

Санкт-Петербургский государственный университет

Кафедра информатики

Группа 19.Б09-мм

Вильданов Эмир Флоридович

Исследование требований к языку
системного программирования высокого
уровня на примере анализа кода
микроядерной ОС seL4 с проверяемой
корректностью кода.

Отчёт по учебной практике

Научный руководитель:
ст. преп. кафедры информатики, к. ф.-м. н., Салищев С. И.

Санкт-Петербург
2022

Оглавление

1. Введение	3
2. Постановка задачи	5
3. Обзор	8
3.1. Языки программирования	8
3.2. Системы статической проверки кода	12
4. Исследование	14
4.1. Архитектура микроядра	14
4.2. Система верификации	16
4.3. Выявление требований	17
5. Альтернативный инструмент	22
5.1. Обзор	22
5.2. Устройство доказательств	23
6. Заключение	26
Список литературы	28

1. Введение

Разработка большого программного продукта влечёт за собой возникновение большого количества ошибок в коде. Десятилетиями программисты борются с так называемыми "багами" и стараются сделать процесс разработки продукта более надежным. Примерами подобных ошибок являются: утечки памяти, запись данных в нераспределенные участки памяти, обращение к неинициализированной памяти [1], переполнение буфера и разыменование недопустимого указателя. Помимо вопроса удобства, с каждым днём становится всё более важным вопрос безопасности кода. Каждую неделю специалисты информационной безопасности обнаруживают новые уязвимости в коде продуктов, которыми пользуются миллиарды людей [2]. Иногда уязвимости касаются вопроса утечки личных данных или банковских реквизитов, иногда мошенники требуют у компаний выкуп за предоставление доступа к данным, чем подрывают экономику предприятий и стран. Реже дело доходит до угрозы жизни человека, когда злоумышленники берут под контроль устройства передвижения, водо- или газоснабжения, целые электростанции и промышленные предприятия [3].

Программы, которые запускаются на машинах с большой оперативной памятью, зачастую пишутся на языках программирования со сборщиком мусора, который следит за автоматическим выделением и освобождением памяти. Также некоторые языки программирования имеют проверку типов во время исполнения, которая может прервать работу программы, в случае ошибки типов. Проверка типов во время исполнения приводит к замедлению исполнения программы.

Сборка мусора и динамическая проверка типов имеют ограниченное применение для систем реального времени (в частности для операционных систем) и для устройств с ограниченным запасом памяти (например, в случае программирования микропроцессоров). В их случае предпочтительно в принципе не допустить возможности запуска программы с уязвимостью.

Существуют специализированные инструменты по проверки кода,

такие как Valgrind [4], которые проверяют код на корректность путём запуска его в виртуальной среде.

Некоторые современные языки программирования выполняют проверку семантики кода на этапе компиляции более сложные чем обычная проверка типов.

Существуют также системы, статически проверяющие выполнение спецификаций, заданных разработчиком, на коде программы методами математической логики. Для проверки используется система автоматического доказательства теорем, спецификации записываются на языке этой системы. Одна из таких систем, инструмент по проверке корректности кода микроядра seL4 [5], и будет являться главным объектом изучения данной работы.

2. Постановка задачи

Как было сказано ранее, главным объектом изучения данной работы является система проверки безопасности кода микроядра seL4.

Задача проверки произвольного кода на удовлетворение требованиям безопасности обречена на провал. Доказательство корректности программы состоит в доказательстве произвольных утверждений в некоторой логике относительно кода программы для произвольных входных данных. Доказательство утверждений по соответствию Карри-Ховарда связано с выводом типов в языке [6]. Вывод типов в свою очередь связан с запуском программы в денотационной семантике [7], определяемой системой типов, частным случаем которой является операционная семантика исполнения программы, в которой каждое значение имеет уникальный тип. Таким образом, в общем случае проверка корректности программ обладает одним из трех недостатков: алгоритмическая неразрешимость, неполнота проверки или ложные срабатывания.

Если мы ограничимся только системами реального времени, то для них вопрос «Завершится ли данный процесс за конечное время?» в принципе не стоит, потому что это одно из требований, предъявляемых к системе реального времени и не имеющее отношения к безопасности языка. Этот факт уже даёт нам надежду на то, что мы сможем вывести некоторый достаточно полный набор эффективно проверяемых требований безопасности для языка.

Когда мы перешли барьер алгоритмической разрешимости, появляются проблемы практического характера, связанные с балансом сложности применения языка, надежности и вычислительной трудности проверки:

- Пусть существует набор правил, который проверяет исходный код на корректность. Что делать в случае, когда система проверки выдает ложноположительные ошибки на заведомо верных, безопасных программах? Логiku рабочей программы придётся переписывать, придерживаясь строгого набора требований системы проверки. Это неудобно, долго и иногда кажется бессмысленным;

- Пусть система проверки работает безотказно, но сколько времени занимает этот процесс? Если это значение велико настолько, что ожидание проверки обесценивает проделанную работу, то такая система проверки бесполезна;
- Что если система не имеет недостатков, перечисленных выше, но код, удовлетворяющий её требованиям, настолько сложный, что его невозможно читать? Получается, что такой подход существенно обесценивается, поскольку код невозможно поддерживать без специально обученных разработчиков.

