Возможности статической верификации монолитного ядра операционных систем[1]

Е.М. Новиков <novikov@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. У большинства современных, повсеместно используемых операционных систем архитектура ядра в той или иной степени является монолитной, поскольку именно данная архитектура позволяет обеспечить максимальную производительность работы. Как правило, размер монолитного ядра без различных расширений, таких как драйверы устройств, составляет несколько миллионов строк кода на языке программирования Си/Си++ и языке ассемблера. С течением времени исходный код достаточно интенсивно изменяется: добавляется поддержка новой функциональности, оптимизируется выполнение различных операций, исправляются ошибки. Высокая практическая значимость монолитного ядра операционных систем определяет строгие требования к его функциональности, безопасности, надежности и производительности. Те подходы к обеспечению качества программных систем, которые в настоящее время используются на практике, позволяют выявить и исправить достаточно большое количество ошибок, однако ни один из них не позволяет обнаружить все возможные ошибки искомых видов. В этой статье показывается, что различные подходы к статической верификации, которые нацелены на решение данной задачи, имеют существенные ограничения, если их применять к монолитному ядру операционных систем целиком, в первую очередь из-за большого размера и сложности исходного кода, который постоянно изменяется. В качестве первого шага в направлении статической верификации монолитного ядра операционных систем предлагается метод декомпозиции ядра на подсистемы.

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

DOI: 10.15514/ISPRAS-2017-29(2)-4

Для цитирования: Новиков Е.М. Возможности статической верификации монолитного ядра операционных систем. Труды ИСП РАН, том 29, вып. 2, 2017 г., стр. 97-116. DOI: 10.15514/ISPRAS-2016-29(2)-4

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

1. Введение

У большинства современных, повсеместно используемых операционных систем (далее — ОС) архитектура ядра в той или иной степени является монолитной, поскольку именно данная архитектура позволяет обеспечить максимальную производительность работы [1]. Как правило, размер монолитного ядра без различных расширений, таких как драйверы устройств (далее в работе это будет называться монолитным ядром), составляет несколько миллионов строк кода на языке программирования Си/Си++ и языке ассемблера. Традиционно по мере разработки монолитного ядра ОС сохраняется совместимость его программного интерфейса для пользовательских приложений, однако при этом с течением времени исходный код достаточно интенсивно изменяется: добавляется поддержка новой функциональности, оптимизируется выполнение различных операций, исправляются ошибки. Например, за 7,5 лет размер монолитного ядра ОС Linux вырос более чем в 2 раза, и на сегодняшний день составляет около 1,4 миллионов строк кода [2].

Высокая практическая значимость монолитного ядра ОС определяет строгие требования к его функциональности, безопасности, надежности и производительности. В случае ошибок и сбоев в монолитном ядре возможны некорректная работа, повреждение данных и снижение производительности самого ядра, драйверов, модулей и пользовательских приложений, могут быть нарушены права и конфиденциальность данных пользователей ОС.

В данной работе делается обзор тех методов и инструментов, которые уже применяются на практике для обеспечения качества монолитного ядра ОС (раздел 2). Отмечается, что ни один из них не позволяет обнаружить все возможные ошибки искомых видов. На решение данной задачи нацелены различные подходы к статической верификации, которые рассматриваются в разделе 3. Также в этом разделе показывается, что существующие подходы к статической верификации ядра ОС имеют существенные ограничения, если их применять к монолитному ядру операционных систем целиком, в первую очередь из-за большого размера и сложности исходного кода, который постоянно изменяется. Раздел 4 представляет метод декомпозиции монолитного ядра ОС, который позволит применять инструменты автоматической статической верификации для монолитного ядра ОС. В заключении подводятся итоги данной работы.

2. Используемые на практике подходы к обеспечению качества монолитного ядра ОС

В настоящее время на практике качество монолитного ядра ОС обеспечивается посредством экспертизы кода, сборки и запуска ядра, тестирования, статического анализа и за счет исправления ошибок, обнаруженных пользователями. Данные подходы в совокупности позволяют выявить и

исправить достаточно большое количество ошибок. Однако ни один из подходов не позволяет обнаружить все возможные ошибки искомых видов [3].

Например, экспертиза кода широко используется в процессе разработки ядра ОС Linux [4]. Тем не менее, возможности данного подхода ограничены. С учетом того, что монолитное ядро ОС имеет достаточно большой объем сложного исходного кода, который быстро развивается, экспертиза кода не позволяет гарантировать отсутствие ошибок ни для существующего исходного кода, ни для постоянно идущих изменений.

Сборка и запуск ядра, тестирование и статический анализ могут быть проведены автоматизированным образом самими разработчиками монолитного ядра ОС. Помимо этого для проектов с открытым исходным кодом были разработаны специализированные инфраструктуры, например, 0-day[1], kernelci.org[2], OSS-Fuzz[3] и Coverity Scan[4], которые еще больше автоматизировали данные подходы. В случае использования данных инфраструктур разработчикам предлагается рассмотреть сообщения о выявленных ошибках и предпринять соответствующие действия, например, исправить ошибку или пометить, что выдано ложное сообщение об ошибке.

Сборка ядра позволяет выявить ошибки, которые сообщаются инструментами, осуществляющими сборку, в основном компилятором и компоновщиком. В ходе запуска ядра выявляются ошибки, которые происходят при инициализации ядра, например, при монтировании файловых систем. Можно обнаружить те же виды ошибок, что и при тестировании. Примечательно то, что в отличие от других автоматизированных подходов к обеспечению качества сборка и запуск ядра часто осуществляются для различных конфигураций и архитектур, поскольку они не требуют существенных усилий при первоначальной настройке и могут быть выполнены достаточно быстро.

