Компонентная верификация операционных систем[1]

[1,2,3] *В.В. Кулямин <kuliamin@ispras.ru>*
[1,2,3] *А.К. Петренко <petrenko@ispras.ru>*
[1,2,3,4] *А.В. Хорошилов <khoroshilov@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25.*
[2] *Московский государственный университет имени М.В. Ломоносова, 119991, Россия, Москва, Ленинские горы, д. 1.*
[3] *НИУ Высшая школа экономики, 101000, Россия, Москва, ул. Мясницкая, д. 20.*
[4] *Московский физико-технический институт (гос. университет), 141700, Россия, Московская обл., г. Долгопрудный, Институтский пер., д. 9.*

Аннотация. В работе рассматриваются полученные недавно результаты на пути к полномасштабной верификации промышленно используемых операционных систем (ОС). Таковыми считаются не системы, разработанные в целях демонстрации определенной исследовательской идеи, а ОС, активно используемые в каких-то областях экономики и управленческой деятельности и развиваемые на протяжении значительного времени. Предлагается декомпозиция заявленной цели верификации промышленной ОС в целом на задачи верификации различных компонентов ОС и их различных свойств, методы решения которых в дальнейшем можно интегрировать в метод достижения общей цели. На данном этапе пока рассматриваются различные подходы к решению выделенных отдельных задач. Статья является экспликацией опыта верификации различных компонентов нескольких ОС и интеграции используемых для этого технологий, полученного в рамках проектов, проводимых в ИСП РАН.

Ключевые слова: операционная система; верификация; тестирование; статический анализ; дедуктивная верификация; мониторинг

DOI: 10.15514/ISPRAS-2018-30(6)-21

Для цитирования: Кулямин В.В., Петренко А.К., Хорошилов А.В. Компонентная верификация операционных систем. Труды ИСП РАН, том 30, вып. 6, 2018 г., стр. 367-382. DOI: 10.15514/ISPRAS-2018-30(6)-21

[1] Работа поддержана грантом Российского фонда фундаментальных исследований № 18-01-00378

1. Введение

Современные промышленные операционные системы (ОС), используемые для запуска множества разнообразных приложений, очень сложны. Они не только основаны на исходном коде очень большого объема (миллионы и десятки миллионов строк), но также вынуждены иметь огромное число разнородных функциональных возможностей, должны работать на разнообразном аппаратном обеспечении и с большим количеством устройств от совершенно не связанных друг с другом производителей. ОС также предоставляют разработчикам приложений множество различных, постепенно стандартизуемых интерфейсов, которые должны не только корректно работать в огромном разнообразии ситуаций, но и эффективно использовать устройства и аппаратные возможности компьютеров, продолжая надежную работу даже в случае сбоев. В данной работе мы рассматриваем ОС, активно используемые в какой-то отрасли экономики или же ОС общего назначения, поддерживаемые и развиваемые в течении достаточно долгого времени (от пяти лет).

ОС как таковая выполняет две основные задачи.

- Организует выполнение набора приложений на определенном аппаратном обеспечении и множестве устройств так, чтобы приложения, разделяя аппаратные ресурсы, не мешали работе друг друга.
- Предоставляет разработчикам приложений интерфейсы для эффективного и удобного использования аппаратных ресурсов в таком режиме, а также для передачи данных и обеспечения взаимодействия приложений при необходимости.

Важнейшей частью ОС является ядро, работающее в привилегированном режиме процессора и поэтому имеющее неограниченных доступ ко всем ресурсам системы. Ядро управляет доступом приложений к аппаратным ресурсам, устанавливая правила такого доступа и предотвращая их нарушения. Некоторые функции ОС, вообще говоря, не требующие привилегированного режима работы, часто включаются в ядро для повышения производительности.

Приложения взаимодействуют с ядром через системные вызовы, обращения к интерфейсным операциям ядра, переключающим систему в привилегированный режим. Иногда также используются дополнительные способы взаимодействия с ядром, например, специальные виртуальные файловые системы в ОС Linux (*procfs, sysfs, debugfs*). Чтобы сделать удобное для работы разработчиков приложений окружение, ОС обычно предоставляет системные библиотеки и наборы системных утилит, реализующих наиболее часто используемые функции, требующие обращения к ядру. Для решения задач, требующих активности со стороны ядра, таких как работа телекоммуникационных протоколов, управление специализированными устройствами и пр., предоставляются системные службы, которые могут работ

как в привилегированном, так и в обычном пользовательском режиме. На рисунке 1 показана типовая структура ОС общего назначения.

