатов на практике, и их поддержка была прекращена несколько лет назад.

Ни одна из рассмотренных систем статической верификации не поддерживает интеграцию сторонних инструментов статической верификации. Это особенно важно при проверке модулей ядра ОС Linux, поскольку, с одной стороны, для проведения анализа доступно большое количество инструментов статической верификации, реализующих принципиально различные подходы, а с другой стороны, в отличие от Microsoft SDV у разработчиков систем статической верификации нет достаточного количества ресурсов на то, чтобы одновременно вести полноценную поддержку какого-то выделенного инструмента статической верификации.

Существующие системы статической верификации не предоставляют средств для сравнительного анализа результатов верификации и автоматизированной разметки сообщений об ошибках, что важно, поскольку проверяемых модулей ядра ОС Linux, спецификаций правил, инструментов статической верификации и их конфигураций может быть достаточно много, а кроме того, со временем они развиваются.

В связи с этим разработка нового метода и системы статической верификации модулей ядра ОС Linux является актуальной задачей.

1.4. План статьи

Во втором разделе данной статьи представлен метод статической верификации модулей ядра ОС Linux. Показывается, что метод позволяет конфигурировать процесс проверки на каждом из его этапов. Кроме того, описываются модификации метода, необходимые для его использования с целью верификации компонентов ядра других ОС. Архитектура разработанной конфигурируемой системы статической верификации модулей ядра ОС Linux рассмотрена в третьем разделе. Четвертый раздел посвящен результатам практического применения данной системы. В данном разделе анализируются выявленные ошибки в модулях ядра ОС Linux, причины ложных сообщений об ошибках и пропуска ошибок, время проведения статической верификации и причины неуспешного завершения работы системы статической верификации. Помимо этого, в четвертом разделе рассматриваются возможности разработанной конфигурируемой системы статической верификации для сравнения инструментов статической верификации

и их конфигураций. В заключении подводятся итоги, а также представляются направления дальнейшего развития.

2. МЕТОД СТАТИЧЕСКОЙ ВЕРИФИКАЦИИ МОДУЛЕЙ ЯДРА ОС LINUX

Предлагаемый в данной статье метод статической верификации модулей ядра ОС Linux состоит из нескольких шагов, которые описаны в подразделах 2.1–2.5. В подразделе 2.6 описываются модификации метода, которые позволяют применять его для верификации компонентов ядра других ОС.

2.1. Первоначальная подготовка исходного кода к статической верификации

На первом шаге метода осуществляется первоначальная подготовка исходного кода ядра и модулей ОС Linux к статической верификации. В подразделе 1.1 было отмечено, что современные инструменты статической верификации не позволяют проверять ядро ОС целиком. В связи с этим возникает задача разделить исходный код ядра и модулей ОС Linux и соответствующие им команды компиляции и компоновки, описывающие правила сборки, таким образом, чтобы размер получаемых *объектов верификации* был ограничен несколькими тысячами или десятками тысяч строк, но при этом данные объекты включали в себя как можно больше кода, участвующего в реализации функциональности соответствующих модулей. Например, можно выделять в отдельный объект верификации файлы с исходным кодом, которые непосредственно составляют анализируемый модуль, и соответствующие им команды сборки[3]. Модуль ядра может вызывать функции ядра, а также функции из других модулей, поэтому соответствующий объект можно дополнить кодом этих функций.

В предлагаемом методе объекты верификации готовятся автоматически на основе оригинальных скриптов сборки ядра ОС Linux, для чего они модифицируются таким образом, чтобы наряду с собственно сборкой ядра и модулей выводилась информация о соответствующих командах компиляции и компоновки.

[3]Это можно сделать на основе команд компоновки, которые соответствуют модулям, поскольку выходные файлы команд компиляции файлов с исходным кодом являются входными файлами для команд компоновки, в том числе, команд компоновки модулей.

2.2. Построение модели окружения

Для того чтобы проверить получаемые на первом шаге объекты верификации на предмет выполнения правил корректного использования программного интерфейса ядра ОС Linux, на втором шаге метода для этих объектов строится модель окружения и задается точка входа для инструментов статической верификации.

В типичном модуле ядра ОС Linux определяются функция инициализации, обработчики (обработчики инициализации и отключения устройств, обработчики открытия, чтения, записи и закрытия файлов, обработчики аппаратных прерываний и т.д.) и функция выхода. Функция инициализации вызывается при загрузке модуля ядром. Данная функция регистрирует обработчики модуля, которые затем вызываются ядром при необходимости для обработки системных вызовов от пользовательских приложений, аппаратных прерываний и внутренних событий ядра. Перед выгрузкой модуля вызывается функция выхода, в которой происходит освобождение выделенных для модуля ресурсов и дерегистрация обработчиков модуля.

В предлагаемом методе статической верификации модулей ядра ОС Linux множество сценариев взаимодействия ядра и модулей, которые возможны при их работе в реальном окружении, описывается с помощью π-процессов [24, 25]. Это позволяет задавать модель окружения как на детальном уровне для определенных групп обработчиков модулей, так и в виде шаблонов, под которые попадает большинство оставшихся групп. В шаблонах может быть описано, например, что обработчики модулей могут вызываться только после успешного вызова функции инициализации, что отключение устройства может быть выполнено только после его успешной инициализации, а читать из файла можно только после его открытия. Благодаря этому спецификацию π-модели окружения для всех типов модулей ядра ОС Linux удается задать достаточно компактным образом. Кроме того, одна и та же специфи-

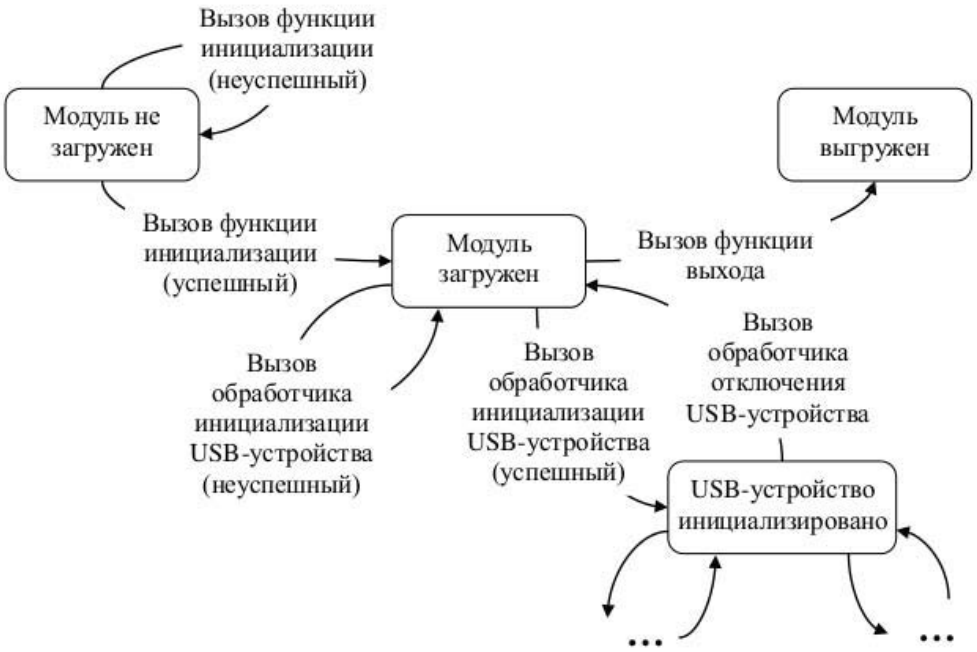

Рис. 1. Схематичное описание модели окружения для драйвера USB-устройства.

кация π-модели окружения может быть использована для различных версий ядра, поскольку в шаблонах не указываются ни конкретные обработчики модулей, ни даже конкретные группы обработчиков. В качестве примера на рис. 1 показано схематичное описание модели окружения для драйвера USB-устройства.

Пользователь может дополнить спецификацию π-модели окружения, например, описать точно определенную группу обработчиков модулей, для которой модель окружения строилась на основе шаблона, если выяснится, что некоторая известная ошибка пропускается из-за того, что модель задает не все возможные сценарии взаимодействия ядра и модулей, или выдается ложное сообщение об ошибке вследствие того, что модель допускает сценарии, невозможные в реальном окружении.