Тестирование требует подготовки специального тестового окружения для того, чтобы осуществить реальное выполнение с различными входными данными. Тестирование позволяет выявить широкий спектр проблем: падения, например, вследствие разыменования нулевого указателя, некорректная функциональность, нарушение прав пользователей, деградация производительности, в том числе зависания, и т.д. Иногда для того чтобы расширить набор проверок, выполняемых в ходе тестирования, и упростить поиск причин ошибок, при сборке монолитного ядра ОС включаются соответствующие конфигурационные опции[5]. При обычной эксплуатации это, как правило, не делается, поскольку может привести к достаточно существенным накладным расходам. При тестировании достаточно сложно достигнуть большого покрытия, проверяются не все возможные ситуации и

[1] https://01.org/lkp.
[2] https://kernelci.org/.
[3] https://github.com/google/oss-fuzz.
[4] https://scan.coverity.com/.
[5] https://github.com/google/kasan/wiki.

часть ошибок пропускается. Для увеличения покрытия, а также поиска некоторых специфичных ошибок требуется прикладывать большие усилия [5, 6].

Статический анализ позволяет обнаружить ошибки на всех возможных путях выполнения, поскольку некоторое представление программы анализируется без ее реального выполнения с некоторыми определенными входными данными. Данное направление активно исследуется в последние годы. На сегодняшний день существует множество коммерческих и академических инструментов, с помощью которых уже удалось получить хорошие результаты [7-10]. Основным недостатком статического анализа является достаточно большое количество ложных сообщений об ошибках, которых, как правило, не бывает при использовании других подходов к обеспечению качества. Для того чтобы решить эту проблему, а также проводить анализ за время, сравнимое по порядку со временем сборки, в инструментах реализуются различные эвристики. Это приводит к тому, что, во-первых, анализируются не все возможные пути, например, не рассматриваются все итерации циклов или вызовы по функциональным указателям. Во-вторых, некоторые потенциальные ошибки могут преднамеренно игнорироваться, поскольку за отведенное время и с учетом сделанных упрощений инструменты не в состоянии определить точно, возможны они на самом деле или нет. Поскольку наборы эвристик в разных инструментах статического анализа отличаются, они находят разные ошибки одних и тех же видов в одних и тех же программных системах. Преимущественно эти инструменты применяются для поиска нарушений общих правил безопасного программирования, таких как разыменование нулевого указателя, выход за границу массива и т.д. Некоторые инструменты позволяют выявлять более специфичные ошибки, например, в ядре ОС Linux: checkpatch[1] и Coccinelle [8].

Достаточно много ошибок в монолитном ядре ОС, которые не обнаружили все рассмотренные ранее подходы, сообщается пользователями, которые сталкиваются с ними в процессе эксплуатации. Для того чтобы упростить этот процесс, разрабатываются специальные средства, которые могут в том числе сообщать разработчикам статистику использования и информацию о падениях автоматически[2]. Несмотря на то, что пользователей монолитного ядра ОС может быть чрезвычайно много, данный подход тем не менее также не гарантирует отсутствие ошибок.

Все рассмотренные подходы к обеспечению качества монолитного ядра ОС необходимо развивать и использовать в дальнейшем, поскольку каждый из них позволяет выявить определенные ошибки, в том числе критичные, и ни один из них не заменяет полностью другой. Наряду с этим требуется предлагать и

[1] https://github.com/torvalds/linux/blob/master/scripts/checkpatch.pl.
[2] https://technet.microsoft.com/en-us/library/cc754364(v=ws.11).aspx, https://support.apple.com/en-us/HT202031.

исследовать новые методы, которые должны быть нацелены на обнаружение всех возможных ошибок искомых видов.

3. Возможности статической верификации монолитного ядра ОС

Потенциально выявить все ошибки искомого вида в программных системах или доказать их корректность можно с помощью методов и инструментов статической верификации. В данном разделе рассматриваются три основных направления в области статической верификации ядра ОС и показывается, что в настоящее время соответствующие подходы не в состоянии решить проблему верификации монолитного ядра ОС целиком по различным причинам.

3.1 Статическая верификация микроядра ОС

Наибольшее продвижение в области статической верификации ядра ОС было сделано для микроядра. Размер микроядра ОС составляет несколько тысяч или десятков тысяч строк кода. Для его статической верификации строятся модели и формальные спецификации, которые покрывают функциональные требования, а также некоторые дополнительные свойства, такие как разделение памяти для пользовательских приложений. Верификация проводится при выполнении определенных условий, например:

- рассматриваются только определенные аппаратные платформы и модели процессоров;
- допускается использовать подмножества языков программирования;
- предполагается корректность работы аппаратного обеспечения и компилятора.

Несмотря на все упрощения и предположения задача полной статической верификации микроядра ОС является чрезвычайно трудоемкой и, как правило, осуществляется только для некоторого подмножества исходного кода.

Например, удалось формально доказать полную функциональную корректность для микроядра seL4, которое состоит из примерно 10 тысяч строк кода на языке программирования Си и языке ассемблера [11]. Размер спецификаций при этом составил примерно 400 тысяч строк. При доказательстве использовались предположения о корректности кода на языке ассемблера, компилятора и аппаратуры. В дополнение к функциональным требованиям позднее для микроядра seL4 были сформулированы такие высокоуровневые свойства, как соблюдение прав доступа и разграничение информационных потоков [12, 13]. Данные свойства важны для построения более надежных программных систем, которые используют микроядро в своей основе. Для них были разработаны соответствующие модели и спецификации, что позволило проверить их выполнимость [14-17]. В [18] подводятся итоги многолетней работы по разработке и статической верификации микроядра

seL4. На момент завершения работы размер спецификаций составил 480 тысяч строк. В целом на разработку и верификацию потребовалось около 2-х и 28-ми человеко-лет соответственно. Важный вывод, который продемонстрировали авторы данного исследования, состоит в том, что разработка программных систем с использованием методов и инструментов формальной верификации существенно более выгодна по стоимости и позволяет достигнуть более высокого уровня качества по сравнению с разработкой программных систем высокой надежности с использованием традиционных подходов.