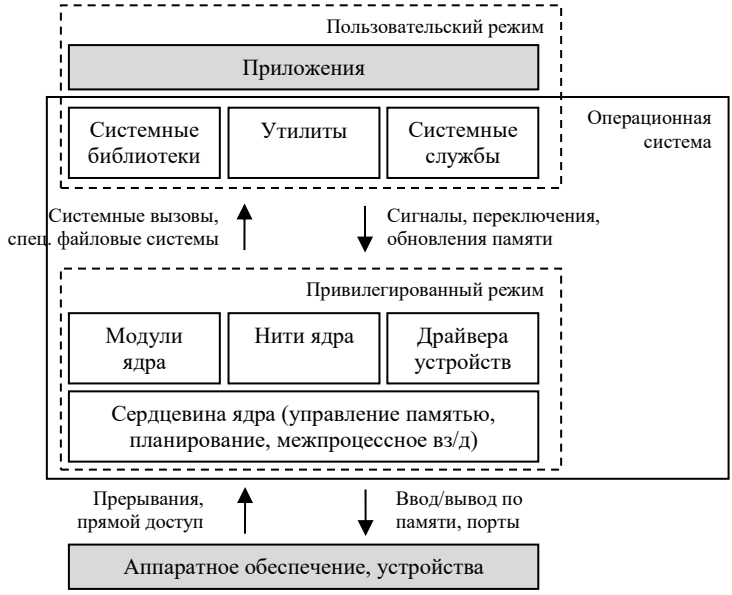

Рис. 1. Типовая структура ОС общего назначения
Fig. 1. Typical structure of general purpose OS

Приведенный выше очень краткий обзор структуры ОС дает некоторое представление о ее сложности. Верификация промышленной ОС также является очень сложной задачей, достаточно упомянуть следующие аспекты.

- Многообразие функциональных возможностей современных ОС создает огромное количество сценариев взаимодействия отдельных функций и множество специфических экстремальных ситуаций, в которых необходим гораздо более тщательный, чем обычно, анализ требуемого поведения.
- Поддержка многозадачности в современных ОС делает проверку корректности поведения гораздо более запутанной.
- Основные функции ОС должны работать, несмотря на сбои в аппаратном и программном обеспечении (ПО); соответственно, необходима верификация устойчивости к сбоям, причем на многих уровнях.
- Современные ОС обычно поддерживают взаимодействие по сети и предоставляют рабочее место для многих пользователей. Это возможно лишь при выполнении некоторых политик защиты данных и задач, накладывающих ограничения на доступ к данным и приложениям и на

взаимодействие между ними. Такие ограничения также должны выполняться несмотря на сбои в ПО и злонамеренные атаки извне и со стороны самих пользователей.

- Поддержка разнородного аппаратного обеспечения и устройств обычно основана на широкой конфигурируемости ОС. Однако верификация всех возможных конфигураций не реализуема на практике, поскольку их число немыслимо огромно.
- Сами по себе объем исходного кода ОС и количество реализуемых им функций огромны. Размер исходного кода ядра Linux версии 4.1 составляет более 20 миллионов строк [1], в том числе около 11.5 миллионов строк приходится на код драйверов, которые разрабатываются и поддерживаются многочисленными сторонними разработчиками. Размер исходного кода Windows XP оценивается в 45 миллионов строк [2]. Общее число функций в системных библиотеках дистрибутива Debian 7.0 около 720 тысяч [3], хотя системных вызовов среди них всего около 350.

Приведенные числовые характеристики показывают, что аккуратная верификация промышленной операционной системы является на настоящий момент недостижимой на практике целью. Однако, определенные методы проверки корректности, эффективности, защищенности и отказоустойчивости современных ОС все равно необходимы. Единственным работоспособным подходом пока является использование всего разнообразия имеющихся методов верификации и анализа для проверки отдельных свойств отдельных компонентов или групп компонентов ОС, начиная с наиболее критичных. Результатами такого подхода обычно являются многочисленные выявляемые ошибки, что, хотя бы, позволяет разработчикам ОС исправлять их, повышая общее качество систем. Несмотря на невозможность сейчас гарантировать корректность, надежность и защищенность разрабатываемых ОС, развитие эффективности и масштабируемости применяемых методов, расширение верифицированной части кода систем и их свойств, позволят постепенно достичь определенных гарантий качества в целом, если не забывать об этой цели в рутине выполнения всех указанных работ.

В данной статье приведен краткий обзор различных подходов к верификации компонентов и свойств промышленных ОС, применяемых в проектах, проводившихся в ИСП РАН на протяжении последних 20 лет. Эти работы интегрируют разнообразные техники верификации, применяя их к компонентам ОС Linux и нескольких специализированных ОС реального времени. Основные используемые методы таковы.

- **Тестирование и динамический анализ**
  Наиболее широко используется тестирование, в виде различных по трудоемкости, обеспечиваемым гарантиям и объему проверяемых свойств методов. Применяемые методы тестирования работоспособности проверяют только основные ограничения в рамках базовых сценариев

работы библиотечных функций и опираются только на покрытие таких функций в качестве критерия полноты. Гораздо более аккуратно и трудоемко тестирование соответствия формальным спецификациям, в рамках которого проверяется строгое выполнение зафиксированных в стандартах требований как в различных сценариях нормального функционирования, так и при возникновении разнообразных исключительных ситуаций, и при различных сценариях взаимодействия разных функций. Вместе с тестированием применяется и верификационный мониторинг, нацеленный на проверку определенных свойств без необходимости разрабатывать специальные тесты.