Подбирать и заполнять шаблоны из спецификации π-модели окружения предлагается автоматически на основе анализа исходного кода, составляющего полученные на предыдущем шаге объекты верификации. Для каждого объекта может быть построено несколько моделей окружения, например, для разных групп обработчиков модулей. С целью того, чтобы впоследствии использовать различные инструменты статической верификации, каждая полученная модель окружения транслируется в функцию на языке Си, из которой вызываются обработчики объекта верификации в том виде и в той последовательности, в которых это происходит при его работе в реальном окружении. Эта функция добавляется к исходному коду объекта верификации и в дальнейшем используется, как точка входа для инструментов статической верификации.

2.3. Построение спецификаций правил корректного использования программного интерфейса ядра ОС Linux

На третьем шаге метода для тех элементов программного интерфейса ядра ОС Linux, которые релевантны специфицируемым правилам, предлагается разработать модель[4], а на основании состояния этой модели и этих правил – задать предусловия данных элементов программного интерфейса.

Подробно процесс построения спецификаций правил корректного использования программного интерфейса ядра ОС Linux описан в диссертации [26]. Там же представлено аспектно-ориентированное расширение языка программирования Си, которое используется

[4]Эта модель является частью модели окружения для модулей ядра ОС Linux.

для задания спецификаций. На рис. 2 представлен пример спецификации правил корректного выделения и освобождения блоков-запросов для USB-устройств (USB Request Blocks – URB), который наглядным образом показывает составные части спецификаций и их взаимосвязь друг с другом и кодом анализируемых модулей.

Следует отметить, что в предлагаемом методе статической верификации модулей ядра ОС Linux сигнатуры элементов программного интерфейса ядра, для которых разрабатывается модель и задаются предусловия, описываются полностью (см. определения USB_ALLOC_URB и USB_FREE_URB на рис. 2). Благодаря этому возможно автоматизированным образом поддерживать согласованность спецификаций правил с программным интерфейсом ядра и его реализацией, поскольку при существенных изменениях в реализации ядра ОС Linux разработчики, как правило, изменяют соответствующие элементы программного интерфейса.

Пользователь может дорабатывать спецификации правил корректного использования программного интерфейса ядра ОС Linux, если окажется, что они недостаточно точные и приводят к пропуску известных ошибок или к ложным сообщениям об ошибках. Также пользовать может предоставить собственные спецификации правил.

На основе спецификаций правил в методе предлагается инструментировать исходный код, полученный на втором шаге, таким образом, чтобы нарушения правил корректного использования программного интерфейса ядра ОС Linux соответствовали достижимости метки LDV_ERROR от заданных точек входа. В результате этого получаются *верификационные задачи* – объекты верификации с поставленными на них задачами достижимости.

2.4. Запуск инструментов статической верификации

На четвертом шаге метода на полученных верификационных задачах запускаются инструменты статической верификации для того, чтобы выявить нарушения проверяемых правил или доказать корректность соответствующих модулей относительно данных правил. В методе предлагается запускать инструменты посредством адаптеров. Адаптер для некоторого инструмента статической верификации выполняет следующее:

- Готовит составляющие объекты верификации Си-файлы, например, применяет к каждому из них стандартный препроцессор языка Си, обрабатывает файлы с помощью инструмента трансформации кода CIL [27], или соединяет все файлы в один с помощью того же CIL (все инструменты статической верификации принимают на вход препроцессированные файлы с исходным кодом, а некоторые инструменты могут анализировать только один файл за раз).

- Запускает инструмент статической верификации, передавая подготовленные Си-файлы и необходимую конфигурацию и ограничивая потребляемые инструментом ресурсы, такие как процессорное время и оперативная память (это необходимо, поскольку инструменты статической верификации могут работать неприемлемо долго и/или потреблять чрезмерно большое количество памяти).

- Обрабатывает статус завершения работы инструмента статической верификации (например, завершен успешно, или завершен по сигналу в связи с превышением ограничения по памяти, или завершен некорректно из-за ошибки в инструменте) и подсчитывает потребленные инструментом ресурсы.

- При неуспешном завершении инструмент статической верификации не выносит определенного вердикта, поэтому в данном случае адаптер должен сгенерировать вердикт *Unknown* и указать причины неуспешного завершения работы, которые он может определить на основе анализа вывода инструмента и информации о статусе завершения работы инструмента.

- В случае успешного завершения инструмент статической верификации либо гарантирует отсутствие нарушений проверяемых правил (выносит вердикт *Safe*), либо говорит об обнаружении возможного нарушения правила

```
#include <linux/usb.h>
#include <verifier/rcv.h>

/* Множество указателей на структуры urb, под
которые для модуля была выделена память. Тип
данных множество set, а также операции с
объектами данного типа определены в
заголовочном файле verifier/rcv.h. */
set URBS = empty;

/* Модельные функции для выделения и
освобождения памяти под структуры urb.
Модельная функция выделения памяти ldv_alloc
определена в заголовочном файле verifier/rcv.h.
*/
struct urb *
ldv_usb_alloc_urb(void)
{
  void *urb;
  urb = ldv_alloc();
  if (urb) {
    add(URBS, urb);
  }
  return urb;
}
void
ldv_usb_free_urb(struct urb *urb)
{
  if (urb) {
    remove(URBS, urb);
  }
}
```

```
/* Описание множества точек использования
элементов программного интерфейса ядра. */
pointcut USB_ALLOC_URB: call(
  struct urb *usb_alloc_urb(int, gfp_t))
pointcut USB_FREE_URB: call(
  void usb_free_urb(struct urb *))

/* Привязка вызовов модельных функций к точкам
использования элементов программного интерфейса. */
around: USB_ALLOC_URB {
  return ldv_usb_alloc_urb();
}
around: USB_FREE_URB {
  ldv_usb_free_urb($arg1);
}

/* Предусловия элементов программного интерфейса
ядра. Макрофункция ldv_assert определяется в
заголовочном файле verifier/rcv.h следующим образом:
#define ldv_assert(e) (e ? 0 : ldv_error())
static inline void ldv_error(void) {
  LDV_ERROR: goto LDV_ERROR;
}
LDV_ERROR является меткой, достижимость которой
равнозначна нарушению проверяемых правил.
*/
before: USB_FREE_URB {
  ldv_assert(contains(URBS, $arg1));
}
after: MODULE_EXIT {
  ldv_assert(is_empty(URBS));
}
```

Рис. 2. Пример спецификации правил корректного выделения и освобождения блоков-запросов для USB-устройств.

(выносит вердикт *Unsafe*) – адаптер должен определить соответствующий вердикт на основе вывода инструмента.

- При вынесении вердикта *Unsafe* вывод инструмента статической верификации сопровождается трассой ошибки, формат которой у разных инструментов может существенно отличаться, поэтому с целью упрощения последующего анализа результатов статической верификации адаптер должен преобразовывать все трассы ошибок к общему формату (см. следующий подраздел).

Для интеграции сторонних инструментов статической верификации пользователю необходимо разработать соответствующий адаптер. При этом можно использовать части адаптеров для других инструментов, поскольку они могут быть устроены схожим образом.

2.5. *Анализ результатов статической верификации*

На заключительном шаге рассматриваемого метода предлагается автоматизировать анализ результатов статической верификации модулей ядра ОС Linux в нескольких направлениях.

Во-первых, упростить анализ трасс ошибок, выдаваемых инструментами статической верификации, – данный вид анализа необходимо выполнять для того, чтобы разобраться в причинах

выявленных ошибок, а также, чтобы определить, какие сообщения об ошибках являются ложными. С этой целью предлагается сначала преобразовывать все трассы ошибок к общему формату (это делается адаптером для соответствующего инструмента статической верификации), а затем единообразно визуализировать трассы, представленные в общем формате, с привязкой к соответствующему исходному коду анализируемых модулей и ядра ОС Linux.

