РАЗРАБОТКА ВЫСОКОПРОИЗВОДИТЕЛЬНОГО И БЕЗОПАСНОГО С ТОЧКИ ЗРЕНИЯ РАБОТЫ С ПАМЯТЬЮ ЯДРА ОС

Комаров Т.И.[1], Чепик Н.А.[2], Иванов М.А.[3]

Целью данной работы *является практическая проверка возможности разработки высокопроизводительного ядра операционной системы (ОС), которое обеспечивает безопасную работу с памятью, без проведения полной процедуры его формальной верификации.*

Метод достижения цели *заключается в выработке подхода к разработке ядра ОС, обеспечивающего его быстродействие и корректность работы с памятью, которая может быть в будущем строго доказана; выборе подходящего языка программирования (ЯП) для разработки прототипа ядра ОС; последующей разработке прототипа ядра ОС в соответствии с выбранным подходом и проведение анализа производительности полученного прототипа.*

Полученные результаты: *представлен подход к разработке ядра ОС, обеспечивающий его высокое быстродействие и корректную работу с памятью; выработаны требования к ЯП, который может быть использован для разработки ядра ОС, обладающего требуемыми свойствами; среди наиболее известных ЯП выбран ЯП Rust для разработки прототипа ядра ОС; разработан прототип ядра ОС на ЯП Rust; выполнена оценка производительности разработанного прототипа ядра ОС; на практике доказана возможность разработки быстрого и безопасного с точки зрения работы с памятью ядра ОС без проведения процедуры формальной верификации.*

Ключевые слова: *операционная система, ядро операционной системы, микроядро, формальная верификация, оценка производительности, безопасное программирование.*

DOI: *10.21681/2311-3456-2019-3-51-56*

Введение

Ни одна действительно сложная вычислительная система не может быть построена без использования разнообразного ПО и, разумеется, без ОС. Обеспечение её корректного функционирования является критически важной задачей при разработке надежных и безопасных компьютерных систем.

Ядро ОС является наиболее важной составляющей ОС, т.к. прочее ПО, как системное, так и прикладное, зависит от корректного функционирования аппаратных и/или прочих программных средств, взаимодействие с которыми и управление которыми возможно только благодаря ядру ОС [1-2]. Если ядро ОС работает с ошибками времени исполнения, то работа любого приложения в ОС не может быть гарантированно правильной и безопасной, что следует из функциональности ядра ОС. Таким образом, для обеспечения корректности функционирования любой программной системы необходимо (и, конечно, не достаточно) обеспечить корректность функционирования ядра ОС [3-5].

Известны случаи, которые на практике продемонстрировали истинность подобных суждений. Так, например, в 2004 году работа марсохода Spirit была парализована на некоторое время практически сразу после высадки на Марсе. Причина – ошибка в программном модуле, отвечавшем за работу с файловой системой DOS, которая привела к циклической перезагрузке всей ОС, т.е. по сути к ошибке типа DoS (отказ в обслуживании). Интересно, что разработчики знали о необходимости доработки этого модуля, но данная задача не рассматривалась в качестве приоритетной [6].

На сегодняшний день были выработаны два основных подхода к обеспечению безопасной работы с памятью в ядре ОС – проведение формальной верификации ядра ОС и разработка ядра ОС на функциональном языке программирования[7-9].

В следующих разделах статьи рассмотрены данные методы, представлено логическое продолжение одного из них. Кроме того, изложены результаты анализа производительности разрабатываемого прототипа микроядра.

Формальная верификация ядра ОС

Впервые безопасность микроядра ОС была доказана в совместном проекте seL4 исследователей из NICTA, Open Kernel Labs и University of New South Wales [10]. Команда исследователей подтвердила соответствие абстрактной функциональной спецификации к высокопроизводительной реализации на C с помощью исполняемой модели (полученной из прототипа Haskell). В своей работе они обозначили основные типы инвариантов, чтобы гарантировать безопасность ядра ОС:

- низкоуровневые инварианты, гарантирующие отсутствие ошибок при работе с памятью (например, отсутствие разыменования нулевого указателя);

1 Комаров Тимофей Ильич, Национальный исследовательский ядерный университет «МИФИ», г. Москва, Россия. E-mail: tikomarov@mephi.ru.