- **Статический анализ.**
  Статический анализ используется также в виде различных методов. Легковесный статический анализ может обнаруживать лишь небольшое число типов ошибок, но делать это весьма эффективно. Более тяжеловесный, использующий формальные модели требований, позволяет находить весьма сложные виды ошибок, но обычно требует достаточно много времени и усилий на подготовку исходных данных, выполнение самого анализа и инспекцию полученных им результатов. Для верификации компонентов ОС мы чаще использовали более сложные, но и более строгие техники статического анализа.
- **Дедуктивная верификация.**
  Дедуктивную верификацию можно использовать на практике для проверки наиболее важных свойств корректности и защищенности. Известно несколько примеров успешной дедуктивной верификации ядра ОС [4-6], но во всех этих случаях размер верифицированного кода существенно меньше размера ядра типичной промышленной ОС. Тем не менее, дедуктивная верификация тоже может быть использована для получения значимых результатов при проверке промышленных ОС.

Ниже мы попытались систематизировать опыт нескольких десятков проектов ИСП РАН, в рамках которых использовались различные техники верификации для разнообразных компонентов операционных систем.

2. Тестирование и мониторинг

Технология тестирования компонентов ОС разрабатывается в ИСП РАН еще со времени его основания в 1994 г. Первая такая технология под названием KVEST [7] использовалась для автоматизированной разработки на основе формальных спецификаций программных контрактов тестовых наборов для ОС реального времени, разработанных и поддерживавшихся Nortel Networks.

2.1 Формальные подходы

Дальнейшее развитие этих методов легло в основу технологии UniTESK [8]. Ее основные элементы могут быть сформулированы следующим образом.

- Функциональные требования к поведению библиотечных функций формулируются в виде программных контрактов, состоящих их предусловий и постусловий функций и инвариантов используемых типов данных. Программные контракты, хотя и представляют собой формальную модель поведения тестируемых компонентов, оформляются с помощью специализированных библиотек или на расширении языков программирования, используемых при разработке этих компонентов; для библиотек ОС это язык С.
- Критерий полноты тестирования задается в виде критерия покрытия ветвлений в программных контрактах. При необходимости выделить специальные ситуации, не возникающие в контрактах явно, они формулируются в виде дополнительных ветвлений, используемых далее инструментами для оценки полноты тестового покрытия.
- Сценарий теста описывается как расширенный конечный автомат, выполнение всех переходов в котором гарантирует достижение выбранного критерия полноты (покрытие ветвлений в контрактах всех задействованных в описании переходов функций). Расширенный автомат редуцируется к конечному за счет задаваемых разработчиком теста ограничений на достижимые состояния и использования конечного множества возможных значений параметров.
- Тестирование выполняется в виде автоматического построения обхода достижимых состояний и переходов автомата, заданного сценарием теста. При этом каждый вызов тестируемой функции сопровождается обращением к ее тестовому оракулу, сгенерированному из контракта и проверяющему корректность результатов ее работы.
- Тестирование параллелизма основывается на семантике чередования [9]. Оно выполняется с помощью сбора всех наблюдаемых событий (вызовов функций, возвращений результата функциями, а также, возможно, других событий, создаваемых тестируемой системой) и построения из них последовательности, в рамках которой все контракты отдельных событий выполняются. Ошибка обнаруживается, если такую последовательность не удается построить.

UniTESK применялся в рамках проекта OLVER [10] для построения тестового набора проверки на соответствие части Core стандарта Linux Standard Base (LSB), описывающей системные библиотеки в объеме, примерно соответствующем стандарту POSIX. Полученный тестовый набор содержит тесты для 1532 функций LSB. Этот же подход использовался при создании тестового набора для проверки соответствия стандарту ARINC-653(I) [11], описывающему 54 функции.

Другой метод построения тестов, не использующий формальные спецификации, но основанный на формальном анализе требований, был использован для построения тестов для математических функций,

работающих с числами с плавающей точкой в рамках системных библиотек, описываемых стандартом POSIX [12]. Метод основан на использовании в качестве источников тестовых данных специальных значений типов с плавающей точкой; значений, мантисса которых удовлетворяет некоторому набору паттернов; границ интервалов специфического поведения тестируемой функции (в рамках которых сохраняется характер монотонности, знак, известные асимптотики и пр.); а также чисел, для которых вычисление точно округленного значения функции наиболее трудоемко, т.е. требует значительно более высокой точности по сравнению с обычными числами. На настоящий момент разработаны тестовые наборы для 120 функций.

2.2 Неформальные методы

Несколько методов построения тестов, используемых в проектах ИСП РАН, не применяют формальные спецификации, но нацелены на обеспечение строгой прослеживаемости требований.

Первый такой метод [13] использует ручное создание параметризованных тестов. Он применялся для построения тестов для более чем 4000 функций из системных библиотек Linux и позволил выявить около 40 ошибок в них.

Другой метод [14] основан на автоматической генерации простейших тестов работоспособности (проверяющих только базовую функциональность) на основе заданных вручную и хранимых в базе данных процедур стандартной инициализации значений типов параметров и проверки простейших свойств корректности для типов результатов. Этот подход обеспечивает лишь самое простейшее тестирование, но позволяет охватить большое число библиотечных функций с небольшими трудозатратами. Он применялся для тестирования библиотек Linux, содержащих около 20000 функций.

