На правах рукописи

Нурмухаметов Алексей Раисович

ПРИМЕНЕНИЕ ДИВЕРСИФИЦИРУЮЩИХ ПРЕОБРАЗОВАНИЙ ДЛЯ ЗАЩИТЫ ОТ ЭКСПЛУАТАЦИИ УЯЗВИМОСТЕЙ

Специальность 05.13.11 —
«Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей»

Автореферат
диссертации на соискание учёной степени
кандидата технических наук

Москва — 2021

Работа выполнена в Федеральном государственном бюджетном учреждении науки Институте системного программирования им. В.П. Иванникова Российской Академии Наук.

Научный руководитель: **Курмангалеев Шамиль Фаимович**,
кандидат физико-математических наук

Официальные оппоненты: **Ильин Вячеслав Анатольевич,**
доктор физико-математических наук,
главный научный сотрудник Курчатовского комплекса НБИКС-природоподобных технологий Национального исследовательского центра «Курчатовский институт»

Волконский Владимир Юрьевич,
кандидат технических наук,
начальник отделения «Системы программирования» ПАО «Институт электронных управляющих машин им. И.С. Брука»

Ведущая организация: Федеральное государственное учреждение «Федеральный научный центр Научно-исследовательский институт системных исследований Российской академии наук»

Защита состоится 25 мая 2021 г. в 14 часов на заседании диссертационного совета Д 002.087.01 при Федеральном государственном бюджетном учреждении науки Институте системного программирования им. В.П. Иванникова Российской Академии Наук по адресу: 109004, г. Москва, ул. А. Солженицына, д. 25.

С диссертацией можно ознакомиться в библиотеке и на сайте Федерального государственного бюджетного учреждения науки Института системного программирования им. В.П. Иванникова Российской академии наук.

Автореферат разослан «__» _____ 2021 года.

Ученый секретарь
диссертационного совета
Д 002.087.01,
к.ф.-м.н. Зеленов С. В.

Общая характеристика работы

Актуальность темы. В настоящее время программное обеспечение применяется для решения прикладных задач практически в любой области человеческой деятельности. Всепроникающая информатизация и развитие интернета вещей привели к тому, что программные продукты используются повсеместно, а их количество колоссально. Поэтому задача бесперебойного и безопасного использования всего многообразия вычислительных устройств является актуальной.

Любое программное обеспечение, написанное на языках с небезопасной работой с памятью, потенциально содержит в себе некоторое количество ошибок. Они могут быть как результатом некачественного программирования и недостатков, допущенных при проектировании, так и результатом внесения специальных закладок в код открытых проектов. Оценить количество ошибок непросто из-за разнообразности исходного кода каждого проекта и неясности точного определения того, что является ошибкой, а что не является. Разные источники дают разные количественные данные, некоторое среднее значение приводимое авторитетными авторами разнится от 0,1 до 20 ошибок на тысячу строк кода. В контексте темы данной работы важно не конкретное количество ошибок, а их наличие. Не каждая ошибка может быть проэксплуатирована. Однако, даже одна единственная ошибка, затерявшаяся от глаз разработчиков в миллионах строк исходного кода большого проекта, может быть с успехом использована злоумышленниками для перехвата управления системы с последующим исполнением вредоносного кода.

Ошибки программирования или специальные закладки особо опасны по причине того, что огромное количество цифровых систем одновременно работают с идентичным программным обеспечением и могут быть массово атакованы. Идентичные исполняемые файлы самого популярного программного обеспечения работают на миллионах компьютеров. Это облегчает масштабное эксплуатирование, одна и та же атака с успехом проходит на множество целей. *Диверсифицирующие преобразования* — это такие преобразования, которые изменяют внутреннюю структуру программы таким образом, чтобы получить большое количество различных её копий, отличающихся друг от друга, но функционально эквивалентных. *Диверсифицированная популяция* — это набор различных копий программы, полученный с помощью диверсифицирующих преобразований. Диверсифицирующие преобразования позволяют уйти от однообразия программного обеспечения.

Поиск и устранение ошибок, приводящих к уязвимостям, в исходном коде — трудоемкая и дорогостоящая задача. Для ее решения существуют статические анализаторы, например Coverity, Klocwork, Svace. Кроме этого, применяется обширное тестирование на всех этапах разработки. Применение всех вышеперечисленных технологий не дает гарантии отсутствия ошибок в большом проекте, что подтверждается данными открытой базы уязвимостей и критиче-

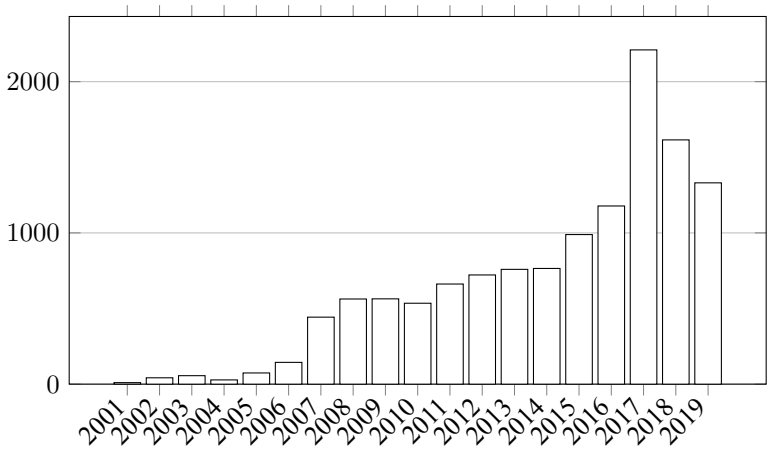

Рис. 1 — Количество опубликованных уязвимостей типа CWE 119 (ошибки работы с границами массивов) согласно базе NIST.

ских ошибок (NIST). По представленным данным на рис. 1 видно, что количество ошибок, связанных с неправильной работой с границами массивов, с течением времени не уменьшается. Из вышесказанного следует, что в условиях наличия в программном обеспечении уязвимостей актуальной становится задача защиты от эксплуатации уязвимостей.

Для решения этой задачи существует ряд технологий: предотвращение выполнения данных (DEP), рандомизация адресного пространства (ASLR), защита стека (stack canary), безопасные функции (FORIFY_SOURCE). Однако непрерывно развивающиеся методы эксплуатации уязвимостей показывают их недостаточность. DEP запрещает исполнять данные, однако оставляет возможность атакующему использовать существующий в памяти процесса код. Поэтому в последнее время широкое распространение получили атаки повторного использования кода (ROP, JOP). Против таких атак разработан ASLR, который позволяет случайно менять базовый адрес специально собранного модуля. Однако при таком подходе внутренняя структура модуля остается одинаковой и в точности соответствует исполняемому файлу на диске. Это может быть использовано атакующим для обхода такой защиты в случае, если каким-то образом ему удалось частично раскрыть базовый адрес модуля или любой из его функций. Примером может служить уязвимость CVE-2013-690, которая использовалась для деанонимизации пользователей сети Tor. Поэтому актуальной становится задача дальнейшего развития и разработки систем автоматической защиты от эксплуатации уязвимостей программного обеспечения на всех стадиях разработки, сборки, распространения и работы программного обеспечения.

Чтобы быть практически применимой, разработанная система защиты от эксплуатации уязвимостей должна удовлетворять следующим условиям:

1. незначительно ухудшать производительность программы,

2. улучшать защиту от эксплуатации методами повторного использования имеющегося в памяти процесса кода,
3. обладать совместимостью с текущими средствами защиты для достижения взаимно-усиливающего эффекта,
4. быть применимой в рамках целой операционной системы,
5. обладать обратной совместимостью.

