Инкрементальное построение спецификаций моделей окружения и требований для подсистем монолитного ядра операционных систем[1]

И.С. Захаров <ilja.zakharov@ispras.ru>
Е.М. Новиков <novikov@ispras.ru>
Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. Методы и инструменты автоматической статической верификации позволяют выявить все ошибки искомых видов в целевых программах при выполнении определенных предположений даже в условиях отсутствия полных моделей и формальных спецификаций. Эта возможность является основой предлагаемого в работе метода инкрементального построения спецификаций моделей окружения и требований для подсистем монолитного ядра операционных систем. Данный метод был реализован в системе статической верификации Klever и применен для проверки подсистемы поддержки терминальных устройств ядра ОС Linux.

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

DOI: 10.15514/ISPRAS-2017-29(6)-2

Для цитирования: Захаров И.С., Новиков Е.М. Инкрементальное построение спецификаций моделей окружения и требований для подсистем монолитного ядра операционных систем. Труды ИСП РАН, том 29, вып. 6, 2017 г., стр. 25-48. DOI: 10.15514/ISPRAS-2017-29(6)-2

1. Введение

Проведенное ранее исследование показало, что одними из наиболее перспективных методов и инструментов для обнаружения всех ошибок искомого вида в монолитном ядре операционных систем (далее — ОС)

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

являются методы и инструменты автоматической статической верификации [1]. Применение данных методов и инструментов на практике затруднено, поскольку они способны верифицировать программы размером несколько тысяч или десятков тысяч строк кода в зависимости от того, насколько целевой код релевантен проверяемым требованиям, от доступных вычислительных ресурсов и от многочисленных настроек, например, от выбора модели памяти и решателя. Размер монолитного ядра ОС без различных расширений, таких как драйверы устройств, может составлять несколько миллионов строк кода [2], поэтому в качестве первого шага в направлении использования методов и инструментов автоматической статической верификации для проверки качества монолитного ядра ОС были предложены различные подходы к декомпозиции ядра на подсистемы [1].

В данной работе предлагается использовать тот подход к декомпозиции монолитного ядра ОС на подсистемы, который основан на принципе разделения четко выраженной функциональности. Преимуществами данного подхода являются:

- возможность выделять достаточно компактные по размеру подсистемы. Если некоторая подсистема окажется слишком большой или сложной для инструментов автоматической статической верификации, то всегда можно продолжить декомпозицию, основываясь на том же принципе;
- отсутствие необходимости разрабатывать модель функций целевой подсистемы для ее проверки, поскольку инструменты автоматической статической верификации сами строят модель всех функций целевой подсистемы;
- достаточно простая реализация, поскольку для задания подсистем нужно перечислить названия соответствующих директорий и/или файлов с их исходным кодом.

Опыт предыдущего применения инструментов автоматической статической верификации на практике продемонстрировал необходимость моделировать окружение для целевых программ и задавать требования к ним достаточно точным образом [3-6]. При проверке таких сценариев взаимодействия целевой подсистемы монолитного ядра ОС с ее окружением[2], которые никогда не осуществляются на практике в силу тех или иных причин, могут быть выданы ложные сообщения об ошибках. Также инструменты автоматической статической верификации могут пропустить ошибки, если не моделируются какие-то пути выполнения, которые возможны при работе подсистемы в реальном окружении. Аналогично, если требования сформулированы

[2] Окружением для подсистемы монолитного ядра ОС являются другие подсистемы монолитного ядра ОС, различные расширения, такие как драйверы устройств, а также, возможно, косвенным образом аппаратура и приложения из пользовательского пространства.

недостаточно точным образом, то могут быть как выданы ложные сообщения об ошибках, так и пропущены ошибки соответствующих видов.

Примечательным является то, что в отличии от методов и инструментов формальной (дедуктивной) статической верификации, методы и инструменты автоматической статической верификации не предполагают разработку полной модели и формальных спецификаций, которые покрывают все функциональные требования и некоторые дополнительные свойства. Обнаруживать ошибки определенных видов, а также доказывать их отсутствие при выполнении определенных предположений возможно даже при наличии неточных моделей и спецификаций. По мнению авторов работы это обусловлено совокупностью следующих факторов.

- При использовании методов и инструментов автоматической статической верификации традиционно не делается попытка доказать полную формальную корректность целевой программы в смысле соответствия реализации ее модели и спецификациям, а ищутся нарушения достаточно общих требований. К числу таких требований относится, например, выполнение правил безопасного программирования (отсутствие разыменований нулевого указателя, выходов за границы буферов и т.п.) и правил корректного использования программного интерфейса, например, корректное использование специфичного механизма синхронизации для многопоточных программ.
- Проверяется корректность не отдельных функций, а всей программы в целом. При этом для всех функций, входящих в целевую программу, автоматически строится их модель достаточно точная для проверки заданных требований.
- Разработчики постоянно предлагают новые и совершенствуют существующие методы статической верификации, которые уже способны автоматически строить достаточно точные модели и автоматически доказывать выполнимость заданных требований для средних по размеру программ с разумными ограничениями на используемые вычислительные ресурсы и общее время проверки.
- При автоматической статической верификации делаются определенные предположения. Например, инструменты игнорируют код на языке ассемблера. Как правило, это не приводит к проблемам, поскольку такого кода в монолитном ядре ОС не так много [2]. При необходимости можно разработать соответствующую модель на языке программирования Си.

В следующем разделе предлагается метод инкрементального построения спецификаций моделей окружения и требований для подсистем монолитного ядра ОС, который во многом опирается на вышеупомянутую возможность методов и инструментов автоматической статической верификации. Раздел 3

посвящен реализации предложенного метода. В разделе 4 описан процесс инкрементального построения спецификаций моделей окружения и требований для подсистемы поддержки терминальных устройств ядра ОС Linux. Также в данном разделе представлены результаты верификации, полученные на различных этапах данного процесса. В заключении делаются выводы по результатам проведенного исследования и намечаются дальнейшие возможные шаги.

2. Метод инкрементального построения спецификаций моделей окружения и требований для подсистем монолитного ядра ОС

Универсального метода построения моделей и спецификаций для статической верификации программ не существует. Для каждого конкретного проекта подходящий метод выбирается исходя из множества различных ограничений. Наиболее важными по мнению авторов работы являются следующие.