Исходя из перечисленных выше доводов, была сформулирована цель данной работы: на примере системы проверки микроядра seL4 изучить требования к системному языку программирования высокого уровня, удовлетворяющего требованиям полноты возможных к реализации алгоритмов, разумного времени проверки и эстетичности кода. Для достижения поставленной цели были поставлены следующие задачи:

1. Ознакомиться с существующими аналогами систем, проверяющих код на безопасность. Выявить их возможные недостатки с точки зрения перечисленных выше факторов;
2. Изучить архитектуру и исходный код микроядра seL4;
3. Изучить синтаксис функционального языка программирования Isabelle, предназначенного для написания доказательств теорем;
4. Изучить документацию по верификации кода в репозитории seL4. Ознакомиться с абстрактным представлением микроядра в контексте математической модели;
5. Исследовать блок верификации seL4:
 - (a) Выявить требования, налагаемые на код микроядра
 - (b) Проанализировать свободу написания кода, удовлетворяющего выявленным требованиям

- (с) Проанализировать время выполнения проверок
- (d) Проанализировать эстетический аспект кода, удовлетворяющего выявленным требованиям

3. Обзор

При поверхностном изучении кода доказательства безопасности seL4 стало понятно, что система проверки микроядра — это очень сложная система с большим количеством зависимостей и логической базой. Поэтому для ознакомления с принципом «memory-safety check» было принято решение изучить иные системы статической проверки, которые также проверяют код на безопасность, но содержат более понятную структуру проверки и более широко раскрытую документацию.

3.1. Языки программирования

В первую очередь были рассмотрены языки программирования, поскольку многие из них сами по себе предоставляют очень хорошую проверку на безопасность. Наверное, каждый программист сталкивался со следующими ошибками в процессе написания кода:

- Выход за пределы массива
- Использование не выделенной памяти
- Использование неинициализированной переменной

Чаще всего поддержка безопасности по перечисленным выше и некоторым другим пунктам осуществляется уже в процессе работы программы. Ниже будут рассмотрены особенности языков, которые осуществляют проверки именно на этапе компиляции.

3.1.1. Rust

В поисках языка программирования со статической проверкой кода на безопасность все пути ведут к языку Rust. У него одна из самых подробных документаций с понятными примерами среди всех языков программирования и большое сообщество программистов.

По заявлениям авторов языка, система безопасности Rust обеспечивает следующие требования безопасности программ:

1. Отсутствие состояния гонки;
2. Невозможность ссылки на освобождённый участок памяти;
3. Невозможность разыменования нулевого указателя;
4. Отсутствие неинициализированных переменных;
5. Невозможность утечки памяти;
6. Невозможность повторного освобождения памяти;

Безопасность кода обеспечивается путём соблюдения следующих основных принципов:

1. RAII principle: Получение некоторого ресурса неразрывно совмещено с инициализацией, а освобождение — с уничтожением объекта;
2. Generic, Traits, Type inference: Система типов, проверка совместимости интерфейсов и автоматическое определение типов позволяют предотвратить ошибки несовместимости типов;
3. Ownership: С каждым участком памяти программы связан её *владелец*:
 - В один момент времени у одного участка памяти может существовать только один *владелец*
 - Когда *владелец* меняется или пропадает (например, это может быть связано с присвоением участка памяти другой переменной или при переходе из одной области видимости в другую), участок памяти становится недоступным

Понятие владения связано с аффинной системой типов языка;

4. References и Borrowing: Существуют изменяемые и неизменяемые ссылки (как и изменяемые и неизменяемые переменные).

Переменные-ссылки на динамически-выделенную память закрывают доступ к памяти в случае копирования ссылки или при переходе в другую область видимости программы. Вот четыре правила обращения с ссылками в языке Rust:

- Может существовать любое количество неизменяемых ссылок на один участок памяти
- Может существовать только одна изменяемая ссылка на участок памяти
- Не может существовать одновременно изменяемая и неизменяемая ссылка на один участок памяти
- Изменяемая ссылка может быть привязана только к изменяемому объекту

5. **Lifetime:** В случаях, когда компилятор не может самостоятельно определить продолжительность жизни ссылки на участок памяти, необходимо делать это вручную для предотвращения доступа к ранее освобождённой памяти.

Более подробно ознакомиться с идеологией Rust можно с помощью официальной документации [8] или с помощью статьи [9], в которой авторы изучают возможности языка и говорят о том, как компилятор старается предотвратить ошибки в программах. Изучить принцип работы системы проверок Rust можно в документации по анализу кода [10].

3.1.2. Swift

Одной из отличительных черт языка программирования Swift от других языков является принцип «эксклюзивного доступа к памяти». На этапе компиляции производится проверка доступа программы на запись и чтение одного и того же участка памяти с целью предотвращения *состояния гонки*, когда содержимое памяти в ходе работы может непредсказуемо меняться. Наиболее ясно работу данного принципа можно видеть на примере функций с параметрами типа *in-out*:

```

var stepSize = 1

func increment(_ number: inout Int) {
    number += stepSize
}

increment(&stepSize)
// Error: conflicting accesses to stepSize

```

Рис. 1: Swift. Функция с inout параметром. Код

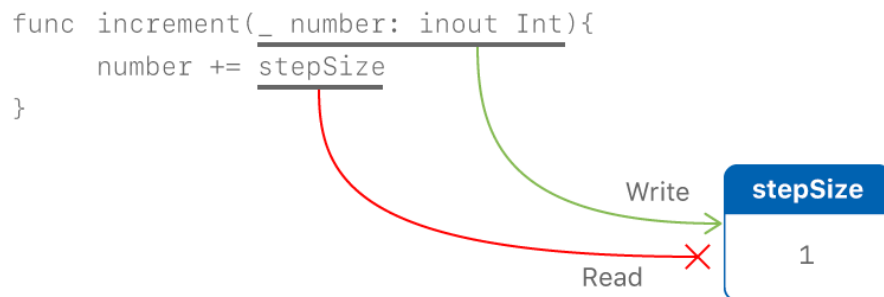


Рис. 2: Swift. Функция с inout параметром. Представление

На участке кода на рис. 1 и рис. 2, взятого из официальной документации Swift [11], можно видеть пример кода с ошибочным обращением с глобальной памятью, где происходит чтение и запись одного участка памяти.

