НАУЧНО-ТЕХНИЧЕСКИЙ ОТЧЁТ

МЕТОДЫ МОДЕЛИРОВАНИЯ И АНАЛИЗА ИЗОЛЯЦИИ РЕСУРСОВ В МИКРОЯДРЕ ОПЕРАЦИОННОЙ СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ

ИССЛЕДОВАНИЕ ВЫПОЛНЕНО ПРИ ФИНАНСОВОЙ ПОДДЕРЖКЕ РФФИ В РАМКАХ НАУЧНОГО ПРОЕКТА № 16-01-00356

Руководитель

_____ Пакулин Н.В.

СПИСОК ИСПОЛНИТЕЛЕЙ

Руководитель, к.ф.-м.н.	_____	Пакулин Н.В.
вед. научн. сотр., к.ф.-м.н.	_____	Камкин А.С.
вед. научн. сотр., к.ф.-м.н.	_____	Хорошилов А.В.
вед. научн. сотр., д.ф.-м.н.	_____	Бурдонов И.Б.
вед. научн. сотр., д.ф.-м.н.	_____	Косачев А.С.
ст. научн. сотр., к.ф.-м.н.	_____	Мутилин В.С.
мл. научн. сотр.	_____	Смолов С.А.
мл. научн. сотр.	_____	Маллачиев К.А.

1 Введение

Проект посвящен задаче верификации разделения ресурсов в операционных системах для встроенных систем.

Задача разделения ресурсов в таких системах является критически важной для обеспечения функциональной корректности и надежности работы системы.

Был проведён анализ существующих подходов к статической верификации встроенных систем. Было установлено, что анализ кода в области встроенных систем бурно развивается, но на практике используется не так широко, как в случае ПО для рабочих станций. Несмотря на большой опыт, накопленный в области статического анализа, проверки на моделях, дедуктивной верификации и других подходов к оценке корректности исходного кода программ без их исполнения, тестирование остаётся основным инструментом обеспечения качества и надежности встроенных систем. Проблема тестирования таких систем очевидна — далеко не все возможные ситуации могут быть достоверно воспроизведены и сконструированы при тестировании. Некоторые маловероятные ситуации могут никогда не возникнуть при тестировании, но привести к фатальным последствиям, если реализуются на практике.

Сложность анализа встроенных систем вызвана как особенностями кода — наличием ассемблерных вставок, поддержкой многоплатформенности, — так и особенностями функционирования: прежде всего, наличием асинхронности — поддержкой планировщика и системных вызовов, переключением контекстов и т. п.

Существует проект, в котором небольшая операционная система — микроядро гипервизора seL4 — было полностью верифицировано. Но это очень большая и трудоемкая работа: для ядра размеров около 10 тыс. размер доказательства со вспомогательными инструментами составляет более 500 тыс. строк. Проект по доказательству seL4 длится уже более 12 лет.

В нашем проекте мы исследуем возможности анализа и доказательства отдельных свойств без столь всеобъемлющего подхода, как seL4.

Исследование показало, что в операционных системах реального времени, использующихся для управления критическими объектами (такими как самолёты), используются статически сконфигурированные адресные пространства. В обычных операционных системах общего назначения возможно добавление или удаление страниц физической памяти, приписанных к прикладному процессу. По этой причине в ОС общего назначения возможны коллизии доступа к физической памяти из различных адресных пространств.

В ОСРВ для критических систем, таких как VxWorks 653, PikeOS, Lynx OS, используется принципиально иной подход к управлению физической памятью. Физическая память для каждого прикладного или системного раздела задается ещё на этапе конфигурации системы. При выполнении функционального ПО ядро такой операционной системы не использует подкачку страниц, не может аллоцировать дополнительные страницы или

удалить часть страниц, которые использовались прикладным ПО. Такие ограничения накладываются требованиями детерминированного и надежного выполнения:

- в бортовых системах может не быть внешнего хранилища для вытесненных страниц,
- процедура подкачки может занимать непредсказуемый интервал времени и нарушить требования жесткого реального времени для переключения между прикладными задачами,
- произвольная аллокация-деаллокация физической памяти может привести к заполнению памяти и непредсказуемым ситуациям нехватки физической памяти прикладным задачам.

По этим причинам в ОСРВ используется подход априорного статического разделения физической памяти между ядром, системными и прикладными задачами. Разделение страниц памяти задаётся в конфигурации сборки, транслируется в параметры загружаемого образа и используется для настройки MMU при переключении контекстов.