В аналогичном проекте была поставлена цель верифицировать всю низкоуровневую программно-аппаратную систему, включая микроядро ОС реального времени PikeOS, которое уже использовалось в промышленности [19, 20]. В работе [21] были обсуждены разработка спецификаций и верификация для одного из наиболее важных нефункциональных свойств ядра ОС, а именно, для разделения памяти между пользовательскими приложениями. Примечательно то, что в данном проекте был также верифицирован исходный код на языке ассемблера [22].

В работе [23] было предложено изменить подход к разработке ядра ОС и спецификаций, благодаря чему удалось существенно сократить их размер и время разработки. Как и для PikeOS, удалось формально верифицировать код на языке ассемблера. Отмечается, что хотя в статьях часто декларируется полная формальная верификации, на деле часть кода остается непроверенной. Последняя работа не является исключением из данного правила.

В другой работе было предложено использовать сразу несколько методов анализа и верификации с целью проверки выполнения различных свойств для микроядра ОС [24]. Отмечается, что данный подход применим и для обеспечения качества монолитного ядра ОС. В статье представлены только первоначальные результаты проекта, так что не понятно, в какой мере удалось добиться поставленной цели хотя бы для рассматриваемого микроядра ОС.

В отличие от предыдущих исследований в [25] было предложено разрабатывать и верифицировать ядро ОС ExpressOS, которое позволяет запускать критичные пользовательские приложения ОС общего назначения Android. Статическая верификация была проведена только для проверки выполнения наиболее важных свойств безопасности, таких как разделение памяти и безопасное межпроцессное взаимодействие. Это позволило существенно сократить размер спецификаций. Было продемонстрировано, что разработанное ядро ОС оказалось неподверженным большинству известных серьезных уязвимостей. Несколько тестов показали, что по производительности оно не существенно уступало неверифицированному ядру ОС общего назначения.

Работа [26] интересна тем, что была статически верифицирована одна из наиболее важных частей существующего ядра ОС реального времени, а именно, планировщик. Была доказана функциональная корректность, а также безопасность работы с памятью. Это исследование демонстрирует, что существующие методы и инструменты позволяют формально верифицировать

небольшие важные части ядра. Однако процесс оказался достаточно трудоемким, поскольку потребовалось переписать исходный код таким образом, чтобы его нотация поддерживалась используемым инструментом верификации. Аналогичная работа была проделана для ядра ОС Linux [27]. Однако авторы указали достаточно большое количество ограничений, что ставит под сомнение возможность практического использования предложенного подхода.

Распространить опыт статической верификации микроядра ОС на монолитное ядро ОС в настоящее время не представляется возможным по следующим причинам:

- Размер монолитного ядра превышает размер микроядра на 2-3 порядка, при этом исходный код постоянно изменяется [2]. Соответственно для полной верификации потребуются чрезвычайно большие усилия высококвалифицированных инженеров. Можно попытаться тщательно верифицировать только наиболее критичные компоненты, но это не дало бы гарантии отсутствия ошибок определенных видов во всем монолитном ядре ОС в целом.
- При разработке монолитного ядра ОС используется вся мощь языков программирования, в том числе всевозможные расширения, например, GNU. Существующие инструменты для разработки спецификаций и верификации не поддерживают это.
- При проектировании микроядра разработчики стараются сделать так, чтобы впоследствии было проще разрабатывать спецификации и проводить статическую верификацию и сертификацию. Монолитное ядро ОС проектируется и разрабатывается по другим принципам, что существенно затрудняет его верификацию.

3.2 Использование специализированных языков программирования и архитектур

Существует достаточно много работ, в которых предлагается использовать специализированные языки программирования или архитектуры для того, чтобы разрабатывать более надежное ядро ОС, а также достаточно сильно упростить его последующую верификацию.

Например, в [28] базовая часть ядра ОС была разработана на типизированном языке ассемблера. Для нее были разработаны спецификации и доказана корректность относительно данных спецификаций. Основная часть ядра была разработана на языке программирования C#. Благодаря предложенной архитектуре для нее удалось автоматически проверить выполнимость свойств безопасности работы с памятью. Необходимо отметить, что данные свойства не исчерпывают все необходимые свойства безопасного и надежного ядра ОС.

В [29, 30] предлагается использовать высокоуровневый язык программирования для разработки частей ядра ОС с целью их постепенной

интеграции в существующее ядро ОС, разработанное на низкоуровневых языках программирования. Это позволяет избежать некоторых ошибок, в том числе нарушения достаточно сложных функциональных требований. Однако подход не нацелен на возможность проверки всех необходимых свойств. Авторы уделили внимание тому, что новые компоненты должны работать как можно более эффективно, однако оценка накладных расходов не была сделана. Авторы рассмотрели несколько примеров с реализацией компонентов ядра ОС, которые не используется широко, но имеют достаточно высокоуровневую реализацию. Применимость данного подхода к компонентам монолитного ядра ОС, которые имеют достаточно сложную реализацию, не изучалась.

Также было предложено использовать альтернативные средства разработки только для сложных типов данных с указателями [31]. Авторы задали их специальным образом для микроядра ОС Fiasco.OC. Затем это представление было автоматически транслировано в исходный код на языке программирования Си++, на котором написан остальной исходный код данного микроядра. Авторы уверяют, что такой подход не вносит существенных накладных расходов, но зато позволяет автоматически проверять выполнимость некоторых свойств, а именно, безопасности работы с памятью. Проверка других свойств не рассматривалась.

Еще один подход к разработке и верификации программных систем состоит в том, что они полностью пишутся на специализированном языке программирования, причем одновременно с реализацией задаются спецификации [32]. Это позволяет проводить верификацию непосредственно по мере разработки. Автор рассмотрел несколько тестовых примеров, поэтому остается неясным, насколько данный подход масштабируем и эффективен на практике.

Помимо использования специализированных средств программирования предлагается использовать также и специализированные аппаратные средства [33, 34]. Авторы отмечают, что архитектура большинства используемых на сегодняшний день программно-аппаратных систем была построена по принципу эффективного использования достаточно ограниченного объема вычислительных ресурсов. Данный принцип необходимо пересматривать таким образом, чтобы гарантировать безопасность работы ценой дополнительных ресурсов.

