ИНСТИТУТ СИСТЕМНОГО ПРОГРАММИРОВАНИЯ ИМ В.П.
ИВАННИКОВА РОССИЙИСКОЙ АКАДЕМИИ НАУК

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»

Диссертация на соискание учёной степени
кандидата технических наук

Научный руководитель:
к.ф.-м.н
Курмангалеев Шамиль Фаимович

Москва — 2020

Оглавление

Стр.

Введение

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. Ди-

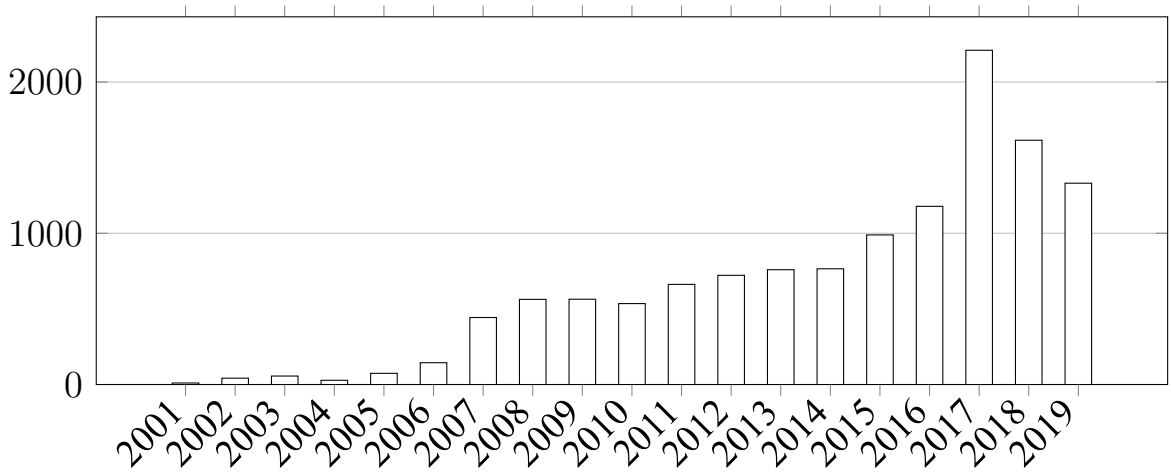

Рисунок 1 — Количество опубликованных уязвимостей типа CWE 119 (ошибки работы с границами массивов) согласно базе NIST.

версифицирующие 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 критических 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 соответ-

ствует 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 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.

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 рас-

пространять 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 возвратно-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.

Объем и структура работы. Диссертация состоит из введения, четырёх глав, заключения и двух приложений. Полный объём диссертации составляет 141 страницу с 25 рисунками и 6 таблицами. Список литературы содержит 72 наименования.

Глава 1. Обзор предметной области

В большинстве современных компаний обрабатывается и хранится много данных в цифровом формате. Острой проблемой является обеспечение их конфиденциальности, целостности и доступности. По этой причине бесспорной является важность компьютерной безопасности, которая может обеспечиваться различными механизмами. В данной работе изучается и развивается область компьютерной безопасности, направленная на противодействие эксплуатации уязвимостей программного обеспечения.

Уязвимость — представляет собой 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**.

Под противодействием эксплуатации уязвимостей подразумеваются меры направленные на усложнение и удорожание процедуры, с помощью которой злоумышленник нарушает компьютерную безопасность, а также меры, направленные на снижение потенциального ущерба от нее. Для достижения поставленной цели в данной работе развивается и используется подход, использующий диверсификацию программного кода.

Диверсификация кода — это процесс изменения внутренней структуры программы, направленный на получение большого количества различных ее копий, отличающихся друг от друга.

Существует большое количество типов уязвимостей и методов их эксплуатации. Далее в данном разделе приводится обзор существующих методов защиты и задается модельный вектор атаки, для противодействия которому будет предлагаться описываемый в данной работе метод защиты.

1.1 Поясняющий и мотивирующий пример

Приведем пример для наглядного пояснения определений, введенных выше, и введения новых (эксплойт, нагрузка). В листинге 1.1 приводится пример программы на языке Си, в которой имеется ошибка переполнения буфера на стеке. Язык Си не имеет автоматического контроля целостности памяти и вся ответственность по контролю корректности доступа к буферу лежит на разработчике. Так случается, что разработчики периодически делают ошибки при работе с буферами в памяти, что подтверждается статистикой по классификатору CWE-119 [8], приводимой базой уязвимостей [9] и приведенной на рисунке 1.

Листинг 1.1 Пример программы с ошибкой переполнения буфера

```
void vul(char *b) {
  int buf[10];
  memcpy(buf, b, strlen(b));
  return;
}

int main(int argc, char** argv) {
  int big_buffer[1000];
  vul(argv[1]);
  return 0;
}
```

В приведенном примере программы в функции `vul` происходит переполнение буфера `buf` данными, на которые указывает `b`. Переполнение происходит при условии того, что `strlen(b)` > 10. В результате переполнения буфера происходит перезапись данных на стеке, лежащих после буфера. В данном примере содержимое по указателю `b` контролируется входными данными, а именно первым аргументом строки команды запуска.

Стек функций устроен таким образом, что на нем вместе с локальными переменными функции помещается адрес возврата из функции. При этом адрес возврата лежит после буфера по ходу записи в него, что хорошо показано на рис. 1.1. Следовательно при выходе за границу буфера можно перезаписать, в том числе, адрес возврата. В случае если перезаписываемые данные поступают из входных данных, то злоумышленный пользователь может контролировать место в коде, куда вернется управление из функции.

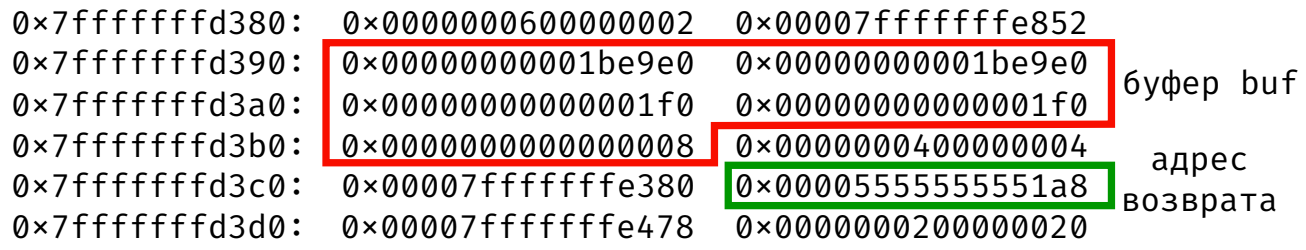

Рисунок 1.1 — Стековый кадр функции перед копированием буфера внутри функции `vul` модельного примера.

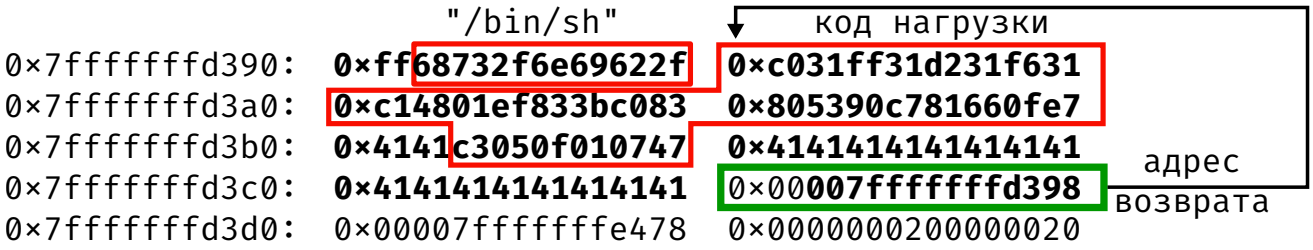

Рисунок 1.2 — Стековый кадр функции после переполнения буфера.

Не вдаваясь в детали процесса обнаружения уязвимостей, которые могут обнаруживаться средствами статического анализа кода или с помощью подхода случайного тестирования, приведем сценарий возможной эксплуатации уязвимости для примера 1.1 в следующих предположениях: архитектура набора команд процессора x86_64 с соглашением о вызовах System V (Linux), а также полное отсутствие защит от эксплуатации уязвимостей. После обнаружения потенциально эксплуатируемой ошибки переполнения буфера `buf` внутри функции `vul` с возможностью контроля данных буфера через параметр командной строки `argv[1]` возможно сформировать **эксплойт**. Эксплойтом называются входные данные, которые приводят к эксплуатации уязвимости. В данном примере его размер должен быть не меньше, чем 64 байта (размер буфера $4 * 10 = 40$ байт, 16 байт данных до адреса возврата, сам адрес возврата 8 байт, см. рис. 1.1). Первые 56 байт эксплойта могут быть почти любыми, а последние 8 должны содержать адрес, на который атакующий планирует передать управление. На самом деле, в примере 1.1 копирование буфера входных данных прервётся на первом нулевом символе (потому что копируется `strlen(b)` байт), поэтому атакующий не сможет использовать в своём эксплойте более одного нулевого символа.