- Возможности и особенности языков моделирования и спецификации, а также непосредственно самих инструментов статической верификации. Например, существует подход, который предлагает разрабатывать программы на специализированном языке программирования таким образом, чтобы спецификации задавались одновременно с реализацией [7]. Такой подход не применим для статической верификации существующих программ. Иногда различные пути, удобные при различных вариантах использования, существуют даже в рамках одного подхода [8]. Часто возможностей существующих инструментов статической верификации оказывается недостаточно: некоторые конструкции языков программирования и их расширений не поддерживаются или для проведения проверки требуется чрезвычайно большое количество вычислительных ресурсов [9-10].
- Квалификация самих верификаторов[3]. Для построения качественных моделей и спецификаций требуется очень хорошее понимание целевой программы, используемых языков моделирования и спецификации, а также возможностей и особенностей инструментов статической верификации. Требования к целевым программам в неформальном виде могут быть не полны или даже противоречивы. Документация, описывающая языки моделирования и спецификации, а также инструменты статической верификации, может быть не достаточно качественной. Как правило, необходимые знания верификаторы приобретают по мере непосредственного выполнения

[3] В данной статье под верификаторами понимаются люди, которые занимаются разработкой моделей, спецификаций, проведением статической верификации и анализом результатов верификации.

конкретных проектов, а не на каких-либо обучающих курсах.

- Различные экономические, организационные и политические ограничения. Известно, что трудоемкость формальной (дедуктивной) верификации чрезвычайно большая и может превосходить трудоемкость разработки программы на порядок [11]. В связи с этим полные модели и формальные спецификации строятся только для наиболее критичных программ или их компонентов общим размером несколько тысяч или десятков тысяч строк кода. В других случаях используют альтернативные методы обеспечения качества программного обеспечения, например, тестирование или статический анализ.

Предлагаемый метод инкрементального построения спецификаций моделей окружения и требований для подсистем монолитного ядра ОС учитывает, с одной стороны, необходимость статической верификации большого, сложного и не всегда хорошо документированного кода, а с другой стороны, возможность методов и инструментов автоматической статической верификации обнаруживать ошибки определенных видов, а также доказывать их отсутствие при выполнении определенных предположений даже при наличии не достаточно точных моделей и спецификаций. Метод состоит из трех шагов, которые рассмотрены далее в соответствующих подразделах.

2.1 Покрытие кода целевых подсистем монолитного ядра ОС

Во введении было отмечено, что инструменты автоматической статической верификации могут пропустить ошибки, если не проверяются какие-то пути выполнения, которые возможны при работе в реальном окружении. Типичным примером служит код, который недостижим из указанной точки входа программы[4] ни на одном из возможных путей выполнения. Для того, чтобы увеличить покрытие кода, необходимо построить соответствующие модели окружения. Например, для программ с функцией *main* в модели окружения требуется вызвать данную функцию со всеми возможными и допустимыми значениями ее аргументов или хотя бы некоторым их подмножеством[5]. Аналогичным образом необходимо вызывать библиотечные функции с различными значениями их аргументов, в правильном порядке и в правильном контексте. При проверке отдельных компонентов или подсистем задача моделирования окружения становится еще более важной, поскольку при анализе вдобавок может отсутствовать как тот код программы, который

[4] Инструменты автоматической статической верификации анализируют целевую программу, начиная с некоторой определенной точки входа. Например, такой точкой входа может быть функция *main*.

[5] Инструменты автоматической статической верификации предлагают специальный механизм, позволяющий задавать сразу все возможные значения определенного базового типа или некоторые диапазоны таких значений.

использует программный интерфейс целевого компонента/подсистемы, так и тот код программы, который используется ими.

В условиях большого количества, сложности и размера подсистем монолитного ядра ОС[6] и недостаточно точного неформального описания требований к их программному интерфейсу, в данной работе для моделирования всевозможных сценариев взаимодействия целевой подсистемы с ее окружением предлагается следующее.

- Анализировать подсистему совместно с теми расширениями монолитного ядра ОС, которые ее используют и для которых разрабатывать модели окружения может быть проще по тем или иным причинам[7]. Например, в случае ядра ОС Linux — вместе с загружаемыми модулями, для которых уже разработаны некоторые достаточно точные спецификации моделей окружения [6].
- Разрабатывать дополнительные спецификации моделей окружения в случае необходимости. Данный шаг является опциональным, однако, как будет показано в следующем разделе, без него не всегда удается увеличить покрытие кода целевой подсистемы монолитного ядра ОС. Если покрыть дополнительный код можно за счет совместного анализа подсистемы с некоторым расширением монолитного ядра ОС, то предлагается воспользоваться данной возможностью в первую очередь, а не разрабатывать дополнительную спецификацию модели окружения.

В обоих случаях в первую очередь необходимо обращать внимание на покрытие наиболее важных и больших участков кода целевой подсистемы монолитного ядра ОС. Например, перед использованием некоторого программного интерфейса обязательно нужно выполнить инициализацию используемых им структур данных. Если одно расширение монолитного ядра ОС позволяет покрыть больше кода целевой подсистемы, чем другое, то предпочтение следует отдать первому.

Стоит отметить, что при таком подходе к покрытию кода целевых подсистем монолитного ядра ОС могут покрываться не все возможные пути выполнения, которые возможны при работе в реальном окружении. Например, некоторый программный интерфейс подсистемы может предоставлять возможности, которые не используются ни одним из выбранных расширений монолитного ядра ОС, а имеющиеся спецификации моделей окружения могут не описывать работу с данным программным интерфейсом или делать это не достаточно

[6] Например, для ядра ОС Linux предполагается порядка 100 подсистем размером порядка 10 тысяч строк кода [1].

[7] Вообще говоря, при таком подходе предполагается, что расширения монолитного ядра ОС корректным образом используют программный интерфейс его подсистем. Выявлять ошибки, которые возникают в противном случае, предполагается другими средствами [5-6, 12-14].

точным образом. Тем не менее это соответствует общей идее метода, поскольку предполагается разрабатывать спецификации моделей окружения инкрементальным образом. При необходимости и возможности спецификации можно доработать, благодаря чему покрыть дополнительный код.

2.2 Уточнение спецификаций моделей окружения

Пути выполнения, которые покрываются в рамках выполнения предыдущего шага метода, могут никогда не осуществляться на практике. Например, из-за неточности спецификаций моделей окружения функции могут быть вызваны с недопустимыми значениями аргументов, в неправильном порядке или в неправильном контексте. Кроме того, некоторые пути выполнения могут быть покрыты, поскольку инструменты автоматической статической верификации делают предположения. Например, предполагается, что функции, для которых при анализе нет ни определений, ни моделей, не имеют побочных эффектов и возвращают произвольное значение допустимое типом. Если в реальности некоторая функция всегда возвращает 0 в случае успеха и некоторое отрицательное целое число, обозначающее код ошибки, то предположение, что функция может также вернуть положительное целое число может привести к анализу невыполнимых путей. В конечном итоге инструменты автоматической статической верификации могут выдать ложные сообщения об ошибках, которые существенно затрудняют анализ результатов верификации.