Данные требования продиктованы желанием разработать и реализовать дополнительную систему защиты, которая может быть внедрена в существующие дистрибутивы операционной системы на базе Linux. Требования практической применимости усложняют задачу, делая ее научно-технически сложной. Для ее успешного решения полезным является богатый опыт в области запутывания и оптимизации программного кода, накопленный в ИСП РАН.

Целью диссертационной работы является разработка и реализация методов диверсификации программного кода для защиты от эксплуатации уязвимостей.

Для достижения поставленной цели необходимо было решить **следующие задачи**:

1. Произвести обзор методов эксплуатации уязвимостей и существующих механизмов защита от них; выяснить их недостатки и преимущества; изучить применимость диверсифицирующих преобразования для защиты от эксплуатации уязвимостей; выбрать инструментальные средства на базе которых будут выполняться диверсифицирующие преобразования.
2. Разработать и реализовать метод запутывания программного кода на уровне промежуточного представления компилятора для генерации диверсифицированной популяции исполняемых файлов.
3. Разработать и реализовать метод диверсификации внутренней структуры исполняемого файла при загрузке его в память.
4. Произвести оценку разработанных методов с точки зрения производительности и эффективности защиты от эксплуатации уязвимостей; выявить преимущества и недостатки разработанных методов и дальнейшие направления их развития.

Основные положения, выносимые на защиту:

1. Разработан метод диверсификации программного кода на уровне промежуточного представления компилятора для генерации диверсифицированной популяции исполняемых файлов.
2. Разработан метод диверсификации внутренней структуры исполняемого файла при загрузке его в память во время запуска приложения. Данный метод позволяет добиться рандомизации адресного пространства с зернистостью до функций.
3. Разработан метод оценки эффективности защиты от эксплуатации уязвимостей, который был применён для оценки эффективности мелкозер-

нистой рандомизации адресного пространства и позволил исправить и улучшить её реализацию.

4. На основе предложенных методов реализована дополнительная система защиты для операционных систем семейства Linux, которая обладает приемлимыми характеристиками по ухудшению производительности программы, совместима с текущими средствами защиты для достижения взаимноусиливающего эффекта, применима в рамках целой ОС, а также обеспечивает дополнительную защиту от атак повторного использования кода.

Научная новизна. В рамках диссертационной работы получены следующие основные результаты, обладающие научной новизной:

1. Предложен метод диверсификации промежуточного представления программы внутри компилятора, позволяющий генерировать большое количество исполняемых файлов приложения. Данный метод позволяет распространять пользователям уникальные версии программ, что осложняет планирование атаки на всех пользователей.
2. Предложен метод мелкозернистой рандомизации внутренней структуры программы на уровне функций, который позволяет при каждом запуске программы уникально изменять адресное пространство процесса.
3. Предложен метод оценки эффективности реализованного метода с точки зрения успешности противодействия атакам, повторно использующим имеющийся в памяти процесса код.

Практическая ценность работы. Предложены алгоритмы диверсификации программного кода во время компиляции и загрузки программы, позволяющие препятствовать эксплуатации существующих уязвимостей. Разработан метод оценки эффективности реализованной защиты и с его помощью показано, что реализованные методы защиты эффективно противодействуют эксплуатации уязвимостей методами повторного использования кода, в том числе возратно-ориентированного программирования. Метод оценки эффективности представляет теоретическую ценность и может быть использован для оценки эффективности других диверсифицирующих методов и их реализаций.

Все предложенные алгоритмы были реализованы в промышленных операционных системах CentOS 7 и Debian 10 в рамках работ по коммерческому контракту с компанией ЗАО «МВП «СВЕМЕЛ». На данные разработки получены сертификаты о государственной регистрации программ для ЭВМ: обфусцирующий компилятор для затруднения эксплуатации уязвимостей (№ 2016661393), инструмент «Faslr» для усиления системной защиты при запуске программ в ОС Linux (№ 2017660041). Данные методы представляют практическую ценность и используются в промышленных ОС для усиления защиты от эксплуатации уязвимостей.

Методология и методы исследования. Результаты диссертационной работы были получены с использованием методов диверсификации программного

кода. Математическую основу данной работы составляют теория множеств, комбинаторика и теория вероятностей

Апробация работы. Основные результаты диссертации докладывались на следующих конференциях:

1. XXI Международная научная конференция студентов, аспирантов и молодых учёных «Ломоносов-2014» (Москва, Россия, 2014 г.);
2. Открытая конференция по компиляторным технологиям (Москва, Россия, 2015 г.);
3. Научно-практическая конференция «Открытая конференция ИСП РАН 2016» (Москва, Россия, 2016 г.).
4. Открытая конференция ИСП РАН им. В.П. Иванникова (Москва, Россия, 2017 г.).

Личный вклад. Все представленные в диссертации результаты получены лично автором.

Публикации. Основные результаты по теме диссертации изложены в 7 печатных изданиях, 6 из которых изданы в журналах, рекомендованных ВАК[1—6], 1 — в тезисах докладов[7]. Публикации [3; 6] входят в международную систему цитирований Scopus и Web of Science.

В работе [4] все результаты принадлежат автору. В совместных работах [2; 3; 7] личный вклад автора заключается в описании диверсифицирующих преобразований и их реализации в рамках компиляторной инфраструктуры GCC. В работах [5; 6] автору принадлежит описание метода мелкогранулярной рандомизации и результатов противодействия эксплуатации уязвимостей. В работе [1] автору принадлежит описание существующих запутывающих компиляторов в обзорной части статьи и замеры замедления исполнения программ от различных преобразований в разделе экспериментальных результатов.

Диссертационная работа была выполнена при поддержке следующих грантов:

1. Грант РФФИ № 14-01-00462 А «Исследование и разработка методов запутывания программного обеспечения».
2. Грант РФФИ № 17-01-00600 А «Исследование и разработка методов защиты от эксплуатации ошибок в программах и методов обхода таких защит».

Объем и структура работы. Диссертация состоит из введения, пяти глав, заключения и приложений. Полный объем диссертации 140 страниц текста с 25 рисунками и 6 таблицами. Список литературы содержит 64 наименования.

Содержание работы

Во **введении** обосновывается актуальность исследований, проводимых в рамках данной диссертационной работы, ставятся цели и задачи диссертационной работы, формулируется научная новизна и практическая значимость пред-

ставляемой работы, а также приводятся основные положения, выносимые на защиту.

Первая глава посвящена обзору предметной области по защите от эксплуатации имеющихся в программах уязвимостей. Производится обзор литературы и программных средств, связанных с решаемой задачей, приводятся определения ключевых понятий.

Уязвимостью — называется недостаток в системе или программном обеспечении, используя который злоумышленник может нарушить конфиденциальность, целостность или доступность информации. Процесс нарушения данных атрибутов безопасности с использованием уязвимости называется **эксплуатацией**. Противодействием эксплуатации уязвимостей или защитой от эксплуатации уязвимостей называются меры направленные на усложнение и удорожание эксплуатации ПО, а также на снижение потенциального ущерба от эксплуатации уязвимостей.

Для противодействия уязвимостям развивается и используется подход, использующий диверсификацию программного кода. **Диверсификация кода** — это процесс изменения внутренней структуры программы, направленный на получение большого количества различных ее копий, функционально эквивалентных, но отличающихся друг от друга.

В разделе 1.1 приводится поясняющий и мотивирующий пример программы с уязвимостью переполнения буфера на стеке. На её примере показывается базовый механизм перехвата потока управления при переполнении буфера, даются определения понятиям **эксплойт** и **код нагрузки**. Эксплойтом называются входные данные, которые приводят к эксплуатации уязвимости. Кодом нагрузки называются входные данные, добавленные или встроенные внутрь эксплойта, которые при эксплуатации интерпретируются как код. Код нагрузки может выполнять вызов системной оболочки, чтение или запись данных, подготовку к выполнению следующей ступени многоуровневого эксплойта (открытие сетевого соединения, отключение защитных механизмов ОС и т.д.).