Анализ показал, что в современных встроенных системах задача разделения ресурсов сводится к задаче разделения физической памяти. Та, в свою очередь (за исключением систем общей памяти, таких как RTEMS), сводится к задачам верификации конфигурации виртуальной памяти и верификации подсистем загрузки задач и переключения задач.

Для обеспечения корректного разделения ресурсов в операционной системе реального времени валидация собственно функций управления MMU в ядре ОСРВ играет минимальное значение. Эти функции всего лишь загружают в MMU параметры адресных пространств, подготовленных на этапе конфигурации решения. Разделение страниц физической памяти обусловлено статической конфигурацией, и во время исполнение это разделение между адресными пространствами прикладных задач не изменяется.

Соответственно, для кода ядра достаточно показать, что во время исполнения ядра ОСРВ области памяти, в которых находятся параметры конфигурации адресных пространств, не изменяются. Так как управление MMU существенно зависит от аппаратуры, то верификация такого требования не может быть полностью автоматизирована – необходим этап «ручного» анализа, для установления зависимостей между данными из конфигурации и обращениями к MMU.

2 Обзор методов статической верификации встроенных систем

Методы статической верификации, то есть такой верификации, при которой не исполняется код, в случае встроенных операционных систем можно разделить на три класса:

1. доказательство полной корректности ядра операционной системы;

2. поиск возможных ошибок в ядре ОС или отдельном модуле операционной системы;
3. доказательство корректности отдельного модуля.

2.1 Доказательство полной корректности ядра операционной системы

На текущий момент есть только один пример всестороннего доказательства ядра операционной системы — это микроядро seL4, разработанное в университете Нового Южного Уэльса, Австралия [1],[2]. Микроядро написано на языке высокого уровня Haskell, для которого разработана схема трансляции в язык Си и последующая компиляция в машинный код. Разработчики seL4 сформулировали ряд требований безопасности, надежности и функциональной корректности, которые формализовали посредством языка Isabelle/HOL. Выдающийся результат этого коллектива заключается в том, что они сумели формально доказать, что реализация микроядра на Haskell удовлетворяет предикатам Isabelle/HOL. Более того, они не остановились на этом результате и разработали технику доказательства корректности трансляции ядра seL4 [3] в машинный код различных семейств процессоров ARM и, тем самым, показали, что на процессорах семейства ARM бинарный образ, полученный из исходных кодов Haskell, удовлетворяет требованиям высокого уровня.

Размер формальной спецификации требований, вспомогательных теорий и лемм, а также дополнительных функций на языке Isabelle/HOL составляет более 530 тыс. строк. Работа над проектом заняла более 12 лет.

В контексте нашего проекта необходимо отметить подходы верификации операций с памятью seL4. Исторически первым был разработан метод верификации корректности обращений к памяти в смысле функциональной корректности — никакая операция с памятью не приведёт к возникновению исключения. Модель памяти и арифметики указателей, совместимая с общей моделью seL4 была разработана в 2008 г. в диссертации Harvey Tuch. «Formal memory models for verifying C systems code» [4]. Данная работа выходит за рамки задачи, которую мы ставим в этом проекте, однако приведена в обзоре, так как она послужила базой для другой диссертации, которая решает задачу, аналогичную поставленной в данном проекте.

Речь идёт о диссертации 2011 г. Rafal Kolanski «Verification of programs in virtual memory using separation logic»[5]. В этой работе был предложен подход к описанию разделения доступа к виртуальной памяти, предложены модели различных механизмов виртуальной памяти и разработана теория на языке Isabelle/HOL, совместимая с моделью seL4. Для доказательства корректности разделения памяти используется подход логики разделов (separation logic), в которой добавлены специальные термы, определяющие различные домены, и сформулированы правила вывода, позволяющие доказать разделение привилегий между задачами. В настоящее время общая теория разделения привилегий (не только оперативной памяти) в доказательстве составляет более 10 тыс. строк.

Подход к описанию разделения ресурсов идеологически близок к тому, что мы предлагаем в данной работе, однако без полной модели микроядра нереализуем.

2.2 Поиск возможных ошибок в ядре ОС или отдельном модуле операционной системы

Это направление анализа встроенных систем бурно развивается в последнее время. Прежде всего, необходимо отметить попытки применения инструментов статического анализа общего назначения для выявления типовых ошибок в модулях операционной системы. Прежде всего, это различные попытки использовать такие инструменты статического анализа, как PC- Lint[5], Cppcheck[6], Saturn[7], Coverity[8], Klockwork[9].