Дополнительно к эксплойту (или внутрь него) может добавляться **код нагрузки** — входные данные, которые при эксплуатации интепретируются как код, который выполняет некоторую последовательность действий, интересных атаку-

Листинг 1.2 Код нагрузки для вызова системной оболочки

```
xor esi, esi
xor edx, edx
xor edi, edi
xor eax, eax
add eax, 0x3b
sub edi, 1
shl rdi, 15
add di, 0x5390
add byte ptr [rdi+7], 1
syscall ; execve("/bin/sh", 0, 0);
ret
```

ющему. Это может быть как вызов системной оболочки, так и чтение или запись некоторых данных, подготовку к выполнению следующего ступени многоуровневого эксплойта (открытие сетевого соединения, отключение защитных механизмов ОС и т.д.). В рассматриваемом примере будем использовать в качестве кода нагрузки вызов системной оболочки. Код нагрузки на языке ассемблера приведён на листинге 1.2. Код нагрузки в данном примере составлен таким образом, чтобы избежать использования тех инструкций, в машинном представлении которых содержатся нулевые байты.

На рис. 1.2 приводится кадр стека функции после переполнения буфера. Входной буфер, который выделен на рисунке жирным шрифтом, был сформирован следующим образом:

– строка `"/bin/sh\xff"`, байт `\xff` будет исправлен на нуль-терминатор в ходе выполнения кода нагрузки;
– код нагрузки для вызова системной оболочки (лист. 1.2);
– некоторое количество букв 'A' для того, чтобы довести размер буфера до 56 байт;
– байты, которые формируют на стеке адрес возврата `0x00007fffffffd398`, который передаст управление на лежащий на стеке код нагрузки для вызова системной оболочки.

При выходе из функции управление передастся на код нагрузки, который подготовит значения регистров для совершения системного вызова `execve("/bin/sh", 0, 0)`. Хорошо видно, что в описанном примере эксплуатация состоит из двух этапов: переполнение буфера, приводящее к передаче управления по контролируемому адресу; и исполнение кода нагрузки, выполня-

ющего вызов системной оболочки. На самом деле, два эти этапа достаточно независимы друг от друга. Эксплуатация может происходить не только путём переполнения буфера на стеке, но и другими способами (переполнение буфера на куче [10], форматная строка [11], целочисленное переполнение [12]). Код нагрузки может абсолютно произвольно сочетаться с совершенно различными методами эксплутации.

Описанная в приведённом примере процедура эксплуатации была широко распространена до внедрения средств защиты от эксплуатации уязвимостей. Характерной особенностью данного примера было внедрение кода нагрузки в память эксплуатируемого процесса. В данное время такой эксплуатации помешают следующие технологии: ограничение выполнения данных (DEP [13]), рандомизация размещения адресного пространства (ASLR [14]), протектор стека (stack canary [15]). Данные технологии были разработаны в качестве средства защиты и представляют собой реализации трех разных подходов соответственно: ограничивающий поведение, контролирующий целостность, основанный на непредсказуемости. В нижеследующих разделах будет объясняться принцип работы этих трех базовых средств защиты, их преимущества и недостатки.

1.2 Протектор стека

Протектор стека (англ. stack canary) — метод защиты от уязвимостей переполнения буфера на стеке, который позволяет частично контролировать целостность потока управления, а именно ребра возврата из функций. Протектор стека реализован в компиляторах, которые в прологе функции размещают специальное проверочное значение на стеке между локальными переменными и адресом возврата так, как показано на рис. 1.3. В эпилоге функции происходит проверка этого значения на изменение. Если произошло переполнение буфера, то проверочное значение изменится. В этом случае произойдет прерывание процесса исполнения программы и вызов обработчика исключительной ситуации.

Проверочное значение, располагаемое перед адресом возврата, составлено следующим образом: символы терминаторы функций копирования (нуль-терминатор, CR, LF, -1), а также случайные байты. Пример этого значения от-

```
0x7f..d380:  0x0000000600000002  0x00007ffffffe852
0x7f..d390:  0x00000000001be9e0  0x00000000001be9e0  буфер buf
0x7f..d3a0:  0x00000000000001f0  0x00000000000001f0
0x7f..d3b0:  0x0000000000000008  0xe98ad79478cacb00  протектор стека
0x7f..d3c0:  0x00007ffffffe380   0x00005555555551a8  адрес возврата
0x7f..d3d0:  0x00007ffffffe478   0x0000000200000020
```

Рисунок 1.3 — Стековый кадр функции с протектором стека

мечен на рис. 1.3 как протектор стека. Протектор стека был впервые предложен и реализован в работе Cowan [16]. Несмотря на дополнительные действия, производимые в начале и конце каждой функции, влияние на производительность оказывается незначительным, меньше 1 % по оценке Szekeres [17]. Кроме того, протектор стека не требует модификации исходного кода программы и не имеет проблем с совместимостью. Эти преимущества позволили протектору стека стать повсеместно используемой защитой от переполнения буфера на стеке. Все современные промышленные компиляторы имеют реализацию данной технологии и применяют ее по умолчанию при стандартной компиляции приложений.

Существует несколько способов обхода протектора стека. Во-первых, всегда существует вероятность угадать значения протектора стека, поэтому нельзя полностью исключать вариант попытки перебора проверочного значения. Кроме того, в зависимости от момента инициализации проверочного значения и природы уязвимого приложения пространство перебора может быть сильно ограничено. Такая ситуация описывается в работе Bittau [18], где повторно запускаемый процесс веб-сервера nginx имеет одно и тоже проверочное значение, которое быстро подбирается побайтовой перезаписью. Во-вторых, в некоторых операционных системах возможна утечка проверочного значения через контексты других процессов.

Рассмотрим эксплойт, приведённый в разделе 1.1, для примера 1.1 в присутствии протектора стека. При выходе за границу буфера произойдет перезапись проверочного значения `0xe98ad79478cacb00` на значение `0x4141414141414141`. Перед возвратом управления из функции произойдет проверка испорченного значения с эталонным, хранящимся в глобальной памяти. Поскольку эти значения не совпадают, то произойдет вызов обработчика исключительной ситуации, который прервёт исполнение программы с сообщением об испорченном стеке. Более того, в примере 1.1 даже точное знание проверочного

значения не позволит осуществить эксплуатацию, потому что проверочное значение содержит нулевой символ. Нуль-терминатор ограничит размер копируемых входных данных и адрес возврата останется нетронутым.

Листинг 1.3 Пример программы с ошибкой переполнения буфера, приводящей к перезаписи указателя на функцию

```
typedef void (*f_type)(void);
struct T {
  int buf[10];
  f_type foo;
};
void foo() { printf("foo called\n"); }
void vul(char *b) {
  struct T s = { {0}, foo };
  memcpy(s.buf, b, strlen(b));
  s.foo();
  return;
}
int main(int argc, char** argv) {
  int big_buffer[1000];
  vul(argv[1]);
  return 0;
}
```

Однако, переходя к примеру на листинге 1.3, стоит отметить, что протектор стека проверяет только адрес возврата. Он не предотвращает ни саму перезапись буфера, ни перезапись других данных, расположенных на стеке. Контролируя целостность потока управления на рёбрах возврата из функций, он не гарантирует целостность других передач управления. Например, в функции на стеке может лежать адрес функции, которая вызывается внутри этой функции. В случае, если при переполнении буфера происходит перезапись значения указателя на функции, то при вызове по этому указателю произойдёт перехват потока управления. Для наглядности на листинге 1.3 приводится пример программы с такой уязвимостью.

Эксплуатация такой уязвимости практически в точности повторяет эксплойт, описанный в предыдущем разделе. Единственной разностью будет лишь уменьшенный размер, поскольку перезаписать нужно лишь указатель, который лежит непосредственно за буфером. Кадр функции после переполнения изображён на рисунке 1.4, где жирным шрифтом выделен эксплойт. При вызове функции `s.foo` произойдёт передача управления на код нагрузки, расположенный на стеке внутри буфера `s.buf`. Код нагрузки аналогично предыдущему примеру вы-

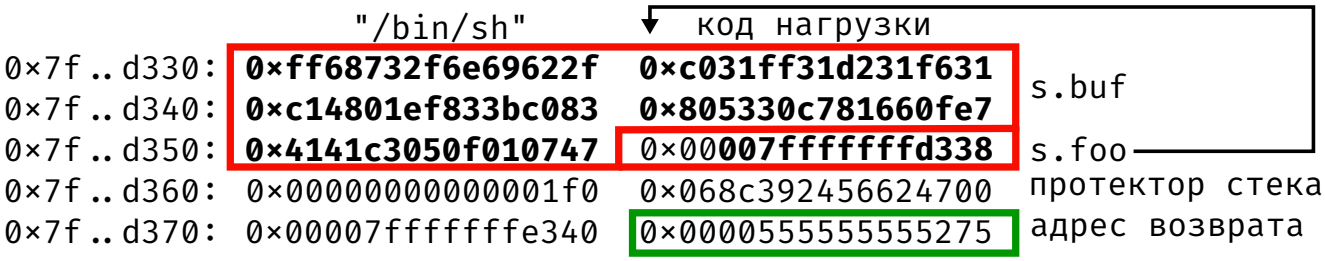

Рисунок 1.4 — Стековый кадр функции с протектором стека при эксплуатации переполнения буфера, приводящей к перезаписи указателя на функцию

зовет оболочку системного интерпретатора. В этом примере проверочное значение и адрес возврата остаются нетронутыми. Даже если бы эксплойт испортил их, то не помешал бы перехвату потока управления, поскольку проверка протектора стека срабатывает только перед выходом из функции, а управление перехватилось внутри функции.