На втором шаге метода предлагается постепенно уточнять спецификации моделей окружения в той степени, насколько это необходимо для проверки выполнения заданных требований для целевых подсистем монолитного ядра ОС с приемлемым количеством ложных сообщений об ошибках. Например, большинство подсистем обращаются к подсистеме управления памятью, поэтому с большой вероятностью потребуется разработать ее модель[8]. Моделировать те подсистемы, которые не используются целевыми, не потребуется.

2.3 Разработка и уточнение спецификаций требований

В данной работе предлагается проверять в первую очередь те же требования, которые традиционно проверяются при статической верификации расширений монолитного ядра ОС [5-6, 12-13]. К ним относятся правила безопасного программирования, которым должны следовать все программы на языке программирования Си, и правила корректного использования программного интерфейса монолитного ядра ОС. Кроме того, необходимо разрабатывать

[8] В случае с монолитным ядром ОС Linux такая модель уже отчасти разработана, поскольку загружаемые модули также используют подсистему управления памятью [5-6, 12-14].

дополнительные спецификации требований в тех случаях, когда требуется проверять специфичные правила корректности, которым должны следовать только подсистемы монолитного ядра ОС.

Несмотря на сходство требований, проверяемых при статической верификации подсистем монолитного ядра ОС, с требованиями к его расширениям, для проверки подсистем может потребоваться дополнительные уточнения спецификаций. Например, в отличии от расширений монолитного ядра ОС, которые могут загружаться и выгружаться динамически и на момент завершения работы должны освобождать все выделенные для них ресурсы тем или иным способом, подсистемы монолитного ядра ОС не всегда обязаны это делать, поскольку нет смысла освобождать выделенные ресурсы, когда ОС завершает свою работу.

3. Реализация предложенного метода

Предложенный метод был реализован для подсистем ядра ОС Linux в системе статической верификации Klever [12-14]. При этом были задействованы все те возможности, которые уже были реализованы в Klever для поддержки статической верификации загружаемых модулей ядра ОС Linux. Дополнительно были реализованы следующие возможности.

- Для реализации шага метода, представленного в подразделе 2.1:
  - Проверка файлов и директорий с исходным кодом целевой подсистемы ядра ОС Linux совместно с исходным кодом одного из загружаемых модулей.
  - Возможность напрямую вызывать функции программного интерфейса из моделей окружения.
- Для реализации шага метода, представленного в подразделе 2.2:
  - Поддержка нескольких программных интерфейсов, имеющих близкую синтаксическую структуру. Например, могут регистрироваться несколько наборов функций-обработчиков, сохраняемых в переменных с одним и тем же структурным типом.
  - Запрет использования программного интерфейса в моделях окружения в тех случаях, когда он еще не зарегистрирован.
- Для решения проблемы, которая была упомянута в подразделе 2.3:
  - Возможность отключать проверку финального состояния на момент завершения работы подсистемы.

Для загружаемых модулей ядра ОС Linux большинство данных возможностей также необходимы, но они не являются столь существенными. Без отсутствия соответствующей поддержки для подсистем ядра ОС Linux либо не удавалось

достичь приемлемого покрытия кода, либо выдавалось чрезвычайно много ложных сообщений об ошибках.

Помимо доработки компонентов системы статической верификации Klever было исправлено несколько ошибок и сделано несколько дополнительных оптимизаций в инструменте автоматической статической верификации CPAchecker [15].

4. Процесс инкрементального построения спецификаций моделей окружения и требований для подсистемы поддержки терминальных устройств ядра ОС Linux

Реализация предложенного метода инкрементального построения спецификаций моделей окружения и требований для подсистем монолитного ядра ОС была использована для статической верификации подсистемы поддержки терминальных устройств ядра ОС Linux версии 3.14 (далее — TTY-подсистемы) в конфигурации *allmodconfig* на архитектуре *x86_64*. В состав TTY-подсистемы входят 11 файлов общим размером около 12 тысяч строк кода.

В данном разделе описан процесс инкрементального построения спецификаций моделей окружения и требований для данной подсистемы. Для каждого этапа этого процесса представлены соответствующие результаты верификации и сделаны выводы о возможных направлениях для их улучшения.

Все эксперименты проводились на виртуальных машинах OpenStack с 4 виртуальными ядрами процессора модели Intel Xeon E312xx (Sandy Bridge) и 32 ГБ оперативной памяти[9]. После всех доработок версия системы статической верификации Klever была 89d823d[10], а версии инструмента автоматической статической верификации CPAchecker для проверки соответствующих требований следующие:

- Правила корректного использования программного интерфейса ядра ОС Linux и оценка покрытия кода — *ldv-bam:25793*[11].
- Безопасная работа с памятью — *smg_witness_for_ldv:26912*.
- Поиск состояний гонок — *CPALockator:26692*.

Для генерации верификационных задач использовался тот набор спецификаций моделей окружений и требований, а также те параметры, которые используются в указанной версии системы статической верификации Klever по умолчанию. Кроме того, были уточнены некоторые существующие спецификации и разработаны новые (см. подразделы данного раздела). На решение каждой верификационной задачи отводилось максимум 15 минут процессорного времени и 10 ГБ оперативной памяти.

[9] http://www.bigdataopenlab.ru/about.html.

[10] Хэш коммита в Git-репозитории https://forge.ispras.ru/projects/klever/repository.

[11] Здесь и далее в данном списке название ветки и номер ревизии в SVN-репозитории https://svn.sosy-lab.org/software/cpachecker/.

4.1 Покрытие функций TTY-подсистемы при ее проверке с одним загружаемым модулем

В соответствии с предложенным методом TTY-подсистема проверялась вместе с загружаемыми модулями, которые ее использовали. В первую очередь была проведена оценка покрытия ее функций при использовании одного загружаемого модуля *drivers/tty/serial/jsm/jsm.ko*. На данном этапе не проводилась проверка выполнимости какого-либо конкретного требования, так как целью запуска инструмента автоматической статической верификации была оценка покрытия кода сверху. Результаты покрытия функций TTY-подсистемы для данного эксперимента приведены в табл. 1.

Табл. 1. Покрытие функций TTY-подсистемы при ее проверке с загружаемым модулем drivers/tty/serial/jsm/jsm.ko.

Table 1. Function coverage of the TTY subsystem checked together with loadable module drivers/tty/serial/jsm/jsm.ko.