В разделе 1.2 описывается защита от уязвимости переполнения буфера, называемая протектор стека (англ. stack canary). Это такой метод защиты от уязвимостей переполнения буфера на стеке, который позволяет контролировать целостность потока управления по рёбрам возврата из функций. Описываются преимущества данного подхода и способы его обхода.

В разделе 1.3 даётся описание технологии ограничения выполнения данных (DEP), также известной под названием $\texttt{W} \oplus \texttt{X}$. Такая защита присутствует в большинстве современных операционных систем и опирается на аппаратные возможности современных процессоров (т.н. NX-бит). Данная технология позволяет операционной системе отображать все страницы памяти, содержащие данные программы (куча и стек) недоступными для исполнения, а все страницы, содержащие код программы, недоступными для записи. DEP ограничивает возможность внедрения нового кода в память процесса, а также делает невозможным дописывание и изменение имеющегося кода. Технология DEP, как впро-

чем и протектор стека, не устраняет саму уязвимость, а предотвращает сценарии эксплуатации, приводящие к произвольному выполнению кода. Уязвимость присутствует внутри приложения и может быть использована в других сценариях эксплуатации, например, для атаки отказа в обслуживании (DoS). Таким образом, DEP и протектор стека понижают опасность ряда уязвимостей с выполнения произвольного кода до отказа в обслуживании.

Повсеместное внедрение технологии ограничения выполнения данных (DEP) привело к развитию методов эксплуатации, получивших общее название **атаки повторного использования кода**. Суть таких методов заключается в использовании имеющегося в адресном пространстве процесса кода для реализации кода нагрузки. Самыми известным методами такого типа являются: **атака возврата в библиотеку** и **возвратно-ориентированное программирование** (ROP).

Раздел 1.4 посвящён описанию атаки методом возврата в библиотеку. Данный метод атаки является исторически первым методом обхода защиты ограничения выполнения данных (DEP). Показывается как с помощью перезаписи адреса возврата передаётся управление на функции из стандартной библиотеки Си, и каким образом через стек передаются значения аргументов функции.

В разделе 1.5 описывается возвратно-ориентированное программирование (англ. Return-Oriented Programming, ROP). ROP — это метод эксплуатации уязвимостей, использующий короткие последовательности инструкций из кода программы. Эти последовательности, как правило, оканчиваются инструкциями возврата и используются для составления кода нагрузки, а называются **гаджетами**.

Демонстрируется пример ROP-цепочки и показывается принцип её действия. Кроме того, показывается, что указатель стека выполняет собой роль виртуального счётчика команд, поэтому ROP-цепочку можно считать кодом для некоторой виртуальной машины. Эта виртуальная машина задаётся набором всех гаджетов в адресном пространстве эксплуатируемого процесса.

Возвратно-ориентированное программирование явилось логичным обобщением метода возврата в библиотеку. В литературе было показано, что оно применимо для различных архитектур набора команд (x86, x86_64, ARM, SPARC, MIPS, RISC-V). Кроме того, в качестве гаджетов могут быть использованы совершенно разнообразные последовательности инструкций. В том числе, инструкции непрямой передачи управления (jmp REG, см. JOP), инструкции вызова функции (см. PCOP), функциональные примитивы служебного назначения (см. DLOP, PIROP), функции целиком (см. FOP, COOP). Более того, к настоящему моменту были сделаны значительные шаги в сторону автоматизации процесса создания ROP-цепочек и появились публично доступные инструменты (т.н. ROP-компиляторы). Наличие таких инструментов упрощает злоумышленникам разработку эксплойтов. С другой стороны такие инструменты могут быть использованы для оценки эффективности методов защиты, в том числе тех, которые предлагаются в диссертации.

В разделе 1.6 приводится пример эксплойта с возвратно-ориентированным программированием в присутствии защиты от выполнения данных (DEP). Данный пример, как и многие эксплойты из реальной жизни состоит из двух этапов. На первом этапе с помощью ROP-цепочки происходит отключение защиты выполнения данных (DEP) для страницы памяти со стеком. На ОС Linux это делается с помощью системного вызова `mprotect`. На втором этапе выполняется основной код нагрузки, который представляет собой обычный ассемблерный код, сохранённый на стеке. Стоит отметить, что эксплойты, подобные представленному, существенным образом опираются на конкретное расположение секций кода и стека в адресном пространстве программы. Метод защиты под названием **рандомизация размещения адресного пространства** (ASLR) устраняет постоянность адресов, внося случайность в выбор базовых адресов этих секций.

Механизм ASLR описывается в разделе 1.7. ASLR представляет собой защитный механизм операционной системы, позволяющий размещать по случайным адресам важные элементы адресного пространства процесса: образ программы, динамические библиотеки, стек, куча. Каждому из перечисленных элементов присваивается случайный базовый адрес один раз в момент запуска приложения. Данный механизм осложняет атаки повторного использования кода. Однако, в отличии от DEP и протектора стека ASLR предоставляет вероятностную защиту.

В разделе 1.8 приводится обзор недостатков ASLR и основных способов его обхода. Среди методов обхода стоит отметить: подбор и перебор адресов, использование утечки информации о размещении кода программы или библиотеки в памяти процесса, использование для построения атаки нерандомизированных областей памяти, частичная перезапись указателей на код.

В разделе 1.9 описываются существующие подходы к мелкозернистой рандомизации размещения адресного пространства. Среди описанных в литературе можно выделить четыре подхода, реализованных в следующих инструментах: Selfrando, XIFER, Oxymoron, pagerando. Из них в открытый доступ выложен только Selfrando.

В проекте Selfrando используется подход к мелкозернистой рандомизации наиболее близкий к развиваемому в данной работе. Для осуществления рандомизации во время статического связывания собирается информация о границах функций и местоположении всех ссылок на код в программе. Для этого используется обертка над системным компоновщиком. Собранная информация вместе с библиотекой RandoLib (которая осуществляет перестановку функций во момент запуска программы) записывается внутрь исполняемого файла. Практическим применением Selfrando является усиление защиты Tor браузера, а также показана практическая применимость защиты на примере конкретных CVE. Увеличение энтропии по сравнению с обычным ASLR оказывается значительным. Даже для маленьких файлов в 10 КБ размера построение цепочки из трех гаджетов потребует перебора как минимум 39 бит энтропии, что значительно больше 29 в сравнении с обычным ASLR. Уменьшение производительности оценивается в

0,71 % при увеличении потребляемой памяти на 0,2 %. Однако, исполняемый файл с Selfrando с точки зрения антивирусных средства защиты трудно отличим от самомодифицирующихся вредоносных программ. Кроме того, процесс сертификации такого кода может быть затруднительным. Подход Selfrando рассчитан на сборку отдельных приложений с усиленной рандомизацией, но плохо применим при масштабировании такого решения для защиты всех приложений в ОС.

Инструмент XIFER представляет собой прототип инструмента бинарного переписывания. Во время динамического связывания исполняемых файлов и библиотек происходит дизассемблирование и построение графа связей между кодом и данными. С помощью построенного графа после перемешивания восстанавливаются связи между инструкциями и данными, а также исправляются инструкции передачи управления между разными кусками кода. Инструмент XIFER поддерживает произвольную зернистость перемешивания (вплоть до одной инструкции). Однако экспериментальным путём установлено, что разбиение на куски меньше, чем 6 инструкций крайне нежелательно, поскольку снижает производительность катастрофически. Компромиссным вариантом между производительностью и энтропией является разбиение на 13 частей для 32-битной архитектуры и 17 частей для 64-битной архитектуры. XIFER поддерживает два режима работы с библиотеками: с разделением библиотек и без разделения библиотек. При первом режиме код и данные библиотеки рандомизируются только один раз при первой её загрузке в систему. Второй режим позволяет рандомизировать библиотеку для каждого процесса отдельно, за счет этого увеличивается общесистемное потребление памяти, но увеличивается защищенность от атаки через недоверенные или скомпроментированные процессы ОС.