Код, работающий в ядре операционной системы, обладает рядом свойств, которые препятствуют успешному применению обычных статических анализаторов: реактивность (обработка внешних событий), наличие ассемблерных вставок, многоплатформенность, взаимодействие с аппаратурой. По этой причине в настоящее время активно развиваются инструменты, объединяющие статический анализ и решатели для логических ограничений.

Так, подход, основанный на предикатной абстракции [10] и символьной проверке на моделях[11] булевых программ является основой, на которой построен верификатор драйверов компании Microsoft[12]. Статический анализатор SLAM[13,14] объединяет различные методы анализа кода и выявляет большой класс ошибок с высокой точностью. В контексте нашего проекта у SLAM и Windows Driver Verifier есть два принципиальных ограничения — эти инструменты работают только для ядра ОС Windows, и класс ограничений, необходимый для решения задачи верификации разделения ресурсов, в них затруднительно выразить. При этом подход компании Microsoft позволил поднять на новый уровень качество драйверов для ОС Windows и служит, пожалуй, одним из немногих примеров индустриальных инструментов анализа кода ядра операционной системы.

Институт системного программирования РАН ведёт аналогичный проект по верификации драйверов ОС Linux: Linux Driver Verification [15,16] Верификатор использует метод «уточнения абстракций по контр-примерам» (Counter Example Guided Abstraction Refinementб CEGAR) [17] и технологически построен на инструментах BLAST [18] и CPAChecker[19]. Это открытый проект. Некоторые из его участников входят в состав коллектива данного проекта. Поэтому мы рассчитываем переиспользовать опыт, полученный при верификации драйверов Linux, для исследования корректности модуля управления виртуальной памятью в операционной системе реального времени.

2.2.1 Доказательство корректности отдельного модуля

Методы, представленные в предыдущем разделе, направлены прежде всего на выявление ошибок некоторого класса. Для этого код модуля операционной системы трансформируется, в него добавляются специальные инструментальные конструкции, и

затем, например, решается задача достижимости определённых точек в инструментированном коде, таких как выход из функции без разблокирования мьютекса.

Задача доказательства корректности отдельного модуля близка, но преследует иную цель: строится доказательство (на самом деле, ищется контрпример), что некоторое нежелательное состояние никогда не наступит. Например, доказательство отсутствия взаимных блокировок, или бесконечных циклов, или доказательство иных требований надежности. Эти требования формулируются на том или ином языке темпоральных спецификаций, и затем инструмент автоматически строит абстрактную модель модуля и средствами проверки на моделях показывает, что код модуля удовлетворяет условию. Широкую известность для решения этой задачи снискал статический анализатор BLAST [18]. Однако, применению этого метода в повседневной практике препятствуют два обстоятельства:

- проверка на моделях имеет высокую вычислительную сложность, поэтому для модулей более 5 тыс. строк решение задачи поиска контрпримера может не завершиться за разумное время или в разумные ресурсы;
- требуется сформулировать требование в специальной (довольно сложной) нотации.

Тем не менее, мы планируем воспользоваться указанной техникой в нашем проекте. Один из членов команды проекта является ответственным за поддержку BLAST [20].

3 Уточнение постановки задачи

В ходе анализа литературы и исследования примеров встроенных операционных систем с поддержкой реального времени мы обнаружили, что можно выделить четыре категории управления доступом к памяти, для которых задача определения корректности разделения памяти ставится существенно разными способами. Ниже мы рассмотрим, как выбор механизмов управления памятью влияет на выбор средств верификации.

1. Общая память. Все прикладные задачи, а также ядро ОС, выполняются в общем адресном пространстве. Примером такой ОС может служить открытая ОСРВ RTEMS. Общая память и выполнение прикладных задач на уровне привилегий ядра позволяет минимизировать накладные расходы на переключение контекстов, и используется прежде всего в космической технике, на очень медленных радиационно-стойких процессорах.
2. Виртуальная память. Адреса, которые использует приложение (и ОС) не являются адресами физической памяти, и преобразуются процессором с использованием специальных таблиц (сегменты, таблицы страниц, TLB). Принципиальное в контексте проекта отличие от предыдущего случая заключается в том, что таблицы трансляции адресов настраиваются ядром ОС, и в очень ограниченном числе мест — как правило, только при создании и переключении задач. Режимы доступа к областям памяти обеспечиваются аппаратурой виртуальной памяти (memory management unit, MMU). Ниже кратко рассмотрены три основных механизма организации виртуальной памяти. Как правило, выбор конкретного механизма не

является произвольным, и зависит от возможностей, которые предоставляет аппаратура.