1.3 Ограничение выполнения данных

В предыдущих разделах были приведены два примера эксплуатации уязвимостей модельных примеров 1.2, 1.4. В обоих случаях эксплуатация происходила с помощью внедрения в память работающего процесса (в частности на стек) кода нагрузки, который выполнял вызов оболочки системного интерпретатора.

Для предотвращения такого внедрения была придумана технология ограничение выполнения данных (DEP), также известная как W ⊕ X. Такая защита присутствует во всех современных операционных системах и опирается на аппаратные возможности современных процессоров (NX-бит). Данная технология позволяет операционной системе отображать все страницы памяти, содержащие данные программы (куча и стек), как недоступные для исполнения, а все страницы, содержащие код программы, как недоступные для записи.

DEP ограничивает возможность внедрения нового кода в память процесса. Действительно, при эксплуатации модельных примеров 1.1, 1.3 в момент передачи управления на код нагрузки, расположенный на стеке, произойдёт генерация исключения и прерывание исполнения программы, потому что стек недоступен для исполнения. При наличии DEP невозможно также дописать или изменить

имеющийся в памяти процесса код из-за недоступности соответствующих страниц на запись.

Стоит заметить, что данная технология не устраняет саму уязвимость, а предотвращает определённые сценарии ее эксплуатации с целью выполнения произвольного кода. Уязвимость остается внутри приложения и атакующий по-прежнему может использовать ее, например, для реализации атаки отказа в обслуживании (DoS). Таким образом, ограничение выполнения данных понижает опасность ряда уязвимостей с выполнения произвольного кода до отказа в обслуживании. DEP реализован таким образом, что имеет крайне незначительное влияние на производительность. Кроме того, для его поддержки не нужно внесения никаких изменений в программу со стороны разработчика. Перечисленные преимущества привели к тому, что в нынешний момент времени все современные операционные системы и большинство процессоров, используемых в высоко-производительных системах, поддерживают и по умолчанию используют DEP.

Повсеместное внедрение технологии ограничения выполнения данных привело к развитию методов эксплуатации, получивших общее название **атаки повторного использования кода**. Суть таких методов заключается в аккуратном использовании уже 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 [19]). Далее данные методы эксплуатации обсуждаются подробнее, а также приводится пример ROP эксплойта при наличии протектора стека и DEP.

1.4 Атака возврата в библиотеку

Атака методом возврата в библиотеку явилась исторически первым методом обхода защиты ограничения выполнения данных (DEP). Одно из первых упоминаний данного метода встречается в почтовой рассылке [20], наряду с другими модифицированными более поздними вариантами [21]. Из-за наличия DEP возможно использовать только имеющийся в памяти процесса код. Он может переписать адрес возврата на стеке адресом некоторой функции из памяти процесса. Следовательно, на нее передастся управление при выходе из текущей функции.

Кроме того, в случае с соглашением о вызовах `cdecl` данные, лежащие на стеке, будут интерпретироваться как аргументы вызываемой функции.

Как правило, в эксплуатации такого типа вызываются функции из стандартной библиотеки Си (`libc`). Именно функции из этой библиотеки являются популярными целями нагрузки по нескольким причинам: эта библиотека загружается вместе с почти каждой программой, написанной на языке Си; стандартная библиотека предоставляет программный интерфейс к большинству системных вызовов ядра операционной системы, таких как создание новых процессов, открытие сетевых соединений и прочее. Именно поэтому данный тип атаки получил свое название атака возврата в библиотеку.

```
0×ffffb810: 0×00000000  0×6e69622f "/bin/sh"
0×ffffb818: 0×0068732f  0×0804fcf0 → system
0×ffffb820: 0×41414141  0×ffffb814 → "/bin/sh"
```

Рисунок 1.5 — Часть стека функции после переполнения при атаке возврата в библиотеку.

На рисунке 1.5 показана часть кадра стека функции после переполнения во время эксплуатации методом возврата в библиотеку. Нагрузка устроена следующим образом: адрес возврата перезаписывается адресом функции `system = 0x0804fcf0` из `libc`; следом за ним кладется произвольный адрес возврата из функции `0x41414141`; далее идёт единственный аргумент функции, который является указателем на нуль-терминированную строку команды для запуска, размещенную следом на стеке, в данном примере это вызов оболочки системного интерпретатора `"/bin/sh"`. Более продвинутые и сложные варианты атаки возврата в библиотеку позволяют делать неограниченное количество вызовов функций [22].

1.5 Возвратно-ориентированное программирование

Возвратно-ориентированное программирование (англ. Return-Oriented Programming, 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. Возвратно-ориентированное программирование относится к типу атак, которые повторно используют код, расположенный в исполняемой памяти процесса. Последовательность инструкций, заканчивающаяся инструкцией возврата, называется **гаджет**.

Покажем логику действия возвратно-ориентированного программирования на примере. Предположим, что в адресном пространстве эксплуатируемой программы имеются следующие последовательности инструкций по указанным адресам:

```
0x5555555AAAA: pop ebx; ret
...
0x5555555BBBB: pop ecx; ret
...
0x5555555FFFF: mov [ecx], ebx; ret
```

Данные гаджеты позволяют создать код нагрузки, который позволяет записывать в заданное место памяти заданное значение. Например, с помощью таких гаджетов можно сохранить в секцию данных программы строку `"/bin/sh"`. Код нагрузки в формате ROP будет располагаться на стеке эксплуатируемой программы так, как показано на рисунке 1.6. Рассмотрим его работу в предположении, что адрес возврата находился на стеке по адресу `0x7fffffffd398`.

Действительно, адрес возврата перезаписан адресом `0x5555555BBBB` первого гаджета. При возврате из функции вызывается этот гаджет. Он загружает со стека значение `0x80031f630` в регистр `ebx`. Затем управление передаётся с помощью инструкции `ret` следующему гаджету, лежащему по адресу `0x5555555AAAA`. Второй гаджет загружает со стека значение `0x0068732f6e69622f` (строка `"/bin/sh"`) и передаёт управление третьему гаджету, который расположен по адресу `0x5555555FFFF`. Третий гаджет сохраняет значение регистра `ebx` по адресу, лежащему в регистре `ecx`.

На первый взгляд, может показаться, что термин код нагрузки некорректно применять в случае с ROP-эксплойтом. Действительно, ROP-эксплойт образован адресами возврата и значениями, которые загружаются на регистры. Все они являются данными. Однако, можно заметить, что при выполнении ROP-эксплойта регистр `rsp` выполняет роль виртуального счетчика команд. Адреса возврата являются кодами операций, а значения представляют собой закодированные внутри операции непосредственные константы. Вполне корректно говорить, что состояние памяти эксплуатируемого процесса, а точнее набор всех гад-

```
0×7f..d390: 0×4141414141414141  0×000005555555BBBB → pop ebx; ret
0×7f..d3a0: 0×0068732f6e69622f  0×000005555555AAAA → pop ecx; ret
0×7f..d3b0: 0×000000080031f630  0×000005555555FFFF → mov [ecx], ebx
                                                    → ret
```

Рисунок 1.6 — Код нагрузки в формате ROP, записывающий значение "/bin/sh" по адресу 0x80031f630 и расположенный на стеке эксплуатируемой программы

жетов в этом состоянии, задаёт некоторую виртуальную машину. ROP-эксплойт является программой для этой виртуальной машины. Поэтому вполне корректно называть ROP-эксплойт кодом нагрузки. Впервые явную аналогию между эксплойтом и программой для некоторой «странной» виртуальной машины сформулировал Dullien [23].

Возвратно-ориентированное программирование лучше всего изучено в литературе среди других методов повторного использования кода. Все началось с работы по адаптации атаки возврата в библиотеку под платформу x86_64 под авторством Krahmer [24]. Дело в том, что на данной платформе соглашение о вызовах отличается от x86: первые аргументы передаются через регистры, а не через стек. Данное различие вынудило искать в секциях кода инструкции (гаджеты) загрузки значений на регистры, чтобы передать аргументы в функцию. Однако Krahmer не придумал запоминающегося названия, такого как ROP, а называл свой подход «метод эксплуатации заимствованными кусками кода».

Логичным продолжением подхода Krahmer стало обобщение, проделанное в работе Shacham [19]. Именно в данной работе встречается в печати первое употребление термина ROP, который стал впоследствии повсеместно используемым. Shacham описывает алгоритмы поиска гаджетов в двоичном коде программы и приводит примеры набора гаджетов, который достаточен для выполнения произвольного кода (Тьюринг полный набор команд). Кроме того, он показал на примере стандартной библиотеки Си libc, что количество находящихся в ней гаджетов достаточно для построения Тьюринг-полного набора команд.