2 Чепик Надежда Анатольевна, Межрегиональное общественное учреждение «Институт инженерной физики», г. Москва, Россия. E-mail: chepiknadya@yandex.ru.

3 Иванов Михаил Александрович, доктор технических наук, профессор, заведующий кафедрой безопасности цифровой экономики РГУ нефти и газа (НИУ) имени И.М. Губкина, г. Москва, Россия. E-mail: msadozen18@mail.ru.

- типизированные инварианты, которые следят за строгой типизацией в реализациях;
- инварианты структуры данных, которые обеспечивают безопасность с использованием структур данных;
- алгоритмические инварианты, которые описывают специфичные свойства реализации ядра ОС.

Одним из главных достижений доказательства безопасности микроядра seL4 является то, что ядро гарантировано безопасно работает с памятью, полностью исключаются такие ошибки как переполнение буфера, разыменование нулевого указателя, использование указателей неправильного типа, утечки памяти. Однако процесс формальной верификации микроядра seL4 потребовал 20 человеко-лет. Данный подход продемонстрировал, сколько требуется времени для проверки такой сложной программы, как ОС. Некоторая часть этой работы может использоваться для формальной верификации нового ядра. Однако аналогичная проверка нового ядра потребует по предварительным оценкам около 6 человеко-лет, что показывает, что даже при относительно незначительных различиях в реализации процесс формальной верификации требует очень много времени.

Разработка ядра ОС на функциональном языке программирования

Альтернативным подходом к решению задачи формальной верификации микроядра является использование функционального языка программирования, а не языка низкого уровня, такого как C, для реализации микроядра. Чисто функциональные языки обладают множеством преимуществ, таких как строгая типизация, полиморфизм, абстрактные типы данных, модульная система. Всех этих преимуществ уже должно быть достаточно, чтобы рекомендовать чисто функциональные языки для реализации микроядра ОС, но самым главным преимуществом является тот факт, что чисто функциональные языки гарантируют безопасность в работе с памятью и сборку мусора. При реализации ядра ОС на чисто функциональном языке можно использовать все преимущества языка для доказательства его безопасности, также это дает возможность использовать математический аппарат для формальных рассуждений и доказательств любых свойств реализаций ядра.

На практике, однако, оказывается, что из-за требований к реализации ядра ОС, использовать преимущества чисто функционального языка нет возможности. Например, эти языки предоставляют надежные гарантии безопасности при работе с памятью, полагаясь на систему времени выполнения, которая контролирует структуру и конфигурацию памяти. Однако основной функцией ОС является управление памятью. Большинство языков не поддерживают возможность прямого доступа к аппаратным средствам управления памятью, но такая возможность имеется через внешние вызовы. Подход, основанный на определении интерфейсов внешних вызовов, позволяет писать программы на чисто функциональных языках программирования, но при этом для соблюдения корректности работы ядра ОС надо гарантировать безопасность работы кода внешних вызовов.

Данная проблема при разработке ядра ОС на чисто функциональном языке программирования, безопасного с точки зрения работы с памятью, был решена R. Leslie в своей диссертационной работе [11]. Суть данного подхода состоит в модифицировании среды выполнения функционального языка программирования, разработке слоя аппаратных абстракций и реализации ядра ОС на выбранном языке. Свойства функционального языка программирования гарантируют отсутствие ошибок при работе с памятью, что позволяет осуществить доказательство безопасности ядра ОС, а несоответствие между требованиями к реализации ОС и возможностями чисто функциональных языков преодолевается путем изоляции небезопасных операций с памятью на уровне слоя аппаратных абстракций, который хорошо взаимодействует с функциональным языком.