Помимо описанных методов построения тестов был разработан метод конфигурационного тестирования – выбора представительного набора конфигураций ОС, основанный на использовании так называемых покрывающих наборов и позволяющий проверить взаимодействие различных конфигурируемых элементов ее функциональности друг с другом [15].

2.3 Динамический анализ и тестирование устойчивости к сбоям

Для использования методов мониторинга при проверке свойств ядра Linux в ИСП РАН разработана среда KEDR [16]. Она позволяет перехватывать обращения одного из модулей ядра к другим модулям и проверять некоторые свойства их корректности в динамике. На основе этой среды были реализованы следующие техники.

- KEDR Leak Check, предназначенный для обнаружения утечек памяти при работе модулей ядра. Он несколько более удобен, чем аналогичный

инструмент kmemleak [17], входящий в дистрибутивы Linux, но не применим для проверки работы сердцевины ядра.

- Kernel Strider [18], предназначенный для обнаружения гонок по данным, т.е., ситуаций, в ходе которых параллельные нити читают и пишут в одну область памяти в неопределенном порядке. Этот инструмент собирает информацию о работе заданного модуля ядра, которая затем может быть проанализирована с помощью ThreadSanitizer [19], инструмента обнаружения гонок от Google.
- KEDR Fault Simulation [20], предназначенный для тестирования отказоустойчивости. Этот инструмент позволяет записать обычное выполнение теста определенного модуля, а затем на основе этой записи создать набор тестов, в каждом из которых при одном из вызовов функций других модулей происходит сбой. С помощью этого инструмента было выявлено несколько ошибок при обработке сбоев в рамках реализаций файловых систем, таких как ext4.

Другим примером инструмента мониторинга, обнаруживающего гонки по данным, является RaceHound [21], реализующий для Linux те же идеи, которые были использованы в инструменте DataCollider [22] для ОС Windows. Этот инструмент позволяет проанализировать выбранный набор инструкций в рамках одной нити, выявить в динамике область памяти, с которой работают заданные инструкции, установить на запись в них аппаратную точку останова и вставить дополнительные интервалы ожидания при доступе к памяти из других нитей. Если это приводит к доступу к помеченной области, такая ситуация фиксируется как гонка.

3. Статический анализ

Для повышения производительности большая часть кода промышленных ОС работает в привилегированном режиме, что может привести при некорректной работе этого кода к повреждению важных данных системы. Особенно эта проблема актуальна для кода драйверов, который разрабатывается сторонними программистами, обычно не знакомыми близко с правилами написания корректного кода, действующими для ядра. В результате больше половины ошибок, обнаруживаемых в ядре Linux, связано с кодов драйверов [23]. Практически то же соотношение наблюдается и для Windows [24].

Чтобы уменьшить число ошибок, совершаемых при написании кода ядра, нужны специализированные инструменты, способные проверять правила корректного использования интерфейсных функций ядра. В Microsoft Research для этой цели разработан Static Driver Verifier [25] (в первых версиях называвшийся SLAM), способный решать эту задачу для Windows. Аналогичный инструмент для Linux, названный Linux Driver Verifier [26,27]

(LDV), разработан в ИСП РАН. Поддерживаемый им метод верификации описывается следующим образом.

- Правила корректного использования интерфейсных функций ядра формулируются в виде программных контрактов в нотации, расширяющей язык C. Они интерпретируются как аспектные вставки, которые вставляются во все места вызова соответствующих функций из проверяемого модуля. При нарушении этих правил, вставляемый код создает специфическую ошибочную ситуацию.
- Для функций модуля создается модель использования, задающая все возможные сценарии обращений к ним извне. Это важно, поскольку многие модули ядра и драйвера не вызываются явно, обращения к ним организует само ядро по определенным правилам.
- Код проверяемого модуля обрабатывается инструментом, генерирующим аспектные вставки и создание ошибок, а также дополняется моделью использования.
- Основную проверку выполняет инструмент статической верификации (обычно используются BLAST [28] или CPAChecker [29]), который пытается найти сценарий работы полученного кода, при котором ошибочная ситуация становится достижимой. Если это удается, значит обнаружена ошибка, которая состоит в нарушении определенного правила корректного использования функций ядра при некотором сценарии его работы. Достижимость анализируется при помощи метода, управляемого контрпримерами уточнения абстракций (CEGAR [30]), который строит все более точные модели работы кода, пока либо не обнаружит достижимость ошибки в рамках и модели, и реального кода, либо в рамках очередной модели не покажет ее недостижимость.

LDV обнаруживает 5-8 ошибок практически в каждом очередном выпуске ядра Linux, общее число найденных и исправленных разработчиками ошибок уже более 350. При этом с помощью LDV регулярно проверяется код примерно 4000 модулей ядра.

Еще один инструмент статического анализа CPALocator [31] предназначен для выявления гонок в коде ядра и разработан на основе CPAChecker.

4. Дедуктивная верификация

Дедуктивная верификация обычно считается одним из самых аккуратных и строгих методов верификации. В то же время для продуктивного применения она требует значительных затрат труда высококвалифицированных специалистов. Систематический обзор попыток использования дедуктивной верификации кода ОС можно найти в [32].