1. Виртуальная память с поддержкой сегментов. У каждой задачи есть фиксированный набор дескрипторов сегментов, задающих базовый адрес сегмента в виртуальной и физической памяти, размер сегмента и, возможно, допустимые режимы доступа к соответствующей области памяти. Сегментная модель виртуальной памяти реализована в процессорах семейства x86 и x86_64, а также в процессорах PowerPC для встроенных применений компании IBM. Важная особенность поддержки сегментной организации памяти в x86 заключается в том, что переключение сегментов допускается на уровне привилегий пользователя. В PowerPC соответствующие регистры доступны только в режиме ядра.
2. Виртуальная память с поддержкой страничной адресации. Страничная адресация представляет собой существенный шаг вперёд по сравнению с сегментной адресацией. Страничная адресация позволяет разбить виртуальную память на произвольное число блоков (страниц), в отличие от фиксированного набора сегментов. Для каждой страницы можно задать режимы доступа и смещения в виртуальном и физическом адресном пространствах, а в некоторых архитектурах процессоров можно указать и размер страниц. Страничный режим адресации поддерживается в процессорах x86, ARM, PowerPC 7xx (IBM). Переключение между таблицами страниц возможно только в режиме ядра. Таблицы страниц хранятся в оперативной памяти, доступной ядру, и для ускорения трансляции адресов кэшируются внутри процессора в translation lookaside buffer, TLB.
3. Виртуальная память с поддержкой прямого доступа в TLB. В некоторых процессорах для встроенных применений (MIPS, PowerPC Book E) предоставляется прямой доступ к TLB. Это позволяет разработчикам ОСРВ полностью контролировать процесс переключения контекстов задач и детерминированно оценивать связанные с этим процессом латентности. Каждая запись в TLB определяет отображение из некоторой непрерывной области виртуальной памяти в непрерывную область физической памяти и режимы доступа к этой области.

Исходная задача верификации разделения ресурсов между задачами, которые выполняются в ОСРВ, должна ставиться индивидуально для каждого способа организации доступа к памяти в силу существенной разницы в том, как обеспечивается поддержка разделения ресурсов аппаратурой процессора.

Рассмотрим насколько влияет многоядерность на постановку задачи верификации. Различают несколько видов многоядерности:

- SMP — симметричная мультипроцессорность, в рамках которой на всех процессорах выполняется одна копия операционной системы, которая управляет всеми ресурсами.

- AMP — асимметричная мультипроцессорность, в рамках которой на разных процессорах выполняются независимые копии операционных систем. Каждая копия ОС называется AMP-модулем.
- Смешанная мультипроцессорность: на непересекающихся наборах процессоров выполняются SMP системы, каждая из которых является отдельным AMP-модулем.

ОС ответственного применения имеют ограничения на реализацию SMP. Рассмотрим ограничения, который декларирует промышленный стандарт для гражданских бортов ARINC-653. В рамках указанного стандарта вводится понятие раздела (partition). Разделом называется приложение, имеющее выделенное адресное пространство. Адресные пространства разделов различны. ОС переключает разделы согласно расписанию. Таким образом достигается разделение разделов во времени и в пространстве. Согласно ARINC-653 в SMP системе единовременно может выполняться только один раздел.

Не смотря на жесткое разделение адресных пространств между разделами, разделам разрешено иметь выделенные области общей памяти. Для этого в конфигурации разделам явно приписываются данные области памяти.

В ОС, построенных на AMP и смешанной мультипроцессорностях, можно выделить подсистему, отвечающую за загрузку AMP-модулей. Будем называть данную подсистему загрузчиком (loader).

Во встраиваемых ОС со статической конфигурацией конфигурация памяти полностью создается на инструментальной машине. Это позволяется значительно упростить как разработку кода, выполняющегося на целевом вычислителе, так и его анализ. Статичность конфигурации позволяет проводить ее анализ на инструментальной машине.

После доказательства корректности конфигурации памяти остается только доказать, что код выполняемый на целевом вычислителе работает в соответствии с заданной конфигурацией.