В последующее десятилетие было опубликовано множество работ, развивающих другие методы повторного использования кода. Некоторые из этих методов являются развитием возвратно-ориентированного программирования, другие же представляют альтернативные подходы. Среди разных методов стоит отметить

следующие: JOP [25], COOP [26; 27], BROP [18], DOP [28], DLOP [29], PCOP [30], FOP [31], PIROP [32].

Кроме того, к настоящему моменту сделаны значительные шаги в сторону автоматизации процесса создания ROP нагрузок для эксплойтов. Одной из самых значимых работ по данному направлению является статья Schwartz [33]. В ней описывается подход к разработке и реализации инструмента по автоматическому поиску гаджетов и созданию из них цепочек, заданной функциональности, так называемый ROP-компилятор. Авторы утверждают, что их компилятор способен генерировать ROP-нагрузки для 80 % приложений из Linux `/usr/bin`, размер которых больше 80 КБ. Они не выложили инструмент в открытый доступ, однако в последующие годы появились в открытом доступе инструменты схожей функциональности. Подробный обзор и описание инструментов и методов автоматизированной генерации эксплойтов методами повторного использования кода выходит за рамки тематики данной работы. Наиболее полный обзор подобных методов предоставлен в работе [34]. Однако использование таких инструментов полезно при оценке эффективности методов защиты, которая проводится для предлагаемого в диссертации метода защиты в главе 4. С другой стороны справедливо и обратное: развитие методов и инструментов для автоматической генерации ROP-цепочек увеличивает важность и необходимость исследования методов защиты от ROP-атак.

1.6 Пример эксплойта с возвратно-ориентированным программированием при наличии защиты от выполнения данных

Для демонстрации возможностей возвратно-ориентированного программирования можно обратиться к примеру такого эксплойта. Целевая программой для такого эксплойта приводится на листинге 1.4. Эта программа читает данные из файла и копирует некоторое их количество внутрь буфера в структуре, лежащей на стеке. Размер копируемых данных задаётся в первом байте файла. Кроме того, в структуре после буфера лежит указатель на функцию, которая вызывается после заполнения буфера. Модельный пример собирается с протектором стека и с защитой ограничения выполнения данных.

Листинг 1.4 Пример программы с ошибкой переполнения буфера для эксплуатации методом возвратно-ориентированного программирования

```
typedef void (*f_type)(void);
struct T { int buf[30]; f_type foo; };
void foo() { printf("foo called\n"); }
void vul(int fd) {
  struct T s = { {0}, foo };
  unsigned char size = 0;
  read(fd, &size, 1);
  read(fd, &s.buf, size);
  s.foo();
  return;
}
int main(int argc, char** argv) {
  vul(open("./input", O_RDONLY));
  return 0;
}
```

Типичная эксплуатация программ в таком случае разбивается на два ключевых этапа:

1. Этап с возвратно-ориентированным кодом нагрузки, его целью является отключение защиты, ограничивающей выполнение данных. На ОС Linux это делается с помощью вызова `mprotect`, в случае с ОС Windows вызывается `VirtualProtect`. Данные системные вызовы позволяют изменять разрешения (запись, чтение, выполнение) для заданных страниц виртуальной памяти.
2. Этап с основным кодом нагрузки. В условиях отключенной защиты выполнения данных он представляет собой обычный код нагрузки, описанный машинными инструкциями, сохранёнными на стеке. По аналогии с примером 1.2 он выполняет вызов оболочки системного интерпретатора.

По аналогии с примером эксплуатации при наличии протектора стека (рис. 1.4) перехват потока управления происходит путём перезаписи указателя функции `s.foo`. Однако, из-за наличия DEP нельзя передать управление непосредственно на код нагрузки, расположенный на стеке. Поэтому управление передаётся на код, который уже расположен в адресном пространстве программы. Такой код, как уже упоминалось выше называется гаджетом в том случае, если он оканчивается инструкцией возврата. Для текущего примера (стек после переполнения буфера изображён на рис. 1.7) управление передаётся на гаджет, который осуществляет сдвиг стека. Сдвиг стека необходим для того, чтобы передать

Рисунок 1.7 — Кадр стека с двумя этапа кодов нагрузки при эксплуатации модельного примера 1.4

управление на код нагрузки в формате возвратно-ориентированного программирования. Указатель стека после сдвига указывает на `0x7ffffffd2f0`, что является началом кода нагрузки первого этапа, которая совершает вызов `mprotect` отключения защиты выполнения данных для региона памяти, содержащего стек текущего процесса.

Код нагрузки первого этапа в формате возвратно-ориентированного программирования состоит из трёх гаджетов:

1. `pop rdi; ret` — загружает первый аргумент, задающий начальный адрес памяти для изменения прав доступа;
2. `pop rsi; ret` — загружает второй аргумент, задающий размер региона памяти права доступа которого будут изменены;
3. `pop rdx; pop rbx; ret` — загружает третий аргумент, задающий новые права доступа `PROT_READ|PROT_WRITE|PROT_EXEC`.

После подготовки аргументов происходит вызов функции `mprotect`. В качестве её адреса возврата указывается адрес начала второй части кода нагрузки `0x7ffffffd340`. Функция `mprotect` разрешает странице памяти, содержащей текущий стековый кадр, быть исполняемой. При возврате из неё управление передаётся на код нагрузки, который выполняет вызов оболочки системного интерпретатора.

Стоит отметить, что представленный эксплойт существенным образом опирается на конкретное расположение секций кода и стека в адресном пространстве программы. Метод защиты под названием **рандомизация размещения адресно-**

го пространства (ASLR) устраняет постоянность адресов, внося случайность в выбор базовых адресов этих секций.

1.7 Рандомизация размещения адресного пространства

Рандомизация размещения адресного пространства (англ. Address Space Layout Randomization, ASLR) — 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 имеющегося в памяти процесса кода. И более того, осложняет все типы атак, при которых атакующий полагается на неизменность положения элементов памяти процесса (код, стек, данные).

Например, пример эксплуатации из раздела 1.6 полагается на неизменность положения стека от запуска к запуску и на неизменность положения гаджетов. Действительно, используются адреса четырёх гаджетов, адрес функции `mprotect`, расположение стека. При наличии ASLR эксплуатация подобным образом будет работать только в тех случаях, когда атакующий угадает расположение всех этих элементов. Важно отметить, что рандомизация размещения адресного пространства предоставляет вероятностную защиту от эксплуатации уязвимостей.

Рандомизация размещения адресного пространства реализована следующим образом. Компилятор генерирует код приложения таким образом, чтобы он был *позиционно-независимым*. Это означает, что такой код может быть расположен по любому адресу с сохранением работоспособности и без необходимости проводить процесс разрешения релокаций. Далее динамический загрузчик/компоновщик операционной системы во время загрузки присваивает случайный базовый адрес секции кода исполняемого файла и динамическим библиотекам. Базовые адреса этих элементов изменяются при каждом запуске программы случайным образом.

Рандомизация размещения адресного пространства представляет собой типичный пример диверсифицирующего преобразования. Данный механизм позволяет сделать каждый запуск программы уникальным и непохожим на другие запуски как на других компьютерах, так и на одном и том же. Это ограничивает атакующему количество потенциально эксплуатируемых целей.

Одна из первых практических реализаций рандомизации размещения адресного пространства была предложена коллективом PaX [14]. В адресном пространстве приложения выделяется три области:

– область исполняемого файла (`exec`), включающая в себя секции кода, инициализированных и неинициализированных данных,
– область памяти отображаемой системными вызовами `mmap` (куча, динамические библиотеки и другие),
– пользовательский стек кадров функций (`stack`).

Адрес каждой области формировался с учетом случайного добавочного значения $\Delta_{exec}, \Delta_{mmap}, \Delta_{stack}$ соответственно для каждой области по следующим формулам:

$$\begin{aligned} exec' &= exec_{base} + \Delta_{exec}, \\ mmap' &= mmap_{base} + \Delta_{mmap}, \\ stack' &= stack_{base} + \Delta_{stack}. \end{aligned} \tag{1.1}$$

Случайность добавочного значения ограничена в текущих операционных системах значениями представленными в таблице 1, данные в этой таблице взяты из онлайн-документации Arch Linux [35]. Они получены с помощью утилиты `paxtest` [36]. В каждом из столбцов приведено количество бит, случайно выбираемых в базовом адресе соответствующей области памяти. Заметим, что количество случайно выбираемых бит ограничено для областей `mmap, exec` требованиями выравнивания страниц памяти (12 младших бит), а также необходимостью поддержки выделения больших областей памяти (4 старших бита) на 32-битной архитектуре.