Во-вторых, автоматизированно размечать сообщения об ошибках (указывать была ли найдена ошибка или сообщение об ошибке является ложным), если соответствующие трассы ошибок оказываются похожими на уже проанализированные трассы ошибок, например, имеют одинаковые деревья или стеки вызываемых функций. Благодаря этому можно существенно сократить трудозатраты на анализ результатов. Например, если инструмент статической верификации выдает ложное сообщение об ошибке при проверке модуля некоторой версии ядра ОС Linux, то, как правило, он выдает достаточно похожую трассу ошибки при проверке данного модуля в следующей версии ядра, и благодаря автоматизированной разметке сообщений об ошибках можно не проводить анализ повторно.

В-третьих, предоставить статистику и возможность сравнения результатов статической верификации, что необходимо, поскольку проверяемых модулей ядра ОС Linux, спецификаций правил, инструментов статической верификации и их конфигураций достаточно много, а кроме того, со временем они развиваются.

В-четвертых, обеспечить возможность совместного анализа результатов статической верификации без существенных трудозатрат на организацию взаимодействия.

2.6. Адаптация метода с целью верификации компонентов ядра других ОС

Предложенный в данном разделе метод может быть применен для верификации компонентов ядра других ОС. С этой целью для целевой ОС необходимо:

- Предложить подход к подготовке объектов верификации. Получать информацию о составе и опциях сборки компонентов ядра можно на основе оригинальных скриптов сборки, так же как и для модулей ядра ОС Linux.
- Построить модель окружения для анализируемых компонентов ядра ОС. Для этого нужно изучить особенности взаимодействия компонентов между собой и с окружением ОС и сформулировать ограничения на множество сценариев взаимодействия между ними. Строить модель окружения, опираясь на данные ограничения, можно вручную либо на основе некоторого формального описания.
- Задать спецификации правил, соответствующих ошибкам искомого вида. Например, для поиска нарушений правил корректного использования программного интерфейса ядра ОС сначала нужно определить релевантные правилам элементы программного интерфейса, а затем разработать модель и задать предусловия данных элементов программного интерфейса.

Дальнейшие шаги, интеграция и запуск инструментов статической верификации и анализ результатов статической верификации, остаются такими же, как и для модулей ядра ОС Linux.

2.7. Выводы

В предложенном методе статической верификации модулей ядра ОС Linux можно конфигурировать процесс проверки на каждом из его этапов:

- При подготовке объектов верификации можно выбрать, каким образом разделить исходный код модулей и ядра и соответствующие им команды сборки.
- Можно дополнить спецификацию π-модели окружения.
- Можно уточнить существующие спецификации правил корректного использования программного интерфейса ядра, а также добавить собственные спецификации.
- Для проведения анализа можно задать особые конфигурации и ограничения на потребляемые ресурсы для интегрированных

инструментов статической верификации, а кроме того, можно интегрировать сторонние инструменты статической верификации.

Возможность конфигурируемости послужила одной из причин, которая позволила довести реализующую метод систему до достаточно зрелого уровня. В настоящее время она продолжает активно развиваться, а с ее помощью уже было выявлено более 150 критичных ошибок в модулях ядра ОС Linux.

3. КОНФИГУРИРУЕМАЯ СИСТЕМА СТАТИЧЕСКОЙ ВЕРИФИКАЦИИ LINUX DRIVER VERIFICATION TOOLS

Предложенный метод статической верификации модулей ядра ОС Linux был реализован в конфигурируемой системе статической верификации Linux Driver Verification Tools (LDV Tools) [28].

3.1. Архитектура Linux Driver Verification Tools

Архитектура LDV Tools представлена на рис. 3. Компоненты и подкомпоненты LDV Tools, а также инструменты статической верификации изображены в центре в порядке их вызова. Слева изображены входные данные для конфигурируемой системы статической верификации. Справа показан порядок формирования отчета по результатам статической верификации, загрузка отчета в базу данных LDV и компонент LDV Analytics Center, предназначенный для автоматизации анализа результатов статической верификации.

3.2. LDV Core

Процесс статической верификации модулей ядра ОС Linux начинается с запуска компонента LDV Core. Данный компонент вызывает подкомпонент Kernel Manager, который создает на диске копию ядра ОС Linux, предоставленного пользователем в виде архива, директории или Git-репозитория. В дальнейшем все компоненты и подкомпоненты LDV Tools работают только с копией ядра. Kernel Manager модифицирует оригинальные скрипты сборки ядра, чтобы впоследствии получить информацию о составе модулей и опциях их сборки.

Далее LDV Core вызывает Build Command Extractor, который запускает сборку ядра ОС Linux. По мере выполнения сборки данный подкомпонент перехватывает поток команд компиляции и компоновки файлов ядра. По этим командам Build Command Extractor выделяет те файлы с исходным кодом, которые относятся к верифицируемым модулям.

Получаемые команды компиляции и компоновки передаются подкомпоненту Command Stream Divider, который формирует на их основе объекты верификации. В настоящее время каждый объект верификации соответствует ровно одному модулю ядра, благодаря чему в дальнейшем обрабатываются и анализируются все файлы с исходным кодом, которые составляют модули. В будущем планируется составлять объекты для групп взаимосвязанных модулей, а также дополнять их тем кодом ядра, от которого зависят анализируемые модули. Пользователю будет предоставлена возможность выбирать ту или иную стратегию разделения исходного кода ядра и модулей ОС Linux и соответствующих им команд сборки на объекты верификации.

Каждый из подготовленных объектов верификации LDV Core подает на вход подкомпоненту Driver Environment Generator [29]. Также данному подкомпоненту на вход поступает спецификация π-модели окружения, входящая в состав LDV Tools. При необходимости пользователь может дополнить данную спецификацию.

Driver Environment Generator определяет функцию инициализации, обработчики и функцию выхода по исходному коду, входящему в объекты верификации. Затем, на основе этих данных и спецификации π-модели окружения для каждого объекта строится промежуточное представление модели окружения, которое Driver Environment Generator транслирует в исходный код на языке Си, который добавляется к коду объектов верификации.

3.3. Domain Specific C Verifier

Компонент Domain Specific C Verifier связывает объекты верификации с указанными спецификациями правил и запускает на них указанный