С учетом вышеперечисленного задачу разделения ресурсов можно разбить по подзадачи:

1. Верификация конфигурации памяти: необходимо удостовериться в том, что:
    1) в рамках одного AMP-модуля разделы не имеют общей физической памяти кроме явно специфицированной;
    2) В рамках одного AMP-модуля каждый раздел и ядро не имеют общих как виртуальных адресов, так и физических адресов;
    3) в рамках нескольких AMP-модулей ядра и разделы не имеют общих физических адресов, кроме явно специфицированных.
2. Для верификации загрузчика необходимо показать, что:
    1) загрузчик загружает память согласно статической конфигурации памяти;
    2) загрузчик после запуска AMP-модуля больше не обращается к областям памяти, относящимся к данному модулю.

3. Для верификации ядра ОС необходимо показать, что:

    1) при переключении разделов ядро ОС перенастраивает память согласно статической конфигурации памяти;

    2) в любой момент времени ядро может обращаться к памяти только того раздела, который выполняется в данный момент согласно расписанию.

Помимо методов статического анализа для верификации разделения ресурсов могут использоваться методы динамического анализа. Методы динамического анализа рассмотрены в разделе 3.3

3.1 Верификация конфигурации памяти

Сформулируем требования к разделения памяти в формальном виде.

Множество $\mathrm{A} \subset \mathbb{N}_0$ определяет множество всех адресов. При отображении виртуальных адресов на физические некоторому множеству из *P(A)* ставится в соответствие множество из *P(A)* такой же мощности, где *P(A)* — множество всех подмножеств множества *A*.

Отображение виртуальных адресов на физический будем задавать в виде четверки $(ampid, taskid, vaddrs, paddrs) \in Q,\ \ Q \equiv \mathbb{N}_0 \times \mathbb{N}_0 \times P(A) \times P(A)$, мощности множеств *vaddrs* и *paddrs* должны совпадать. Такая четверка определяет одно из множеств виртуальных адресов раздела *taskid* в AMP-модуле *ampid* и соответствующее ему множество физических адресов. Обозначим *M* множество всех таких четверок, описывающих отображение виртуальных адресов на физические в конфигурации памяти.

Введем функции для получения элементов описанной четверке:

get_ampid(ampid,taskid,vaddrs,paddrs) = ampid
get_taskid(ampid,taskid,vaddrs,paddrs) = taskid
get_vaddrs(ampid,taskid,vaddrs,paddrs) = vaddrs
get_paddrs(ampid,taskid,vaddrs,paddrs) = paddrs

Так как на один и тот же физический адрес могут быть отображены несколько вирутальных адресов, то введем предикат описывающий могут ли заданные разделы разделять заданное множество физических: *SHARED(ampid_1, taskid_1, ampid_2, taskid_2, paddr)* принимает значение *ИСТИНА*, если к адресу *paddr* могут получать доступ как раздел *taskid_1* в AMP-модуле *ampid_1*, так и раздел *taskid_2* в AMP-модуле *ampid_2*.

Теперь условия корректности конфигурации памяти можно определить следующим образом:

1. $$\forall m_1, m_2 \in M, addr \in A : get\_ampid(m_1) = get\_ampid(m_2) \wedge get\_taskid(m_1) \neq get\_taskid(m_2) \wedge (addr \in get\_paddrs(m_1) \cap get\_paddrs(m_2)) \rightarrow SHARED(get\_ampid(m_1), get\_taskid(m_1), get\_ampid(m_2), get\_taskid(m_2), addr))$$

2. $$\forall m_1, m_2 \in M : get\_ampid(m_1) = get\_ampid(m_2) \wedge get\_taskid(m_1) = 0 \wedge get\_taskid(m_2) \neq 0 \rightarrow ((get\_paddrs(m_1) \cap get\_paddrs(m_2) = \varnothing) \wedge (get\_vaddrs(m_1) \cap get\_vaddrs(m_2) = \varnothing))$$

3. $$\forall m_1, m_2 \in M, addr \in A : get\_ampid(m_1) \neq get\_ampid(m_2) \wedge (addr \in get\_paddrs(m_1) \cap get\_paddrs(m_2)) \rightarrow SHARED(get\_ampid(m_1), get\_taskid(m_1), get\_ampid(m_2), get\_taskid(m_2), addr)$$

3.1.1 Разработанный инструмент верификации конфигурации памяти

В рамках проекта были сформулированы требования корректности конфигурации адресных пространств.