Компилятор Swift также выдаст ошибку, если попытаться осуществить двойной доступ на запись в один участок памяти. Компилятор способен выявить данные ошибки только в области видимости функции. В глобальной области видимости ошибки данного рода компилятор выявить не может.

3.1.3. SPARK

Авторы языка SPARK, основанного на языке Ada и позиционируемого как более надёжный язык, говорят, что при создании они вдохновлялись идеологией Rust. Поэтому изучение системы проверки данного языка может быть необходимо только в случае необходимости найти более удобной и простой альтернативы языку Rust.

3.2. Системы статической проверки кода

Помимо систем проверки кода, встроенных в компиляторы языков программирования, существуют также внешние системы статического анализа кода, предназначенные для проверки уже существующего кода (чаще всего, на языке программирования C).

Поиски подобных систем не привели к большим результатам:

- Проприетарные проекты (например, проект компании Parasoft под названием C++test [12]) активно поддерживаются, но их код закрыт;
- Небольшие исследовательские проекты разных университетов и исследовательских групп выкладывают статьи, но чаще всего не оказывается хорошей документации и отсутствуют или устаревают ссылки на код.

Примером открытой системы может служить проект, разработанный инженерами университета Иллинойс [13]. Ссылка на реализацию проекта оказалась не действительной, но осталась статья от 2003 года, в которой они описывают требования, дополнительно налагаемые на язык C для обеспечения его безопасности:

1. Некоторые тривиальные требования, легко достижимые в контексте LLVM:
 - Все переменные и выражения должны быть строго типизированы;
 - Запрещено приведение к типу «указателя» переменной любого другого типа;
 - Тип «множество» может содержать только приводимые друг к другу типы данных;
2. Требования для достижения безопасности указателей:
 - Локальный указатель должен быть инициализирован перед обращением;

- Адрес на стеке не может быть сохранён на куче или в глобальной переменной и не может быть возвращён из функции;
- Нельзя напрямую работать с функциями выделения и освобождения памяти. Участки памяти для работы должны быть покрыты «потокками» памяти, запрещающими переопределение этой памяти другим потокам или стандартным функциям языка. С помощью такого подхода мы избегаем неявной проблемы обращения к освобождённой памяти: мы можем случайно обратиться к участку памяти, но будем уверены, что в нём содержится нужный нам тип данных.

3. Требования для достижения безопасности массивов:

- Индекс доступа к элементу массива должен лежать в пределах размера массива;
- Размер динамически создаваемого массива должен быть положительным числом;
- Если обращение к массиву происходит внутри цикла, то:
 - Вектор $[a, b]$ границ цикла должны быть получены путём аффинного преобразования вектора $[c, d]$, где c – длина массива, d – значение переменной вне цикла (или наоборот)
 - Выражение индекса обращения к элементу массива должно состоять из переменных циклов и размера массива
 - Если выражение индекса обращения к элементу массива содержит в себе переменную, независимую от индексов цикла, оно не должно зависеть от значения переменной

4. Исследование

4.1. Архитектура микроядра

4.1.1. Определение микроядра

Целью данной работы является изучение системы проверки кода микроядра seL4, а не устройства самого микроядра. Но изучение кода проверки какой-то системы невозможно без хотя бы поверхностного понимания принципа работы проверяемой системы.

Первым делом стоит понять, что seL4 – это микроядро, которое, в отличие от ядра, содержит только минимальную функциональность операционной системы, заключающуюся в работе с:

- виртуальной памятью;
- потоками;
- межпроцессным взаимодействием;
- примитивами драйверов устройств.

На рис. 3 можно видеть сравнение возможностей ядра и микроядра. Как раз за счёт уменьшения количества строк кода (в районе 10 тысяч в сравнении с 20 миллионами у ядра Linux) seL4 и предоставляет возможность написания кода, удовлетворяющего условиям системы проверки.

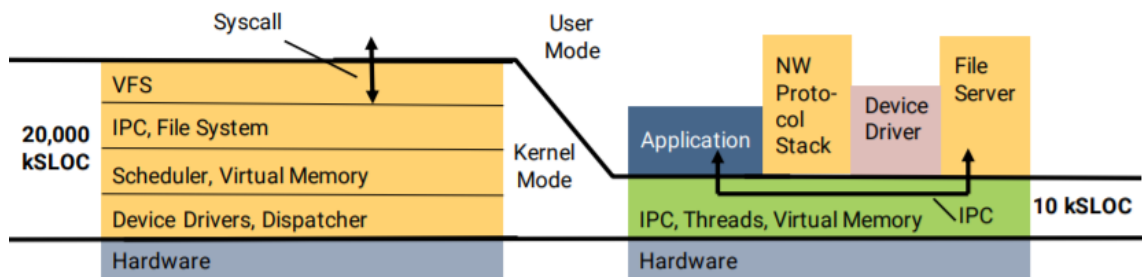


Рис. 3: Сравнение монолитного ядра и микроядра

4.1.2. Capability

«Capability» [14] – это архитектурная модель безопасности операционной системы, которая контролирует полномочия доступа к памяти. В отличие от известного принципа ACL, который предполагает определение полномочий на чтение, изменение и запись для конкретного ресурса для различных программ или потоков, Capabilities предполагает ведение списка полномочий для конкретной программы или потока. На рис. 4 можно видеть разницу двух подходов.