В рамках одного из проектов ИСП РАН дедуктивная верификация использовалась для проверки свойств защищенности специализированной ОС,

основанной на ядре Linux и предназначенной для применения в государственных учреждениях для работы с секретными данными [33,34]. Эта ОС реализует достаточно сложную модель политик безопасности (МРОСЛ ДП-модель), интегрирующую механизм контроля доступа, основанного на ролях, контроля целостности и многоуровневую защиту. Эти механизмы реализованы с помощью инфраструктуры Linux Security Modules (LSM) [35], обеспечивающую перехват всех операций доступа к данным или процессам в ядре Linux.

В рамках проекта МРОСЛ ДП-модель была формализована на языке Event-B и верифицирована с помощью среды интерактивного дедуктивного анализа Rodin. Все свойства безопасности модели (например, что процесс с низким уровнем доступа не может получить доступ на чтение к высококонфиденциальным данным, или что процесс не может получить доступ к данным, не имея привязки к роли, обладающей правом на такой доступ) были записаны в виде инвариантов и доказаны. Далее, интерфейсные функции LSM были специфицированы с помощью контрактов, построенных по аналогии с контрактами похожих операций МРОСЛ ДП-модели, и для них также были сформулированы и доказаны аналогичные свойства безопасности.

К сожалению, интерфейс операций модели и интерфейс LSM сильно отличаются, и для построения контрактов операций LSM потребовалось создать отдельную формальную модель, свойства безопасности которой являются переформулировкой свойств безопасности исходной модели в терминах типов данных, с которыми работают операции LSM. Полученные во второй модели контракты операций LSM были переписаны на расширении языка C, называемом ACSL и используемом в инструменте дедуктивной верификации кода Frama C/Jessie [36]. Используемая реализация LSM в дальнейшем должна быть верифицирована на соответствие полученным контрактам. Эта работа пока не закончена, поскольку верификация кода функций ядра потребовала существенной доработки инструментов дедуктивной верификации.

Хотя верификация кода еще не доведена до конца, множество ошибок было выявлено как в самой исходной модели политик безопасности, так и в коде, при попытках проведения верификации отдельных функций. Исправление этих ошибок, хотя и не дает полных гарантий защищенности системы, существенно повышает доверие к ней.

5. Заключение

В статье рассмотрены задачи верификации и анализа современных промышленных операционных систем и проведен систематический обзор техник верификации, используемых для проверки отдельных компонентов и свойств промышленных ОС в проектах ИСП РАН. Хотя конечная цель полномасштабной верификации такой ОС пока недостижима, опыт использования рассмотренных методов показывает, что за последние годы в

сообществе исследователей и разработчиков достигнуто существенное продвижение на пути к ней.

Использование методов и инструментов различных типов, использующих тестирование, мониторинг, статический и дедуктивный анализ, может взаимно обогатить их и позволить получать более аккуратные и полные результаты за счет заимствования специфических техник моделирования или анализа определенных свойств. Это хорошо видно на примере использования одинаковых моделей памяти в рамках инструментов статического анализа и дедуктивной верификации [37].

Список литературы

[1]. Why is the Linux kernel 15+ million lines of code? Режим доступа: https://unix.stackexchange.com/questions/223746/why-is-the-linux-kernel-15-million-lines-ofcode/223770, дата обращения 10.12.2018.

[2]. How Many Lines of Code in Windows XP? Режим доступа: https://www.facebook.com/windows/posts/155741344475532, дата обращения 10.12.2018.

[3]. Герлиц Е.А., Кулямин В.В., Максимов А.В., Петренко А.К., Хорошилов А.В., Цыварев А.В. Тестирование операционных систем. Труды ИСП РАН, том 26, вып. 1, 2014 г., стр. 73-108. DOI: 10.15514/ISPRAS-2014-26(1)-3.

[4]. Bevier W.R. Kit: a Study in Operating System Verification. IEEE Transactions on Software Engineering, vol. 15, no. 11, 1989, pp. 1382-1396.

[5]. Alkassar E., Paul W.J., Starostin A., Tsyban A. Pervasive Verification of an OS Microkernel. Lecture Notes in Computer Science, vol. 6217, 2010, pp. 71-85.

[6]. Klein G., Andronick J., Elphinstone K., Murray T., Sewell T., Kolanski R., Heiser G. Comprehensive Formal Verification of an OS Microkernel. ACM Transactions on Computer Systems, vol. 32, no. 1, 2014.

[7]. Burdonov I., Kossatchev A., Petrenko A., Galter D. KVEST: Automated Generation of Test Suites from Formal Specifications. Lecture Notes in Computer Science, vol. 1708, 1999, pp. 608-621.

[8]. Bourdonov I.B., Kossatchev A.S., Kuliamin V.V., Petrenko A.K. UniTesK Test Suite Architecture. Lecture Notes in Computer Science, vol. 2391, 2002, pp. 77-88.

[9]. Kuliamin V.V., Petrenko A.K., Pakoulin N.V., Kossatchev A.S., Bourdonov I.B. Integration of Functional and Timed Testing of Real-Time and Concurrent Systems. Lecture Notes in Computer Science, vol. 2890, 2003, pp. 450-461.