Таблица 1 — Качество рандомизации размещения адресного пространства по областям

Ядро Linux	exec	mmap	stack
Linux 32-bit	8	13	19
Linux 32-bit hardened	18	22	24
Linux 64-bit	28	28	30
Linux 64-bit hardened	32	40	40

1.8 Недостатки рандомизации размещения адресного пространства и методы её обхода

Практически все современные операционные системы используют рандомизацию размещения адресного пространства (ASLR). Данная технология позволяет размещать код и данные по случайным адресам, что усложняет успешную эксплуатацию ошибок, но не исключает её возможность. В текущей главе приводится обзор недостатков ASLR и основных способов обхода ASLR, среди которых стоит отметить: подбор и перебор адресов 1.8.1; использование утечки информации о размещении кода программы или библиотеки в памяти процесса 1.8.2; использование для построения атаки нерандомизированных областей памяти 1.8.3; частичная перезапись указателей на код 1.8.4.

1.8.1 Перебор адресов областей памяти процесса

В работе Shacham [21] приводится пример атаки возврата в библиотеку на веб-сервер Apache в присутствии ASLR. Приводимый авторами эксплойт был устроен таким образом, что требовал подбора лишь одного адреса функции `system` из стандартной библиотеки `libc`. Адрес аргумента уже присутствовал на стеке в кадре предыдущей функции. Эксплуатация производилась для приложения запущенного на 32-битной архитектуре x86. Веб-приложение представляет собой процесс, который для каждого входящего соединения создаёт копию своего адресного пространства с помощью системного вызова `fork` (стандартные реализации ASLR не производят рандомизации размещения адресного простран-

ства в этот момент). По результатам экспериментов Shacham [21] выходит, что в среднем для успешной эксплуатации требовалось около 216 секунд. И действительно, согласно таблице 1 пространство перебора в случае 32-битных приложений ограничено. Для обычных ядер Linux требуется $2^7 = 128$, а для усиленных $2^{17} = 131072$. Такие небольшие пространства перебора исчерпываются при современных скоростях вычислительных машин в худшем случае за несколько минут. Shacham также показывает, что даже если бы при каждом вызове `fork` генерировалось новое размещение адресного пространства, то для эксплойта с подбором одного адреса время увеличилось бы всего лишь в два раза.

Однако, подбор адресов для такого эксплойта в случае 64-битной архитектуры потребует слишком большого времени. Согласно таблице 1 потребуется среднее количество попыток 2^{27} для успешной эксплуатации. При скорости перебора равной скорости перебора для 32-битной архитектуры это займёт несколько суток. Таким образом, рандомизация размещения адресного пространства на 32-битных архитектурах не обеспечивает надежной защиты от перебора. Недостаточность рандомизации для таких архитектур не потеряют своей актуальности до тех пор пока они будут использоваться, не стоит также забывать о мобильных устройствах, подавляющее большинство из которых в данный момент используют 32-битные архитектуры набора команд.

1.8.2 Утечка информации о размещении областей памяти процесса

Защита ASLR эффективна лишь в том случае, если составителю эксплойта неизвестно размещение адресного пространства. В случае, если каким-либо образом составитель эксплойта получает знание о местоположении стека или кода в адресном пространстве, то он использует это знание для составление работоспособного эксплойта. Тем самым, надежность защиты методом ASLR уязвима перед раскрытием адресного пространства. Раскрытие адресного пространства может происходить из-за непосредственной уязвимостей утечки данных об адресном пространстве или из-за наблюдения за побочными эффектами.

Для получения информации о размещении областей памяти процесса может быть использована уязвимость форматной строки как это было показано в

работах Scut [37] и Liu [38], Durden [39]. Специальные форматные строки позволяют читать данные со стека. В этих данных могут находиться адреса кода (адрес возврата, указатели на функции) и указатели на данные на стеке. По этим данным можно определить расположение кода и стека в адресном пространстве процесса. Кроме того, в некоторых случаях с помощью форматной строки возможно считать память по произвольному адресу.

Метод обхода ASLR с помощью наблюдения за побочными эффектами описывается в работе Евтюшкина [40]. Для реализации описанного метода требуется возможность запускать атакующему процессы с измерением времени работы. Атака использует буфер адресов перехода, который нужен для ускорения обработки переходов управления. В буфере создается коллизия адресов перехода между двумя процессами: один контролируется атакующим, другой подвергается атаке. На основании данной коллизии возможно раскрыть назначение перехода на основе изменения времени выполнения перехода. Данный способ позволяет раскрыть (иногда лишь частично) размещение адресного пространства атакуемого процесса. Характерное время раскрытия измеряется сотнями миллисекунд и не вызывает прерывания работы атакуемого процесса.

Достаточно часто случается так, что для обход защиты ASLR используется комбинация нескольких методов. К примеру, в работе Bittau [18] предлагается метод эксплуатации BROP, основанный на уязвимости переполнения буфера, при котором атакующий может переписывать произвольное количество байт на стеке у протектора стека и адреса возврата. Целевым приложением для эксплойта является веб-сервер, который повторно запускает обработчик для нового соединения без повторной рандомизации адресного пространства. Используя это, оказывается возможным подобрать примерно за 640 итераций адрес возврата из функции. Знание адреса возврата частично раскрывает память процесса, что позволяет проводить дальнейший подбор функций и гаджетов.

Кроме того, существуют подходы к раскрытию размещения адресного пространства ядра операционной системы. Данные методы работают в предположении наличия локального доступа к непривилегированному пользователю с возможностью запускать произвольный код. Они также используют недостаточно большую энтропию размещения кода ядра ОС, которая может содержать лишь 1024 различных слотов для ядра Linux [41]. К таким подходам относится работа Hund [42] и работа Gu [43].

1.8.3 Использование нерандомизированных областей памяти процесса

Рандомизация адресного пространства может быть эффективна лишь в том случае, если применяется ко всему адресному пространству. В случае, если в адресном пространстве остаются области кода по фиксированным адресам, то они могут быть использованы для конструирования эксплойта, повторно использующего этот код. Поэтому одним из самых распространённых методов обхода ASLR является подход повторного использования кода из нерандомизированных областей кода.

Одним из методов использования нерандомизированных областей для обхода ASLR является метод возврата в таблицу связывания процедур (англ. Procedure Linkage Table — PLT). Данный метод отличается от метода возврата в библиотеку лишь тем, что вместо адресов функций используются адреса ячеек в таблице связывания. Таблица связывания процедур содержит код заглушки для вызова каждой глобальной функции используемой библиотеки. Инструкция вызова в программе не вызывает напрямую функцию `system` из библиотеки. Вместо этого, она передает управление на код заглушки в таблице связывания процедур `system@PLT`. Этот код заглушки с помощью динамического компоновщика находит адрес функции и копирует его в соответствующую ячейку таблицы глобальных смещений (англ. Global Offset Table - GOT). Разрешение адреса функции происходит лениво, только один раз и только при первом вызове функции. В дальнейшем при каждом вызове `system@PLT` заглушка читает адрес функции из GOT и передает туда управление.

Таблица GOT и PLT не меняет свое местоположение в том случае, если сам исполняемый файл собран без поддержки позиционно-независимого кода (как пример отсутствует флаг `-pie` у компилятора `gcc`). Таким образом, в случае если библиотека размещена в адресном пространстве случайно, а исполняемый файл нет, то эксплойт может передавать управление на заглушки функций в таблице связывания процедур (PLT). Кроме того, существует возможность осуществлять вызов других функций (например `system`) методом переписывания ячеек таблицы глобальных смещений (GOT). Данный метод описывается в работе Roglia [44].

Метод переписывания ячеек таблицы GOT опирается на тот факт, что ASLR меняет только базовый адрес динамической библиотеки при загрузке в память,

при этом относительные смещения функций внутри библиотеки сохраняются постоянными. Данный факт позволяет атакующему поступать следующим образом. Находится функция, которая в эксплуатируемой программе вызывается до точки эксплуатации, т.е. ее адрес уже правильно посчитан и расположен в соответствующей ячейке GOT. Пусть это будет функция `printf`. Целевой функцией эксплойта является, как правило, функция `system`. Считается разница между адресами этих функций и добавляется к ячейке функции `printf` в таблице GOT. Данные операции производятся с помощью ROP-гаджетов. После перезаписи можно вернуться к процедуре возврата в PLT, либо считать из GOT правильный адрес и перейти по нему с помощью специального ROP-гаджета (например `jmp eax`).

Данные методы эксплуатации становятся неактуальными при применении рандомизации размещения адресного пространства для исполняемого файла, потому что секции PLT и GOT меняют свое местоположение. Однако в данный момент не все исполняемые файлы в популярных ОС собираются с поддержкой ASLR. Это хорошо видно по данным приведенным в работе Федотова [45]. Отчасти это объясняется тем, что данные изменения просто не были внесены в процедуру сборки соответствующих пакетов, а с другой стороны объясняется тем, что сборка приложений в виде позиционно-независимого кода отрицательно сказывается на производительности. По данным отчета Payer [46] среднее замедление на наборе тестов SPEC CPU® 2006 [47] для архитектуры x86 — 9,4 %, а для архитектуры x86_64 — 2,3 %.