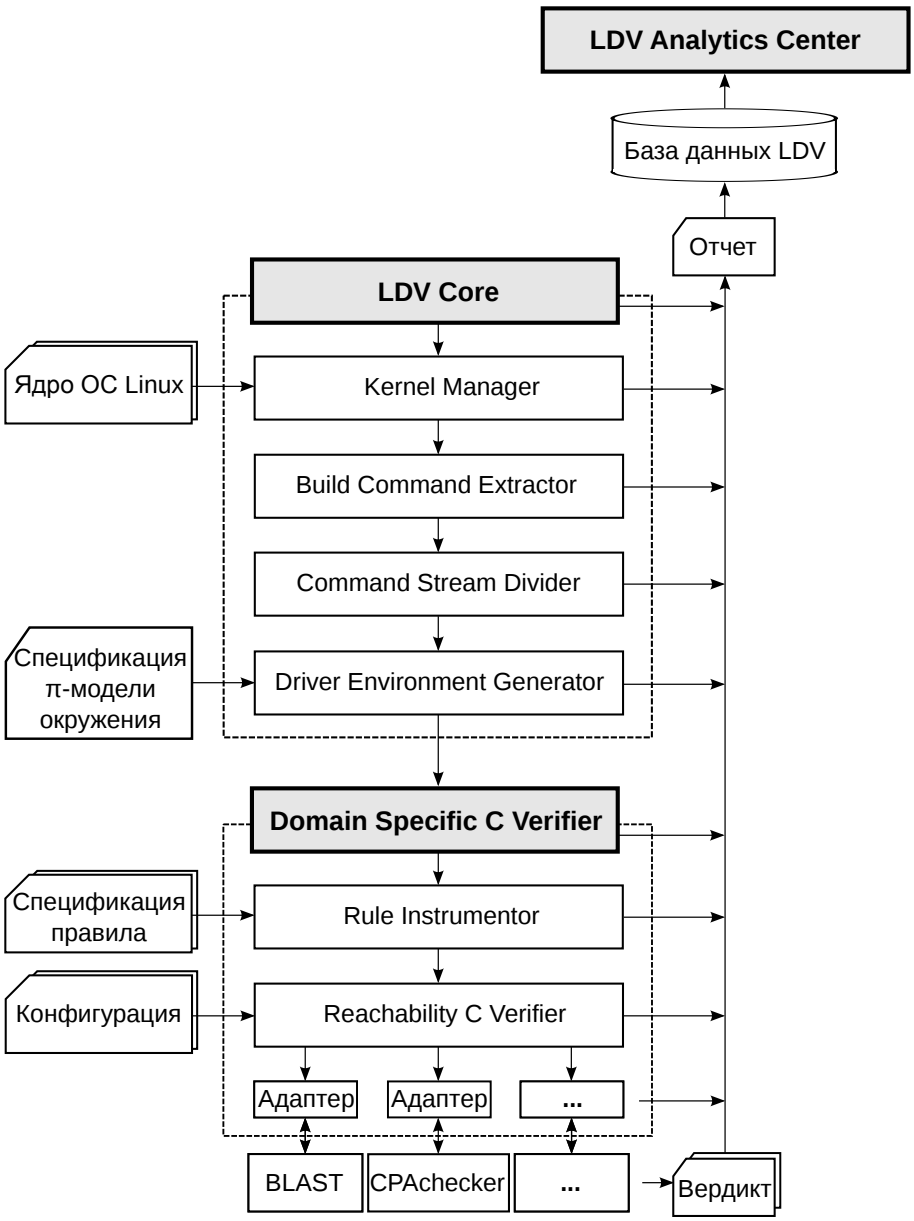

Рис. 3. Архитектура системы статической верификации Linux Driver Verification Tools.

инструмент статической верификации в заданной конфигурации.

Спецификации правил, доступные для проверки, хранятся в базе, где каждой спецификации соответствует уникальный идентификатор. При необходимости пользователь может использовать свою базу спецификаций правил, добавить в существующую базу свои собственные спецификации правил либо модифицировать существующие спецификации.

Для каждого из указанных идентификаторов спецификаций правил Domain Specific C Verifier вызывает Rule Instrumentor. Данный подкомпонент по идентификатору извлекает из базы информацию о проверяемой спецификации, на основе которой он инструментирует файлы, входящие в объект верификации, так, чтобы была поставлена задача достижимости для инструментов статической верификации (нахождение решения данной задачи соответствует возможному нарушению проверяемых правил).

Для проведения статической верификации используются либо те инструмент и конфигурация, которые указаны пользователем, либо, при отсутствии данных указаний, те, которые используются в LDV Tools по умолчанию. Взаимо-

#	Task	Kernel	Rule	Total	Safe	Unsafe	Unknown	Verdicts		
								True	False	?
1	Task description BLAST, 15Gb	linux-3.13-rc1	08_1a	5994	5004	92	898	2	79	11
2			101_1a	5994	5112	1	881	-	1	-
3			106_1a	5994	5008	23	963	2	20	1
4			10_1a	5994	5175	8	811	-	8	-
5			118_1a	5994	5115	13	866	-	12	1
6			129_1a	5994	5028	7	959	2	5	-
7			132_1a	5994	5055	42	897	13	14	15
8			134_1a	5994	5074	3	917	3	-	-
9			146_1a	5994	5028	11	955	-	1	10
10			147_1a	5994	5017	21	956	3	1	17
11			150_1a	5994	5119	3	872	1	2	-
12			32_7a	5994	5037	81	876	8	70	3
13			39_7a	5994	5050	76	868	5	58	13
14			68_1	5994	4867	119	1008	4	115	-
15			77_1a	5994	5184	1	809	1	-	-

Рис. 4. Результаты статической верификации, сгруппированные по версии ядра ОС Linux (3.13-rc1) и по идентификаторам спецификаций правил.

действие с инструментом статической верификации осуществляет подкомпонент Reachability C Verifier, который вызывает инструмент статической верификации посредством соответствующего адаптера. Данному адаптеру передаются верификационная задача, конфигурация и ограничения на ресурсы для инструмента статической верификации.

На сегодняшний день в состав конфигурируемой системы статической верификации LDV Tools входят адаптеры для таких инструментов статической верификации, как BLAST [30] (используется по умолчанию), CPAchecker [31], UFO [32] и CBMC [22]. Кроме того, пользователь может интегрировать в LDV Tools сторонние инструменты статической верификации путем реализации адаптеров для этих инструментов.

3.4. Формирование итогового отчета и его загрузка в базу данных LDV

Вердикты, которые выдает инструмент статической верификации, обрабатываются всеми компонентами и подкомпонентами LDV Tools в обратном порядке (рис. 3). При этом на каждом этапе промежуточный отчет дополняется информацией о работе соответствующих компонентов и подкомпонентов. В итоге формируется финальный отчет с результатами статической верификации модулей ядра ОС Linux, который может быть загружен в базу данных LDV.

3.5. LDV Analytics Center

LDV Analytics Center – это компонент конфигурируемой системы статической верификации LDV Tools, который позволяет автоматизированным образом проводить несколько видов анализа результатов статической верификации, загруженных в базу данных LDV:

- Статистический анализ (можно сгруппировать результаты статической верификации, например, по версии ядра ОС Linux, идентификаторам спецификаций правил и т.д. – рис. 4).
- Анализ результатов статической верификации, полученных для конкретных спецификаций, модулей и т.д. – рис. 5.
- Сравнительный анализ (например, можно сравнить результаты статической верификации, полученные для разных версий ядра ОС Linux, – рис. 6).

С целью упростить анализ трасс ошибок LDV Analytics Center использует подкомпонент Error

Task	Kernel	Rule	Module	Entry point	Verifier	Error trace	KB Verdict	KB Tags
Task description BLAST, 15Gb	linux-3.13-rc1	106_1a	drivers/base/firmware_class.ko	ldv_main0	blast	...	False positive	env_gen
			drivers/char/ppdev.ko	ldv_main0	blast	...	False positive	multi_module
			drivers/hid/hid.ko	ldv_main1	blast	...	False positive	kernel_model
			drivers/hsi/clients/hsi_char.ko	ldv_main0	blast	...	False positive	env_gen
			drivers/iio/industrialio.ko	ldv_main0	blast	...	False positive	rule_model
			drivers/infiniband/core/ib_ucm.ko	ldv_main0	blast	...	False positive	init_global_var
			drivers/media/pci/ddbridge/ddbridge.ko	ldv_main0	blast	...	False positive	multi_module
			drivers/media/usb/pvrusb2/pvrusb2.ko	ldv_main5	blast	...	True positive	obsolete
			drivers/mtd/ubi/ubi.ko	ldv_main3	blast	...	False positive	pointer_analysis
				ldv_main4	blast	...	False positive	env_gen
			drivers/net/wireless/mac80211_hwsim.ko	ldv_main0	blast	...	True positive	new
			drivers/scsi/dpt_i2o.ko	ldv_main0	blast	...	False positive	init_global_var
			drivers/scsi/scsi_mod.ko	ldv_main0	blast	...	False positive	rule_model

Рис. 5. Анализ сообщений об ошибках, выдаваемых инструментом статической верификации BLAST при проверке спецификации правил, которые описывают корректную регистрацию USB-устройств класса Gadget (справа показаны результаты разметки сообщений об ошибках: *True positive* – выявлена ошибка, *False positive* – ложное сообщение об ошибке).

Kernel	Rule	Total changes	Safe → Unknown	Unsafe → Safe	Unsafe → Unknown	Unknown → Safe	Unknown → Unsafe
linux-3.12-rc1 → linux-3.13-rc1	39_7a	310	23	1	2	75	10

Рис. 6. Сравнение результатов статической верификации модулей ядра ОС Linux версий 3.12-rc1 и 3.13-rc1 (часть ошибок была исправлена – переходы из *Unsafe*, для большего количества модулей удалось доказать корректность – переходы в *Safe* или найти возможные ошибки – переходы в *Unsafe*).

Trace Visualizer [33]. Данный подкомпонент визуализирует трассы ошибок, представленные в общем формате, в виде веб-страниц. Пользователю предоставляется удобная навигация по трассам ошибок и соответствующему им исходному коду модулей, ядра и контрактных спецификаций. Пример визуализированной трассы ошибки приведен на рис. 7.