[10]. Grinevich A., Khoroshilov A., Kuliamin V., Markovtsev D., Petrenko A., Rubanov V. Formal Methods in Industrial Software Standards Enforcement. Lecture Notes in Computer Science, vol. 4378, 2006, pp. 456-466.

[11]. Maksimov A. Requirements-based conformance testing of ARINC 653 real-time operating systems. In Proc. of the International Conference on Data Systems in Aerospace (DASIA 2010), 2010.

[12]. Kuliamin V. Standardization and Testing of Mathematical Functions. Lecture Notes in Computer Science, vol. 5947, 2009, pp. 257-268.

[13]. Khoroshilov A., Rubanov V., Shatokhin E. Automated Formal Testing of C API Using T2C Framework. In Proc. of the International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2008), 2008, pp.56-70.

[14]. Zybin R.S., Kuliamin V.V., Ponomarenko A.V., Rubanov V.V., Chernov E.S. Automation of broad sanity test generation. Programming and Computer Software, vol. 34, no. 6, 2008, pp. 351-363.

[15]. Кулямин В.В. Комбинаторная генерация программных конфигураций ОС. Труды ИСП РАН, том 23, 2012 г., стр. 359-370. DOI: 10.15514/ISPRAS-2012-23-20.

[16]. Shatokhin E. Using Dynamic Analysis to Hunt Down Problems in Kernel Modules. Presentation at LinuxCon Europe, 2011. Режим доступа: http://linuxtesting.org/2011-LinuxConEurope-Shatokhin-KEDR.pdf, дата обращения 10.12.2018.

[17]. kmemleak description. Режим доступа: https://www.kernel.org/doc/Documentation/kmemleak.txt, дата обращения 10.12.2018.

[18]. Kernel Strider. Режим доступа: https://code.google.com/p/kernel-strider/, дата обращения 10.12.2018.

[19]. Serebryany K., Iskhodzhanov T. ThreadSanitizer: data race detection in practice. In Proc. of the Workshop on Binary Instrumentation and Applications (WBIA 2009), 2009, pp. 62-71.

[20]. Цыварев А.В., Хорошилов А.В. Использование симуляции сбоев при тестировании компонентов ядра ОС Linux. Труды ИСП РАН, том 27, вып. 5, 2015 г., стр. 157-174. DOI: 10.15514/ISPRAS-2015-27(5)-9.

[21]. Race Hound tool. Режим доступа: http://forge.ispras.ru/projects/race-hound/, дата обращения 10.12.2018.

[22]. Erickson J., Musuvathi M., Burckhardt S., Olynyk K. Effective data-race detection for the kernel. Proc. of the USENIX Conference on Operating systems design and implementation, 2010, pp. 151-162.

[23]. Мутилин В.С., Новиков Е.М., Хорошилов А.В. Анализ типовых ошибок в драйверах операционной системы Linux. Труды ИСП РАН, том 22, 2012 г., стр. 349-374. DOI: 10.15514/ISPRAS-2012-22-19.

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

[25]. 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. In Proc. of the ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys), 2006, pp. 73-85.

[26]. Мутилин В.С., Новиков Е.М., Страх А.В., Хорошилов А.В., Швед П.Е. Архитектура Linux Driver Verification. Труды ИСП РАН, том 20, 2011 г., стр. 163-187.

[27]. Захаров И.С., Мандрыкин М.У., Мутилин В.С., Новиков Е.М., Петренко А.К., Хорошилов А.В. Конфигурируемая система статической верификации модулей ядра операционных систем. Труды ИСП РАН, том 26, вып. 2, 2014 г., стр. 5-42. DOI: 10.15514/ISPRAS-2014-26(2)-1.

[28]. 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), vol. 5, 2007, pp. 505-525.

[29]. Beyer D., Keremoglu M.E. CPAchecker: A tool for configurable software verification. Lecture Notes in Computer Science, vol. 6806, 2011, pp. 184-190.

[30]. Clarke E., Grumberg O, Jha S., Lu Y., Veith H. Counterexample-Guided Abstraction Refinement. Lecture Notes in Computer Science, vol. 1855, 2000, pp. 154-169.

[31]. Андрианов П.С., Мутилин В.С., Хорошилов А.В. Конфигурируемый метод поиска состояний гонок в операционных системах с использованием предикатных абстракций. Труды ИСП РАН, том 28, вып. 6, 2016 г., стр. 65-86. DOI: 10.15514/ISPRAS-2016-28(6)-5.

[32]. Klein G. Operating system verification – An overview. Sadhana, vol. 34, no. 1, pp. 27-69.

[33]. Devyanin P.N., Khoroshilov A.V., Kuliamin V.V., Petrenko A.K., Shchepetkov I.V. Formal Verification of OS Security Model with Alloy and Event-B. In Proc. of the International Conference on Abstract State Machines, 2014, pp. 309-313.

[34]. Devyanin P.N., Khoroshilov A.V., Kuliamin V.V., Petrenko A.K., Shchepetkov I.V. Comparison of specification decomposition methods in Event-B. Programming and Computer Software, vol. 42, no. 4, 2016, pp. 198-205.