1.8.4 Частичная перезапись указателей на код

В разных частях адресного пространства программы содержатся указатели на код, например адреса возврата на стеке, адреса библиотечных функций в таблице связывания, обычные указатели на функции как на стеке, так и в глобальной памяти или на куче. Каждый такой указатель содержит актуальный адрес функции в рандомизированном адресном пространстве. Технология ASLR устроена таким образом, что она не меняет младшие биты указателей. Таким образом, с помощью частичной перезаписи некоторого количества младших байтов, возможно передать управление на функцию, находящуюся на той же странице памяти с тем

местом, на которое указывал перезаписанный указатель. Частичная перезапись может осуществляться при помощи уязвимости записи за границу буфера или её обобщением под названием write-what-where [48].

Примером такого подхода является работа Ward [49], в которой описывается метод построения ROP-цепочек из гаджетов, лежащих внутри рандомизированных библиотек. Для предлагаемого метода существенно, чтобы базовый адрес исполняемого файла программы не рандомизировался. Следовательно, не рандомизируются базовые адреса таблиц PLT и GOT. Это позволяет авторам частично перезаписать значения указателей, хранящихся в этих таблицах. В конкретном примере, который приводят авторы, они ограничены параграфом памяти, внутрь которого указывал оригинальный указатель (из-за того, что уязвимость позволяет изменять лишь последний байт указателя). Данный факт сильно ограничивает авторов в доступном наборе гаджетов и заставляет использовать сложные гаджеты с нетривиальными побочными эффектами.

Другим примером атаки с частичной перезаписью указателей является работа Göktas [50]. Авторы представляют метод, который позволяет составить возвратно-ориентированный эксплойт в условиях присутствия DEP и полного ASLR (адрес загрузки исполняемого файла тоже рандомизируется). Главной идеей их подхода является, так называемый «массаж стека», идея которого заключается в том, что стек при выходе из функции не очищается. Следовательно, в памяти остаются значения предыдущих адресов возврата и, возможно, других указателей на код. Неинициализированные локальные значения позволяют избегать затирания значений из старых кадров стека. Как итог, авторы представляют метод, который позволяет с помощью аккуратно подобранных входных данных сформировать в области стека последовательность указателей на код, которые перемежаются местами с некоторыми данными. Далее при помощи уязвимости записи отдельного значения за пределы массива исправляются младшие байты указателей так, чтобы они указывали на ROP гаджеты. При необходимости таким же изменениям подвергаются данные на стеке (параметры гаджетов). В конце концов, происходит перехват управления и начинается исполнение ROP-эксплойта.

1.8.5 Недостатки рандомизации размещения адресного пространства

Резюмирую предыдущие разделы (1.8.1, 1.8.2, 1.8.3, 1.8.4) можно сделать вывод, что основными недостатками существующей реализации ASLR является: недостаточная энтропия на 32-битных архитектурах, изменение только базового адреса исполняемых модулей и библиотек, неполное применение ASLR ко всему адресному пространству процесса, опасность утечки адресов.

Изменение только базового адреса исполняемых модулей и библиотек приводит к тому, что весь код каждого отдельного модуля (динамической библиотеки, исполняемого файла или загрузчика) находится в памяти непрерывным куском подряд. При этом относительные смещения функций не меняются от запуска к запуску, и в точности соответствуют значениям, которые можно прочитать из расположенного на диске файла. Более того, почти все современные ОС, реализующие ASLR, не рандомизируют отдельно базовый адрес размещения каждой динамической библиотеки. Все библиотеки загружаются в память подряд, начиная со случайного адреса. Таким образом, даже утечка одного адреса приводит к раскрытию значительных областей памяти процесса. Изменение относительных смещений между кодом внутри модуля может исправить данный недостаток.

Недостаточная энтропия рандомизации размещения адресного пространства на 32-битных архитектурах приводит к тому, что перебор является эффективным сценарием атаки на ряд приложений. Данный недостаток может быть исправлен путём увеличения энтропии. Энтропия может быть повышена путем разнообразных диверсифицирующих преобразований программного кода.

Таким образом, применение диверсифицирующих преобразований, в том числе, изменение относительного порядка функций внутри модуля позволит устранить три из четырёх перечисленных недостатков.

1.9 Мелкозернистая рандомизация размещения адресного пространства

Одним из подходов преодоления недостатков, описанных в главе 1.8, является мелкозернистая рандомизация размещения адресного пространства. Ключе-

вым отличием от обычной рандомизации адресного пространства является уменьшение областей памяти, базовый адрес которых изменяется случайно, и изменение их относительных смещений.

При наличии технологии DEP (ограничения выполнения данных) для составления эксплойта может быть использован только существующий в памяти процесса код. Для этого нужно знать, где он расположен. Например, для построения ROP-цепочки нужно знать адреса как минимум нескольких гаджетов. Изменение относительных смещений внутри кода позволяет предотвратить вычисление адресов всех гаджетов по одному утекшему адресу. При составлении эксплойта возникнет необходимость подбирать все адреса отдельно. В случае, если перебор адресов гаджетов возможен без нарушения работоспособности процесса, то такой перебор потребует больше времени.

Подходящим способом уменьшения зернистости рандомизации размещения адресного пространства является *диверсификация*. Диверсификация может производиться разными способами в разные моменты времени жизненного цикла программного обеспечения. В данном разделе будут рассматриваться способы мелкозернистой рандомизации при запуске программы, представленные в литературе.

1.9.1 Рандомизация при запуске программы

Selfrando

В проекте Selfrando, описанном в работе Conti [51], используется подход наиболее близкий к развиваемому в главе 3. Авторы ставят перед собой цель — разработать практически применимый подход рандомизации размещения адресного пространства, который бы увеличивал энтропию рандомизации по сравнению с ASLR и обладал устойчивостью к утечке адресов. Практическим применением разработанного инструмента является усиления защиты Tor браузера.

Для осуществления рандомизации на уровне функций Selfrando собирает информацию о границах функций в исполняемых файлах во время статическо-

го связывания. Для этого используется обертка над системным компоновщиком. Кроме того, данная обертка собирает информацию о местоположении всех ссылок на код в программе. Вся эта информация вместе с библиотекой RandoLib записывается внутрь исполняемого файла. Точка входа меняется функцию из RandoLib, которая отвечает за процесс перемешивания местами функций.

Процесс перемешивания функций при запуске программы начинается с генерации случайной перестановки всех имеющихся в программе функций. Затем функции располагаются в памяти согласно этому порядку и происходит исправление ссылок на код по информации, сохраненной с этапа статической компоновки. После завершения этого процесса освобождается и зануляется область памяти, содержавшая библиотеку RandoLib и информацию о границах функций. Последним шагом является передача управления на действительную точку входа в программу.

Авторам пришлось столкнуться с некоторыми техническими сложностями в процессе реализации. Некоторые Си++ функции или инструкции ассемблера имеют специфические требования на выравнивание. Для того, чтобы избежать проблем с этим, авторы использовали заведомо большее выравнивание, что незначительно сказалось на увеличении потребляемой памяти (около 0,3 %). Другой проблемой явилась раскрутка стека, происходящая при обработке исключений и выводе стека вызовов при отладке. Современные компиляторы опускают генерацию указателя кадра стека (–fomit-frame-pointer) и вместо этого создают специальную метаинформацию для идентификации кадра. Авторам пришлось изменять ее соответствующим образом в процессе перемешивания при запуске программы. Последняя описанная авторами проблема затрагивает совместимость с компиляторными проверками ошибок обращения с адресами и памятью (AddressSanitizer). Для поддержки вывода человекочитаемой трассы ошибки компилятор создает специальную таблицу соответствия символьных имен функций с адресами в памяти. Авторам также пришлось вносить изменения в эту карту в процессе загрузки.

Экспериментальная проверка качества защиты на конкретных примерах CVE прошла успешно. Авторы оценивают увеличение энтропии по сравнению с ASLR как значительное. 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 случае обычной ASLR. Увеличение времени

загрузки Tor браузера оказалось незначительным, около 350 мс и составило 2,4 с всего в среднем. Уменьшение производительности измерялось на наборе тестов SPEC CPU® 2006 и составило 0,71 % при увеличении потребляемой памяти на 0,2 %.

Стоит отметить, что сбор информации о функциях и релокациях извне компоновщика является сложной задачей, вследствие чего Selfrando не в полной мере поддерживает TLS (Thread Local Storage). Включение кода для рандомизации в сам исполняемый файл не только дополнительно увеличивает его размер, но ещё и мешает работе других средств защиты. Исполняемый файл с Selfrando с точки зрения антивирусных средства защиты трудно отличим от самомодифицирующихся вредоносных программ. Итого, подход Selfrando рассчитан на упрощение сборки отдельных приложений с поддержкой рандомизации, но плохо применим в масштабах всей операционной системы сразу.