Была разработана нотация на базе YAML для спецификации конфигурации адресных пространств и были сформулированы требования корректности адресных пространств:

- области физической памяти, не помеченные как «разделяемые», не пересекаются;
- области физической памяти, не помеченные как «разделяемые», не пересекаются с областями физической памяти, помеченными как «разделяемые»;
- дополнительные аппаратно-зависимые требования (например, ограничения на режимы доступа для прикладных задач и для ядра) для отдельных областей памяти;
- дополнительные аппаратно-зависимые свойства для доступа к страницам физической памяти, на которую отображены конфигурационные пространства устройств (например, режимы кэширования).

Указанный результат является новым, в открытых источниках нет аналогичных работ по определению ограничений (constrains) на конфигурацию памяти для ОСРВ (RTOS memory map). Тем не менее, очевидно, что во всех коммерческих ОСРВ существуют инструменты валидации карты памяти той или иной степени детальности.

Также в рамках проекта был разработан прототип программного средства на языке Python, который для заданной конфигурации физической памяти проверяет требования корректности адресных пространств.

3.2 Метод верификации корректности работы с памятью в коде ОСРВ

Помимо доказательства корректности конфигурации памяти необходимо показать, что код выполняемый на целевом вычислителе, работает в точном соответствии с заданной конфигурацией памяти.

В процессе работы ОС проходит разные стадии, которые важно отличать, так как в разных стадиях существуют разные требования к корректности ОСРВ.

С точки зрения работы с памятью можно выделить следующие этапы выполнения загрузчика:

1. Начальная инициализация загрузчика и инициализация областей физической памяти всех AMP-модулей (запись данных в области физической памяти).
2. Последовательный старт всех AMP-модулей.

3. На каждом AMP-модуле происходит:
    1. Настройка MMU (memory management unit) в соответствии со статической конфигурацией памяти.
    2. Запуск ядра ОС.

Во время работы ядро ОС (в рамках одного AMP-модуля) может находится в одном из двух режимов, постоянно переключаясь между ними:

1. Выполнение одного раздела (текущего по расписанию).
2. Переключение с одного раздела на другой. Во время переключения разделов происходит перенастройка MMU на новый раздел. Переключение разделов происходит неатомарно, поэтому необходимо явно выделять данную стадию.

Будем считать, что код ОСРВ (как ядра, так и загрузчика) является корректным, если:

1. Загрузчик настраивает MMU (memory management unit) в соответствии со статической конфигурацией памяти для каждого AMP-модуля.
2. После окончания стадии начальной инициализации областей физической памяти загрузчик обращается либо к памяти загрузчика, либо к памяти текущего AMP-модуля.
3. В зависимости от режима ядра ОСРВ
    а) Если ядро ОСРВ находится в режиме выполнения одного раздела, то ядро ОСРВ может обращаться либо к собственной памяти, либо к памяти текущего раздела. То есть ядру запрещено обращаться к памяти других разделов.
    б) Если ядро ОСРВ находится в режиме переключения «старого» раздела на «новый», то ядро должно:
        (1) перенастраивать MMU в соответствии со статической конфигурацией памяти «нового» раздела.
        (2) Не обращается к памяти ни одного раздела.

В рамках проекта был разработан метод позволяющий проводить анализ выполнения перечисленных условий корректности кода. Предлагаемый метод позволяет анализировать объектный код ОСРВ при условии, что объектный код сопровождается отладочной информацией.

Метод основывается на моделировании аппаратуры. Для моделирования используется программа на языке Си. Программа хранит модель, состоянием модели является совокупность состояния процессора (всех его ядер), состояния памяти и режимов работы экземпляров ОСРВ в разных AMP-модулях.

Объектный код транслируется в данную программу. Каждой машинной инструкции процессора ставится в соответствие функция на языке Си, которая модифицирует состояния процессора и памяти в соответствии со спецификацией на процессор.

Инструкциям можно задавать пред- и постусловия — условия, которые проверяются соответственно в моменты начала и окончания функции, которая соответствует заданной инструкции. Часть предусловий на инструкции напрямую следуют из спецификации на процессор, другие можно задавать дополнительно.

Для инструкций, взаимодействующих (чтение, запись или исполнение) с памятью, вводится дополнительное предусловие — результат предиката is_valid_addr(state, config, addr). Данный предикат определяет допустимость обращения к памяти по адресу addr в рамках модели, находящейся в состоянии state, для конфигурации памяти config.