Все данные подходы могут быть использованы при построении программно-аппаратных систем специального назначения, но не применимы к верификации монолитного ядра современных, повсеместно используемых ОС, поскольку разработчики предпочитают использовать всю мощь общецелевых языков программирования и инструментов, а работать оно должно эффективно на оборудовании общего назначения. Данное направление не пользуется большой популярностью на практике, но вызывает достаточно большой интерес у исследователей.

3.3 Автоматическая статическая верификация

На сегодняшний день наиболее значимые результаты при статической верификации компонентов монолитного ядра повсеместно используемых ОС были получены при использовании инструментов автоматической статической верификации, таких как SLAM [35], BLAST, CPAchecker, CBMC и др. [36, 37]. Данные инструменты позволяют доказывать за приемлемое время выполнимость специфицированных свойств для программных систем размером несколько десятков тысяч строк кода на различных языках программирования. Они были успешно использованы для статической верификации драйверов ОС Microsoft Windows [38] и модулей ядра ОС Linux [39-42], большинство из которых укладываются в указанное ограничение по размеру. Удалось выявить сотни ошибок, которые были признаны разработчиками [38, 42].

Изначально инструменты автоматической статической верификации были нацелены на проверку такого свойства, как достижимость, к которому можно свести, например, правила корректного использования программного интерфейса [38-42]. Вообще говоря, таким образом можно выразить и традиционные функциональные требования. Позднее была добавлена поддержка проверки таких свойств, как безопасность работы с памятью и завершимость [43, 44].

Инструменты автоматической статической верификации не используют эвристики так, как это делают инструменты статического анализа, а выполняют точный анализ всех путей выполнения. Благодаря этому они позволяют выявить все ошибки искомого вида в программных системах или доказать их корректность.

Опыт использования данных инструментов показал, что они выдают большое количество ложных сообщений об ошибках [45]. Для получения более качественных результатов верификации необходимо разрабатывать достаточно точные спецификации моделей окружения и проверяемых правил [46-48]. Также инструменты могут потреблять чрезвычайно большое количество ресурсов, в первую очередь, процессорного времени и оперативной памяти, что особенно остро проявляется при увеличении размера анализируемых программных систем.

В следующем разделе данной статьи показывается, как предлагается учитывать эти особенности при использовании инструментов автоматической статической верификации для монолитного ядра ОС.

4. Метод декомпозиции монолитного ядра ОС на подсистемы

В предыдущем разделе было показано, что наиболее перспективными методами и инструментами для обнаружения всех ошибок искомого вида в монолитном ядре ОС являются методы и инструменты автоматической статической верификации. Данные инструменты позволяют анализировать

исходный код монолитного ядра ОС в оригинальном виде, что особенно важно ввиду его большого размера и высокого темпа развития, и выявлять ошибки, которые обусловлены некорректной работой с памятью и некорректным использованием программного интерфейса[1]. Автором работы было показано, что эти ошибки являются одними из наиболее распространенных для монолитного ядра ОС Linux [2].

Как было отмечено, на текущий момент инструменты автоматической статической верификации способны верифицировать промышленные программы размером порядка нескольких десятков тысяч строк кода. Для того чтобы применить их для верификации монолитного ядра ОС, размер которого составляет несколько миллионов строк кода, необходимо декомпозировать монолитное ядро на подсистемы, которые возможно проверять по отдельности. Декомпозиция должна быть проведена таким образом, чтобы, во-первых, инструменты автоматической статической верификации могли анализировать получившийся исходный код с разумными ограничениями на используемые вычислительные ресурсы и общее время проверки, для чего все подсистемы должны быть ограничены по размеру и сложности. Во-вторых, необходимо иметь возможность постепенно повышать качество верификационных результатов (сокращать количество ложных сообщений об ошибках и находить ошибки новых видов) за счет инкрементальной разработки спецификаций моделей окружения и проверяемых свойств. Такой подход уже достаточно хорошо проявил себя при верификации драйверов ядра ОС Microsoft Windows и модулей ядра ОС Linux, поэтому с большой вероятностью аналогичных результатов удастся добиться и для подсистем монолитного ядра.

В настоящий момент автору представляется наиболее правильным проводить декомпозицию монолитного ядра ОС на группы файлов (подсистемы) по принципу разделения четко выраженной функциональности. Таким образом, следует выделить подсистему управления памяти, планировщик, компоненты сетевой подсистемы и т.д. Поскольку такому подходу, как правило, следуют при разработке, ожидается, что провести декомпозицию на практике будет достаточно легко.

Каждая из выделенных подсистем, скорее всего, окажется достаточно компактной по размеру, а значит, будет поддаваться анализу. В том случае, если какая-нибудь подсистема получится слишком большой или сложной, потребуется провести дополнительную декомпозицию. В предположении, что размер каждой подсистемы составит порядка 10 тысяч строк кода, общее количество подсистем для типового монолитного ядра ОС составит порядка 100.

[1] Автору неизвестен опыт удачного применения инструментов автоматической статической верификации для проверки выполнения других свойств для промышленных программных систем. Однако в ближайшем будущем такой опыт скорее всего появится, поскольку это направление пользуется большой популярностью у исследователей и активно развивается.

Задавать группы файлов предлагается по их принадлежности к директориям. В том случае, если файлы одной директории потребуется отнести к различным подсистемам, то их надо будет перечислить явным образом для каждой из этих подсистем. Такой подход позволит проще адаптироваться к другим версиям монолитного ядра ОС, поскольку директории создаются, удаляются и перемещаются намного реже, чем файлы.