Capabilities API решает такую известную проблему, как «Confused deputy problem» [15], которая возникает, когда операционная система вынуждена решать, чьи права доступа ей использовать. В случае с Capabilities эта проблема решается за счёт того, что пользователю придётся вручную получить у операционной системы право на работу с потенциально уязвимым ресурсом.

**Capabilities and ACLs:
Symmetric But Very Different**

	Word Processor	Uninstaller	Doom Game
Operating System	Read Authority	Write Authority	No Authority
Confidential Docs	R/W Authority	No Authority	No Authority
Saved Games	No Authority	No Authority	R/W Authority

Individual
Access
Control
List

Set Of Capabilities

Рис. 4: Сравнение Capability API и ACL

4.1.3. Термины

Ниже приведён список терминов, понимание которых пригодится в процессе изучения блока доказательства seL4 (в будущем список может быть расширен):

1. CNode – объект для хранения Capability;

2. IPC Endpoints – интерфейсы межпроцессного взаимодействия;
3. VSpace – виртуальное адресное пространство;
4. IRQ (Control) – прерывание, Capability отслеживания прерывания конкретного устройства;
5. Untyped – участок памяти, предназначенный для создания на нём объектов ядра;
6. TCB – Thread Control Block, объект исполняемого процесса.

Более подробно ознакомиться с устройством микроядра можно с помощью документации инструкций [16] и статьи, описывающей формальную спецификацию ядра [17].

4.2. Система верификации

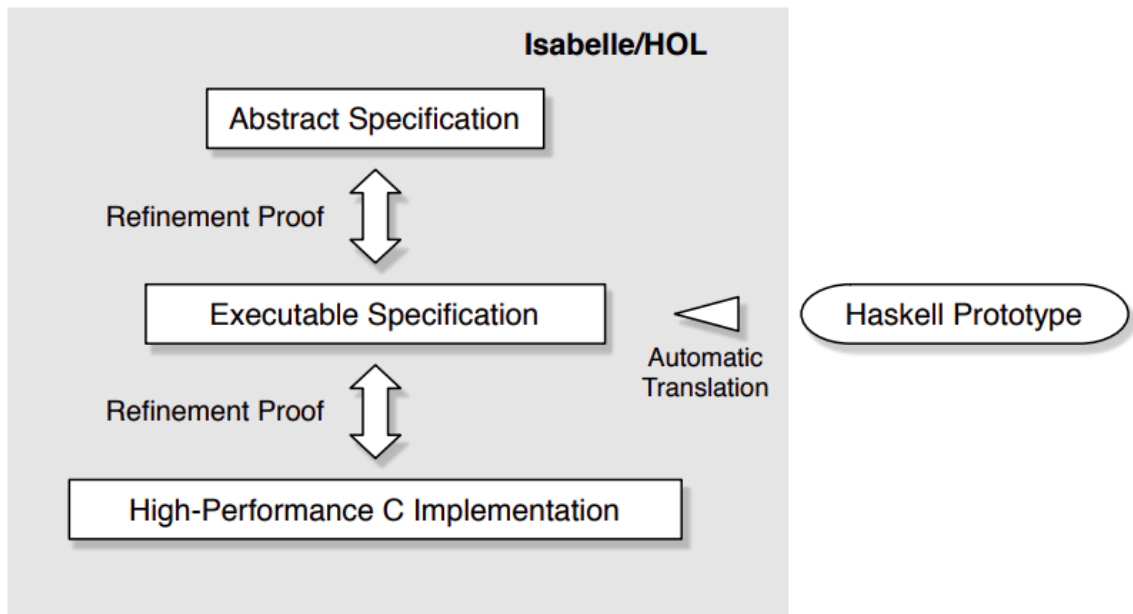


Рис. 5: Уровни абстракции верификации

Процесс верификации seL4 строится на 3 основных компонентах, их можно увидеть на рис. 5. Все они представляют собой математическую модель, представленную в виде кода на языке Isabelle:

1. Абстрактная спецификация. На этом уровне без уточнения конкретной реализации описывается ожидаемое поведение микроядра: описываются аргументы, кодировки, виды возникающих ошибок и, например, ограничения на размеры памяти. Пример кода можно видеть на рис. 6;
2. Функциональная реализация. Промежуточный слой между абстрактной спецификацией и её реализацией на языке C. Получается путём автоматической трансляции кода на языке Haskell. Представляет из себя реализацию спецификации с конкретными структурами данных и типами. Пример кода можно видеть на рис. 7;
3. Машинная реализация. Также получается путём автоматической трансляции кода на C. Строится по стандарту C99. Доказательства удовлетворения получившейся модели требованиям функциональной реализации играет ключевую роль в верификации микроядра.

```
schedule = do
    change_current_domain;
    next_domain <- gets cdl_current_domain;
    threads      <- gets (active_tcb_in_domain next_domain);
    next_thread <- select threads;
    switch_to_thread (Some next_thread)
od
```

Рис. 6: Пример кода абстрактной спецификации

4.3. Выявление требований

4.3.1. Подготовка

Если примеры кода на рис. 6 и рис. 7 кажутся довольно хорошо читаемыми, то, например, код, представленный на рис. 8, кажется совсем

```

schedule :: Kernel ()
schedule = do
    curThread <- getCurThread
    action <- getSchedulerAction
    case action of
        ResumeCurrentThread -> return ()
    ...

```

Рис. 7: Пример кода функциональной реализации

непонятным. По этой причине возникла необходимость глубже изучить синтаксис языков функционального программирования, а именно Haskell (это самый популярный функциональный язык программирования с хорошей документацией), а затем и Isabelle, основной язык доказательств.

Для изучения языка Haskell была выбрана книга под названием «Get Programming with Haskell» [18].

Для изучения языка Isabelle была прочитана официальная документация [19] от авторов языка.