Определение свойства безопасности при работе с памятью для операций на уровне аппаратных абстракций не является простым. В различных языках программирования под безопасностью обычно подразумевается отсутствие неконтролируемых ошибок, связанных с семантикой языка, а под безопасностью памяти – отсутствие набора ошибок при работе с ней. Если все операции уровня аппаратных абстракций реализованы правильно, то они должны быть безопасны с точки зрения работы с памятью в соответствии с определением безопасности памяти. Однако этого недостаточно, т.к. операция уровня аппаратных абстракций может повредить среду выполнения, т.к. она имеет доступ ко всей физической памяти и системным данным времени выполнения. Поэтому в дополнение к данному определению безопасности при работе с памятью необходимо доказать, что операции уровня аппаратных абстракций не повреждают среду выполнения и правильно распределяют память среди множества процессов. Основные характеристики безопасности слоя аппаратных абстракций могут быть выражены в виде двух свойств:

- Целостность среды выполнения. Память, занятая средой выполнения, должна отличаться от памяти, используемой для других целей.
- Целостность адресного пространства. Память, которая имеет представление в каком-либо адресном пространстве (например, таблица страниц или каталог таблиц страниц), должна отличаться от памяти, используемой для других целей.

Таким образом, свойства безопасности при работе с памятью для слоя аппаратных абстракций – это сочетание целостности среды выполнения, целостности адресного пространства и традиционного понимания безопасности при работе с памятью (отсутствие ошибок разыменования нулевого указателя и т.п.). Если реализация слоя аппаратных абстракций удовлетворяет этому определению, то можно уверенно говорить, что ядро не выйдет из строя из-за того, что программа, работающая поверх абстрактного слоя, смогла испортить среду выполнения функционального языка программирования.

Абстрактная модель аппаратного обеспечения, управляющего памятью

Для обеспечения целостности среды выполнении и целостности адресного пространства памятью в работе [11] была предложена абстрактная модель аппаратного обеспечения, управляющего памятью. Далее будет описан разработанный метод построения абстрактной модели и доказательства безопасной работы ядра с памятью.

Модель обеспечения целостности адресного пространства и среды выполнения состоит из множества компонентов: типы данных, предикаты, множество состояний, домены защиты и политика безопасности. Для решения задачи обеспечения целостности среды выполнения требуется описание структуры памяти, используются разные типы данных. Модель должна определять разные типы страниц памяти, такие как страница среды выполнения, обычная, таблица страниц, каталог таблиц страниц, удаленная:

```
data Page = Environment PageData
          | Normal PageData
          | PageDirectory DirectoryPageData
          | PageTable TablePageData
          | NotInstalled
```

Для того чтобы узнать, в каком состоянии находится страница и используется ли она, применяются предикаты:

$$isEnvironment, isNormal, isPageDirectory, isPageTable, isInstalled :: Page \rightarrow Bool. \quad (1)$$

Также в модели существует предикат для состояний *wellFormed*, благодаря которому никакое действие ОС не выведет ее из безопасного состояния, а переведет ее в другое безопасное состояние:

$$\forall s \in State, \forall op \in A. wellFormed\ s \Rightarrow wellFormed\ step(s, op), \quad (2)$$

где *A* – это множество действий в модели, а *State* – это множество безопасных состояний.

Модель также содержит в себе домены защиты. Все домены имеют доступ к ресурсам (части глобального состояния) и набору действий, которые могут в них выполняться. Политика безопасности, входящая в модель, описывает влияние данных действий на состояние системы путем указания доменов, которым доступны результаты действий какого-либо другого домена. Например, ядро может быть формализовано как набор доменов (набор пользовательских процессов) с политикой безопасности, которая запрещает процессам доступ к адресному пространству друг друга.

Вся ОС в модели представляется как набор доменов, соответствующих основным ее компонентам: программному интерфейсу для доступа ядра к аппаратному обеспечению, ядру, среде выполнения и пользовательским программам. Каждый из этих компонентов имеет возможность выполнять различные действия и получать доступ к различным частям состояния системы. Рассмотрим данные домены более подробно:

```
data D = E | H | K | U
```

Домен окружения E является абстрактным представлением системы времени выполнения языка программирования. Он содержит все страницы среды выполнения и разрешает к ним произвольный доступ. Домен *H* – это набор действий, выполняемых интерфейсом *H* для доступа ядра к аппаратному обеспечению. Домен ядра *K* - это домен ядра-клиента, использующего интерфейс *H* для доступа к аппаратному обеспечению. Домен разрешает доступ на чтение, запись к страницам, отображенным в пространстве ядра. Домен пользовательского программного обеспечения *U*. Все пользовательские программы сгруппированы, поскольку с точки зрения модели важно их влияние на состояние системы в целом, а не их взаимодействие между собой. Данный домен разрешает доступ на чтение, запись к страницам, отображенным в адресное пространство пользователя.