Файл TTY-подсистемы	Процент покрытых функций	Количество покрытых функций	Общее количество функций
drivers/tty/n_tty.c	0%	0	70
drivers/tty/pty.c	10%	3	30
drivers/tty/sysrq.c	10%	5	50
drivers/tty/tty_audit.c	0%	0	12
drivers/tty/tty_buffer.c	0%	0	20
drivers/tty/tty_io.c	9%	11	116
drivers/tty/tty_ioctl.c	0%	0	28
drivers/tty/tty_ldisc.c	0%	0	36
drivers/tty/tty_ldsem.c	0%	0	20
drivers/tty/tty_mutex.c	0%	0	5
drivers/tty/tty_port.c	0%	0	25
Всего	**5%**	**19**	**412**

Детальный анализ покрытия функций TTY-подсистемы показал, что покрытие обеспечено за счет инициализации TTY-подсистемы, которая отчасти схожа с инициализацией загружаемых модулей ядра ОС Linux. Вывод: необходимо анализировать TTY-подсистемы с другими загружаемыми модулями помимо *drivers/tty/serial/jsm/jsm.ko*.

4.2 Покрытие функций TTY-подсистемы при ее проверке с загружаемыми модулями терминальных устройств из директории *drivers/tty*

По аналогии с предыдущем подразделом был проведен эксперимент, в котором TTY-подсистема проверялась вместе со всеми входящими в состав ядра ОС Linux загружаемыми модулями терминальных устройств из директории *drivers/tty*. Всего таких модулей оказалось 38: *drivers/tty/{cyclades.ko, ipwireless/ipwireless.ko, isicom.ko, moxa.ko, mxser.ko, n_gsm.ko, n_hdlc.ko, n_r3964.ko, n_tracerouter.ko, n_tracesink.ko, nozomi.ko, rocket.ko, serial/{8250/{8250.ko, 8250_dw.ko, 8250_pci.ko, serial_cs.ko}, altera_jtaguart.ko, altera_uart.ko, arc_uart.ko, clps711x.ko, fsl_lpuart.ko, ifx6x60.ko, jsm/jsm.ko, kgdboc.ko, max3100.ko, mfd.ko, mrst_max3110.ko, pch_uart.ko, rp2.ko, sccnxp.ko, serial_core.ko, sh-sci.ko, st-asc.ko, timbuart.ko, uartlite.ko}, synclink.ko, synclink_gt.ko, synclinkmp.ko}*. TTY-подсистема проверялась с каждым из этих модулей по-отдельности. Статическая верификация TTY-подсистемы одновременно со всеми этими модулями не рассматривалась ввиду слишком высокой сложности и большого объема исходного кода такого объединения. Результаты покрытия функций TTY-подсистемы приведены в табл. 2.

Табл. 2. Покрытие функций TTY-подсистемы при ее проверке с загружаемыми модулями терминальных устройств из директории drivers/tty.

Table 2. Function coverage of the TTY subsystem checked together with TTY loadable modules from directory drivers/tty.

Файл TTY-подсистемы	Процент покрытых функций	Количество покрытых функций	Общее количество функций
drivers/tty/n_tty.c	0%	0	70
drivers/tty/pty.c	10%	3	30
drivers/tty/sysrq.c	10%	5	50
drivers/tty/tty_audit.c	0%	0	12
drivers/tty/tty_buffer.c	70%	14	20
drivers/tty/tty_io.c	18%	21	116
drivers/tty/tty_ioctl.c	10%	3	28
drivers/tty/tty_ldisc.c	11%	4	36
drivers/tty/tty_ldsem.c	40%	8	20
drivers/tty/tty_mutex.c	0%	0	5
drivers/tty/tty_port.c	28%	7	25
Всего	**16%**	**65**	**412**

Детальный анализ непокрытых функций файла TTY-подсистемы *drivers/tty/tty_port.c* показал следующее[12]:

- Исправление ошибок в компоненте генерации моделей окружения системы статической верификации Klever, которые перечислены в разделе 3, и уточнение спецификаций моделей окружения *tty_operations* (соответствующий заголовочный файл — *include/linux/tty_driver.h*) и *tty_port_operations* (соответствующий заголовочный файл — *include/linux/tty.h*) поможет покрыть 17 из 18 непокрытых функций[13]. Более того, более углубленный анализ показал, что перечисленные выше проблемы могут помешать повысить покрытие функций для всех остальных улучшений, которые представлены далее в этом списке.
- Проверка TTY-подсистемы с другими загружаемыми модулями помимо загружаемых модулей терминальных устройств из директории *drivers/tty* поможет покрыть 16 из 18 непокрытых функций. Однако расширение списка загружаемых модулей может потребовать существенно больше времени на проведение статической верификации, особенно при проверке выполнения большого количества требований.
- Статическая верификация с использованием альтернативной конфигурации, отличной от *allmodconfig*, поможет покрыть 3 из 18 непокрытых функций.
- Статическая верификация загружаемых модулей и подсистем ядра ОС Linux, которые собираются для других архитектур, может помочь покрыть 17 из 18 непокрытых функций. Однако данная возможность не поддерживается системой статической верификации Klever.
- Анализ дополнительных подсистем ядра ОС Linux совместно с TTY-подсистемой мог бы помочь покрыть 13 из 18 непокрытых функций, но данный подход представляется нецелесообразным, поскольку это может достаточно сильно усложнить верификационные задачи из-за увеличения объема анализируемого кода.
- Для покрытия одной функции *tty_port_register_device_attr* требуется использовать многомодульный анализ, то есть анализировать одновременно несколько загружаемых модулей, что ограничено поддерживается системой статической верификации Klever и достаточно сильно усложняет верификационные задачи.

[12] Подробные результаты анализа доступны по следующей ссылке https://getbox.ispras.ru/index.php/s/qvMVWq8HbUUR0v1.

[13] Здесь и далее дается приближенная оценка, поскольку покрытие функций может не всегда увеличиться на указанное значение в силу наличия второстепенных проблем, которые не учитываются при оценке.

Последние три возможности из списка выше в рамках данной работы более не будут рассматриваться, поскольку либо отсутствует необходимая поддержка в системе статической верификации Klever, либо ожидается существенное усложнение задач для инструментов автоматической статической верификации.

Выводы:

- Проверка TTY-подсистемы со всеми загружаемыми модулями терминальных устройств из директории *drivers/tty* увеличила покрытие функций всего на 10%.
- В первую очередь необходимо исправить ошибки в генераторе моделей окружения и уточнить спецификации моделей окружения *tty_operations* и *tty_port_operations*. По результатам этого будет определено, нужно ли анализировать TTY-подсистему с другими загружаемыми модулями помимо загружаемых модулей терминальных устройств из директории *drivers/tty* или в другой конфигурации, отличной от *allmodconfig*.