4.3.2. Необходимое ПО

Для работы с кодом доказательства разработчики и авторы статей предлагают воспользоваться инструментом под названием Isabelle/jEdit [20]. Это расширение текстового редактора с открытым исходным кодом. Расширение обеспечивает удобную подсветку кода и красивое представление специальных символов и сообщает об ошибках, возникших в процессе построения доказательства. Инструмент можно получить, скачав Isabelle с официальной страницы <https://isabelle.in.tum.de/>. Запустить программу можно, перейдя в папку *bin* из корня скачанной Isabelle и исполнив команду `./isabelle jedit`.

Сразу после начала изучения доказательств стало понятно, что данный инструмент подходит больше для написания доказательств, а не

```

lemma ccorres_return_cte_cteCap [corres]:
  fixes ptr' :: "cstate ⇒ cte_C ptr"
  assumes r1: "∧s s' g. (s, s') ∈ rf_sr ⇒ (s, xfu g s') ∈ rf_sr"
  and xf_xfu: "∧s g. xf (xfu g s) = g s"
  shows "ccorres ccap_relation xf
    (λs. ctes_of s ptr = Some cte) {s. ptr_val (ptr' s) = ptr} hs
    (return (cteCap cte))
    (Basic (λs. xfu (λ_. h_val (hrs_mem (t_hrs_ (globals s)))
      (Ptr &(ptr' s →['cap_C']))) s))"
  apply (rule ccorres_return)
  apply (rule conseqPre)
  apply vcg
  apply (clarsimp simp: xf_xfu ccap_relation_def)
  apply rule
  apply (erule r1)
  apply (drule (1) rf_sr_ctes_of_clift)
  apply (clarsimp simp: typ_heap_simps)
  apply (simp add: c_valid_cte_def)
done

```

Рис. 8: Пример трудно читаемого кода абстрактной спецификации

для их изучения: инструмент не позволяет удобно перемещаться по большому количеству файлов. Однако расширение — это хороший инструмент для ознакомления с языком Isabelle. В ходе изучения документации авторы дают серию заданий на понимание материала, а точнее предлагают запрограммировать определённые модели и теоремы.

По причине неудобства работы с большим количеством файлов в Isabelle/jEdit был выбран редактор кода Visual Code, для которого существует расширение для языка Isabelle. В нём так же появляется подсветка кода и отображение спецсимволов, но становится более удобно перемещаться по структуре папок.

4.3.3. Подход

Выявление требований состоит из двух этапов:

1. Прочтение статьи. Разработчики системы верификации выложили две статьи, поясняющие принципы работы их системы: полную [21] и менее подробную [22], описывающую принципы работы системы только в общих чертах. Необходимо ознакомиться с одной из статей, чтобы понимать, что и где необходимо искать в коде

доказательства;

2. Работа с кодом. Происходит посредством клонирования репозитория <https://github.com/seL4/l4v> и его дальнейшего изучения. По своей последовательности соответствует процессу построения доказательства:

- (a) Изучение кода абстрактной спецификации. Соответствует теоремам в папке *spec/abstract*. Полезен только для понимания интерфейса взаимодействия между компонентами микроядра;
- (b) Изучение кода функциональной реализации на Haskell. Полезен для понимания конкретной реализации. Легко читаемый;
- (c) Изучение кода проверки машинной реализации. Соответствует теоремам в папке *proof/crefine*.

В процессе изучения кода были выявлены два типа требования:

1. Архитектурные. Относящиеся к реализации логических компонентов ядра;
2. Общие. Относящиеся к написанию безопасного кода на C.

Следует сказать, что в явном виде данные требования в коде не встречаются. Все они выводятся путём изучения ограничений, накладываемых на CSpace (то есть CapabilitySpace) — обёртку вокруг привычной работы с памятью.

Архитектурные	Общие
Разрешение глобальных переменных для системных структур данных с отслеживанием их инвариантов	Недопущение выхода за пределы массива
Ограничение работы с памятью через Capability blocks	Недопущение входа ядра в неактивное состояние
Ограничение на исполнение в одном потоке	Недопущение арифметических ошибок и ошибок переполнения
	Проверка правильной организованности списков
	Отсутствие пересечений по памяти

Данный список не является полным, поскольку в полной мере осознать архитектурные требования не удалось из-за сложности инструмента и кода самого ядра.

5. Альтернативный инструмент

5.1. Обзор

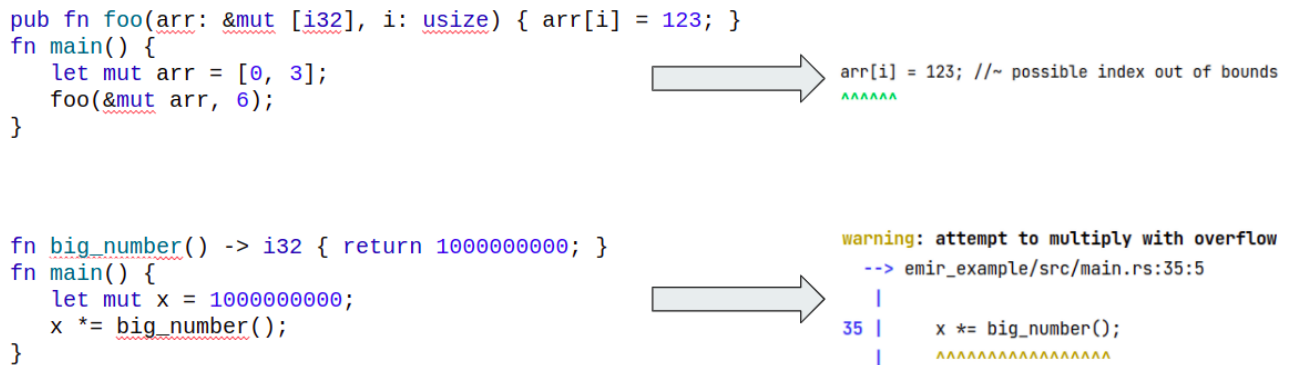
После того, как были выявлены требования к системе верификации кода с использованием Isabelle, была предпринята попытка найти существующий инструмент, который позволял бы более удобно разрабатывать систему проверки.