При использовании такого подхода к декомпозиции монолитного ядра ОС на уровне каждой подсистемы реализация будет достаточно сильно взаимосвязана, при этом моделировать соответствующую функциональность не потребуется, поскольку будет верифицироваться сразу весь исходный код подсистемы. В том случае, если при верификации будут выдаваться ложные сообщения об ошибках вследствие отсутствия при анализе реализаций других подсистем, их нужно будет постепенно моделировать. Например, большинство подсистем обращаются к подсистеме управления памятью, поэтому с большой вероятностью потребуется разработать ее модель. Моделировать те подсистемы, которые не используются другими, не потребуется. Такими подсистемами с большой вероятностью окажутся, например, подсистемы поддержки различных типов устройств, которые часто реализуются в монолитном ядре ОС.

Также для подсистем нужно будет инкрементально разрабатывать:

- Спецификации моделей окружения, отвечающих за обращения к соответствующим подсистемам. Например, описывать всевозможные последовательности вызовов функций со всевозможными значениями их аргументов – соответствующий подход был предложен в работе [48].
- Спецификации правил использования программного интерфейса других подсистем [47]. Для проверки свойства безопасности работы с памятью разрабатывать дополнительные спецификации не потребуется, поскольку их проверка интегрирована в инструменты автоматической статической верификации [43, 44].

При разработке спецификаций нужно обязательно предусмотреть возможность достаточно быстро адаптировать их при условии изменений реализации ядра и требований к его окружению, что уже было учтено в проекте по статической верификации модулей ядра ОС Linux [47].

Можно предложить и другие подходы к декомпозиции монолитного ядра ОС на подсистемы, например, верифицировать отдельно каждый файл монолитного ядра ОС. Такой подход будет существенно более трудоемким, поскольку файлов в монолитном ядре ОС много, например, около 2,2 тысяч для монолитного ядра ОС Linux 4.8 [2], и придется разрабатывать достаточно большую спецификацию модели окружения. Рассматривать по отдельности каждую функцию представляется еще менее перспективным, поскольку

обычно их намного больше, чем файлов, например, около 50 тысяч для монолитного ядра ОС Linux 4.8, и они сильно взаимосвязаны друг с другом.

Можно разделить исходный код монолитного ядра ОС таким образом, чтобы каждая подсистема содержала весь исходный код, который может использоваться при выполнении одного из системных вызовов или одной из функций из программного интерфейса ядра. Ожидается, что общее количество таких подсистем будет намного меньше, чем количество всех функций монолитного ядра ОС, но сравнимо по порядку с количеством файлов. При этом реализация всей связанной функциональности будет рассматриваться единовременно в отличие от того подхода, при котором рассматриваются отдельные файлы, так что моделирование окружения будет намного менее трудоемким. По сравнению с предложенным ранее подходом к декомпозиции на непересекающиеся группы файлов также потребуется разрабатывать существенно меньше моделей, поскольку не будет нужно моделировать другие подсистемы.

При использовании данного подхода с большой вероятностью некоторые подсистемы окажутся достаточно большими и сложными. Например, в большинство подсистем войдет та или иная часть подсистемы управления памятью. Провести статическую верификацию в таком случае окажется затруднительно или невозможно. Еще одной предполагаемой сложностью данного подхода является определение границы подсистем. Например, некоторые функции могут вызываться не напрямую, а посредством функциональных указателей. Поскольку для такого случая нельзя простыми методами определить конкретные функции в общем случае, потребуется выполнить определенное моделирование. Аналогичная ситуация возникает для глобальных переменных, которые могут быть косвенно модифицированы при вызове функций из других подсистем, – это также потребуется моделировать некоторым образом.

Без оценки на практике невозможно определить, какой из двух подходов к декомпозиции монолитного ядра ОС, разделение на непересекающиеся группы файлов или выбор всего исходного кода, относящегося к системным вызовам и функциям программного интерфейса ядра, окажется менее трудоемким и/или позволит получить более качественные результаты верификации. Возможно окажется целесообразным применять оба данных подхода или их композицию для некоторой части или даже для всего монолитного ядра ОС в целом.

5. Заключение

Используемые в настоящее время подходы к обеспечению качества монолитного ядра ОС позволяют выявить и исправить достаточно большое количество ошибок, однако ни один из подходов не позволяет обнаружить все возможные ошибки искомых видов. Решить эту задачу теоретически способна статическая верификация. Большинство существующих подходов к статической верификации ядра ОС имеют существенные ограничения, если их

применять к монолитному ядру ОС целиком. Наиболее перспективными в этом направлении являются методы и инструменты автоматической статической верификации программных систем. В данной работе предлагается метод декомпозиции монолитного ядра ОС на подсистемы с целью их последующей верификации.

Список литературы

[1]. В.Е. Карпов, К.А. Коньков. Основы операционных систем. Курс лекций. Учебное пособие. М.: Интернет-университет информационных технологий, 536 с., 2005.

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

[3]. R.L. Glass. Facts and fallacies of software engineering. Addison-Wesley Professional, 2002.

[4]. J. Corbet, G. Kroah-Hartman. Linux kernel development. How Fast It is Going, Who is Doing It, What They Are Doing and Who is Sponsoring the Work. http://go.linuxfoundation.org/linux-kernel-development-report-2016, 2016.

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

[6]. J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, A. Fedorova. The Linux scheduler: a decade of wasted cores. In Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16), article 1, 16 pages, 2016.

[7]. A. Chou, J. Yang, B. Chelf, S. Hallem, D. Engler. An empirical study of operating systems errors. In Proceedings of the eighteenth ACM symposium on Operating systems principles (SOSP'01), pp. 73-88, 2001.

[8]. J.L. Lawall, J. Brunel, N. Palix, H.R. Rydhof, H. Stuart, G. Muller. 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), pp. 43-52, 2009.

[9]. А. Аветисян, А. Белеванцев, А. Бородин, В. Несов. Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ. Труды ИСП РАН, т. 21, 2011, стр. 23-38.

[10]. N. Palix, G. Thomas, S. Saha, C. Calves, J. Lawall, G. Muller. Faults in Linux: ten years later. In Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11), pp. 305-318, 2011.

[11]. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, S. Winwood. seL4: formal verification of an OS kernel. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP'09), pp. 207-220, 2009.