[35]. Wright C., Cowan C., Morris J., Smalley S., Kroah-Hartman G. Linux Security Module Framework. In Proc. of the Ottawa Linux Symposium, 2002, pp. 6-16.

[36]. Marhé C., Moy Y. The Jessie Plugin for Deductive Verification in Frama-C. Режим доступа: http://krakatoa.lri.fr/jessie.pdf, дата обращения 10.12.2018.

[37]. Мандрыкин М.У., Мутилин В.С. Обзор подходов к моделированию памяти в инструментах статической верификации. Труды ИСП РАН, том 29, вып. 1, 2017 г., стр. 195-230. DOI: 10.15514/ISPRAS-2017-29(1)-12.

Component-based verification of operating systems

[1,2,3] *V.V. Kuliamin <kuliamin@ispras.ru>*
[1,2,3] *A.K. Petrenko <petrenko@ispras.ru>*
[1,2,3,4] *A.V. Khoroshilov <khoroshilov@ispras.ru>*

[1] *V.P.Ivannikov Institute for system programming, Russian Academy of Sciences A. Solzhenitsyn str., 25, Moscow, 109004, Russia.*
[2] *Lomonosov Moscow State University, Leninskie gory, 1, Moscow, 119991, Russia.*
[3] *FCS NRU Higher School of Economics, Myasnitskaya str., 20, Moscow, 101000, Russia.*
[4] *Moscow Institute of Physics and Technology, Institutskiy per., 9, Dolgoprudny, Moscow reg., 141700, Russia.*

Abstract. The paper presents recent results on the way towards accurate and complete verification of industrial operating systems (OS). We consider here OSes, either of general purpose or actively used in some industrial domain, elaborated and maintained for a significant time, and not touching research-related OSes usually developed as a proof-of-concept. In spite of the fact that the stated goal of accurate and complete verification of industrial OS is still unreachable, we consider its decomposition into tasks of verification of various functional OS components and various their properties. The paper shows that many of these tasks can be solved with the help of various modern verification techniques and their combinations. Proposed methods can be lately integrated into an approach to the final goal. The paper summarizes the experience of various OS component and features verification from the projects conducted in ISP RAS in the last years.

Keywords: operating system; software verification; testing; static analysis; deductive verification; monitoring.

DOI: 10.15514/ISPRAS-2018-30(6)-21

For citation: Kuliamin V.V., Petrenko A.K., Khoroshilov A.V. Component-based verification operating systems. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 6, 2018, pp. 367-382 (in Russian). DOI: 10.15514/ISPRAS-2018-30(6)-21

References

[1] Why is the Linux kernel 15+ million lines of code? Available at: https://unix.stackexchange.com/questions/223746/why-is-the-linux-kernel-15-million-lines-ofcode/223770, accessed 10.12.2018.

[2] How Many Lines of Code in Windows XP? Available at: https://www.facebook.com/windows/posts/155741344475532, accessed 10.12.2018.

[3] Gerlits E.A., Kuliamin V.V., Maksimov A.V., Petrenko A.K., Khoroshilov A.V., Tsyvarev A.V. Testing of Operating Systems. Trudy ISP RAN/Proc. ISP RAS, vol. 26, issue 1, 2014, pp. 73-108 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-3.

[4] Bevier W.R. Kit: a Study in Operating System Verification. IEEE Transactions on Software Engineering, vol. 15, no. 11, 1989, pp. 1382-1396.

[5] Alkassar E., Paul W.J., Starostin A., Tsyban A. Pervasive Verification of an OS Microkernel. Lecture Notes in Computer Science, vol. 6217, 2010, pp. 71-85.

[6] Klein G., Andronick J., Elphinstone K., Murray T., Sewell T., Kolanski R., Heiser G. Comprehensive Formal Verification of an OS Microkernel. ACM Transactions on Computer Systems, vol. 32, no. 1, 2014.

[7] Burdonov I., Kossatchev A., Petrenko A., Galter D. KVEST: Automated Generation of Test Suites from Formal Specifications. Lecture Notes in Computer Science, vol. 1708, 1999, pp. 608-621.

[8] Bourdonov I.B., Kossatchev A.S., Kuliamin V.V., Petrenko A.K. UniTesK Test Suite Architecture. Lecture Notes in Computer Science, vol. 2391, 2002, pp. 77-88.

[9] Kuliamin V.V., Petrenko A.K., Pakoulin N.V., Kossatchev A.S., Bourdonov I.B. Integration of Functional and Timed Testing of Real-Time and Concurrent Systems. Lecture Notes in Computer Science, vol. 2890, 2003, pp. 450-461.

[10] Grinevich A., Khoroshilov A., Kuliamin V., Markovtsev D., Petrenko A., Rubanov V. Formal Methods in Industrial Software Standards Enforcement. Lecture Notes in Computer Science, vol. 4378, 2006, pp. 456-466.

[11] Maksimov A. Requirements-based conformance testing of ARINC 653 real-time operating systems. In Proc. of the International Conference on Data Systems in Aerospace (DASIA 2010), 2010.

[12] Kuliamin V. Standardization and Testing of Mathematical Functions. Lecture Notes in Computer Science, vol. 5947, 2009, pp. 257-268.