Выбор пал на инструмент под названием MIRAI. Его исходный код можно увидеть, перейдя по ссылке: <https://github.com/facebookexperimental/MIRAI>. Это абстрактный интерпретатор языка Rust, который в своей реализации инструмент автоматического доказательства теорем Z3. Среди подобных инструментов он считается одним из лучших и у него есть удобный интерфейс для написания кода прямо на языке Rust. Инструмент использует промежуточное представление языка Rust под названием MIR: через интерфейс, предоставляемый компилятором, он вставляет дополнительные проверки на код. Если с помощью этого инструмента можно написать систему проверки, удовлетворяющую выявленным требованиям, можно было бы решить проблему с необходимостью изучения трёх языков программирования для понимания принципов работы системы.

Для проверки работоспособности инструмента, был собран его исходный код и запущена проверка на потенциально опасных программах. В ходе тестирования инструмента удалось выяснить, что он эффективно обнаруживает следующие уязвимости:

1. Ошибки арифметики и переполнения;
2. Выходы за границы массива;
3. Возможность заморозки программы.

На рис. 9 можно видеть предупреждения, которые выдаёт инструмент после запуска его на потенциально опасном участке кода. Запуск стандартного компилятора языка данные ошибки не выявляет.



```

pub fn foo(arr: &mut [i32], i: usize) { arr[i] = 123; }
fn main() {
    let mut arr = [0, 3];
    foo(&mut arr, 6);
}

```

arr[i] = 123; //~ possible index out of bounds
AAAAAA

```

fn big_number() -> i32 { return 1000000000; }
fn main() {
    let mut x = 1000000000;
    x *= big_number();
}

```

warning: attempt to multiply with overflow
--> emir_example/src/main.rs:35:5
35 | x *= big_number();
| AAAAAAAAAAAAAAAAAA

Рис. 9: Пример работы MIRAI

5.2. Устройство доказательств

Попробуем теперь понять, как работает MIRAI. Чтобы переписать доказательства с языка Isabelle на язык Rust, нужно увидеть, каким образом можно дополнить его функциональность, чтобы при запуске инструмент проводил проверку выявленных нами ранее требований.

5.2.1. Общие требования

Напомним, что общие требования — это требования, относящиеся к написанию безопасного кода в целом, то есть, например, недопущение утечек памяти и ошибок арифметики. Поиск подобных уязвимостей — это основная задача MIRAI.

Обработку уязвимостей инструмент осуществляет путём добавления дополнительной логики в процесс обхода кода компилятором. Архитектура компилятора подразумевает применения паттерна программирования под названием «Посетитель»: посещению определённой семантической конструкции соответствует функция-колбэк, которая отвечает за обработку этой конструкции. Таким образом функциональность MIRAI основана на определении этих функций-колбэков.

Так, например, на рисунке 10 можно видеть последовательность вызова колбэков во время обработки присваивания адреса памяти. В результате серии проверок будет, например, обработан случай обращения к уже освобождённому участку памяти.

Для удовлетворения выявленных общих требований к языку необ-

ходимо будет дополнить логику обработки конкретных случаев.