Каждый домен позволяет выполнять определенные действия, влияющие на состояние системы. Например, набор действий для домена *K*, влияющих на состояние системы, ограничивается одним действием:

WriteK va val – запись значения *val* на странице *va*, входящей в адресное пространство ядра.

Если домен *A* может изменять состояние домена *B* через некоторую часть глобального состояния, то такая связь называется интерференционной и обозначается $A \rightsquigarrow B$. Отсутствие интерференционной связи между двумя доменами гарантирует, что *B* не зависит от выполнения *A*. Безопасность, гарантируемая политикой, в значительной степени зависит от модели системы, потому что набор доменов, описание состояния и множество действий влияют на свойства, которые определяет политика [12].

Для создания политики безопасности необходимо определение ресурсов, операций на выделенных ресурсах и взаимодействия между доменами.

При разработке политики безопасности возможны следующие ошибки: определение неправильного набора доменов, неправильное определение последствий действий, создание огромного количества зависимостей между доменами.

Рассмотрим процесс построения модели на примере домена окружения:

- Необходимо определить действия, доступные для домена окружения. Оно всего одно: writeE, которое позволяет записывать значение на страницу среды выполнения.
- Необходимо определить на какие домены может оказывать влияние домен окружения – это все домены, которые позволяют осуществлять чтение данных страниц памяти. Домены H и K позволяют читать информацию со страниц среды выполнения, поскольку они напрямую зависят от состояния страницы среды выполнения. Сам домен окружения также следит за страницами среды выполнения.

Данные ограничения приводят к выделению трех зависимостей для домена окружения: $E \rightsquigarrow E, E \rightsquigarrow H, E \rightsquigarrow K$.

Для доказательства обеспечения целостности необходимо доказать соответствие реализации построенной модели. Для этого каждой реализации действия *opImp* необходимо поставить в соответствие спецификацию *op* и доказать, что для каждого *op* в модели реализация *opImp*:

- сохраняет безопасные состояния

$$\forall s \in State,\ wellFormed\ s \Rightarrow wellFormed\ (opImp\ s); \quad (3)$$

Таблица 1.

Базовая стоимость межпроцессного взаимодействия различных микроядер семейства L4

Имя микроядра	Год выпуска	Процессор	Частота, МГц	Задержка, МГц	Время, мкс
Original	1993	i486	50	250	5.00
Original	1997	Pentium	160	121	0.75
L4/MIPS	1997	R4700	100	86	0.86
L4/Alpha	1997	21064	433	45	0.10
Hazelnut	2002	Pentium 4	1400	2000	1.38
OKL4	2007	XScale 255	400	151	0.64
NOVA	2010	Core i7 (Bloomfield)	2660	288	0.11
seL4	2013	Core i7 (Haswell)	3400	301	0.09
seL4	2013	ARM11	532	188	0.35
seL4	2013	Cortex-A9	1000	316	0.32
Prot.	2018	Cortex-A8	1000	2592	2.59

• реализует спецификацию, когда действие возможно $\forall s \in State, wellFormed\ s \wedge (\exists s', ActionSpec\ op\ s\ s') \Rightarrow actionSpec\ op\ s\ (opImp\ s)$; (4)

• сохраняет состояние системы, когда действие невозможно $\forall s \in State, wellFormed\ s \wedge (\forall s', \neg (actionSpec\ op\ s\ s')) \Rightarrow OpImp\ s = s$. (5)

Например, для обеспечения целостности среды выполнения должно выполняться условие, что на домен окружения может влиять только он сам, но никакой другой домен. Для проверки этого условия необходимо доказать несколько лемм:

сни одно действие, принадлежащее домену U, не изменяет сопоставления виртуальной и физической памяти для любой страницы среды выполнения;

- ни одно действие, принадлежащее домену U, не изменяет домена среды.

Данные леммы показывают, что модель удовлетворяет условиям, которые гарантируют соответствие действий в системе политике безопасности.