[13] Khoroshilov A., Rubanov V., Shatokhin E. Automated Formal Testing of C API Using T2C Framework. In Proc. of the International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2008), 2008, pp.56-70.

[14] Zybin R.S., Kuliamin V.V., Ponomarenko A.V., Rubanov V.V., Chernov E.S. Automation of broad sanity test generation. Programming and Computer Software, vol. 34, no. 6, 2008, pp. 351-363.

[15] Kuliamin V.V. Combinatoric generation of operating system software configurations. Trudy ISP RAN/Proc. ISP RAS, vol. 23, 2012, pp. 359-370 (in Russian). DOI: 10.15514/ISPRAS-2012-23-20.

[16] Shatokhin E. Using Dynamic Analysis to Hunt Down Problems in Kernel Modules. Presentation at LinuxCon Europe, 2011. Available at: http://linuxtesting.org/2011-LinuxConEurope-Shatokhin-KEDR.pdf, accessed 10.12.2018.

[17] kmemleak description. Available at: https://www.kernel.org/doc/Documentation/kmemleak.txt, accessed 10.12.2018.

[18] Kernel Strider. Available at: https://code.google.com/p/kernel-strider/, accessed 10.12.2018.

[19] Serebryany K., Iskhodzhanov T. ThreadSanitizer: data race detection in practice. In Proc. of the Workshop on Binary Instrumentation and Applications (WBIA 2009), 2009, pp. 62-71.

[20] Tsyvaerv A., Khoroshilov A. Using Fault Injection for Testing Linux Kernel Components. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 5, 2015, pp. 157-174 (in Russian). DOI: 10.15514/ISPRAS-2015-27(5)-9.

[21] Race Hound tool. Available at: http://forge.ispras.ru/projects/race-hound/, accessed 10.12.2018.

[22] Erickson J., Musuvathi M., Burckhardt S., Olynyk K. Effective data-race detection for the kernel. Proc. of the USENIX Conference on Operating systems design and implementation, 2010, pp. 151-162.

[23] Mutilin V.S., Novikov E.M., Khoroshilov A.V. Analysis of typical faults in Linux operating system drivers. Trudy ISP RAN/Proc. ISP RAS, vol. 22, 2012, pp. 349-374 (in Russian). DOI: 10.15514/ISPRAS-2012-22-19.

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

[25] 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. In Proc. of the ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys), 2006, pp. 73-85.

[26] Mutilin V.S., Novikov E.M., Strakh A.V., Khoroshilov A.V., Shved P.E. Architectire of Linux Driver Verification. Trudy ISP RAN/Proc. ISP RAS, vol. 20, 2011, pp. 163-187 (in Russian).

[27] Zakharov I.S., Mandrykin M.U., Mutilin V.S., Novikov E.M., Petrenko A.K., Khoroshilov A.V. Configurable Toolset for Static Verication of Operating Systems Kernel Modules. Trudy ISP RAN/Proc. ISP RAS, vol. 26, issue 2, 2014, pp. 5-42 (in Russian). DOI: 10.15514/ISPRAS-2014-26(2)-1.

[28] 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), vol. 5, 2007, pp. 505-525.

[29] Beyer D., Keremoglu M.E. CPAchecker: A tool for configurable software verification. Lecture Notes in Computer Science, vol. 6806, 2011, pp. 184-190.

[30] Clarke E., Grumberg O, Jha S., Lu Y., Veith H. Counterexample-Guided Abstraction Refinement. Lecture Notes in Computer Science, vol. 1855, 2000, pp. 154-169.

[31] Andrianov P.S., Mutilin V.S., Khoroshilov A.V. Adjustable method with predicate abstraction for detection of race conditions in operating systems. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 6, 2016, pp. 65-86 (in Russian). DOI: 10.15514/ISPRAS-2016-28(6)-5.

[32] Klein G. Operating system verification – An overview. Sadhana, vol. 34, no. 1, pp. 27-69.

[33] Devyanin P.N., Khoroshilov A.V., Kuliamin V.V., Petrenko A.K., Shchepetkov I.V. Formal Verification of OS Security Model with Alloy and Event-B. In Proc. of the International Conference on Abstract State Machines, 2014, pp. 309-313.

[34] Devyanin P.N., Khoroshilov A.V., Kuliamin V.V., Petrenko A.K., Shchepetkov I.V. Comparison of specification decomposition methods in Event-B. Programming and Computer Software, vol. 42, no. 4, 2016, pp. 198-205.

[35] Wright C., Cowan C., Morris J., Smalley S., Kroah-Hartman G. Linux Security Module Framework. In Proc. of the Ottawa Linux Symposium, 2002, pp. 6-16.

[36] Marhé C., Moy Y. The Jessie Plugin for Deductive Verification in Frama-C. Available at: http://krakatoa.lri.fr/jessie.pdf, accessed 10.12.2018.

[37] Mandrykin M.U., Mutilin V.S. Survey of memory modeling methods in static verification tools. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 1, 2017, pp. 195-230 (in Russian). DOI: 10.15514/ISPRAS-2017-29(1)-12.