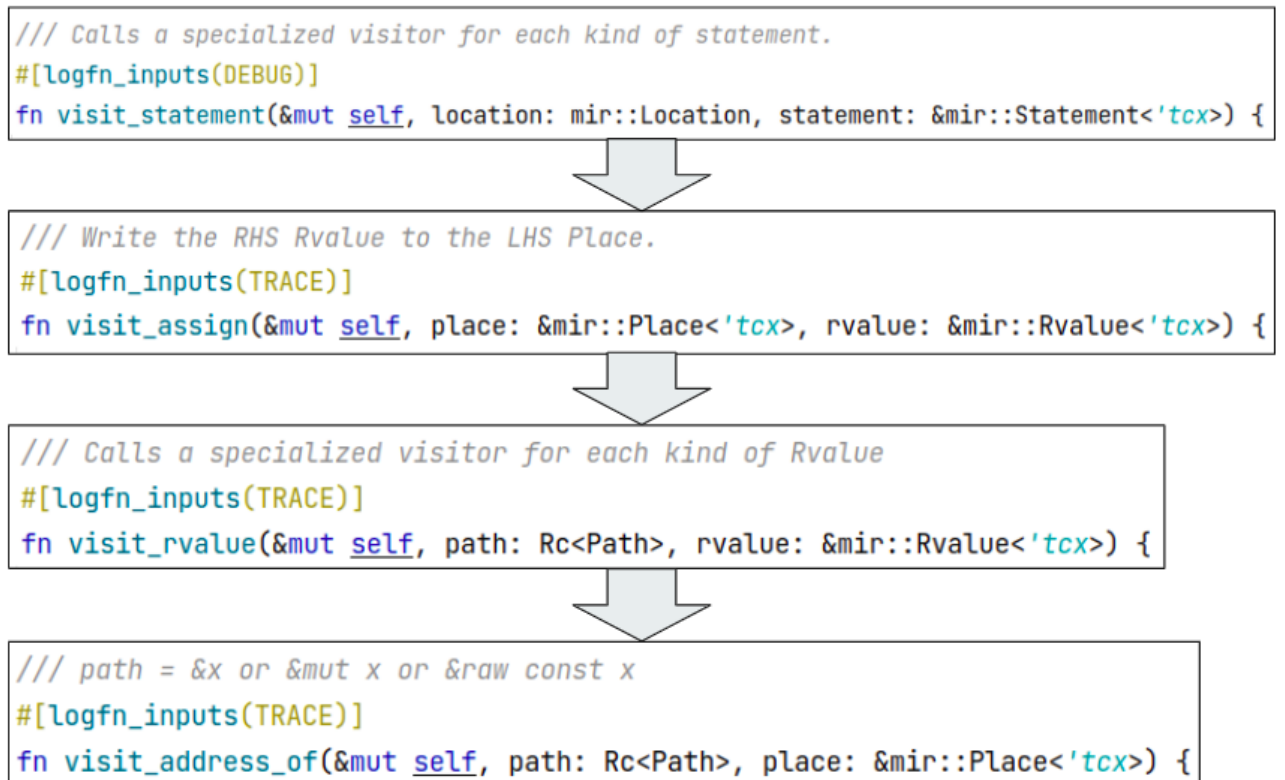


Рис. 10: Последовательность обработки выражения в MIRAI

5.2.2. Архитектурные требования

Архитектурные требования — это дополнительные требования, наложенные на архитектуру микроядра для удовлетворения дополнительных условий безопасности. По словам разработчиков [23] основная часть доказательств теорем на языке Isabelle — это проверка троек Хоара на всех выражениях микроядра, где 80% проверок составляют проверки сохранения инвариантов.

Для осуществления данных проверок можно попробовать воспользоваться пакетами *contracts* и *mirai-annotations* (они на этапе компиляции поддерживаются MIRAI), которые добавляют возможность программирования по контракту. Первый пакет предоставляет возможность устанавливать пост- и предусловия вместе с отслеживанием условия инвариантов. Второй пакет не имеет удобной функциональности по отслеживанию инвариантов, зато добавляет функциональность по ра-

боте с так называемыми «тегами»: они позволяют отслеживать потоки данных и состояний от одних переменных к другим. Пример работы тегов можно увидеть на рис. 11.

```
fn from_bytes_unchecked(bytes: &[u8]) -> PublicKey {
    let public_key = ...;
    add_tag!(&public_key, Tainted);
    public_key
}

fn try_from(bytes: &[u8]) -> Result<PublicKey> {
    let public_key = from_bytes_unchecked(bytes);
    // Perform validity checks. Return Err if the key is invalid.
    ...
    add_tag!(&public_key, Sanitized);
    Ok(public_key)
}

// Check that `sig` is valid for `message` using `public_key`.
fn verify_msg(sig: &Signature, message: &Message, public_key: &PublicKey) -> Result<()> {
    precondition!(does_not_have_tag!(public_key, Tainted) || has_tag!(public_key, Sanitized));
    ...
}
```

Рис. 11: Пример работы mirai-annotations из официальной документации

С помощью пакета *contracts*, например, можно проверять, что выполняются пост- и предусловия внутри структур данных при вызове определённой функции. На рис. 12 можно видеть, как добавляются пост- и предусловия на работу с множествами. Таким образом можно, например, покрыть работу с функциями древовидной структуры данных, которая в микроядре отвечает за отслеживание статуса блоков Capabilities.

```

impl Library {
  fn book_exists(&self, book_id: &str) -> bool {
    self.available.contains(book_id)
    || self.lent.contains(book_id)
  }

  #[debug_requires(!self.book_exists(book_id), "Book IDs are unique")]
  #[debug_ensures(self.available.contains(book_id), "Book now available")]
  #[ensures(self.available.len() == old(self.available.len()) + 1)]
  #[ensures(self.lent.len() == old(self.lent.len()), "No lent change")]
  pub fn add_book(&mut self, book_id: &str) {
    self.available.insert(book_id.to_string());
  }

  #[debug_requires(self.book_exists(book_id))]
  #[ensures(ret -> self.available.len() == old(self.available.len()) - 1)]
  #[ensures(ret -> self.lent.len() == old(self.lent.len()) + 1)]
  #[debug_ensures(ret -> self.lent.contains(book_id))]
  #[debug_ensures(!ret -> self.lent.contains(book_id), "Book already lent")]
  pub fn lend(&mut self, book_id: &str) -> bool {
    if self.available.contains(book_id) {
      self.available.remove(book_id);
      self.lent.insert(book_id.to_string());
      true
    } else {
      false
    }
  }
  ...
}

```

```

pub struct Library {
  available: HashSet<String>,
  lent: HashSet<String>,
}

```

Рис. 12: Пример работы contracts из официальной документации

6. Заключение

Поскольку работа над учебной практикой велась в течении курса, далее приводятся результаты с разбиением на два семестра.

В процессе работы над учебной практикой в осеннем семестре были выполнены следующие задачи:

1. Проведено ознакомление с существующими аналогами систем, проверяющих код на безопасность;
2. Поверхностно изучена архитектура и исходный код микроядра seL4;

В весеннем семестре планировалось выполнить следующие задачи:

1. Изучить основы написания проверки теорем на языке Isabelle;

2. Изучить документацию по верификации кода, написанного на языке Isabelle;
3. Провести анализ блока верификации seL4.

По итогу были достигнуты следующие результаты:

1. Изучены основы написания проверки теорем на языке Isabelle. Предварительно поверхностно проведено ознакомление с функциональным языком программирования Haskell;
2. Изучена документацию по верификации кода, написанного на языке Isabelle;
3. Проведён анализ блока верификации seL4;
4. Частично выявлены требования, предъявляемые системой проверки;
5. Найден инструмент, на основе которого может быть реализована более понятная система проверки кода на безопасность.