Иногда машинные инструкции несут дополнительную семантику, например, при переключения режима работы ОС. Данную семантику тяжело извлечь из объектного кода, поэтому предлагается использовать отладочную информацию, которая позволяет локализовать начало и конец важных для анализа участков кода. Будем называть данные участки *мета-функциями*.

С мета-функцией связанна функции языке Си, модифицирующая состояние модели в соответствии с семантикой данной функции. Например, мета-функция, отвечающая за окончание режима переключения разделов, модифицирует состояние модели, меняя режим ОСРВ на режим выполнения одного раздела. У мета-функций также могут быть заданны пред- и постусловия.

Для мета-функции, которая настраивает MMU, вводится дополнительное постусловии — предикат is_correct_map(state, config). Данный предикат проверяет, что в состоянии state настройки MMU соответствуют конфигурации config. Это означает, в том числе, что в MMU есть все отображение памяти, которые описаны в конфигурации для текущего AMP-модуля и текущего раздела, и только они.

Кроме того, модель позволяет вводить инварианты — предикаты на состояние модели, которые должны выполняться в каждый момент времени. Примером такого инварианта является проверка того, что режим ОСРВ является одним из описанных выше.

В рамках описанной модели сформулируем требования к корректности кода ОСРВ, которые были описаны выше в виде инвариантов, пред- и постусловий на примере архитектуры PowerPC:

- Условия 1 и 3.б.1 сводятся к определению предиката is_correct_map, который является постусловием мета-функции, настраивающей MMU. Данный предикат должен найти в конфигурации части, соответствующие текущему состоянию, и проверить, что MMU настроен в соответствии с ними.
  Конфигурация, с которой работает is_correct_map, строится на основе множества M и предиката SHARED из раздела 3.1
- Условия 2, 3.а и 3.б.2 сводятся к определению предиката is_valid_addr, который является предусловием инструкций, работающих с памятью. Данный предикат должен проверять, что конфигурация допускает обращение к заданному адресу в текущем состоянии.

3.3 Динамический анализ виртуальной памяти

Статический анализ позволяет доказать корректность программы на всех возможных путях её выполнения, но он требует достаточно большого времени и трудозатрат. Динамический анализ может охватить только небольшое количество путей выполнения, но быстрее и с меньшими трудозатратами. Комбинирование статического и динамического анализа позволяет повысить эффективность процесса верификации в целом. За счёт предварительного проведения динамического анализа можно быстрее выявить проблемы и приступить к их устранению. Поэтому дополнительным направлением исследования являлся вопрос о возможности эффективного обнаружения проблем с управлением виртуальной памятью при помощи средств динамического анализа.

В целях реализации данной задачи привлекался набор санитайзеров из стека динамического инструментирования LLVM: AddressSanitizer, MemorySanitizer, UndefinedBehaviorSanitizer. Указанные средства на стадии трансляции автоматически привносят в объектный код различные проверки, которые при соответствующей реализации среды исполнения могут уведомлять тестировщика об ошибках. В числе обнаруживаемых ошибок:

- Обращение к невыделенному адресу, например, за пределами выделенного массива. Обеспечивается посредством отображения основной памяти на статусную с проверкой при каждом обращении.
- Обращение к переменной за пределами её области видимости, например, в случае передачи адреса переменной на стеке в вышестоящую функцию. Реализуется посредством генерации статусного стека в процессе работы функции.
- Обращение к неинициализированной переменной с последующим использованием, например, в качестве условия в конструкции ветвления. Также обеспечивается посредством отображения основной памяти на статусную, но статусом в данном случае является наличие записи в данную область.

В связи с особой специфичностью среды исполнения с жёстким реальным временем, в результате исследования инструменты были портированы на перспективную бортовую ОСРВ JetOS. Возможные затруднения реализации подобного инструментирования в ОС такого класса, способы их обхода, а также средства максимизации эффективности данных инструментов описаны в статье «Dynamic Analysis of ARINC 653 RTOS with LLVM».

4 Список публикаций по проекту

К.М. Маллачиев, Н.В. Пакулин, А.В. Хорошилов Устройство и архитектура операционной системы реального времени Труды Института системного программирования РАН Том 28, выпуск 2 2016 г. Стр. 181-192.