Инструмент Oxymoron реализован на основе фреймворка XIFER. Его ключевой особенностью является возможность разделения памяти с сохранением мелкозернистой рандомизации. Для этого было предложено новое соглашение о вызовах, названное PALACE (англ. Position and Layout Agnostic Code). Код разбивается на куски размером в страницу памяти. Все междустраничные передачи управления становятся косвенными и разрешаются с помощью специальной таблицы RaTTle, которая расположена в неизвестном атакующему месте. Таким образом, содержимое страницы с кодом остаётся неизменным и может быть разделено между разными процессам. При проведении рандомизации необходимо исправлять только значения, расположенные в таблице RaTTle. Разделение позволяет снизить расходы оперативной памяти для всех работающих процессов. Увеличение потребления памяти на отдельный процесс оценивается в 15 %. Падение производительности составляет 2,7 %.

Похожий на Oxymoron подход развивается в инструменте pagerando. С тем отличием, что он реализован внутри компилятора и компоновщика для ОС Android для архитектуры ARM. Аналогичным образом код разделяется на куски размером в страницу памяти и добавляется таблица переходов между страницами. Адрес таблицы хранится в регистре r9, выведенном из общего оборота. Падение производительности оценивается в 6,5 %, а увеличение потребления памяти

оценивается в 19 %, что в настоящий момент мешает внедрению в повсеместное использование.

Вторая глава посвящена описанию *разработки и реализации диверсифицирующих преобразований на уровне промежуточного представления компилятора с целью генерации диверсифицированной популяции исполняемых файлов*.

Предлагаемый метод заключается в следующем:

- Для распространения пользователям вместо одной копии генерируется множество различных копий приложения.
- Генерацию различных копий осуществляет модифицированный компилятор.
- Модифицированный компилятор имеет источник случайности и набор диверсифицирующих преобразований:
    1. Генератор случайных чисел служит источником недетерминированности процесса трансляции.
    2. Преобразование перестановки функций местами внутри модуля или всей программы,
    3. Преобразование перестановки местами базовых блоков внутри функции,
    4. Преобразование добавления локальных переменных и перемешивания их на стеке.
- В процессе трансляции исходного кода запускаются диверсифицирующие преобразования.
- Процесс трансляции необходимо повторять отдельно для каждого пользователя.

В разделе 2.1 обсуждается ситуация однообразия исполняемых файлов приложений. Идентичность исполняемых файлов приложения создает угрозу крупномасштабной эксплуатации в случае, если злоумышленник находит эксплуатируемую уязвимость. Один эксплойт способен эксплуатировать все используемые копии уязвимой версии приложения до момента их обновления.

Одинаковости исполняемых файлов можно избежать путем генерации различных копий заданной версии приложения и распространения каждому конечному пользователю уникальной копии. Другими словами, можно создать диверсифицированную популяцию исполняемых файлов заданной версии приложения. Диверсифицирующие преобразования должны быть достаточно мощными для того, чтобы сгенерировать большое количество разнообразных версий приложения.

В разделе 2.2 предлагается подход для генерации диверсифицированной популяции исполняемых файлов приложения с помощью специального компилятора. Компилятор, оборудованный набором диверсифицирующих преобразований, может генерировать большое количество функционально эквивалентных, но внутренне различных копий компилируемой программы. Поведение программы, специфицированное исходным кодом, у всех копий одинаковое. Разным является поведение в тех случаях, которые не специфицированы напрямую в ис-

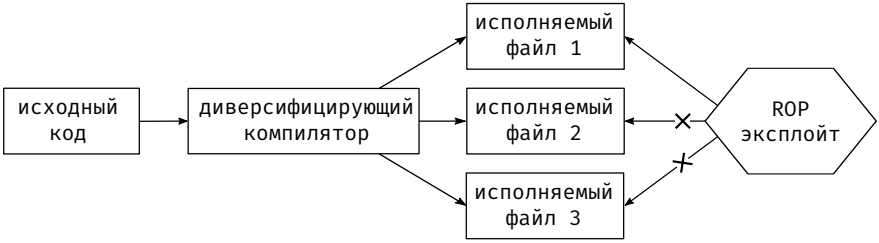

Рис. 2 — Модель атаки на диверсифицированную популяцию исполняемых файлов.

ходном коде программы, стандартом языка или соглашениями о вызовах и общих интерфейсах. Каждая версия программы ведет себя по-разному при попытке эксплуатировать ее неспецифицированное поведение (уязвимость). Таким образом, эксплойт, разработанный для одной версии приложения, не будет работать для других версий приложения. Схематично это изображено на рис. 2.

В разделе 2.3 предлагается источник случайности и обсуждаются возможности для диверсификации на уровне промежуточного представления компилятора: перестановка функций, базовых блоков, локальных переменных, инструкций; добавление инструкций и локальных переменных. В разделах 2.4, 2.5 обсуждаются особенности реализации диверсифицирующих преобразований в компиляторных средах GCC и LLVM соответственно.

В разделе 2.6 описываются реализованные в компиляторах диверсифицирующие преобразования: перестановка функций местами, перестановка базовых блоков, добавление и перемешивание локальных переменных. Раздел 2.7 посвящён анализу влияния реализованных преобразований на производительность. На наборе тестов SPEC CPU® 2006 среднее падение производительности составляет 0,5 %. Падение производительности никакого из тестовых приложений не превосходит 2 %. Примерно у половины тестов в наборе не изменился размер исполняемых файлов, тогда как у другой половины он увеличился на несколько процентов. Среднее увеличение размера исполняемых файлов оценивается в 0,8 %.

Третья глава посвящена описанию мелкозернистой рандомизации внутренней структуры программы при запуске.

В разделе 3.1 описывается предложенный *метод диверсификации внутренней структуры исполняемого файла при загрузке его в память во время запуска приложения с зернистостью до уровня функций*, который состоит из следующих этапов:

1. сбор и сохранение информации о границах функций во время трансляции исходного кода программы,
2. сбор и сохранение информации о ссылках на символы во время статической компоновки,
3. перемешивание функций местами с использованием информации из первого пункта во время запуска программы в загрузчике,

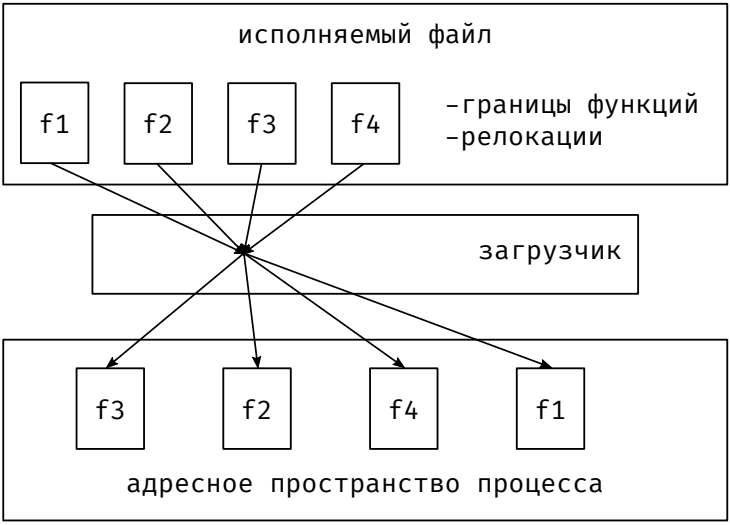

Рис. 3 — Схема работы мелкозернистой рандомизации во время запуска программы.