Возможные дальнейшие шаги развития данной работы — это перевод требований системы верификации seL4 с языка Isabelle на язык Rust с использованием библиотеки Z3. Дополнительно возникает задача изучения самого инструмента автоматического доказательства теорем Z3: необходимо понять принцип его работы и отличия его реализации от реализации Isabelle.

Список литературы

- [1] Naveen Gv. *Memory errors in C++*. URL: https://www.cprogramming.com/tutorial/memory_debugging_parallel_inspector.html (дата обр. 10.12.2021).
- [2] Fortinet. *Cybersecurity statistics*. URL: <https://www.fortinet.com/resources/cyberglossary/cybersecurity-statistics> (дата обр. 10.12.2021).
- [3] Michael Holloway. *Stuxnet Worm Attack on Iranian Nuclear Facilities*. URL: <http://large.stanford.edu/courses/2015/ph241/holloway1/> (дата обр. 10.12.2021).
- [4] Julian Seward. *The design and implementation of Valgrind*. URL: https://courses.cs.washington.edu/courses/cse326/05wi/valgrind-doc/mc_techdocs.html (дата обр. 10.12.2021).
- [5] *seL4 About*. URL: <https://sel4.systems/About/> (дата обр. 10.12.2021).
- [6] Paweł Urzyczyn Morten Heine B. Sørensen. *Lectures on the Curry-Howard Isomorphism*. URL: <http://disi.unitn.it/~bernardi/RSISE11/Papers/curry-howard.pdf> (дата обр. 10.12.2021).
- [7] P. Cousot. *Abstract interpretation*. URL: https://www.di.ens.fr/~cousot/AI/#tth_sEc2.5 (дата обр. 10.12.2021).
- [8] *Rust docs*. URL: <https://doc.rust-lang.org/book/title-page.html> (дата обр. 10.12.2021).
- [9] MINGSHEN SUN HUI XU ZHUANGBIN CHEN. «Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs». В: (2021). URL: <https://arxiv.org/pdf/2003.03296.pdf> (дата обр. 10.12.2021).
- [10] Rust Foundation. URL: <https://rustc-dev-guide.rust-lang.org/part-4-intro.html> (дата обр. 10.12.2021).
- [11] Swift. *Swift Documentation*. URL: <https://docs.swift.org/swift-book/LanguageGuide/MemorySafety.html> (дата обр. 10.12.2021).

- [12] *Parasoft*. URL: <https://www.parasoft.com/products/parasoft-c-ctest/c-c-static-analysis/> (дата обр. 10.12.2021).
- [13] Vikram Adve Dinakar Dhurjati Sumant Kowshik и Chris Lattner. «Memory Safety Without Runtime Checks or Garbage Collection». B: (2003). URL: <http://llvm.cs.uiuc.edu/pubs/2003-05-05-LCTES03-CodeSafety.html> (дата обр. 10.12.2021).
- [14] Professor Fred B. Schneider. *Capability-based Access Control Mechanisms*. URL: <https://www.cs.cornell.edu/courses/cs513/2007fa/L08.html> (дата обр. 10.12.2021).
- [15] *Unintended Proxy or Intermediary*. URL: <https://cwe.mitre.org/data/definitions/441> (дата обр. 10.12.2021).
- [16] Trustworthy Systems Team. «seL4 Reference Manual». B: (2021). URL: <https://sel4.systems/Info/Docs/seL4-manual-latest.pdf> (дата обр. 10.12.2021).
- [17] Trustworthy Systems Team. «Abstract Formal Specification of the seL4/ARMv6 API». B: (2021). URL: <http://sel4.systems/Info/Docs/seL4-spec.pdf> (дата обр. 10.12.2021).
- [18] Will Kurt. *Get Programming with Haskell*. URL: <https://livebook.manning.com/book/get-programming-with-haskell/chapter-1/> (дата обр. 30.05.2021).
- [19] Tobias Nipkow. *Programming and Proving in Isabelle/HOL*. URL: <https://isabelle.in.tum.de/dist/Isabelle2021-1/doc/prog-prove.pdf> (дата обр. 30.05.2021).
- [20] Makarius Wenzel. URL: <https://isabelle.in.tum.de/dist/doc/jedit.pdf> (дата обр. 30.05.2021).
- [21] Trustworthy Systems. *Comprehensive Formal Verification of an OS Microkernel*. URL: https://trustworthy.systems/publications/nicta_full_text/7371.pdf (дата обр. 30.05.2021).

- [22] Trustworthy Systems. *seL4: Formal Verification of an Operating-System Kernel*. URL: [https : / / trustworthy . systems / publications/nicta_full_text/3783.pdf](https://trustworthy.systems/publications/nicta_full_text/3783.pdf) (дата обр. 30.05.2021).
- [23] Trustworthy Systems. *seL4: Formal Verification of an OS Kernel*. URL: [http : / / web1 . cs . columbia . edu / ~junfeng / 09fa - e6998 / papers/sel4.pdf](http://web1.cs.columbia.edu/~junfeng/09fa-e6998/papers/sel4.pdf) (дата обр. 04.06.2021).
- [24] Gernot Heiser. «The seL4 Microkernel An Inroduction». B: (2020). URL: <https://sel4.systems/About/seL4-whitepaper.pdf> (дата обр. 10.12.2021).
- [25] *Capacl*. URL: [http : / / www . skyhunter . com / marcs / capabilityIntro/capacl.html](http://www.skyhunter.com/marcs/capabilityIntro/capacl.html) (дата обр. 10.12.2021).