Также на рис. 7 показан веб-интерфейс Knowledge Base – базы знаний LDV, еще одного подкомпонента LDV Analytics Center. Данный веб-интерфейс позволяет сохранять результаты анализа трасс ошибок инструментов статической верификации. Скрипты, входящие в Knowledge Base, автоматически размечают все новые загружаемые трассы, если они оказываются похожими по тем или иным критериям на ранее сохраненные трассы ошибок. Например, для трассы ошибки, приведенной на рис. 7, вердикт был вынесен автоматически на основе сравнения по идентификатору спецификации правила (*32_7a*), по модулю (*drivers/staging/rtl8188eu/r8188eu.ko*), по точке входа (*ldv_main55*) и по стеку вызываемых функций (*call_stacks_ne*) с трассой ошибкой, проанализированной для ядра ОС Linux более ранней версии.

Сравнение трасс по стеку вызываемых функ-

Рис. 7. Визуализированная трасса ошибки (выявлено нарушение спецификации правил корректного использования мьютексов в одном потоке).

ций в настоящее время используется в Knowledge Base по умолчанию. Пользователь также может сравнивать трассы ошибок по деревьям вызываемых функций или задать свой собственный скрипт сравнения.

Для проведения анализа LDV Analytics Center предоставляет веб-интерфейс, что позволяет использовать его совместно нескольким пользователям. Благодаря этому удается существенно сократить трудозатраты на анализ получаемых результатов статической верификации.

3.6. Выводы

Разработанная система статической верификации LDV Tools допускает конфигурирование на каждом из этапов своей работы. Пользователь может выбрать, каким образом разделить исходный код ядра и модулей и соответствующие ему команды сборки на объекты верификации; дополнить спецификацию π-модели окружения; модифицировать существующие и предоставить собственные спецификации правил; а также указать инструмент статической верификации, его конфигурацию и ограничения на ресурсы, которые он может использовать при проверке модулей ядра ОС Linux. Благодаря этому система статической верификации LDV Tools в полной мере реализует метод, предложенный в разделе 2.

4. РЕЗУЛЬТАТЫ ПРАКТИЧЕСКОГО ПРИМЕНЕНИЯ LINUX VERIFICATION TOOLS

Конфигурируемая система статической верификации LDV Tools применяется на практике более четырех лет. За это время разработчикам LDV Tools удалось выявить более 150 ошибок в модулях ядра ОС Linux, а также определить наиболее существенные проблемы в системе статической верификации и используемых инструментах статической верификации.

4.1. Анализ выявленных ошибок в модулях ядра ОС Linux

На сегодняшний день с помощью LDV Tools было выявлено 150 ошибок, которые были

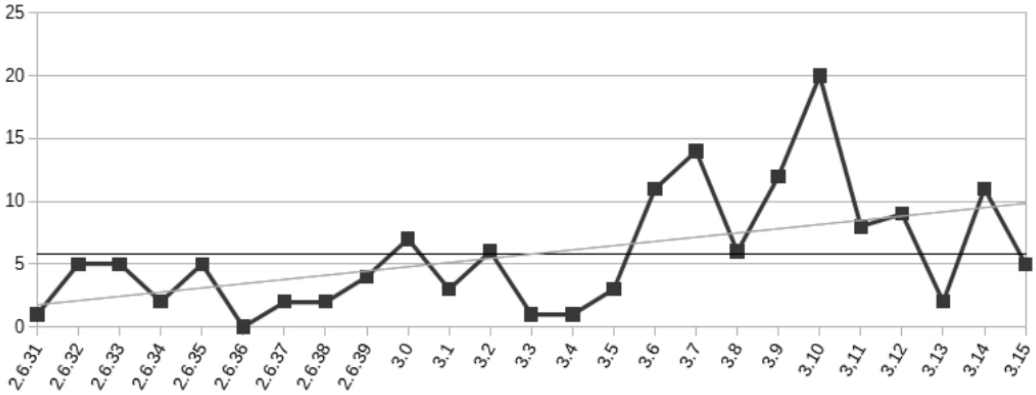

Рис. 8. Зависимость количества исправлений выявленных с помощью LDV Tools ошибок в модулях ядра ОС Linux от версии ядра (горизонтальная линия – среднее число, прямая наклонная линия – линейная регрессия).

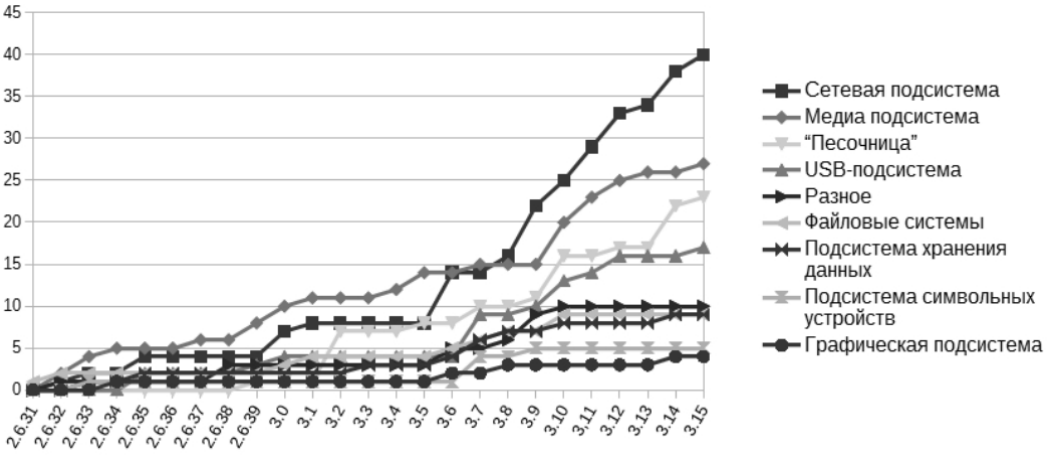

Рис. 9. Зависимость количества исправлений выявленных с помощью LDV Tools ошибок в модулях ядра ОС Linux от версии ядра для различных подсистем ядра.

признаны разработчиками ядра[5] [34]. На рис. 8 представлен график, который демонстрирует, как зависит количество исправлений данных ошибок в модулях ядра ОС Linux от версии ядра. В среднем для ядра версий от 2.6.31 (выпущено 9 сентября 2009) до 3.15-rc2 (выпущено 20 апреля 2014) в каждой версии было исправлено около 6 ошибок. При этом с течением времени количество исправлений ошибок увеличивается. Объясняется это тем, что за четыре года конфигурируемая система статической верификации LDV Tools достигла такого уровня развития, что разработчики применяют ее на практике все более и более активно, а также тем, что постепенно расширяется список проверяемых правил. Сейчас LDV Tools позволяет проверять более 40 правил корректного использования программного интерфейса ядра ОС Linux.

Стоит отметить, что последние 1–2 года с помощью LDV Tools выявляется больше ошибок в модулях ядра ОС Linux, чем разработчики LDV Tools успевают проанализировать и сообщить авторам соответствующих модулей. Это явно свидетельствует о том, что разработанная конфигурируемая система статической верификации имеет достаточно высокий потенциал для выявления новых проблем в модулях.

На рис. 9 методом нарастающего итога представлена зависимость количества исправлений

[5] Кроме ошибок, признанных разработчиками ядра ОС Linux, были выявлены ошибки, которые уже были исправлены к моменту обнаружения, а также ошибки в неподдерживаемых модулях ядра. Эти ошибки не входят в статистику, приводимую в данной статье.

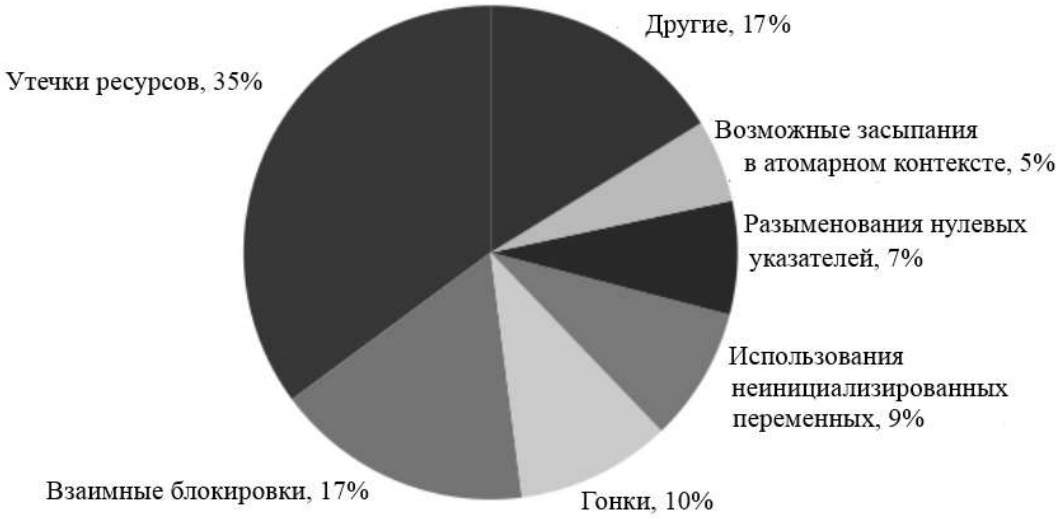

Рис. 10. Распределение возможных последствий выявленных ошибок в модулях ядра ОС Linux.