4.3 Покрытие функций TTY-подсистемы после исправления ошибок в генераторе моделей окружения и спецификаций моделей окружения *tty_operations* и *tty_port_operations*

В рамках данного и всех последующих подразделов TTY-подсистема проверялась вместе со всеми загружаемыми модулями терминальных устройств, которые участвовали в предыдущем эксперименте. Результаты покрытия функций TTY-подсистемы после намеченных в предыдущем подразделе исправлений приведены в табл. 3.

Табл. 3. Покрытие функций TTY-подсистемы после исправления ошибок в генераторе моделей окружения и уточнения спецификаций моделей окружения tty_operations и tty_port_operations.

Table 3. Function coverage of the TTY subsystem obtained with the fixed environment models generator and improved tty_operations and tty_port_operations environment model specifications.

Файл TTY-подсистемы	Процент покрытых функций	Количество покрытых функций	Общее количество функций
drivers/tty/n_tty.c	1%	1	70
drivers/tty/pty.c	96%	29	30
drivers/tty/sysrq.c	20%	10	50
drivers/tty/tty_audit.c	41%	5	12
drivers/tty/tty_buffer.c	80%	16	20
drivers/tty/tty_io.c	83%	97	116

drivers/tty/tty_ioctl.c	75%	21	28
drivers/tty/tty_ldisc.c	97%	35	36
drivers/tty/tty_ldsem.c	85%	17	20
drivers/tty/tty_mutex.c	100%	5	5
drivers/tty/tty_port.c	96%	24	25
Всего	**63%**	**260**	**412**

Детальный анализ непокрытых функций файлов TTY-подсистемы *drivers/tty/{pty.c, tty_audit.c, tty_buffer.c, tty_io.c, tty_ioctl.c, tty_ldisc.c, tty_ldsem.c, tty_port.c}* показал следующее[14].

- Разработка спецификаций моделей окружения *tty_ldisc_ops* (соответствующий заголовочный файл — *include/linux/tty_ldisc.h*), очередей работ (соответствующий заголовочный файл — *include/linux/workqueue.h*) и *class* (соответствующий заголовочный файл — *include/linux/device.h*) поможет покрыть 12, 5 и 1 из 43 непокрытых функций соответственно. При этом отсутствие данных спецификаций моделей окружения может помешать повысить покрытие функций для улучшений, которые представлены далее в этом списке.
- Проверка TTY-подсистемы с другими загружаемыми модулями помимо загружаемых модулей терминальных устройств из директории *drivers/tty* поможет покрыть 6 из 43 непокрытых функций.
- Из-за использования приближений реальное покрытие функций может быть меньше. Например, некоторый конкретный программный интерфейс может считаться покрытым даже несмотря на то, что он не был зарегистрирован, а был зарегистрирован другой программный интерфейс с похожей синтаксической структурой. Подобная неточность может стоить особенно дорого при проверке требований, поскольку при точном анализе будет выяснено, что код функций первого программного интерфейса недостижим, и, соответственно, возможные ошибки в этом коде не будут обнаружены. Для корректного анализа программного интерфейса, который должен покрываться благодаря существующей спецификации моделей окружения *file_operations* (соответствующий заголовочный файл — *include/linux/fs.h*) обязательно необходимо вызывать функцию инициализации TTY-подсистемы *tty_init*. Для спецификации модели окружения *tty_ldisc_ops* — *console_init*.
- Три функции *tty_pair_get_pty*, *tty_throttle* и *ldsem_down_write_trylock* не используются ни в одной подсистеме или загружаемом модуле

[14] Подробные результаты анализа доступны по следующей ссылке https://getbox.ispras.ru/index.php/s/q8VYCYdH8lGon3a.

ядра ОС Linux. Соответственно, покрыть данные функции возможно только за счет их явного вызова из модели окружения. Однако это не кажется целесообразным ввиду отсутствия пользователей данных функций.

- Покрыть функцию *ptmx_open* не удается, поскольку компонент системы статической верификации Klever, который генерирует модели окружения, не поддерживает используемый способ задания программного интерфейса.

Выводы:

- Исправление ошибок в генераторе моделей окружения и уточнение спецификаций моделей окружения *tty_operations* и *tty_port_operations* помогло увеличить покрытие функций TTY-подсистемы почти на 50%.
- В первую очередь необходимо разработать спецификацию модели окружения *tty_ldisc_ops*. Кроме того, модель окружения обязательно должна вызывать функции инициализации TTY-подсистемы *tty_init* и *console_init*.

4.4 Покрытие функций TTY-подсистемы после разработки спецификации модели окружения *tty_ldisc_ops* и вызова функций *tty_init* и *console_init* из модели окружения

С целью дальнейшего повышения покрытия кода, а также для достижимости кода при выполнении более точного анализа, используемого для проверки выполнимости требований, была разработана спецификация модели окружения *tty_ldisc_ops* и из модели окружения были вызваны функции *tty_init* и *console_init*. Получившееся в результате покрытие функций TTY-подсистемы приведено в табл. 4.

Табл. 4. Покрытие функций TTY-подсистемы после разработки спецификации модели окружения tty_ldisc_ops и вызова функций tty_init и console_init из модели окружения.
Table 4. Function coverage of the TTY subsystem obtained with implemented tty_ldisc_ops environment model specification and invocations of tty_init and console_init functions from the environment model.

Файл TTY-подсистемы	Процент покрытых функций	Количество покрытых функций	Общее количество функций
drivers/tty/n_tty.c	98%	69	70
drivers/tty/pty.c	96%	29	30
drivers/tty/sysrq.c	20%	10	50
drivers/tty/tty_audit.c	75%	9	12
drivers/tty/tty_buffer.c	85%	17	20

drivers/tty/tty_io.c	87%	102	116
drivers/tty/tty_ioctl.c	92%	26	28
drivers/tty/tty_ldisc.c	94%	34	36
drivers/tty/tty_ldsem.c	85%	17	20
drivers/tty/tty_mutex.c	100%	5	5
drivers/tty/tty_port.c	92%	23	25
Всего	**83%**	**341**	**412**

В дополнение к результатам предыдущего подраздела детальный анализ непокрытых функций файлов TTY-подсистемы *drivers/tty/{n_tty.c, pty.c, tty_audit.c, tty_buffer.c, tty_io.c, tty_ioctl.c, tty_ldisc.c, tty_ldsem.c, tty_port.c}* показал следующее[15]:

- Разработка спецификаций моделей окружения очередей работ, *class* и *device_attribute* (соответствующий заголовочный файл — *include/linux/device.h*) поможет покрыть 5, 1 и 1 из 31 непокрытых функций соответственно.
- Проверка TTY-подсистемы с другими загружаемыми модулями помимо загружаемых модулей терминальных устройств из директории *drivers/tty* поможет покрыть 3 из 31 непокрытых функций.
- Покрыть 5 из 31 непокрытых функций возможно за счет анализа реализации функций регистрации программных интерфейсов при наличии их моделей.