XIFER

Проект XIFER, описанный в работе Davi [52], представляет собой попытку реализовать идеальный мелко-зернистый рандомизатор адресного пространства. Авторы выдвигают ряд требований по свойствам инструментов такого рода:

- эффективность противодействия атакам переиспользования кода;
- энтропия рандомизации, которая должна быть достаточной для того, чтобы сделать атаку методом перебора невыполнимой на практике;
- устойчивость рандомизации к утечке какого-то одного адреса, т.е. адресное пространство процесса не должно раскрываться по одному адресу;
- частота рандомизации;
- требование к наличию дополнительной входной информации (исходный код, информация о релокациях, отладочная информация);
- покрытие кода, т.е. какая часть адресного пространства рандомизируется;
- совместимость с созданием подписи для программы;
- производительность;
- потребление памяти и диска;
- поддержка рандомизации библиотек;

– целевые архитектуры.

Перечисленные свойства, действительно, могут быть полезными при сравнении аналогичных решений.

Инструмент XIFER представляет собой прототип инструмента бинарного переписывания. Он подгружается в память запускаемого процесса с помощью LD_PRELOAD и подменяет собой все вызовы на те функции, с помощью которых происходит загрузка и динамическое связывание исполняемый файлов и библиотек. При обработки загрузки каждого модуля происходит дизассемблирование с использованием информации о релокациях. Во время дизассемблирования строится граф связей между кодом и данными. С помощью него после разбиения кода на куски и их перемешивания восстанавливаются связи между инструкциями и данными, и поправляются инструкции передачи управления между разными кусками кода.

Разбиение модуля происходит на несколько частей. Их количество вообще говоря задается пользователем, но авторы проводят исследование на предмет необходимости разбиения модулей на совсем небольшие куски с двух точек зрения: падения производительности из-за нарушения локальности и увеличения количества промахов кэша, и с точки зрения энтропии. Их вывод заключается в том, что разбиение на куски меньше, чем 6 инструкций крайне нежелательно, поскольку снижает производительность катастрофически. Компромиссным вариантом между производительностью и энтропией, по мнения авторов, является разбиение на 13 частей для 32-битной архитектуры и 17 частей для 64-битной архитектуры.

Во время загрузки после разбиения происходит перемешивание кусков в памяти и существует опциональный шаг сохранения произведенных преобразований обратно в файл. Стоит отметить, что перемешиванию подвергаются все секции модуля, в том числе и секции данных. После этого происходит выгрузка из памяти процесса самого инструмента и управление передается на входную точку программы. В случае, если необходима отладка рандомизируемого приложения, то поддерживается сохранение актуальной отладочной информации в отдельный файл и ключ, позволяющий воспроизводить процесс рандомизации.

Влияние на производительность оценивалось авторами на SPEC CPU® 2006 и составляет примерно 5 %. Увеличение потребления памяти оценивается авторами в 5 %. Инструмент поддерживает два режима работы с библиотеками: с разде-

лением 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 недоверенный процесс ОС.

Oxymoron

На основе фреймворка бинарного переписывания был разработан проект Oxymoron, описанный в работе Backes [53]. Целью этого проекта было совместить мелкозернистую рандомизацию с возможностью разделения памяти. Прототип инструмента был реализован с помощью фреймворка XIFER.

Для достижения поставленной цели авторами было предложено новое соглашение о вызовах, названное ими PALACE (англ. Position-and-Layout-Agnostic Code — позиционно и размещено независимый код). Согласно ему все ссылки на код и данные, которые на данный момент содержат абсолютные или относительные адреса, заменяются на особые метки, которые разрешаются в действительные адреса дополнительным уровнем абстракции. Для этого добавляется специальная таблица RaTTle, которая отвечает за разрешение этого уровня абстракции. Схематично процедура косвенного вызова с помощью таблицы RaTTle изображена на рис. 1.8. Данная таблица вместе с процессом трансляции меток должна удовлетворять следующим условия: таблица не должна занимать много места; процесс трансляции меток в адреса должен быть быстрым и недоступным для атакующего; работать на современных немодифицированных ОС Linux.

Процедура работы Oxymoron делится на три части:

– трансформация кода,
– разбиение кода,
– рандомизация размещения.

Процесс трансформации кода в PALACE код может происходить в разные моменты времени жизненного цикла программы. Возможны три следующих принципиальных варианта: время компиляции, статическая трансляция до запуска, трансформация во время загрузки программы. Первый подход требует перес-

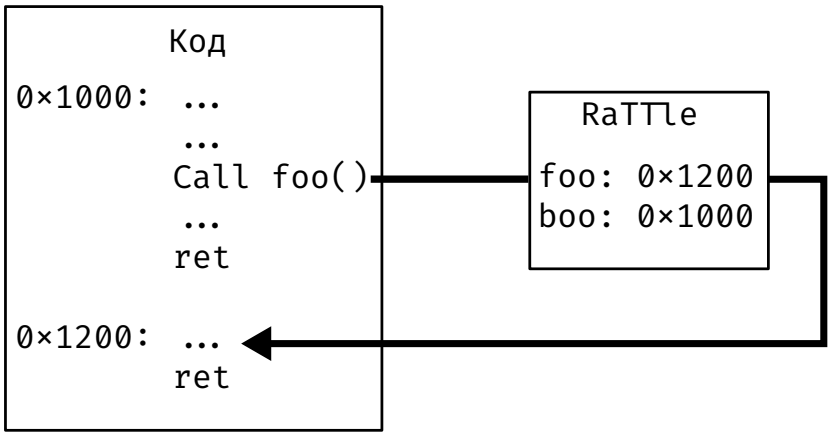

Рисунок 1.8 — Схема осуществления вызовов PALACE через таблицу Rattle

борки приложения и наличия исходного кода приложения. В случае если исходный код недоступен, то остаются варианты статической или динамической работы с бинарным файлом приложения.

Трансформированный PALACE код разбивается на куски размером в страницу памяти. Данные куски перемешиваются случайным образом во время загрузки с присвоением каждому своего случайного адреса в пространстве программы. PALACE код устроен таким образом, что требуется только изменение RaTTle таблицы, которое происходит с использованием стандартного механизма разрешения релокаций динамическим загрузчиком.

Расположение таблицы RaTTle должно быть неизвестно и недоступно атакующему. Для реализации этого авторы используют возможности сегментной адресации памяти процессоров семейства x86. Косвенный вызов через таблицу RaTTle реализуется на этой архитектуре с помощью одной инструкции `call \%fs: *0x4` (AT&T синтакс). В регистре `fs` хранится адрес таблицы RaTTle. Из этой таблицы берётся значение по смещению `0x4`, на которое передаётся управление.

Описанный подход позволяет разделять содержимое отдельной страницы, поскольку ее содержание не зависит от ее расположения в памяти. Разделение позволяет снизить расходы оперативной памяти для всех работающих процессов. Увеличение потребления памяти на отдельный процесс авторы оценивают в 15 %. Падение производительности измерялось на наборе тестов SPEC CPU® 2006 и составило 2,7 %. Авторы измеряли влияние преобразования на частоту промахов кэша и не обнаружили значимого влияния. Размеров файлов на диске увеличивается в среднем на 1,8 %.

pagerando

Проект pagerando, описанный в работе Crane [54], реализует подход, похожий на Oxymoron. Однако, он реализован внутри компилятора и компоновщика. В ходе компиляции код приложения разбивается на куски размером в страницу памяти, при этом по необходимости функции переставляются таким образом, чтобы уменьшить количество пустого места в конце каждой страницы. Данный метод был реализован в компиляторе LLVM и компоновщике gold для операционной системы Android 6, архитектура ARM.

Внутри страницы передача управления осуществляется обычным образом. Для разрешения межстраничных переходов был добавлен уровень неявной абстракции в виде таблицы PGLT (англ. page linkage table - таблица связывания страниц). При межстраничном вызове происходит переход в PGLT, в котором специальным образом хранится и автоматически разрешается при загрузке с помощью релокаций целевой адрес другой страницы.

Адрес таблицы PGLT хранится в регистре r9, который больше ни для чего не используется. Это отрицательно сказывается на производительности. По результатам измерений на наборе тестов Vellamo падение производительности составило 6,5 %, а увеличение потребления памяти оценивается в 19 %.

1.9.2 Рандомизация во время работы

TASR

В работе Bigelow [55] описывается подход к рандомизации адресного пространства программы периодически во время работы. Авторы предлагают проводить рандомизацию после каждого системного вызова, отвечающего за вывод информации из контекста процесса, и перед каждым системным вызовым, отвечающим за ввод информации в контекст процесса, а также перед каждым системным вызовом `fork`. Это позволяет снизить до минимума временное окно утечки ин-

формации о размещении адресного пространства. Любая информация, полученная атакующим от программы при помощи некоторой уязвимости, раскрывающей адресное пространство, перестает быть актуальной к моменту ввода от атакующего в память процесса любых данных.

Авторы ограничиваются только перемещением кода во время работы. Указателей на данные на порядок больше, чем указателей на код, отслеживание всех их привело бы к сильному уменьшению производительности. Тем не менее задача отслеживания во время работы всех указателей на код не является простой. Авторы для ее упрощения используют специальную отладочную информацию об указателях на код, сохраненную с момента компиляции в специальную секцию. Модуль ядра следит за процессами, которые запущены с поддержкой рандомизации во время работы, и по наступлении событий ввода-вывода запускает процесс перемещения кода.