4. исправление значений ссылок на символы с использованием информации из второго пункта.

Концептуальная схема работы предлагаемого метода представлена на рисунке 3.

В разделе 3.2 описываются технические детали реализации предлагаемого метода мелкозернистой рандомизации. Реализация метода производилось для архитектуры x86-64 и ОС Linux, использующей ELF как основной формат исполняемых и библиотечных файлов приложения. Вся информация, необходимая для рандомизации, хранится в дополнительной секции, которая игнорируется при использовании стандартного динамического загрузчика. Это позволяет минимизировать проблемы совместимости.

Во время статического связывания компоновщик обходит все функции в исполняемом файле и сохраняет информацию об их границах в специальную секцию. Кроме того, собирается и сохраняется информация о ссылках на символы в коде. Формат хранения этой информации описывается в разделе 3.2.1. Во время динамического связывания системный загрузчик на основе содержимого специальной секции производит перестановку функций местами и исправление ссылок на код. Перестановка функций происходит при загрузке как исполняемых файлов, так и разделяемых библиотек.

В разделе 3.2.2 отмечается, что в дополнительной секции сохраняется только информация о ссылках на символы, которые необходимо будет исправить после перемешивания функций. Таким образом, относительные ссылки, которые указывают из данной функции внутрь этой же функции, и абсолютные ссылки,

целевой адрес которых не находится ни в одной из функций, не нуждаются в сохранении и последующем исправлении после перестановки. Кроме того, часть ссылок пришлось обрабатывать специальным образом. Многие ссылки, относящиеся к реализации локальной памяти потока (TLS) пришлось рассматривать отдельно для определения формулы для подсчёта целевого адреса. Особая обработка потребовалась также для секции `.eh_frame`, которые используются для размотки стека при обработке исключений. Для получения корректной информации о таких ссылках пришлось реализовать разбор их итогового бинарного содержимого.

В разделе 3.2.3 уточняется, что перестановка функций производится в тот момент, когда исполняемый файл уже корректно загружен в память и произведена, если необходима, динамическая компоновка. Перестановка производится случайным образом, при этом запоминается на какое смещение в байтах была сдвинута функция. Затем исправляются ссылки: к значению каждой ссылки прибавляется смещение целевой функции; если ссылка относительная, то вычитается смещение той функции, в которой содержится исправляемая ссылка. Стоит отметить, что перестановка функций модифицирует код программы в памяти. Как правило, страницы с кодом недоступны для записи. Из-за этого приходится делать системный вызов `mprotect` для временного разрешения их модификации. В случае присутствия в ядре ОС механизмов защиты (SELinux, PaX, seccomp), которые запрещают одному региону памяти быть исполняемым и записываемым, может потребоваться их дополнительная настройка и адаптация для разрешения такого поведения. Поэтому перспективным направлением для дальнейшего развития данного подхода является создание мелкозернистой рандомизации с разделением памяти. После завершения перестановки в памяти не остаётся никакой дополнительной информации, утечка которой могла бы раскрыть размещение функций.

В разделе 3.3 приводятся данные о влиянии мелкозернистой рандомизации на производительность. Тесты на производительность производились на наборе тестов SPEC CPU® 2017. Результаты показывают, что производительность отдельных тестов как ухудшилась до 7 %, так и увеличилась до 4,3 %. Производительность большинства тестов изменилась в пределах 2 % в обе стороны. Изменение показателя производительности для набора тестов SPEC CPU® 2017 приводится в таблице 1.

Таблица 1 — Результаты измерения производительности SPEC CPU® 2017 для мелкозернистой рандомизации

SPEC CPU®	без ранд.	с ранд.	Δ
intspeed	3,85	3,84	0,3 %
fpspeed	7,80	7,80	0 %
intrate	3,09	3,06	1 %
fprate	3,16	3,15	0,3 %

Мелкозернистая рандомизация увеличивает количество действий, производимых системным загрузчиком, при запуске программы. По результатам измерений видно, что :

– время работы загрузчика увеличилось в 1.1–33 раз, при среднем относительном замедлении около 6;
– абсолютное время загрузки не превышает 30 мс.

Размер исполняемых файлов в среднем увеличился на 31 % из-за специальной секции со служебной информацией о границах функций и ссылках. Стоит отметить также, что динамическая разделяемая библиотека, собранная с поддержкой мелкозернистой рандомизации, перестает быть разделяемой между разными процессами. При запуске каждого процесса перемешивание функций происходит независимо случайным образом, поэтому в каждом процессе данная библиотека представлена по-разному. Это увеличивает защищенность системы в целом, поскольку исключается возможность утечки размещения адресного пространства данной библиотеки из других процессов операционной системы. Однако, из-за отказа от разделяемости библиотек увеличивается общее потребления оперативной памяти в операционной системе. По данным измерений потребление памяти увеличивается в 3–4 раза, потребляемая память серверной установки увеличивается до 200 МБ, а настольной до 2,5 ГБ.

В **четвертой главе** приводится описание метода оценки эффективности реализованных методов защиты от эксплуатации уязвимостей и производится оценка эффективности реализованной мелкозернистой рандомизации.

Метод оценки эффективности реализованного метода защиты состоит из следующих шагов:

– Определение модели атаки (повторное использование кода, ROP).
– Оценка количества гаждетов, остающихся на своих местах после преобразований.
– Оценка работоспособности эксплойтов на уязвимостях, синтетически внедренных в программы (набор из 670 штук):
    1. Внедрение в программу уязвимости.
    2. Создание эксплойта в формате ROP-цепочки.
    3. Проверка работоспособности ROP-цепочки.
– Оценка работоспособности эксплойтов на реальных примерах уязвимостей из базы CVE.

В начале раздела 4 определяется модель атаки, используемая при оценке эффективности. Атакующий имеет в своем распоряжение исполняемый файл и предполагает, что в атакуемой операционной системе среди защитных механизмов имеется только DEP. Кроме того, он нашел какую-то уязвимость и умеет её эксплуатировать. Атакующий создает нагрузку эксплойта в виде ROP-цепочки. Созданный таким образом эксплойт подается на вход программе, адресное пространство которой изменяется при запуске с помощью мелкозернистой рандомизации.

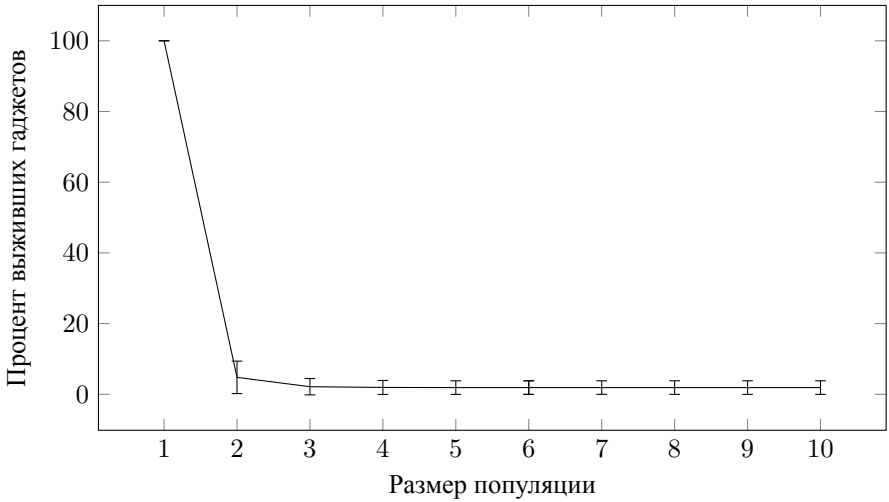

Рис. 4 — Среднее относительное количество выживших гаджетов в зависимости от размера популяции

В предложенной модели атаки эксплойт не будет работать из-за изменения относительного положения функций. Используемые в ROP-цепочке гаджеты поменяют своё положение. Перехваченный поток управления попадёт вместо гаджета на некоторое произвольное место. На архитектуре x86 из-за плотности набора инструкций практически любое произвольное место программы является началом какой-то последовательности инструкций. Исполнение случайных инструкций с большой вероятностью закончится на обращении в невыделенную память или попыткой исполнения несуществующей инструкции. Таким образом, атака понизит свою опасность с удаленного выполнения кода до отказа в обслуживании.