ошибок в модулях ядра ОС Linux от версии ядра для различных подсистем ядра. Представленные графики демонстрируют, что в целом количество исправляемых ошибок в подсистемах сохраняется пропорциональным. Скачки на графиках объясняются изменением анализируемой кодовой базы. Например, для сетевой подсистемы на ядре версии 3.6 скачок произошел, потому что с помощью LDV Tools помимо модулей, которые представляют драйверы сетевых устройств, стали также верифицировать модули, входящие в основную сетевую подсистему ядра. В подсистему "песочница" попадают новые модули ядра, которые уже имеют достаточно высокий уровень качества, но по тем или иным техническим причинам пока не могут быть перенесены в одну из основных подсистем. Поскольку в новых модулях может выявляться большое количество ошибок, поведение соответствующего графика достаточно непредсказуемое.

Диаграмма на рис. 10 демонстрирует распределение возможных последствий выявленных ошибок. Видно, что в большинстве случаев благодаря предложенным исправлениям удалось избежать утечек ресурсов[6]. На втором и третьем местах идут соответственно взаимные блокировки[7] (как следствие отсутствия освобождения примитивов синхронизации в потоке, в котором они были захвачены), гонки[8] (как следствие освобождения примитивов синхронизации в потоке, в котором они не были захвачены). Стоит отметить, что был исправлен ряд ошибок, которые имеют специфичные для модулей ядра ОС Linux проявления. Например, из-за возможных засыпаний в атомарном контексте может существенно замедлиться работа ядра, а в некоторых случаях даже может зависнуть вся ОС.

Большинство ошибок (около 70%) были обнаружены в коде обработки ошибок, например, после неудачного выделения памяти или проблем при инициализации устройства. Это объясняется тем, что при использовании ОС и при тестировании подобные ситуации происходят достаточно редко, тогда как при статической верификации рассматриваются все возможные пути выполнения.

Практически все ошибки были выявлены с помощью инструмента статической верификации BLAST [30]. Это объясняется тем, что этот инструмент используется в LDV Tools по умолчанию, а также тем, что он был специально оптимизирован для анализа модулей ядра ОС Linux. 4 ошибки были выявлены с помощью инструмента статической верификации CPAchecker [31] (CPAchecker предоставляет дополнительные возможности по анализу функциональных указа-

[6]Определение утечки ресурса. http://cwe.mitre.org/data/definitions/401.html.

[7]Определение взаимной блокировки. http://cwe.mitre.org/data/definitions/833.html.

[8]Определение гонки. http://cwe.mitre.org/data/definitions/362.html.

телей и битовой арифметики по сравнению с BLAST).

В нескольких случаях по результатам обсуждений с разработчиками модулей и ядра ОС Linux исправлений ошибок, найденных с помощью конфигурируемой системы статической верификации LDV Tools, пересматривался дизайн соответствующих подсистем ядра или модулей[9]. Зачастую выявляемые ошибки являлись наведенными вследствие, например, некорректной обработки ошибок[10] или отсутствия инициализации данных[11] (исходно обнаруженные проблемы – утечки памяти).

Другие подходы к обеспечению качества программных систем теоретически смогли бы обнаружить ошибки, выявленные с помощью LDV Tools. Часть этих ошибок могла бы проявиться при использовании ядра ОС Linux либо при проведении тестирования. Однако с одной стороны, для этого может потребоваться достаточно много времени, в том числе для разработки специфичных тестовых сценариев, поскольку большинство ошибок были обнаружены в коде обработки ошибок. А с другой стороны, для таких ошибок, как утечки ресурсов и гонки, бывает достаточно сложно точно идентифицировать их причины.

Большинство выявленных ошибок вероятно можно было бы обнаружить с помощью инструментов, реализующих легковесные подходы статического анализа. Для этого необходимо разработать подходящие для инструментов формальные описания правил корректного использования программного интерфейса ядра ОС Linux, запустить проверку и провести анализ результатов. При этом часть ошибок может быть пропущена (например, несколько десятков ошибок были выявлены при проведении межпроцедурного анализа и при анализе потока управления со сложными зависимостями, что ограниченно поддерживается в легковесных подходах статического анализа), а анализ результатов может быть затруднен из-за большого количества ложных сообщений об ошибках.

4.2. Анализ причин ложных сообщений об ошибках

При проведении экспериментов, результаты которых приведены в данном и трех последующих подразделах, конфигурируемая система статической верификации LDV Tools версии 0.5 запускалась на компьютере с четырехядерным процессором с частотой 3.4 GHz (Intel Core i7-2600), 16 гигабайтами оперативной памяти и ОС Ubuntu 12.04 (ядро ОС Linux версии 3.5, 64-битная архитектура). Для проверки использовался инструмент статической верификации BLAST версии 2.7.2 в конфигурации, используемой по умолчанию. На максимальное количество времени и оперативной памяти, которые мог использовать BLAST на решение одной верификационной задачи, были установлены ограничения 15 минут и 15 гигабайт соответственно.

В таблице 1 приведено распределение ложных сообщений об ошибках по их причинам для трех спецификаций правил. Для составления этого распределения были проанализированы результаты статической верификации всех модулей ядра ОС Linux версии 3.12-rc1 (примерно 4200 модулей[12]). По таблице видно, что модулей, для которых были выданы ложные сообщения об ошибках, не так много относительно общего количества анализируемых модулей (в среднем около 1,5%). Основными причинами ложных сообщений об ошибках являются:

- Нехватка при анализе кода взаимосвязанных модулей, например, когда анализируемый модуль вызывает функции, определенные в других модулях.
- Неточная модель окружения, например, вызовы обработчиков модулей в неправильном порядке.
- Недостаточно точный анализ инструмента статической верификации BLAST, например, вследствие неполноты анализа алиасов.

[9] Обсуждение исправлений ряда ошибок в драйверах сетевых устройств. https://lkml.org/lkml/2012/8/14/128.

[10] Пример существенных исправлений в коде драйвера после обнаружения в нем ошибки. http://linuxtesting.ru/results/report?num=L0130.

[11] Пример исправления инициализации структур драйвера. http://linuxtesting.ru/results/report?num=L0116.

[12] В LDV Tools для одного модуля может генерироваться несколько моделей окружения. В таблице 1 и далее числа приведены с учетом всех моделей окружения для модулей.

Таблица 1. Распределение ложных сообщений об ошибках по их причинам для трех спецификаций правил и всех модулей ядра ОС Linux 3.12-rc1.

Спецификация правила / Причина ложных сообщений об ошибках	Корректное использование мьютексов в одном потоке[13]	Корректное выделение и освобождение блоков-запросов для USB-устройств[14]	Корректная регистрация USB-устройств класса Gadget[15]	Итого
Нехватка при анализе кода взаимосвязанных модулей	4	29	5	38
Неточная модель окружения	24	43	7	74
Неточная спецификация правил	18	6	3	27
Недостаточно точный анализ инструмента статической верификации	17	34	5	56
Итого	63	112	20	195

4.3. Анализ причины пропуска ошибок

Методы статической верификации изначально были нацелены на выявление всех возможных ошибок искомого вида или на доказательство корректности анализируемых программных систем относительно проверяемых правил. Тем не менее по тем или иным причинам возможен пропуск ошибок.