Выводы:

- Разработка спецификации модели окружения *tty_ldisc_ops* и вызов функций *tty_init* и *console_init* из модели окружения помогли увеличить покрытие функций TTY-подсистемы на 20%.
- Дальнейшие улучшения либо затруднены, либо позволят увеличить покрытие функций несущественным образом (см. дополнительные рассуждения в подразделах 4.2 и 4.3).

4.5 Проверка выполнения требований для TTY-подсистемы

В данной работе для TTY-подсистемы проверялись требования 16 спецификаций: *generic:memory*, *linux:{alloc:{irq, spinlock, usb lock}*, *arch:io*, *drivers:base:{class, dma-mapping}*, *fs:sysfs*, *kernel:{locking:{mutex, rwlock, spinlock}*, *module*, *rcu:update:lock}*, *net:register*, *usb:register}*, *sync:race*. Отсутствие нарушений требований для всех загружаемых модулей терминальных устройств из директории *drivers/tty*, которые проверялись совместно с TTY-подсистемой, было доказано для 4 спецификаций:

[15] Подробные результаты анализа доступны по следующей ссылке https://getbox.ispras.ru/index.php/s/HQLsKieKfpZSIGE.

linux:{alloc:spinlock, fs:sysfs, kernel:{locking:rwlock, rcu:update:lock}}. В дальнейших экспериментах данные спецификации больше не участвовали.

В 4 спецификациях требований с идентификаторами *linux:{alloc:{irq, usb lock}, net:register, usb:register}* были обнаружены ошибки, которые не позволяли генерировать верификационные задачи. После исправления данных ошибок повторилась ситуация, как и для предыдущих 4 спецификаций требований.

Только для одной спецификации требований, *linux:kernel:locking:mutex*, удалось обнаружить 9 нарушений. Однако все они оказались ложными сообщениями об ошибках по причине неточности спецификации модели окружения *tty_operations*, а именно, перед/после вызова функции-обработчика для открытия терминального устройства необходимо захватывать/освобождать мьютекс *legacy_mutex*. Аналогично нужно поступать для некоторых других функций-обработчиков *tty_operations*.

Для 7 спецификаций требований на выполнение проверки не хватило отведенного процессорного времени, причем для 3 из них (*linux:{drivers:base:class, kernel:module}, sync:race*) — для всех загружаемых модулей терминальных устройств из директории *drivers/tty*, которые проверялись совместно с TTY-подсистемой.

Для спецификации требований *generic:memory* для успешного построения модели инструментом автоматической статической верификации потребовалось следующее.

- Для всех загружаемых модулей терминальных устройств из директории *drivers/tty*, которые проверялись совместно с TTY-подсистемой, был включен дополнительный заголовочный файл *linux/user_namespace.h*, который содержит определение типа *struct user_namespace*.
- Для загружаемых модулей *drivers/tty/{cyclades.ko, isicom.ko, moxa.ko, rocket.ko}* в состав верификационных задач был добавлен дополнительный файл ядра *kernel/timer.c* с определением типа *struct tvec_base*, используемого при определение таймеров.

После этого при проверке данной спецификации для 37 из 38 загружаемых модулей терминальных устройств из директории *drivers/tty*, которые проверялись совместно с TTY-подсистемой, были обнаружены утечки памяти. Все выданные нарушения оказались ложными сообщениями об ошибках, поскольку, как было отмечено в подразделе 2.3, для подсистем монолитного ядра ОС не требуется освобождать выделенные ресурсы. Отключение проверки соответствующего требования для *generic:memory* привело к тому, что на проверку перестало хватать процессорного времени для всех загружаемых модулей терминальных устройств из директории *drivers/tty*, которые проверялись совместно с TTY-подсистемой.

4.6 Упрощение моделей окружений для проверки выполнения требований для TTY-подсистемы в случаях нехватки процессорного времени

В предыдущем подразделе было показано, что несмотря на достаточно небольшой размер TTY-подсистемы для проверки выполнения спецификаций некоторых требований не хватает процессорного времени, причем для 4 — для всех загружаемых модулей терминальных устройств из директории *drivers/tty*, которые проверялись совместно с TTY-подсистемой. В рамках данного подраздела исследуются причины данной проблемы, а также рассматриваются некоторые возможности ее преодоления.

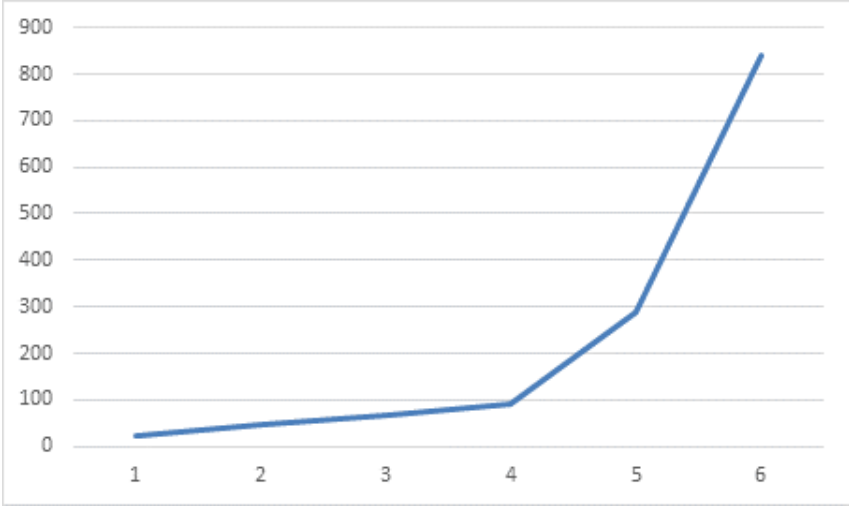

Рис. 1. Зависимость потребления процессорного времени (в секундах) от количества экземпляров программных интерфейсов.
Fig. 1. Usage of CPU time (in seconds) depending on the number of API instances.

Оказалось, что в TTY-подсистеме есть достаточно большое количество экземпляров программных интерфейсов, таких как *tty_operations* и *file_operations*. Для одного из самых требовательных к вычислительным ресурсам требований (отсутствие состояний гонок — *sync:race*) было измерено потребление процессорного времени для различного числа экземпляров программных интерфейсов — рис. 1. Видно, что потребление процессорного времени увеличивается практически экспоненциально, что делает невозможной проверку подсистем ядра ОС Linux и загружаемых модулей с большим количеством экземпляров программных интерфейсов. Для других требований картина может отличаться, однако такое поведение вполне ожидаемо.