Однако, существует вероятность, что перестановка функций произойдёт таким образом, что используемые гаджеты не поменяют своего местоположения. Такому условию всегда удовлетворяет исходный порядок функций. Его вероятность равна $1/N!$. Эта оценка является оптимистичнее реальности, поскольку в программе могут быть похожие куски кода в разных местах, например, прологи и эпилоги функций. И тем не менее, для небольших программ размером до 10 функций среднее количество попыток запуска до успешной эксплуатации меньше миллиона.

Для оценки реальной вероятности проэксплуатировать программу при перестановке функций в настоящей диссертации развивается экспериментальный метод проверки качества защиты. В рамках него в разделе 4.2 оценивается количество выживших гаджетов, в разделе 4.3 работоспособность реальных ROP-цепочек, в разделе 4.4 работоспособность эксплойтов на реальных примерах.

В разделе 4.2 вводится определение понятию выживший гаджет в предположении о том, что имеется некоторая популяция снимков памяти одной и той же программы с разных запусков. Тогда **выживший гаджет** — это такой гаджет исполняемого файла, которой находится по одному и тому же адресу в каждом снимке памяти из популяции. Составленная из выживших гаджетов ROP-цепочка работоспособна на каждом снимке памяти из популяции. На рисунке 4 представлена кривая, отражающая среднее арифметическое значение доли выживших гаджетов по всем программам из тестового набора. Кроме того, у каждой точки отложено среднеквадратичное отклонение от среднего значения. Характер формы представленной кривой напоминает экспоненциально убывающую последовательность с некоторым константным смещением по оси абсцисс вверх примерно на 2 %. Данное остаточное количество выживших гаджетов, наблюдаемое независимо от размера популяции, объясняется наличием кода в неперемешиваемых секциях (INIT, FINI, PLT).

По собранным данным также можно оценить вероятность гаджета остаться на своем месте, если считать что m – количество файлов, n_j – количество гаджетов в j файле, k_i^j – количество файлов, в которых гаджет g_i^j остался на своем месте.

$$\frac{\sum_{j=1}^{m} \left(\frac{\sum_{i=1}^{n_j} k_i^j}{10 n_j} \right)}{m} = 0{,}05 \qquad (1)$$

Приблизительная вероятность гаджета остаться на своем месте (по тестовому набору файлов) оценивается примерно в 5 %.

В разделе 4.3 приводятся результаты эксперимента по проверке работоспособности ROP-цепочек. Для этого на наборе файлов из CentOS 7, располагаемых в директориях /usr/bin/, /usr/sbin (количество 487), проверялась работоспособность ROP-цепочек, созданных полуавтоматизированными методами. Для рассмотрения были взяты только файлы, собранные без поддержки позиционно-независимого кода (PIE). Это необходимо для того, чтобы исключить влияние системной ASLR на успешность ROP-цепочки эксплойта. Использовалось пять типов модельной нагрузки разной сложности (вызов функции с разным количеством аргументов и вызов оболочки системного интерпретатора). Сложность ROP-цепочек в зависимости от типа нагрузки показана на рисунке 5, а также количество сгенерированных цепочек каждого типа модельной нагрузки.

В разделе 4.3.2 описывается процесс генерации ROP-цепочек. Для их генерации использовался разработанный в ИСП РАН инструмент MAJORCA. Схематично процесс создания цепочек состоял из следующих этапов:

– В зависимости от типа цепочки определялся набор регистров, нуждающихся в инициализации значениями. Примеры тестировались на машине x86_64, что однозначно определяло соглашение о вызовах и названия регистров, через которые передавались параметры в функции:

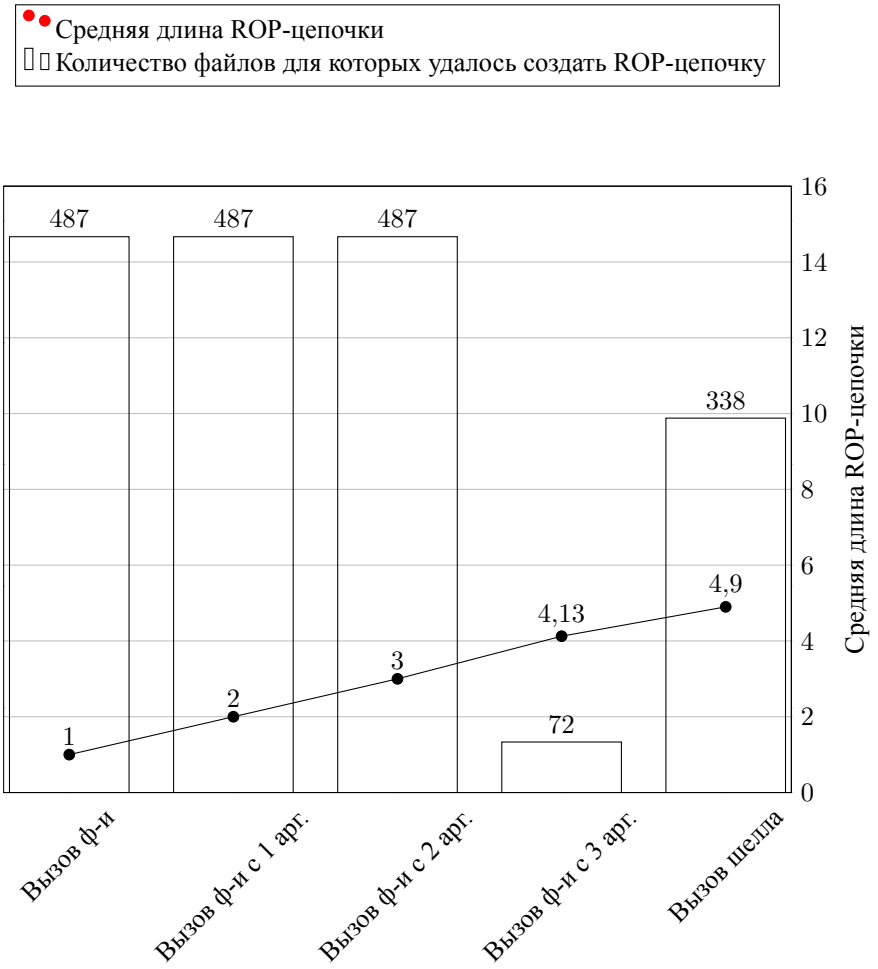

Рис. 5 — Статистика по результатам создания ROP-цепочек для модельных типов нагрузок. По оси абсцисс отложены типы созданных ROP-цепочек. Столбцы показывают количество исполняемых файлов для которых были успешно созданы ROP-цепочки. Сплошной линией показано среднее количество ROP-гаджетов, которые потребовались для создания цепочки соответствующего типа.

RDI, RSI, RDX соответственно для первого, второго и третьего аргумента.

- Производился поиск гаджетов, которые инициализируют определенные регистры. Инициализация производилась путем загрузки значений на регистры со стека.
- Для цепочки вызова интерпретатора системной оболочки происходил также поиск гаджета записи значения из регистра в память с подходящими ему гаджетами загрузки значения на регистр со стека.
- Склеивание гаджетов друг с другом в конечную ROP-цепочку происходило с помощью конкретных значений, которые должны загружать на регистры гаджеты инициализации регистров значениями.
- В конец цепочки дописывался адрес функции, которую необходимо вызвать.