Для оценки количества пропущенных ошибок конфигурируемая система статической верификации LDV Tools запускалась на 34 модулях ядра ОС Linux разных версий, в которых имелись известные нарушения правил корректного использования программного интерфейса ядра. С помощью инструмента статической верификации BLAST удалось обнаружить 16 из этих ошибок, с помощью CPAchecker[16] – 14. BLAST нашел все те же ошибки, которые обнаружил CPAchecker. Для обнаружения 2 ошибок, которые нашел BLAST, CPAchecker не хватило отведенного времени.

Основные причины пропуска остальных ошибок у использованных инструментов статической верификации следующие: недостаток в верификационных задачах кода взаимосвязанных модулей (5), недостаточно точная модель окружения (5), недостаточно точная спецификация правил (3), ошибки в инструментах статической верификации (BLAST: 4, CPAchecker: 3), нехватка памяти (BLAST: 1) и нехватка времени (CPAchecker: 2).

4.4. Анализ времени верификации модулей ядра ОС Linux

Эксперименты показали, что на проверку всех

[13] http://forge.ispras.ru/issues/1940.
[14] http://forge.ispras.ru/issues/3233.
[15] http://forge.ispras.ru/issues/2742.

[16] Ревизия SVN 8244, конфигурация, используемая по умолчанию.

драйверов, которые входят в состав ядра ОС Linux 3.12-rc1 и которые могут быть представлены в виде модулей (примерно 3300), по одной спецификации правил в среднем потребовалось около 25 часов процессорного времени. На проверку всех модулей ядра – около 36 часов процессорного времени. Примерно 70% всего времени анализа заняла работа инструмента статической верификации.

Стоит отметить, что в настоящее время конфигурируемая система статической верификации LDV Tools не работает в параллельном режиме (поскольку инструментам статической верификации требуется большое количество оперативной памяти), поэтому количество ядер процессора практически не влияет на общее время верификации.

4.5. Анализ причин неуспешного завершения работы LDV Tools

Из всех модулей ядра ОС Linux 3.12-rc1 со всеми сгенерированными моделями окружения не удалось верифицировать около 800 (примерно 15% от общего количества) по следующим причинам: ошибки в Command Stream Divider (около 12% от 800), ошибки в Driver Environment Generator (около 11%), ошибки в Rule Instrumentor (около 19%), ошибки в BLAST (около 29%), превышение допустимого ограничения по памяти (около 22%) и по времени (около 7%) у BLAST.

4.6. Возможности LDV Tools для сравнения инструментов статической верификации и их конфигураций

Конфигурируемая система статической верификации LDV Tools позволяет сравнивать инструменты статической верификации и их конфигурации на такой большой, сложной и динамически развивающейся программной системе, как модули ядра ОС Linux [35, 36]. Благодаря этому для проведения статической верификации модулей ядра ОС Linux с целью выявления ошибок удается подобрать инструменты и их конфигурации оптимальным образом с точки зрения уменьшения пропущенных ошибок, количества ложных сообщений об ошибках и потребляемых ресурсов. Кроме того, удается выявить препятствия применения инструментов статической верификации на практике, например, что в инструменте есть некоторые ошибки или выдаваемые им трассы ошибки содержат недостаточно информации для их наглядной визуализации.

Несколько последних лет с помощью конфигурируемой системы статической верификации LDV Tools готовился набор верификационных задач DeviceDrivers64 для ежегодных соревнований, проходящих в рамках мероприятий ETAPS/Competition on Software Verification [12, 13]. При подготовке набора разработчики LDV Tools отобрали те верификационные задачи, при решении которых были выявлены нарушения правил корректного использования программного интерфейса ядра или инструментам статической верификации потребовалось большое количество ресурсов. С 2013 года DeviceDrivers64 стал самым большим набором по количеству задач (более половины от количества всех задач).

В таблице 2 приведены информация о количестве верификационных задач в наборе DeviceDrivers64, победители ежегодных соревнований ETAPS/Competition on Software Verification на наборе DeviceDrivers64 и их итоговые результаты (набранное количество баллов и время работы). Данные результаты показывают, какие инструменты статической верификации необходимо использовать при проведении проверки модулей ядра ОС Linux.

4.7. Выводы

Анализ результатов практического применения конфигурируемой системы статической верификации LDV Tools наглядным образом продемонстрировал, что инструменты статической верификации могут быть успешно использованы для выявления нарушений правил корректного использования программного интерфейса ядра ОС Linux в модулях. Также он позволил определить наиболее существенные проблемы в LDV Tools:

- Недостаток в верификационных задачах кода взаимосвязанных модулей.
- Недостаточно точная спецификация π-модели окружения.

Таблица 2. Победители ежегодных соревнований ETAPS/Competition on Software Verification на наборе верификационных задач DeviceDrivers64.

Набор верификационных задач DeviceDrivers64	1-е место	2-е место	3-е место
SV-COMP 2012 41 задача, максимально 66 баллов	BLAST 55 баллов *1400 с*	CPAchecker-Memo 49 баллов *500 с*	SATabs 32 баллов *3200 с*
SV-COMP 2013 1237 задач, максимально 2419 баллов	UFO 2408 баллов *2500 с*	CPAchecker-Explicit 2340 баллов *9700 с*	BLAST 2338 баллов *2400 с*
SV-COMP 2014 1428 задач, максимально 2766 баллов	BLAST 2682 баллов *13000 с*	UFO 2642 баллов *5700 с*	FrankenBit 2639 баллов *3000 с*

- Недостаточно точные спецификации правил.
- Ошибки в компонентах и подкомпонентах LDV Tools.

и используемых инструментах статической верификации:

- Недостаточно точный анализ алиасов, функциональных указателей, битовой арифметики и т.д.
- Ошибки в инструментах, например, при разборе файлов с исходным кодом, подаваемых им на вход.
- Чрезмерное потребление ресурсов.

Благодаря решению данных проблем в будущем можно существенно сократить количество пропущенных ошибок и ложных сообщений об ошибках, избежать неуспешного завершения работы системы статической верификации, а также оптимизировать время ее работы.

5. ЗАКЛЮЧЕНИЕ

Современные методы и инструменты статической верификации уже позволяют доказать выполнимость специфицированных свойств для средних по размеру программных систем за приемлемое время. Благодаря этому с помощью данных инструментов можно выявить все нарушения правил корректного использования программного интерфейса ядра ОС в модулях. Опыт статической верификации большого количества сложных и активно развивающихся модулей ядра ОС Linux показал важность профессионального решения большого количества инженерных задач, без которого не удается использовать преимущества даже самых передовых методов статической верификации.

В предложенном в работе методе статической верификации можно конфигурировать процесс проверки на каждом из его этапов, благодаря чему реализующая метод конфигурируемая система статической верификации LDV Tools продолжает успешно развиваться, а кроме того уже позволила выявить более 150 критичных ошибок в модулях ядра ОС Linux.

В работе было показано, что модули ядра ОС Linux являются уникальным полигоном для испытания инструментов статической верификации. Это позволяет, с одной стороны, подобрать для проведения верификации оптимальные по ряду критериев инструменты и их конфигурации. С другой стороны, положительные результаты такого комплексного подхода являются новой движущей силой для развития самих методов и инструментов статической верификации.

В ходе практического применения конфигурируемой системы статической верификации LDV Tools были выявлены проблемы, которые определяют основные направления дальнейшего развития:

- Анализ групп взаимосвязанных модулей.
- Дополнение спецификации π-модели окружения.

- Уточнение существующих спецификаций правил.
- Исправление ошибок в компонентах и подкомпонентах LDV Tools.

Кроме того, планируются развивать следующие направления:

- Продолжение анализа изменений в модулях ядра ОС Linux с целью выявления новых критичных правил корректного использования программного интерфейса ядра и разработка новых спецификаций правил.
- Интеграция новых инструментов статической верификации и подбор оптимальных конфигураций инструментов для проверки тех или иных правил.
- Многоаспектная статическая верификация модулей ядра ОС Linux для выявления всех возможных нарушений нескольких проверяемых правил за один запуск инструмента статической верификации.
- Статическая верификация модулей ядра для различных архитектур, таких как ARM, MIPS, S/390 и PowerPC, а также наиболее часто используемых конфигураций ядра.
- Использование и развитие методов регрессионной статической верификации для проверки изменений в коде модулей и ядра ОС Linux.
- Распараллеливание статической верификации модулей ядра ОС Linux.
- Применение предложенного метода статической верификации модулей ядра ОС Linux для проверки компонентов ядра других ОС.