Чтобы обойти данную проблему, авторы данного исследования изучили несколько возможных упрощений модели окружения. Во-первых, для

спецификации требований *sync:race*, вместо точной модели окружения, когда для каждого экземпляра программного интерфейса создавался отдельный поток, была использована модель окружения с двумя одинаковыми потоками, которые начинаются непосредственно от начала инициализации TTY-подсистемы и загружаемых модулей ядра ОС Linux. В результате было выдано почти 2,5 тысячи предупреждений, анализ части которых явным образом продемонстрировал, что необходимо использовать точную модель окружения.

Была сделана попытка увеличить максимальный объем вычислительных ресурсов, которые можно использовать на решение каждой верификационной задачи до 150 минут процессорного времени (больше исходного ограничения в 10 раз) и 25 ГБ оперативной памяти (больше исходного ограничения в 2,5 раза). Соответствующий эксперимент для *linux:{drivers:base:class, kernel:{locking:mutex, module}}* позволил обнаружить одно нарушение требования спецификации *linux:drivers:base:class*, которое оказалось ложным сообщением об ошибке (причина и способ его устранения рассмотрены далее). Однако для всех остальных верификационных задач по-прежнему не хватало процессорного времени.

Применительно ко всем спецификациям требований была сделана попытка ограничить количество экземпляров программных интерфейсов. Во-первых, были определены и запрещены те экземпляры программных интерфейсов, для которых регистрация не поддерживалась в спецификациях моделей окружения или была недостижима. Для TTY-подсистемы таких экземпляров оказалось 3, для всех загружаемых модулей терминальных устройств из директории *drivers/tty*, которые проверялись совместно с TTY-подсистемой, — 8. Проверка спецификации требований *generic:memory* после этого показала, что этого упрощения модели окружения не достаточно, поскольку снова не хватило процессорного времени на решение всех верификационных задач.

Радикально сократить время статической верификации помог перебор связанных по регистрации экземпляров программных интерфейсов (то есть рассматривались одновременно только те экземпляры, один из которых регистрируется при инициализации, а остальные регистрируются друг в друге). При этом количество верификационных задач увеличилось с 38 до 55 и получились следующие результаты верификации.

- Для *generic:memory* удалось обнаружить одну ошибку в загружаемом модуле *drivers/tty/serial/mfd.ko* (независимо для 2 верификационных задач). Также было выдано 2 ложных сообщения об ошибках из-за неточности анализа. Отсутствие ошибок удалось доказать для 14 верификационных задач, а для решения оставшихся 37 верификационных задач по-прежнему не хватило процессорного времени. Такого результата следовало ожидать, поскольку для проверки данной спецификации требований используется наиболее тяжеловесный анализ среди всех рассматриваемых спецификаций

требований. Увеличение ограничения на максимальное процессорное время с 15 до 60 минут позволило доказать корректность еще для 5 из 37 верификационных задач.

- Для *sync:race* удалось обнаружить две ошибки в загружаемых модулях *drivers/tty/serial/{st-asc.ko, pch_uart.ko}* (2 верификационные задачи). Для 8 верификационных задач, включая одну из предыдущих, были выданы ложные сообщения об ошибках из-за неточности анализа (7) и модели окружения (1). Отсутствие ошибок удалось доказать для 28 верификационных задач. Решение 4 верификационных задач завершилось неуспешно по разным причинам. Для решения 14 верификационных задач по-прежнему не хватило процессорного времени.
- При проверке спецификации требований *linux:kernel:locking:mutex* было выдано 2 ложных сообщения об ошибках с той же причиной, которая была указана в предыдущем подразделе. Для 46 верификационных задач была доказана корректность, для 7 — не хватило процессорного времени на проверку.
- Для спецификации требований *linux:kernel:module* было выдано 1 ложное сообщение об ошибке из-за неточности модели окружения. Для 49 верификационных задач была доказана корректность, для 4 — не хватило процессорного времени на проверку, для 1 — возникла проблема при обработке трассы ошибки.
- Для *linux:drivers:base:class* было выдано большое количество ложных сообщений об ошибках из-за проблемы, которая указана в подразделе 2.3, поэтому потребовалось запретить проверку финального состояния. После этого для всех верификационных задач было доказано отсутствие нарушений соответствующих требований.
- Для *linux:arch:io* было обнаружено 4 ошибки в загружаемых модулях *drivers/tty/{cyclades.ko, serial/{altera_jtaguart.ko, arc_uart.ko, mfd.ko}}* (для 5 верификационных задач). Для 3 верификационных задач были выданы ложные сообщения об ошибках из-за неточности анализа (2) и модели окружения (1). Для 38 — доказана корректность. Для 9 — не хватило процессорного времени на проверку.
- Для всех загружаемых модулей, при статической верификации которых для *linux:drivers:base:dma-mapping* и *linux:kernel:locking:spinlock* не хватало процессорного времени, перебор связанных по регистрации экземпляров программных интерфейсов помог доказать корректность.

Таким образом, сделанные упрощения моделей окружений позволили существенным образом улучшить результаты верификации, в том числе было обнаружено 7 ошибок в загружаемых модулях терминальных устройств из директории *drivers/tty*, которые проверялись совместно с TTY-подсистемой.

Из-за недостаточной точности анализа инструмента автоматической статической верификации были выданы ложные сообщения об ошибках для 9 верификационных задач, из-за неточности моделей окружения — для 4 верификационных задач.

Обнаружить нарушения проверяемых требований в самой TTY-подсистеме во время проведения экспериментов не удалось. Это демонстрирует высокое качество подсистем ядра ОС Linux. Кроме того, размер исходного кода рассматриваемых загружаемых модулей на порядок превышает размер целевой подсистемы. Ошибки в TTY-подсистеме могут быть найдены, если продолжить улучшать спецификации моделей окружения и требований, а также предлагать новые подходы для сокращения сложности верификационных задач.

5. Заключение

В данной работе был предложен метод инкрементального построения спецификаций моделей окружения и требований для подсистем монолитного ядра ОС, который позволяет использовать методы и инструменты автоматической статической верификации для поиска ошибок в этих подсистемах.

При проведении данного исследования были доработаны компоненты системы статической верификации Klever, а также уточнены существующие и разработаны новые спецификации моделей окружения и требований. Практически все сделанные улучшения в большей или меньшей степени полезны для статической верификации загружаемых модулей ядра ОС Linux, поэтому с течением времени планируется обобщить их и поддерживать для различных вариантов использования.