А.Н. Емеленко, К.А. Маллачиев, Н.В. Пакулин Разработка отладчика для операционной системы реального времени Труды Института системного программирования РАН Том 28, выпуск 2 2016 г. Стр. 193-204.

Маллачиев К.А., Пакулин Н.В., Хорошилов А.В., Буздалов Д.В. Использование модульного подхода во встраиваемых операционных системах. Труды ИСП РАН, том 29, вып. 4, 2017 г., стр. 283-294 (на английском). DOI: 10.15514/ISPRAS2017-29(4)-19

Маллачиев К.А., Хорошилов А.В Построение модульного программного обеспечения на основе однородной компонентой модели. Труды ИСП РАН, том 30, вып. 3, 2018 г., стр. 135-148 (на английском языке). DOI: 10.15514/ISPRAS2018-30(3)-10

Vitaly Cheptsov, Alexey Khoroshilov "Dynamic Analysis of ARINC-653 RTOS with LLVM" Proceedings of Ivannikov ISPRAS Open Conference, 22-23 November, 2018, Moscow, Russia, (в процессе публикации в IEEE Xplore)

5 Ссылки

[1] D. Potts, R. Bourquin, L. Andresen, J. Andronchik, G. Klein, G. Heiser. Mathematically Verified Software Kernels: Raising the Bar for High Assurance Implementations. © 2014 General Dynamics. https://sel4.systems/Info/Docs/GD-NICTA-whitepaper.pdf

[2] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski and Gernot Heiser. Comprehensive formal verification of an OS microkernel. ACM Transactions on Computer Systems, Volume 32, Number 1, pp. 2:1-2:70, February, 2014

[3] Thomas Sewell, Magnus Myreen and Gerwin Klein. Translation validation for a verified OS kernel. ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 471-481, Seattle, Washington, USA, June, 2013

[4] Harvey Tuch. Formal memory models for verifying C systems code. PhD Thesis, UNSW, Sydney, Australia, August, 2008

[5] Rafal Kolanski. Verification of programs in virtual memory using separation logic. PhD Thesis, UNSW, Sydney, Australia, July, 2011

[6] Stephen C. Johnson. Lint, a C program checker. In Comp. Sci. Tech. Rep. Murray Hill, 1978.

[7] Cppcheck — инструмент статического анализа для C/C++. http://cppcheck.sourceforge.net/. Ссылка проверена 2016-12-19.

[8] Coverity — инструмент статического анализа. https://www.synopsys.com/software-integrity/products/static-code-analysis.html. Ссылка проверена 2016-12-19.

[9] Klocwork — инструмент статического анализа. http://www.klocwork.com/products-services/klocwork

[10] Graf, S., Sa ̈idi, H.: Construction of abstract state graphs with PVS. In: Computer-Aided Verification (CAV). pp. 72–83. LNCS, Springer (1997)

[11] Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, L.J.: Symbolic model checking: 10 20 states and beyond. Information and Computation (1992)

[12] Ball, T., Bounimova, E., Levin, V., Kumar, R., Lichtenberg, J.: The static driver verifier research platform. Proceedings of CAV 2010.

[13] Ball, T., Rajamani, S.: The SLAM project: debugging system software via static analysis. In: Principles of Programming Languages (POPL). pp. 1–3. ACM (2002)

[14] Ball, T., Rajamani, S.K.: The SLAM toolkit. In: CAV. LNCS, vol. 2102, pp. 260–264. Springer (2001)

[15] 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. Труды ИСП РАН, том 26-2, 2014. С. 5-42.

[16] В.С. Мутилин, Е.М. Новиков, А.В. Хорошилов. Анализ типовых ошибок в драйверах операционной системы Linux. Труды ИСП РАН, том 22, 2012. С. 349-372.

[17] М.У. Мандрыкин, В.С. Мутилин, А.В. Хорошилов. Введение в метод CEGAR — уточнение абстракции по контрпримерам. Труды ИСП РАН, том 24, 2013. С. 219-292.

[18] Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker BLAST. STTT 9(5-6), 505–525 (2007)

[19] Beyer, D., Keremoglu, M.E.: CPACHECKER: A tool for configurable software verification. Proc. Computer Aided Verification (CAV). pp. 184–190. LNCS 6806, Springer (2011)

[20] В. Мутилин. Predicate Analysis with BLAST 2.7. Доклад на 2-м международном соревновании по верификации программного обеспечения (SV-COMP-2013), Рим, Италия. 14 марта 2013 г.