[12]. J. Andronick, D. Greenaway, K. Elphinstone. Towards proving security in the presence of large untrusted components. In Proceedings of the 5th international conference on Systems software verification (SSV'10), pp. 9-9, 2010.

[13]. G. Klein. From a verified kernel towards verified systems. In Proceedings of the 8th Asian conference on Programming languages and systems (APLAS'10), pp. 21-33, 2010.

[14]. I. Kuz, G. Klein, C. Lewis, A. Walker. capDL: a language for describing capability-based systems. In Proceedings of the first ACM asia-pacific workshop on Workshop on systems (APSys'10), pp. 31-36, 2010.

[15]. T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, G. Klein. seL4 enforces integrity. In Proceedings of the Second international conference on Interactive theorem proving (ITP'11), pp. 325-340, 2011.

[16]. T. Murray, D. Matichuk, M. Brassil, P. Gammie, G. Klein. Noninterference for operating system kernels. In Proceedings of the Second international conference on Certified Programs and Proofs (CPP'12), pp. 126-142, 2012.

[17]. M. Daum, N. Billing, G. Klein. Concerned with the unprivileged: user programs in kernel refinement. Formal Aspects of Computing, vol. 26, issue 6, pp. 1205-1229, 2014.

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

[19]. C. Baumann, B. Beckert, H. Blasum, T. Bormer. Formal Verification of a Microkernel Used in Dependable Software Systems. In Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security (SAFECOMP'09), pp. 187-200, 2009.

[20]. E. Alkassar, W. J. Paul, A. Starostin, A. Tsyban. Pervasive verification of an OS microkernel: inline assembly, memory consumption, concurrent devices. In Proceedings of the Third international conference on Verified software: theories, tools, experiments (VSTTE'10), pp. 71-85, 2010.

[21]. C. Baumann, T. Bormer, H. Blasum, S. Tverdyshev. Proving Memory Separation in a Microkernel by Code Level Verification. In Proceedings of the 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW '11), pp. 25-32, 2011.

[22]. S. Schmaltz, A. Shadrin. Integrated semantics of intermediate-language c and macro-assembler for pervasive formal verification of operating systems and hypervisors from VerisoftXT. In Proceedings of the 4th international conference on Verified Software: theories, tools, experiments (VSTTE'12), pp. 18-33, 2012.

[23]. R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. Wu, S.-C. Weng, H. Zhang, Y. Guo. Deep Specifications and Certified Abstraction Layers. SIGPLAN Not. 50, 1, pp. 595-608, 2015.

[24]. M. Děcký. A road to a formally verified general-purpose operating system. In Proceedings of the First international conference on Architecting Critical Systems (ISARCS'10), pp. 72-88, 2010.

[25]. H. Mai, E. Pek, H. Xue, S. T. King, P. Madhusudan. Verifying security invariants in ExpressOS. In Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems (ASPLOS '13), pp. 293-304, 2013.

[26]. J.F. Ferreira, C. Gherghina, G. He, S. Qin, W.-N. Chin. Automated verification of the FreeRTOS scheduler in Hip/Sleek. Int. J. Softw. Tools Technol. Transf. 16, 4, pp. 381-397, 2014.

[27]. A. Gotsman, H. Yang. Modular verification of preemptive OS kernels. In Proceedings of the 16th ACM SIGPLAN international conference on Functional programming (ICFP '11), pp. 404-417, 2011.

[28]. J. Yang, Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10), pp. 99-110, 2010.

[29]. M. Danish, H. Xi. Operating system development with ATS: work in progress. In Proceedings of the 4th ACM SIGPLAN workshop on Programming languages meets program verification (PLPV'10), pp. 9-14, 2010.
[30]. M. Danish, H. Xi. Using Lightweight Theorem Proving in an Asynchronous Systems Context. In Proceedings of the 6th International Symposium on NASA Formal Methods, vol. 8430, pp. 158-172. 2014.
[31]. B.W. Cronkite-Ratcliff. Development of automatically verifiable systems using data representation synthesis. In Proceedings of the 2013 companion publication for conference on Systems, programming, & applications: software for humanity (SPLASH '13), pp. 109-110, 2013.
[32]. K.R.M. Leino. Developing verified programs with dafny. In Proceedings of the 2013 International Conference on Software Engineering (ICSE '13), pp. 1488-1490, 2013.
[33]. A. DeHon, B. Karel, T. F. Knight, Jr., G. Malecha, B. Montagu, R. Morisset, G. Morrisett, B. C. Pierce, R. Pollack, S. Ray, O. Shivers, J. M. Smith, G. Sullivan. Preliminary design of the SAFE platform. In Proceedings of the 6th Workshop on Programming Languages and Operating Systems (PLOS '11), article 4, 5 pages, 2011.
[34]. A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, A. Tolmach. A verified information-flow architecture. In Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'14), pp. 165-178, 2014.
[35]. T. Ball, E. Bounimova, R. Kumar, V. Levin. SLAM2: Static driver verification with under 4% false alarms. In Proceedings of the 10th International Conference on Conference on Formal Methods in Computer-Aided Design (FMCAD'10), pp. 35-42, 2010.
[36]. D. Beyer. Status Report on Software Verification (Competition Summary SV-COMP 2014). In Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and of Analysis Systems (TACAS'14), LNCS 8413, pp. 373-388, 2014.
[37]. D. Beyer. Software Verification and Verifiable Witnesses (Report on SV-COMP 2015). In Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and of Analysis Systems (TACAS'15), LNCS 9035, pp. 401-416, 2015.
[38]. T. Ball, V. Levin, S.K. Rajamani. A decade of software model checking with SLAM. Communications of the ACM, vol. 54, issue 7, pp. 68-76, 2011.
[39]. T. Witkowski, N. Blanc, D. Kroening, G. Weissenbacher. Model checking concurrent Linux device drivers. In Proceedings of the 22nd IEEE/ACM international conference on Automated Software Engineering (ASE'07), pp. 501-504, 2007.
[40]. H. Post, W. Küchlin. Integrated static analysis for Linux device driver verification. In Proceedings of the 6th International Conference on Integrated Formal Methods (IFM'07), LNCS, vol. 4591, pp. 518-537, 2007.
[41]. Д. Бейер, А.К. Петренко. Верификация драйверов операционной системы Linux. Труды ИСП РАН, т. 23, стр. 405-412, 2012. DOI: 10.15514/ISPRAS-2012-23-23
[42]. И.С. Захаров, М.У. Мандрыкин, В.С. Мутилин, Е.М. Новиков, А.К. Петренко, А.В. Хорошилов. Конфигурируемая система статической верификации модулей ядра операционных систем. Программирование, т. 41, выпуск 1, стр. 44-67, 2015.
[43]. T. Ströder, C. Aschermann, F. Frohn, J. Hensel, J. Giesl. AProVE: Termination and Memory Safety of C Programs (Competition Contribution). In Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15), LNCS 9035, pp. 417-419, 2015.
[44]. M. Kotoun, P. Peringer, V. Šoková, T. Vojnar. Optimized PredatorHP and the SV-COMP Heap and Memory Safety Benchmark. In Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'16), vol. 9636, pp. 942-945, 2016.
[45]. D. Engler, M. Musuvathi. Static analysis versus model checking for bug finding. In Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'04), LNCS, vol. 2937, pp. 191-210, 2004.
[46]. T. Ball, S.K. Rajamani. SLIC: A specification language for interface checking of C. Technical Report MSR-TR-2001-21, Microsoft Research, 2001.
[47]. Е.М. Новиков. Развитие метода контрактных спецификаций для верификации модулей ядра операционной системы Linux. Диссертация на соискание ученой степени кандидата физико-математических наук, Институт системного программирования РАН, 2013.
[48]. A. Khoroshilov, V. Mutilin, E. Novikov, I. Zakharov. Modeling Environment for Static Verification of Linux Kernel Modules. In Proceedings of the 9th International Ershov Informatics Conference (PSI'14), LNCS 8974, pages 400-414, 2014.