В разделе 4.3.3 описывается способ внедрения в каждое приложение эксплуатируемой уязвимости. Он осуществляется путём внедрения кода с уязвимостью и кода проверки в адресное пространство процесса через LD_PRELOAD. Внедренный код содержит уязвимость переполнения буфера на стеке при чтении данных из файла в момент прямо перед запуском функции main программы. Вместе с уязвимостью внедряется проверка корректного выполнения кода нагрузки.

Каждую созданную ROP-цепочку подавали на вход тестируемого приложения. Данная процедура повторялась по 100 раз для каждой цепочки для каждой программы. Кроме того, выбор местоположения целевой функции ROP-эксплойта влияет на процент успешных запусков, как показано на рисунке 7. Поэтому проводились две серии измерений: серия с краевой целевой функцией (КЦ) и серия с средней целевой функцией (СЦ).

На рисунке 6 изображена усредненная зависимость статистики эффективности противодействия. По оси абсцисс отложены пять точек, соответствующих модельным нагрузкам: вызов функции без аргумента, вызов функции с одним аргументом, вызов функции с двумя аргументами, вызов функции с тремя аргументами, вызов оболочки системного интерпретатора (/bin/sh). Две сплошные линии соответствуют усредненным значениям относительного количества в процентах успешных запусков ROP-цепочек для соответствующих модельных примеров. Первой КЦ серии соответствует верхняя линия, для нее также подписаны значения отдельных точек. Второй СЦ серии тестов соответствует нижняя сплошная линия. Синяя прерывистая линия показывает усредненную кривую для обеих серий экспериментов. Из результатов эксперимента, отображенного на данном графике, можно сделать вывод, что процент успешности резко падает с увеличением длины ROP-цепочки (точнее сказать с увеличением количества требуемых для создания цепочки ROP-гаджетов), и для нетривиальных модельных нагрузок стремится к нулю.

В разделе 4.4 приводятся результаты работоспособности на реальных примерах уязвимостей: gv CVE-2004-1717, rsync CVE-2004-2093, proftpd CVE-2006-

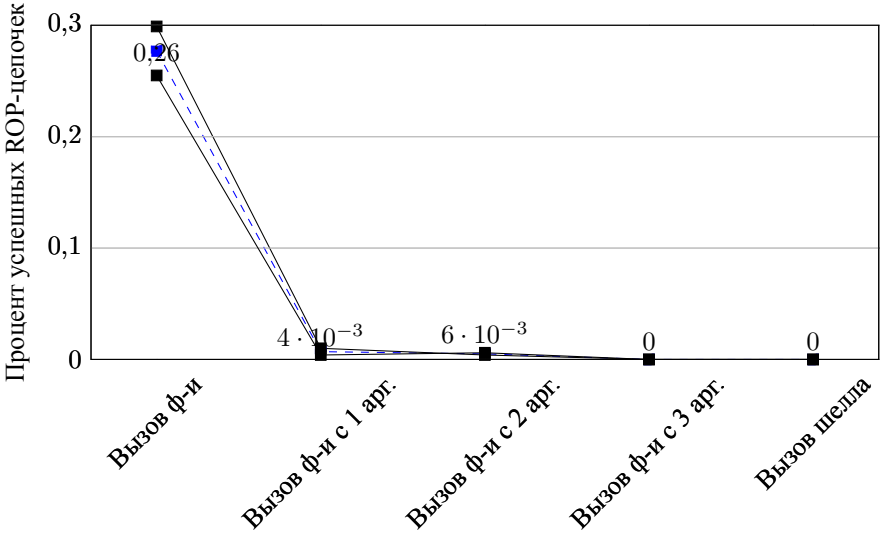

Рис. 6 — Статистика по эффективности противодействия атакам повторного использования кода в виде ROP-цепочек.

6563, opendchub CVE-2010-1147, nginx CVE-2013-2028, torque CVE-2014-0749. На каждом примере было показано, что мелкозернистая рандомизация позволяет предотвращать эксплуатацию уязвимых приложений.

В **пятой главе** проводятся исследования задачи о случайных перестановках с точки зрения математики.

Для оценки вероятности каждой функции остаться на своём месте были исследованы два крайних случая: случай, когда длины всех функций в программе одинаковы; случай, когда длины всех функций в программе различны, а кроме того длина никакой функции из программы не равна сумме длин любого набора функций из программы. Математическое ожидание количества функций, остающихся на своем месте в случае равенства всех длин функций равно 1. Математическое ожидание в случае существенно различных длин функций можно, как показано в главе 5, посчитать по формуле

$$MX = \sum_{i=1}^{n} \frac{(i-1)!(n-i)!}{n!} \tag{2}$$

Кроме того, показано, что математическое ожидание количества неподвижных функций для реальной программы находится между значениями двух крайних случаев при любом количестве функций.

В разделе 5 показывается, что при перестановке функций вероятность крайних функций остаться на своём месте значительно превосходит вероятность средних функций остаться на своём месте. Наглядно это изображено на рисун-

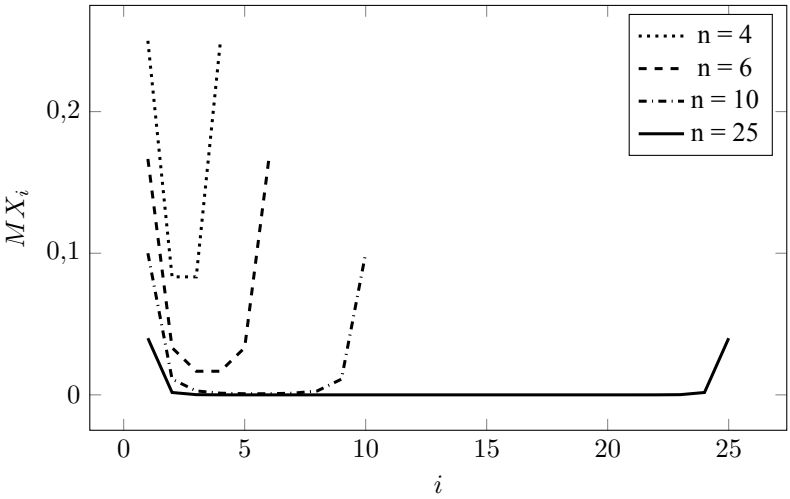

Рис. 7 — Графики зависимости вероятности функции под номером i остаться на своем месте в предположении о существенном различии их длин для $n \in [4{,}6{,}10{,}25]$

ке 7. Данный вывод повлиял на реализацию мелкозернистой рандомизации. В неё было добавлено смещение базового адреса всех функций на некоторую случайную величину. Это позволило выровнять вероятность нахождения крайних и средних функций на своём месте, что показано в разделе 4.3.4.

В **шестой главе** приводится описание программной реализации *дополнительной системы защиты для ОС семейства Linux* на основе методов, предложенных в главах 2 и 3, *которая обладает приемлимыми характеристиками по ухудшению производительности программы, совместима с текущими средствами защиты для достижения взаимноусиливающего эффекта, применима в рамках целой ОС, а также обеспечивает дополнительную защиту от атак повторного использования кода*. Архитектура реализованной системы дополнительной защиты представлена на рисунке 8.

В разделе 6.1 описывается реализованная дополнительная функциональность компилятора GCC, позволяющая генерировать диверсифицированную популяцию копий приложения. Для этого в компилятор добавлены дополнительные модули: генератор случайных чисел, преобразование для перестановки местами функций в пределах единицы трансляции, преобразование для добавления и перестановки локальных переменных, преобразование для перестановки местами базовых блоков в пределах функции. Раздел продолжается описанием перечисленных модулей. Кроме того, даётся описание алгоритма случайной перестановки объектов. Приводится описание реализации перестановки базовых блоков, а также приводится способ передачи компоновщику от компилятора информации о границах функций.