Разработка ядра ОС на безопасном языке программирования

Подход, предложенный R. Leslie, является, на взгляд авторов данной статьи, правильным с теоретической точки зрения, поэтому он и был использован в качестве отправной точки для теоретических исследований, но его практическое воплощение для разработки L4-совместимого микроядра на языке программирования Haskell является не слишком удачным.

Haskell, как язык программирования, имеет следующие особенности:

- Большая и сложная среда исполнения, разработанная для использования в окружении ОС.
- Использование построенного по принципу «stop the world» сборщика мусора, который может приостанавливать работу программы на непредсказуемые промежутки времени.
- Ленивые вычисления – пока результаты вычислений не требуются, они не подсчитываются.
- Сложности анализа производительности, оптимизации и использования ассемблерных вставок.

Следствием неверно выбранного R. Leslie языка программирования для разработки стала низкая производительность прототипа – в 60 раз более низкая, чем у аналогичного по функциональности микроядра Pistachio, разработанного на C++ [6].

Авторами данной статьи предлагаются следующие критерии для выбора языка программирования, подходящего для разработки ядра ОС:

- Наличие безопасного подмножества языка, при использовании которого гарантируется безопасная работа с памятью.
- Наличие небезопасного подмножества языка, которое может быть использовано для реализации низкоуровневых примитивов и разработки безопасного интерфейса доступа к ним.
- Отсутствие сборщика мусора.
- Простая и компактная среда выполнения, которая может быть использована в окружении без ОС.
- Компиляция в высокопроизводительный машинный код.
- Простота анализа производительности, оптимизации и использования ассемблерных вставок.

Относительно молодой язык программирования Rust удовлетворяет всем перечисленным выше требованиям [13-15] и был выбран в качестве языка для разработки прототипа микроядра авторами данной статьи.

В качестве референсной реализации микроядра было выбрано микроядро seL4, функциональность которого была во многом воспроизведена.

Для определения производительности полученного прототипа микроядра была использована стандартная методика, применяемая для оценки быстродействия микроядер – измерение базовой стоимости межпроцессного взаимодействия, т.е. времени, необходимого для передачи сообщения от одного потока другому.

В качестве аппаратной платформы для измерения производительности прототипа микроядра была использована отладочная плата BeagleBone Black, построенная

на базе системы на кристалле AM3358 Sitara, содержащей процессорное ядро ARM Cortex-A8.

Далее представлены результаты измерения производительности различных микроядер семейства L4 [16], к которым добавлены результаты оценки производительности разработанного прототипа микроядра (табл. 1).

Как видно из представленной выше таблицы, производительность разработанного прототипа микроядра приблизительно в 8 раз ниже, чем у одного из наиболее производительных микроядер seL4, т.е. фактически на порядок выше, чем у прототипа, разработанного R. Leslie.

Следует отметить, что авторами данной статьи не проводились работы по оптимизации прототипа, а его функциональность не является законченной, т.е. указанная оценка является очень примерной, и скорее всего будет пересмотрена в будущем.

Выводы

1. Представлен подход к разработке ядра ОС, обеспечивающий его высокое быстродействие и корректную работу с памятью.

2. Выработаны требования к ЯП, который может быть использован для разработки ядра ОС, обладающего требуемыми свойствами.

3. Среди наиболее известных ЯП выбран ЯП Rust для разработки прототипа ядра ОС.

4. Разработан прототип ядра ОС на ЯП Rust.

5. Выполнена оценка производительности разработанного прототипа ядра ОС.

Таким образом, поставленная цель была достигнута – на практике была проверена и доказана возможность разработки высокопроизводительного ядра ОС, которое обеспечивает корректную работу с памятью и при этом не требует проведения процедуры полной формальной верификации.