Static verification of operating system monolithic kernels

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

Abstract. The most of modern widely used operating systems have monolithic kernels since this architecture aims at reaching maximum performance. Usually monolithic kernels without various extensions like device drivers consist of several million lines of code in the programming language C/C++ and in the assembly language. With time, their source code evolves quite intensively: a new functionality is supported, various operations are optimized, bugs are fixed. The high practical value of operating system monolithic kernels defines strict requirements for their functionality, security, reliability and performance. Approaches for software quality assurance which are currently used in practice help to identify and to fix quite a number of bugs, but none of them allows to detect all possible bugs of kinds sought for. This article shows that different approaches to static verification, which are aimed at solving this task, have significant restrictions if applied to monolithic kernels as a whole, primarily due to a large size and complexity of source code that is constantly evolving. As a first step towards static verification of operating system monolithic kernels a method is proposed for decomposition of kernels into subsystems.

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

DOI: 10.15514/ISPRAS-2017-29(2)-4

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

References

[1]. V.E. Karpov, K.A. Kon'kov. Fundamentals of operating systems. Kurs lekcij. Uchebnoe posobie [Lectures. Study material]. M.: Internet-universitet informacionnyh tehnologij, 536 p., 2005 (in Russian).

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

[3]. R.L. Glass. Facts and fallacies of software engineering. Addison-Wesley Professional, 2002.

[4]. J. Corbet, G. Kroah-Hartman. Linux kernel development. How Fast It is Going, Who is Doing It, What They Are Doing and Who is Sponsoring the Work. http://go.linuxfoundation.org/linux-kernel-development-report-2016, 2016.

[5]. A.V. Cyvarev, V.A. Martirosjan. Testing of Linux File System Drivers. Trudy ISP RAN/Proc. ISP RAS, volume 23, pp. 413-426, 2012 (in Russian). DOI: 10.15514/ISPRAS-2012-23-24

[6]. J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, A. Fedorova. The Linux scheduler: a decade of wasted cores. In Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16), article 1, 16 pages, 2016.

[7]. A. Chou, J. Yang, B. Chelf, S. Hallem, D. Engler. An empirical study of operating systems errors. In Proceedings of the eighteenth ACM symposium on Operating systems principles (SOSP'01), pp. 73-88, 2001.

[8]. J.L. Lawall, J. Brunel, N. Palix, H.R. Rydhof, H. Stuart, G. Muller. 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), pp. 43-52, 2009.

[9]. A. Avetisjan, A. Belevancev, A. Borodin, V. Nesov. Using static analysis for finding security vulnerabilities and critical errors in source code. Trudy ISP RAN/Proc. ISP RAS, volume 21, pp. 23-38, 2011 (in Russian).

[10]. N. Palix, G. Thomas, S. Saha, C. Calves, J. Lawall, G. Muller. Faults in Linux: ten years later. In Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11), pp. 305-318, 2011.

[11]. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, S. Winwood. seL4: formal verification of an OS kernel. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP'09), pp. 207-220, 2009.

[12]. J. Andronick, D. Greenaway, K. Elphinstone. Towards proving security in the presence of large untrusted components. In Proceedings of the 5th international conference on Systems software verification (SSV'10), pp. 9-9, 2010.

[13]. G. Klein. From a verified kernel towards verified systems. In Proceedings of the 8th Asian conference on Programming languages and systems (APLAS'10), pp. 21-33, 2010.

[14]. I. Kuz, G. Klein, C. Lewis, A. Walker. capDL: a language for describing capability-based systems. In Proceedings of the first ACM asia-pacific workshop on Workshop on systems (APSys'10), pp. 31-36, 2010.

[15]. T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, G. Klein. seL4 enforces integrity. In Proceedings of the Second international conference on Interactive theorem proving (ITP'11), pp. 325-340, 2011.