На различных этапах работы были выявлены и сообщены разработчикам новые проблемы в инструменте автоматической статической верификации CPAchecker. Все эти проблемы были устранены.

Реализация предложенного метода инкрементального построения спецификаций моделей окружения и требований для подсистем монолитного ядра ОС была использована для статической верификации подсистемы поддержки терминальных устройств ядра ОС Linux версии 3.14, которая проверялась совместно с загружаемыми модулями терминальных устройств из директории *drivers/tty*. Благодаря построенным спецификациями удалось повысить покрытие функций целевой подсистемы с 5% до 83% и обнаружить 7 ошибок в рассматриваемых загружаемых модулях. О наличии данных ошибок планируется сообщить разработчикам ядра ОС Linux, если они еще не были исправлены в последних версиях. Ошибки в целевой подсистеме ядра ОС Linux выявлены не были.

Авторы работы планируют продолжить исследование совместно с разработчиками инструментов автоматической статической верификации. В первую очередь необходимо исправить известные проблемы, которые затрудняют или делают невозможным процесс инкрементального построения

спецификаций моделей окружения и требований для подсистем ядра ОС Linux, например:

- поддержать генерацию обращений к программному интерфейсу даже при наличии определения функции, ответственной за его регистрацию;
- поддержать возможность захвата/освобождения блокировок (вызова соответствующих функций) перед/после вызова функций-обработчиков из моделей окружения.

Кроме того, целесообразно расширить область применения предложенного метода и разработанного инструментария на несколько других подсистем ядра ОС Linux.

6. Благодарность

Авторы данной работы выражают благодарность Павлу Андрианову, Антону Васильеву, Владимиру Гратинскому и Алексею Полушкину из команды Linux Driver Verification[16] за участие в обсуждении проблем инкрементальной статической верификации подсистем монолитного ядра операционных систем, за помощь при проведении анализа результатов верификации, а также за поддержку в доработке компонентов системы статической верификации Klever и инструмента автоматической статической верификации CPAchecker.

Список литературы

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

[2]. Е.М. Новиков. Развитие ядра операционной системы Linux. Труды ИСП РАН, том 29, вып. 2, 2017 г., стр. 77-96. DOI: 10.15514/ISPRAS-2017-29(2)-3

[3]. 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, volume 2937, pp. 191-210, 2004.

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

[5]. 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.

[6]. 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, vol. 8974, pp. 400-414, 2014.

[7]. 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.

[8]. П.Н. Девянин, В.В. Кулямин, А.К. Петренко, А.В. Хорошилов, И.В. Щепетков. Сравнение способов декомпозиции спецификаций на Event-B. Программирование, т. 42, № 4, стр. 17-26, 2016.

[16] http://linuxtesting.ru/ldv.

[9]. М.У. Мандрыкин, В.С. Мутилин. 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

[10]. М.У. Мандрыкин, А.В. Хорошилов. О дедуктивной верификации Си программ, работающих с разделяемыми данными. Труды ИСП РАН, том 27, выпуск 4, стр, 49-68, 2015. DOI: 10.15514/ISPRAS-2015-27(4)-4

[11]. 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 p., 2014.

[12]. Д. Бейер, А.К. Петренко. Верификация драйверов операционной системы Linux. Труды ИСП РАН, том 23, стр. 405-412, 2012. DOI: 10.15514/ISPRAS-2012-23-23

[13]. 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, том 41, № 1, стр. 44-67, 2015.

[14]. E. Novikov, I. Zakharov. Towards Automated Static Verification of GNU C Programs. In Proceedings of the 11th International Ershov Informatics Conference (PSI'17), LNCS, volume 10742, 2018 (в печати).

[15]. D. Beyer, M.E. Keremoglu. CPAchecker: A tool for configurable software verification. In: Proceedings of the 23rd International Conference on Computer Aided Verification, Berlin, Heidelberg, Springer, pp. 184-190, 2011.

Incremental development of environment model and requirement specifications for subsystems of operating system monolithic kernels

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

Abstract. Methods and tools for automated static verification aim at detecting all violations of checked requirements in target programs under certain assumptions even without complete models and formal specifications. The given feature form a basis of the suggested method for incremental development of environment model and requirement specifications for subsystems of operating system monolithic kernels. This method was implemented on top of static verification framework Klever. It was evaluated by checking the Linux kernel TTY subsystem. During this study some Klever components were improved. Besides, we fixed some existing and developed new environment model and requirement specifications. Almost all made changes also helps at static verification of loadable modules of the Linux kernel. Developers of automated static verification tool CPAchecker fixed several issues that we revealed and reported during the research. Overall developed specifications allowed to increase function coverage of the TTY subsystem from 5% to 83%. Moreover, we revealed 7 bugs in loadable modules verified together with the TTY subsystem.

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

DOI: 10.15514/ISPRAS-2017-29(6)-2

For citation: Zakharov I.S., Novikov E.M. Incremental development of environment model and requirement specifications for subsystems of operating system monolithic kernels. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 6, 2017, pp. 25-48 (in Russian). DOI: 10.15514/ISPRAS-2017-29(6)-2

References

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

[2]. E.M. Novikov. Evolution of the Linux kernel. Trudy ISP RAN/Proc. ISP RAS, vol. 29, issue 2, pp. 77-96, 2017 (in Russian). DOI: 10.15514/ISPRAS-2017-29(2)-3

[3]. 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, volume 2937, pp. 191-210, 2004.

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

[5]. 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 (in Russian).

[6]. 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, vol. 8974, pp. 400-414, 2014.

[7]. 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.

[8]. P.N. Devyanin, V.V. Kulyamin, A.K. Petrenko, A.V. Khoroshilov, I.V. Shchepetkov. Comparison of Specification Decomposition Methods in Event-B. Programming and Computer Software, vol. 42, issue 4, pp. 17-26, 2016.

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

[10]. M.U. Mandrykin, A.V. Khoroshilov. Towards Deductive Verification of C Programs with Shared Data. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 4, pp. 49-68, 2015 (in Russian). DOI: 10.15514/ISPRAS-2015-27(4)-4

[11]. 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 p., 2014.

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

[13]. 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, vol. 41, issue 1, pp. 49-64, 2015. DOI: 10.1134/S0361768815010065

[14]. E. Novikov, I. Zakharov. Towards Automated Static Verification of GNU C Programs. In Proceedings of the 11th International Ershov Informatics Conference (PSI'17), LNCS, volume 10742, 2018 (to appear).

[15]. D. Beyer, M.E. Keremoglu. CPAchecker: A tool for configurable software verification. In: Proceedings of the 23rd International Conference on Computer Aided Verification, Berlin, Heidelberg, Springer, pp. 184-190, 2011.