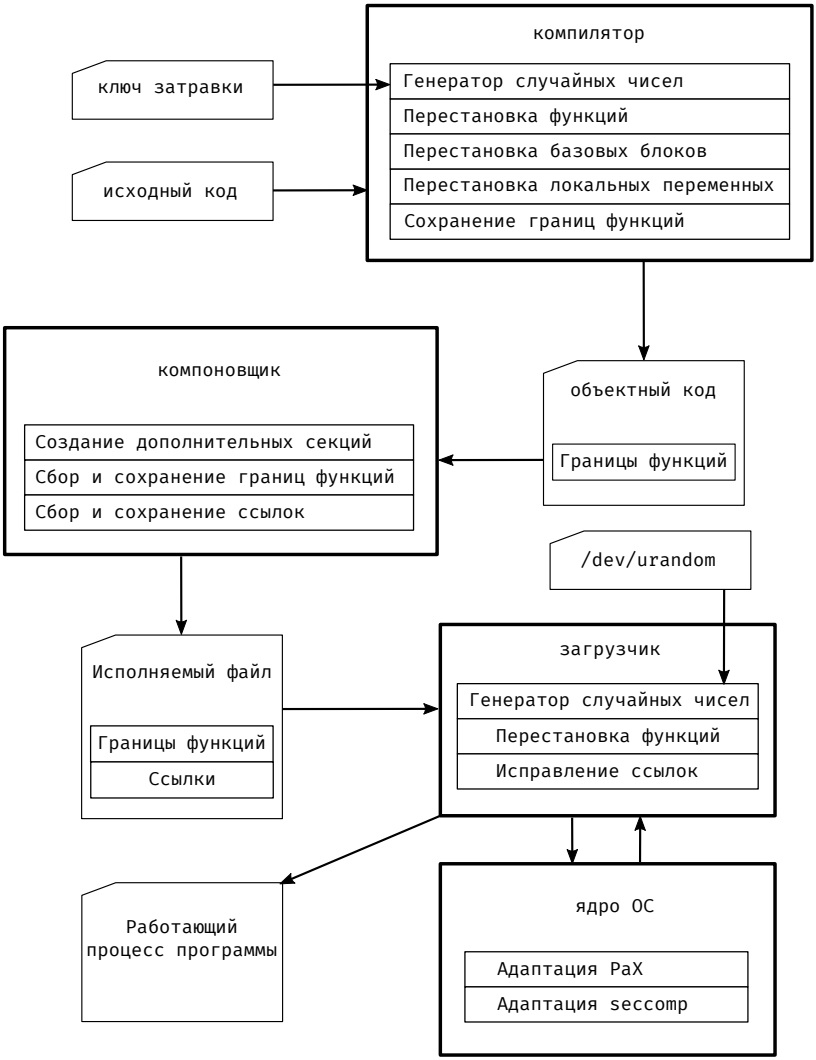

Рис. 8 — Архитектура реализованной системы дополнительной защиты.

В разделе 6.2 описывается реализованная дополнительная функциональность компоновщика ld, являющегося частью пакеты binutils. Дополнительная функциональность позволяет сохранять внутри исполняемого файла формата ELF информацию о границах функций и ссылках. Для этого создаются дополнительные секции `.reloc_infos`, `.note.reloc_infos`. В данном разделе описывается процедура их создания и заполнения.

В разделе 6.3 описывается реализованная дополнительная функциональность загрузчика ld.so, являющегося частью пакета gilbc. Для реализации мелкозернистой рандомизации в загрузчик была добавлена дополнительная функциональность: модуль генератора случайных чисел, функциональность поиска виртуального адреса секции `.reloc_infos`, функциональность чтения из дополнительной секции информации о границах функций и ссылка, функциональность перемешивания функций местами и их сдвига как целого, функциональность исправления ссылок, функциональность изменения и восстановления разрешения на доступ к региону памяти с кодом.

В разделе 6.4 описывается реализованная дополнительная функциональность ядра ОС Linux. Для совместимости имеющихся средств защиты с мелкозернистой рандомизацией произведена адаптация механизмов защиты PaX `mprotect` и `seccomp`. Для PaX `mprotect` добавлена функциональность идентификации работающей мелкозернистой рандомизации и разового разрешения вызова `mprotect` для разрешения записи в секцию кода. Для `seccomp` добавлена функциональность идентификации работающей мелкозернистой рандомизации, сохранения региона памяти загрузчика в структуру процесса, а также изменение функциональности `seccomp` таким образом, чтобы разрешать вызов `mprotect` с аргументом `PAGE_EXEC` для процессов с поддержкой мелкозернистой рандомизации.

В **заключении** перечисляются основные результаты работы:

1. Разработан метод диверсификации программного кода на уровне промежуточного представления компилятора для генерации диверсифицированной популяции исполняемых файлов.
2. Разработан метод диверсификации внутренней структуры исполняемого файла при загрузке его в память во время запуска приложения. Данный метод позволяет добиться рандомизации адресного пространства с зернистостью до функций.
3. Разработан метод оценки эффективности защиты от эксплуатации уязвимостей, который был применён для оценки эффективности мелкозернистой рандомизации адресного пространства и позволил исправить и улучшить её реализацию.
4. На основе предложенных методов реализована дополнительная система защиты для операционных систем семейства Linux, которая обладает приемлимыми характеристиками по ухудшению производительности программы, совместима с текущими средствами защиты для достижения взаимноусиливающего эффекта, применима в рамках целой ОС,

а также обеспечивает дополнительную защиту от атак повторного использования кода.

Публикации автора по теме диссертации

1. Реализация запутывающих преобразований в компиляторной инфраструктуре LLVM / В. Иванников [и др.] // Труды Института системного программирования РАН. — 2014. — Т. 26, № 1. — С. 327—342. — DOI: 10.15514/ISPRAS-2014-26(1)-12.
2. Применение компиляторных преобразований для противодействия эксплуатации уязвимостей программного обеспечения / А. Нурмухаметов [и др.] // Труды Института системного программирования РАН. — 2014. — Т. 26, № 3. — С. 113—126. — DOI: 10.15514/ISPRAS-2014-26(3)-6.
3. Application of Compiler Transformations Against Software Vulnerabilities Exploitation / A. R. Nurmukhametov [et al.] // Program. Comput. Softw. — New York, NY, USA, 2015. — July. — Vol. 41, no. 4. — Pp. 231–236. — DOI: 10.1134/S0361768815040052.
4. *Нурмухаметов А. Р.* Применение диверсифицирующих и обфусцирующих преобразований для изменения сигнатуры программного кода // Труды Института системного программирования РАН. — 2016. — Т. 28, № 5. — С. 93—104. — DOI: 10.15514/ISPRAS-2016-28(5)-5.
5. Мелкогранулярная рандомизация адресного пространства программы при запуске / А. Р. Нурмухаметов [и др.] // Труды Института системного программирования РАН. — 2017. — Т. 29, № 6. — С. 163—182. — DOI: 10.15514/ISPRAS-2017-29(6)-9.
6. Fine-Grained Address Space Layout Randomization on Program Load / A. R. Nurmukhametov [и др.] // Programming and Computer Software. — 2018. — Сент. — Т. 44, № 5. — С. 363—370. — DOI: 10.1134/S0361768818050080.
7. *Нурмухаметов А. Р.*, *Саргсян С. С.* Применение компиляторных преобразований для повышения стойкости программного обеспечения к эксплуатации уязвимостей // Сборник трудов XXI Международной научной конференции студентов, аспирантов и молодых учёных «Ломоносов-2014». — 2014.