[16]. T. Murray, D. Matichuk, M. Brassil, P. Gammie, G. Klein. Noninterference for operating system kernels. In Proceedings of the Second international conference on Certified Programs and Proofs (CPP'12), pp. 126-142, 2012.

[17]. M. Daum, N. Billing, G. Klein. Concerned with the unprivileged: user programs in kernel refinement. Formal Aspects of Computing, vol. 26, issue 6, pp. 1205-1229, 2014.

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

[19]. C. Baumann, B. Beckert, H. Blasum, T. Bormer. Formal Verification of a Microkernel Used in Dependable Software Systems. In Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security (SAFECOMP'09), pp. 187-200, 2009.

[20]. E. Alkassar, W. J. Paul, A. Starostin, A. Tsyban. Pervasive verification of an OS microkernel: inline assembly, memory consumption, concurrent devices. In Proceedings of the Third international conference on Verified software: theories, tools, experiments (VSTTE'10), pp. 71-85, 2010.

[21]. C. Baumann, T. Bormer, H. Blasum, S. Tverdyshev. Proving Memory Separation in a Microkernel by Code Level Verification. In Proceedings of the 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW '11), pp. 25-32, 2011.

[22]. S. Schmaltz, A. Shadrin. Integrated semantics of intermediate-language c and macro-assembler for pervasive formal verification of operating systems and hypervisors from VerisoftXT. In Proceedings of the 4th international conference on Verified Software: theories, tools, experiments (VSTTE'12), pp. 18-33, 2012.

[23]. R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. Wu, S.-C. Weng, H. Zhang, Y. Guo. Deep Specifications and Certified Abstraction Layers. SIGPLAN Not. 50, 1, pp. 595-608, 2015.

[24]. M. Děcký. A road to a formally verified general-purpose operating system. In Proceedings of the First international conference on Architecting Critical Systems (ISARCS'10), pp. 72-88, 2010.

[25]. H. Mai, E. Pek, H. Xue, S. T. King, P. Madhusudan. Verifying security invariants in ExpressOS. In Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems (ASPLOS '13), pp. 293-304, 2013.

[26]. J.F. Ferreira, C. Gherghina, G. He, S. Qin, W.-N. Chin. Automated verification of the FreeRTOS scheduler in Hip/Sleek. Int. J. Softw. Tools Technol. Transf. 16, 4, pp. 381-397, 2014.

[27]. A. Gotsman, H. Yang. Modular verification of preemptive OS kernels. In Proceedings of the 16th ACM SIGPLAN international conference on Functional programming (ICFP '11), pp. 404-417, 2011.

[28]. J. Yang, Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10), pp. 99-110, 2010.

[29]. M. Danish, H. Xi. Operating system development with ATS: work in progress. In Proceedings of the 4th ACM SIGPLAN workshop on Programming languages meets program verification (PLPV'10), pp. 9-14, 2010.

[30]. M. Danish, H. Xi. Using Lightweight Theorem Proving in an Asynchronous Systems Context. In Proceedings of the 6th International Symposium on NASA Formal Methods, vol. 8430, pp. 158-172. 2014.

[31]. B.W. Cronkite-Ratcliff. Development of automatically verifiable systems using data representation synthesis. In Proceedings of the 2013 companion publication for

conference on Systems, programming, & applications: software for humanity (SPLASH '13), pp. 109-110, 2013.

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

[33]. A. DeHon, B. Karel, T. F. Knight, Jr., G. Malecha, B. Montagu, R. Morisset, G. Morrisett, B. C. Pierce, R. Pollack, S. Ray, O. Shivers, J. M. Smith, G. Sullivan. Preliminary design of the SAFE platform. In Proceedings of the 6th Workshop on Programming Languages and Operating Systems (PLOS '11), article 4, 5 pages, 2011.

[34]. A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, A. Tolmach. A verified information-flow architecture. In Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'14), pp. 165-178, 2014.

[35]. T. Ball, E. Bounimova, R. Kumar, V. Levin. SLAM2: Static driver verification with under 4% false alarms. In Proceedings of the 10th International Conference on Conference on Formal Methods in Computer-Aided Design (FMCAD'10), pp. 35-42, 2010.

[36]. D. Beyer. Status Report on Software Verification (Competition Summary SV-COMP 2014). In Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and of Analysis Systems (TACAS'14), LNCS 8413, pp. 373-388, 2014.

[37]. D. Beyer. Software Verification and Verifiable Witnesses (Report on SV-COMP 2015). In Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and of Analysis Systems (TACAS'15), LNCS 9035, pp. 401-416, 2015.

[38]. T. Ball, V. Levin, S.K. Rajamani. A decade of software model checking with SLAM. Communications of the ACM, vol. 54, issue 7, pp. 68-76, 2011.

[39]. T. Witkowski, N. Blanc, D. Kroening, G. Weissenbacher. Model checking concurrent Linux device drivers. In Proceedings of the 22nd IEEE/ACM international conference on Automated Software Engineering (ASE'07), pp. 501-504, 2007.

[40]. H. Post, W. Küchlin. Integrated static analysis for Linux device driver verification. In Proceedings of the 6th International Conference on Integrated Formal Methods (IFM'07), LNCS, vol. 4591, pp. 518-537, 2007.

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

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

[43]. T. Ströder, C. Aschermann, F. Frohn, J. Hensel, J. Giesl. AProVE: Termination and Memory Safety of C Programs (Competition Contribution). In Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15), LNCS 9035, pp. 417-419, 2015.

[44]. M. Kotoun, P. Peringer, V. Šoková, T. Vojnar. Optimized PredatorHP and the SV-COMP Heap and Memory Safety Benchmark. In Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'16), vol. 9636, pp. 942-945, 2016.

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

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

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

[48]. A. Khoroshilov, V. Mutilin, E. Novikov, I. Zakharov. Modeling Environment for Static Verification of Linux Kernel Modules. In Proceedings of the 9th International Ershov Informatics Conference (PSI'14), LNCS 8974, pages 400-414, 2014. DOI: 10.1007/978-3-662-46823-4_32