Литература:

1. Чепик Н.А., Иванов М.А., Комаров Т.И. Построение абстрактной модели целостности адресного пространства ядра операционной системы // Известия Института инженерной физики. 2017. № 3 (45). С. 83-86.
2. Комаров Т.И., Чепик Н.А. Подход к разработке микроядра на функциональном языке программирования // В книге: XIX Международная телекоммуникационная конференция молодых ученых и студентов «МОЛОДЕЖЬ И НАУКА» Тезисы докладов. Ответственный редактор О.Н. Голотюк. 2015. С. 101-102.
3. Матвейчиков И.В. обзор методов динамического встраивания в ядро операционной системы (на примере LINUX) // Безопасность информационных технологий. 2014. Т. 21. № 4. С. 75-82.
4. Гражданкин М.А., Коваль И.А., Львова А.П., Калашникова В.А. Анализ безопасности операционных систем MICROSOFTWINDOWS и LINUX // В сборнике: Студенческая наука для развития информационного общества сборник материалов VII Всероссийской научно-технической конференции. 2018. С. 269-273.
5. Сумкин К.С., Байков И.В., Горелкин Д.О. Безопасность в операционных системах, модель безопасности операционных систем // Промышленные АСУ и контроллеры. 2011. № 7. С. 59-60.
6. G. Reeves and T. Neilson, “The Mars Rover Spirit FLASH anomaly,” 2005 IEEE Aerospace Conference, Big Sky, MT, 2005, pp. 4186-4199.
7. Костюкова Н.И. Безопасность в операционных системах, сетевая безопасность // Альманах современной науки и образования. 2011. № 7. С. 57-61.
8. Скрыпников А.В., Чернышова Е.В., Василенко А.В. Основные угрозы безопасности операционных систем // Евразийский союз ученых. 2015. № 3-4 (12). С. 43-46.
9. Девянин П.Н. О проблеме представления формальной модели политики безопасности операционных систем // Труды Института системного программирования РАН. 2017. Т. 29. № 3. С. 7-16.
10. G. Klein, K. Elphinstone, G. Heiser et al., “sel4: Formal verification of an os kernel,” in Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, ser. SOSP ’09. New York, NY, USA: ACM, 2009, pp. 207–220.
11. Leslie R. A functional approach to memory-safe operating systems. Ph.D. dissertation, Portland State University, Portland, 2011. 342 p.
12. Leslie R. Dynamic Intransitive Noninterference. // IEEE International Symposium on Secure Software Engineering, Washington, D.C., 2006.
13. Иванов С.О., Ильин Д.В., Большаков И.Ю. Сравнительное тестирование языков программирования // Вестник Чувашского университета. 2017. № 3. С. 222-227.
14. Лутфуллин Б.Л., Якупов З.Я. Язык RUST и реализация нечеткой логики // Новая наука: Проблемы и перспективы. 2016. № 6-3 (85). С. 42-50.
15. Берлизов Д.М. Реализация программной транзакционной памяти для языка программирования RUST // В книге: Материалы 54-й Международной научной студенческой конференции МНСК-2016: Информационные технологии 2016. С. 147.
16. K. Elphinstone and G. Heiser, “From l3 to sel4 what have we learnt in 20 years of l4 microkernels?” in Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, ser. SOSP ’13. New York, NY, USA: ACM, 2013, pp. 133–150. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522720.

DEVELOPING A HIGH-PERFORMANCE OS KERNEL SAFE IN TERMS OF WORKING WITH MEMORY

Komarov T.I.[4], Chepik N.A.[5], Ivanov M.A.[6]

The purpose of this paper is to practically check the possibility of developing a high-performance OS kernel to ensure safe work with memory, without having the kernel go through the complete formal verification procedure.

The goal achieving method consists in devising an approach to the development of the OS kernel, ensuring its speed and correctness of working with memory, which can be rigorously proved in the future; choosing a suitable programming language (PL) to develop a prototype OS kernel; subsequent development of the prototype OS kernel in accordance with the chosen approach and carrying out prototype performance analysis.

***Results:** An approach is presented to the development of the OS kernel, ensuring its high speed and correct work with memory; requirements were developed for PL that can be used to develop an OS kernel with the desired properties; PL Rust was chosen from among the best-known PLs to develop the prototype OS kernel; the prototype OS kernel was developed using PL Rust; the developed OS core prototype performance was evaluated; it was proved in practice that a fast and safe (in terms of working with memory) OS kernel can be developed without a formal verification procedure.*

***Keywords:** operating system, operating system kernel, microkernel, formal verification, performance analysis, safe programming*