Разработчикам инструментов статической верификации предлагается разрабатывать более точные методы анализа, исправлять ошибки в инструментах, а также оптимизировать реализацию методов с целью уменьшения потребления ресурсов.

Текущие достижения, проблемы, перспективы и планы регулярно обсуждаются на семинарах Linux Driver Verification Workshop и мероприятиях ETAPS/Competition on Software Verification, в которых наиболее активную роль играют Институт системного программирования РАН, Москва (разработчик конфигурируемой системы статической верификации LDV Tools) и Университет Пассау, Германия (основной разработчик инструмента статической верификации CPAchecker).

Авторы выражают признательность П. Андрианову, В. Гратинскому, В.А. Захарову, М. Макиенко, В. Морданю, О. Стрикову, А. Страху, П. Шведу и И. Щепеткову за активное участие в проектировании и разработке конфигурируемой системы статической верификации LDV Tools.

СПИСОК ЛИТЕРАТУРЫ

1. *Chou A., Yang J., Chelf B., Hallem S., Engler D.* An empirical study of operating system errors // Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP). 2001. P. 73–88.
2. *Swift M., Bershad B., Levy H.* Improving the reliability of commodity operating systems // Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP). 2003. P. 73–88.
3. *Palix N., Thomas G., Saha S., Calves C., Lawall J., Muller G.* Faults in Linux: ten years later // Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2011. P. 305–318.
4. *Мутилин В.С., Новиков Е.М., Хорошилов А.В.* Анализ типовых ошибок в драйверах операционной системы Linux // Труды ИСП РАН. 2012. Т. 22. С. 349–374.
5. *Ball T., Bounimova E., Cook B., Levin V., Lichtenberg J., McGarvey C., Ondrusek B., Rajamani S. K., Ustuner A.* Thorough static analysis of device drivers // Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys). 2006. P. 73–85.
6. *Glass R.L.* Facts and fallacies of software engineering. Addison-Wesley Professional, 2002.
7. *Engler D., Chelf B., Chou A., Hallem S.* Checking system rules using system-specific, programmer-written compiler extensions // Proceedings of the 4th conference on Symposium on Operating System Design & Implementation (OSDI). 2000. V. 4. P. 1–16.

8. *Аветисян А., Белеванцев А., Бородин А., Несов В.* Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ // Труды ИСП РАН. 2011. Т. 21. С. 23–38.

9. *Lawall J. L., Brunel J., Palix N., Rydhof H. R., Stuart H., Muller G.* WYSIWIB: A declarative approach to finding API protocols and bugs in Linux code // Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). 2009. P. 43–52.

10. *Мандрыкин М.У., Мутилин В.С., Хорошилов А.В.* Введение в метод CEGAR – уточнение абстракции по контрпримерам // Труды ИСП РАН. 2013. Т. 24. С. 219–292.

11. *Engler D., Musuvathi M.* Static analysis versus model checking for bug finding // Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), LNCS. 2004. V. 2937. P. 191–210.

12. *Beyer D.* Competition on Software Verification // Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS. 2012. V. 7214. P. 504–524.

13. *Beyer D.* Second Competition on Software Verification // Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS. 2013. V. 7795. P. 594–609.

14. *Мандрыкин М.У., Мутилин В.С., Новиков Е.М., Хорошилов А.В.* Обзор инструментов статической верификации Си программ в применении к драйверам устройств операционной системы Linux // Труды ИСП РАН. 2012. Т. 22. С. 293–326.

15. *Corbet J., Kroah-Hartman G., McPherson A.* Linux kernel development. How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It., 2012. http://go.linuxfoundation.org/who-writes-linux-2012

16. *Ball T., Levin V., Rajamani S.K.* A decade of software model checking with SLAM // Communications of the ACM. 2011. V. 54. I. 7. P. 68–76.

17. *Ball T., Bounimova E., Kumar R., Levin V.* SLAM2: Static driver verification with under 4% false alarms // Proceedings of the 10th International Conference on Conference on Formal Methods in Computer-Aided Design (FMCAD). 2010. P. 35–42.

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

19. *Ball T., Bounimova E., Levin V., Kumar R., Lichtenberg J.* The Static Driver Verifier Research Platform // Proceedings of the 22nd International Conference on Computer Aided Verification (CAV), LNCS. 2010. V. 6174. P. 119–122.

20. *Witkowski T., Blanc N., Kroening D., Weissenbacher G.* Model checking concurrent Linux device drivers. Proceedings of the 22nd IEEE/ACM international conference on Automated Software Engineering (ASE). 2007. P. 501–504.

21. *Post H., Kuchlin W.* Integrated static analysis for Linux device driver verification // Proceedings of the 6th International Conference on Integrated Formal Methods (IFM), LNCS. 2007. V. 4591. P. 518–537.

22. *Clarke E., Kroening D., Lerda F.* A tool for checking ANSI-C programs // Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS. 2004. V. 2988. P. 168–176.

23. *Clarke E., Kroening D., Sharygina N., Yorav K.* SATABS: SAT-based predicate abstraction for ANSI-C // Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS. 2005. V. 3440. P. 570–574.

24. *Мутилин В.С.* Верификация драйверов операционной системы Linux при помощи предикатных абстракций. Диссертация на соискание ученой степени кандидата физико-математических наук, ИСП РАН, 2012.

25. *Захаров И.С., Мутилин В.С., Новиков Е.М., Хорошилов А.В.* Моделирование окружения драйверов устройств операционной системы Linux // Труды ИСП РАН. 2013. Т. 25. С. 85–112.

26. *Новиков Е.М.* Развитие метода контрактных спецификаций для верификации модулей ядра операционной системы Linux. Диссертация на соискание ученой степени кандидата физико-математических наук, ИСП РАН, 2013.

27. *Necula G. C., McPeak S., Rahul S.P., Weimer W.* CIL: Intermediate language and tools for analysis and transformation of C programs // Proceedings of the 11th International Conference on Conference on Compiler Construction, LNCS. 2002. V. 2304. P. 213–228.

28. *Мутилин В.С., Новиков Е.М., Страх А.В., Хорошилов А.В., Швед П.Е.* Архитектура Linux Driver Verification // Труды ИСП РАН. 2011. Т. 20. С. 163–187.

29. *Khoroshilov A., Mutilin V., Novikov E., Zakharov I.* Modeling environment for static verification of Linux kernel modules // Proceedings of the 11th International Andrei Ershov Memorial Conference (PSI), 2014.

30. *Beyer D., Henzinger T., Jhala R., Majumdar R.* The software model checker BLAST: Applications to software engineering // International Journal on Software Tools for Technology Transfer (STTT). 2007. V. 5. P. 505–525.

31. *Beyer D., Keremoglu M.E.* CPAchecker: A tool for configurable software verification // Proceedings of the 23rd International Conference on Computer Aided Verification (CAV), LNCS. 2011. V. 6806. P. 184–190.

32. *Albarghouthi A., Li Y., Gurfinkel A., Chechik M.* UFO: A framework for abstraction and interpolation-based software verification // Proceedings of the 24th International Conference on Computer Aided Verification, LNCS. 2012. V. 7358. P. 672–678.

33. *Новиков Е.М.* Упрощение анализа трасс ошибок инструментов статического анализа кода. Сборник научных трудов научно-практической конференции Актуальные Проблемы Программной Инженерии (АППИ). 2011. С. 215–221.

34. Список ошибок, выявленных в модулях ядра ОС Linux с помощью конфигурируемой системы статической верификации Linux Driver Verification Tools.
http://linuxtesting.org/results/ldv

35. *Мандрыкин М.У., Мутилин В.С., Новиков Е.М., Хорошилов А.В., Швед П.Е.* Использование драйверов устройств операционной системы Linux для сравнения инструментов статической верификации // Программирование. 2012. № 5. С. 54–71.

36. *Бейер Д., Петренко А.К.* Верификация драйверов операционной системы Linux // Труды ИСП РАН. 2012. Т. 23. С. 405–412.