References

1. Chepik N.A., Ivanov M.A., Komarov T.I. Postroyeniye abstraktnykh modeley tselostnosti adresnogo prostranstva yadra operatsionnoy sistemy // Izvestiya instituta inzhenernoy fiziki. 2017. № 3 (45). Pp. 83-86.
2. Komarov T.I., Chepik N.A. Tezisy dokladov. V kn .: XIX Mezhdunarodnaya telekommunikatsionnaya konferentsiya molodykh uchenykh i tudentov «MOLODEZH' I NAUKA» Tezisy dokladov. Ispolnitel'nyy redaktor O.N. Golotyuk. 2015. s. 101-102.
3. Matveychikov I.V. Obzor metodov dinamicheskogo vstraivaniya v yadro operatsionnoy sistemy (na primere LINUX) // Informatsionnyye tekhnologii bezopasnosti. 2014. Tom. 21. № 4. S. 75-82.
4. Grazhdankin M.A., Koval' I.A., L'vova A.P., Kalashnikova V.A. Analiz zashchishchennosti dostupnykh sistem MICROSOFTWINDOWS i LINUX // V sb .: Studencheskaya nauka dlya razvitiya informatsionnogo obshchestva, sbornik materialov VII Vserossiyskoy nauchno-tekhnicheskoy konferentsii. 2018. S. 269-273.
5. Sumkin K.S., Baykov I.V., Gorelkin D.O. Modeli bezopasnosti i dostupa k sistemam // Promyshlennyye ASU i kontrollery. 2011. № 7. S. 59-60.
6. G. Reeves and T. Neilson, "The Mars Rover Spirit FLASH anomaly," 2005 IEEE Aerospace Conference, Big Sky, MT, 2005, pp. 4186-4199.
7. Kostyukova N.I. Bezopasnost' v dostupnykh sistemakh, setevaya bezopasnost' // Al'manakh sovremennoy nauki i obrazovaniya. 2011. № 7. S. 57-61.
8. Skrypnikov A.V., Chernyshova Ye.V., Vasilenko A.V. Osnovnymi ugrozami bezopasnosti yavlyayutsya dostupnyye sistemy // Yevraziyskiy soyuz uchenykh. 2015. № 3-4 (12). Pp. 43-46.
9. Devyanin P.N. O problemakh predstavleniya formal'noy modeli politiki bezopasnosti v sistemakh sistem // Trudy Instituta sistemnogo programmirovaniya Rossiyskoy akademii nauk. 2017. T. 29. № 3. S. 7-16.
10. G. Klein, K. Elphinstone, G. Heiser et al., "sel4: Formal verification of an os kernel," in Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 207–220.
11. Leslie R. A functional approach to memory-safe operating systems. Ph.D. dissertation, Portland State University, Portland, 2011. [Online]. Available: https://pdxscholar.library.pdx.edu/open_access_etds/499.
12. Leslie R. Dynamic intransitive noninterference // First IEEE International Symposium on Secure Software Engineering, 2006. [Online]. Available: http://leslier.com/papers/issse06.pdf.
13. Ivanov S.O., Il'in D.V., Bol'shakov I.YU. Sravnitel'noye testirovaniye yazykov programmirovaniya // Vestnik Chuvashskogo universiteta. 2017. № 3. S. 222-227.14. Lutfullin B.L., Yakupov Z.YA. YAzyk RUST i realizatsiya nechetkoy logiki // Novaya nauka: problemy i perspektivy. 2016. № 6-3 (85). Pp. 42-50.
15. Berlizov D.M. Vnedreniye programmnoy tranzaktsionnoy pamyati dlya yazyka programmirovaniya RUST // Materialy 54-y Mezhdunarodnoy nauchnoy studencheskoy konferentsii ISSC-2016: Informatsionnyye tekhnologii 2016. S. 147.
16. K. Elphinstone and G. Heiser, "From l3 to sel4 what have we learnt in 20 years of l4 microkernels?" in Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 133–150. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522720.

4 Timofey Komarov, National Research Nuclear University "MEPhI", Moscow, tikomarov@mephi.ru.

5 Nadezhda Chepik, Interregional Public Organization "Institute of Engineering Physics", Moscow, chepiknadya@yandex.ru.

6 Mikhail Ivanov, Dr.Sc., Professor, Gubkin Russian State University of Oil and Gas (National Research University), Moscow, msadozen18@mail.ru