Функциональность, отвечающая за перемещение кода и исправление ссылок на код, находится в отдельной библиотеке, которая подгружается ядром в память пользовательского процесса. Она перемещает код в новое место и, используя отладочную информацию, исправляет все устаревшие ссылки на код. Также требуется поправлять адреса возврата на стеке, для чего используется библиотека `libunwind`.

Данный подход имеет ряд ограничений и проблем с совместимостью с нынешней кодовой базой, среди которых: любые странные приведения типов указателей на функции, которые не в состоянии отследить TASR; встроенные ассемблерные инструкции будут требовать ручных аннотаций; проблема возникает и с интерпретаторами (perl и др.); кроме того динамическое выделение памяти под указатели на функции без точного указания их типа (использования `sizeof`) и поддержка пользовательских аллокаторов. Авторы обходят стороной вопрос совместимости их подхода с Си++, тестируя свой инструмент только на тестах из SPEC CPU® 2006, написанных на Си.

Производительность оценивалась авторами на части тестов из набора SPEC CPU® 2006, написанных только на языке Си. Авторы утверждают, что их метод дает только 2 % падения производительности в среднем. Однако они сравниваются с версией приложении, собранной с опцией `-Og`, с которой приложение работает в среднем на треть медленее, чем с общеиспользуемой `-O2`. SPEC CPU® 2006 содержит тесты, которые в основном нагружают процессор, то-

гда как операций ввода-вывода в них содержится немного. По идее метод авторов может замедлять существенно по-другому приложения, которые часто вызывают системные вызовы, вызывающие рандомизацию. Кроме того данный метод подвержен уязвимости частичной перезаписи, потому что относительные смещения кода внутри перемещаемых модулей остаются постоянными.

Shuffler

Другой подход использует проект Shuffler, описанный в работе Williams-King [56]. Ключевой особенностью их работы является изменение семантики указателя. До запуска программы они, используя инструмент бинарного переписывания исполняемых файлов, изменяют в программе все указатели на индексы, по которым в специальной таблице хранятся действительные значения указателей на код. Местоположение данной таблицы неизвестно атакующему. Рандомизация кода производится на уровне функций.

В отдельном процессе параллельно и асинхронно основной программе работает библиотека, отвечающая за перемещение кода на новое местоположение и исправление значений указателей в данной таблице. Изменения самого кода не требуется из-за замены указателей на индексы в таблице. С некоторым интервалом времени исполнение программы прерывается и управление передается со старой версии кода на новую перемешанную по-другому.

Адреса возврата, хранящиеся на стеке, не записываются в эту таблицу. Вместо этого они шифруются операцией XOR с некоторым ключом, который у каждого потока на каждую функцию уникальный и меняется при каждом перемещении. Перед передачей управления запускается синхронно процесс их исправления. Для этого стек разворачивается по кадрам обратно и происходит последовательное исправление всех адресов возврата. Для этого требуется наличие отладочной информации в исполняемом файле.

Все прямые вызовы функций, в том числе те, которые происходят через таблицу связывания процедур (PLT), преобразуются в относительные переходы. Таким образом программа по сути статически связывается с библиотеками в момент

запуска. Данная операция устраняет источник утечки адресов в виде глобальной таблицы смещений (GOT), но не совместимо с использованием `dlopen`.

Инструмент Shuffler не требует доступа к исходному коду, но полагается на наличие в исполняемом файле информации о символах и релокациях. Бинарное переписывание семантики указателей порождает значительное количество специальных случаев, подробно описанных в статье [56], например практическую неотличимость ситуации когда `jmp [rax]` используется для хвостовой рекурсии и таблицы переходов в `PIC` режиме. Кроме того, в текущей реализации инструмент не поддерживает `dlopen` и Си++ исключения.

Предложенный авторами подход замедляет запуск программы за счет дизассемблирования и переписывания бинарных файлов. На наборе тестов SPEC CPU® 2006 замедление из-за запуска приложений составляет 8 %. Падение производительности при периоде рандомизации в 50 мс оценивается авторами в 15 % при незначительном увеличении размера исполняемых файлов.

Глава 2. Запутывание промежуточного представления компилятора для противодействия эксплуатации уязвимостей

2.1 Монокультурность популяции исполняемых файлов приложений

В настоящее время из-за особенностей процедуры создания и распространения готовых приложений пользователям сложилась ситуация однообразия исполняемых файлов. Для целей разработки и отладки программ такая ситуация действительно удобнее. Жизненный цикл прикладного программного обеспечения, написанного на языках Си/Си++ схематично устроен следующим образом:

1. стадия разработки (т.е. стадия написания нового и/или исправления старого кода), во время которой разработчики реализуют новую функциональность и исправляют известные ошибки;
2. стадия сборки исполняемых файлов из исходного кода;
3. стадия распространения готового приложения конечным пользователям;
4. стадия использования готового приложения конечными пользователями;

В процессе последней стадии идет также сбор информации об ошибках работы приложения и пожеланий пользователей о добавлении новых функциональных возможностей. Данная информация накапливается и служит новой отправной точкой для повторения вышеприведенного цикла. Каждый проход по этому циклу порождает новую версию приложения и маркируется уникальным идентификатором, который, как правило, называется версией приложения.

Каждая конкретная версия приложения многократно тиражируется пользователям в неизменном виде, что удобно разработчикам и пользователям по нескольким причинам:

1. пользователям удобно сообщать об ошибках, а разработчикам удобно принимать информацию об ошибках;
2. тривиально проверять целостность приложения с помощью контрольных сумм.

Однако идентичность исполняемых файлов приложения создает угрозу крупномасштабной эксплуатации в случае, если злоумышленник находит эксплуатируемую уязвимость. Специально подобранные входные данные позволяют эксплуатировать все используемые копии уязвимой версии приложения.

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. Внесение разнообразия в программу может быть реализовано на разных этапах: во время написания исходного кода, во время трансляции исходного кода в машинный код, во время статического связывания, во время запуска приложения и динамического связывания.

Внесение диверсификации на каждом из этапов имеет свои плюсы и минусы. Диверсификации во время написания исходного кода имеет наибольшую мощь, потому что ничем не ограничена. Например, разные команды разработчиков могут разрабатывать одно и тоже приложение, используя разные методики, технологии и языки программирования. Такая диверсификация реализовалась естественным образом на рынке популярных приложений. Действительно, существует несколько различных видов браузеров, проигрывателей видео и прочих типов программного обеспечения. Практически все эксплойты заточены под конкретное приложение и не работают для аналогов. Однако аналогичных приложений, решающих одну и ту же задачу, как правило не очень много, в первую очередь, из-за колоссальной стоимости их разработки. Главным минусом такого подхода является практическая невозможность создать большое количество уникальных приложений.

Возможности диверсификации на других этапах ограничиваются. Действительно, компилятор не может изменять или подменять другим алгоритм, написанный программистом. Во время запуска приложения утеряна часть информации об абстракциях высокого уровня. Однако применение диверсификации на этих уровнях обойдется меньшими тратами по сравнению с диверсификацией на уровне написания исходного кода.

Диверсифицирующие преобразования должны быть достаточно мощными для того, чтобы имелась возможность предоставить каждому пользователю уникальную копию программы. Для оценки необходимой мощности можно привести приблизительное количество загрузок самых популярных приложений в магазинах приложений. Для магазина приложений Android Google Play количество загрузок приложения Google Chrome превышает 10^6 (один миллиард) [57]. Конкретное количество загрузок точнее неизвестно. Однако, из приведенной статистики

загрузок и оценки численности людей на планете Земля [58], можно сделать вывод, что достаточная мощность диверсификации должна исчисляться миллиардами уникальных копий, если речь идет о приложениях для мобильных устройств. Для менее используемых типов операционных систем и приложений это оценка может быть меньше.

2.2 Диверсифицированная популяция исполняемых файлов

Компилятор является ключевой частью процесса разработки программного обеспечения, в том числе потому, что он транслирует программу, написанную на языке высокого уровня, в низкоуровневую программу, непосредственно исполняемую компьютером. Одним из свойств любого компилятора является детерминированность процесса трансляции. Это означает, что один и тот же файл исходного кода всегда транслируется в один и тот же исполняемый файл, конечно при условии того, что компиляция производится при одинаковом уровне оптимизации. Таким образом, складывается ситуация, при которой тысячи пользователей обладают идентичным программным обеспечением. Идентичность означает, что вся внутренняя структура одной копии программы в точности повторяется в тысячах других копий. Это облегчает злоумышленникам планирование широкомасштабных атак на информационные инфраструктуры.

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 исходном коде программы, стандартом языка или соглашениями о вызовах и общих интерфейсах. Таким образом, каждая версия программы ведет себя по-разному при попытке эксплуатировать ее неспецифицированное поведение (т.е. уязвимость).

Генерация диверсифицированной популяции исполняемых файлов для одного пакета исходного кода приводит к усложнению и удорожанию разработки эксплойтов. Разработчик, планирующий атаку, должен либо создать эксплойт для каждой копии, либо сделать его достаточно сложным для того, чтобы он мог

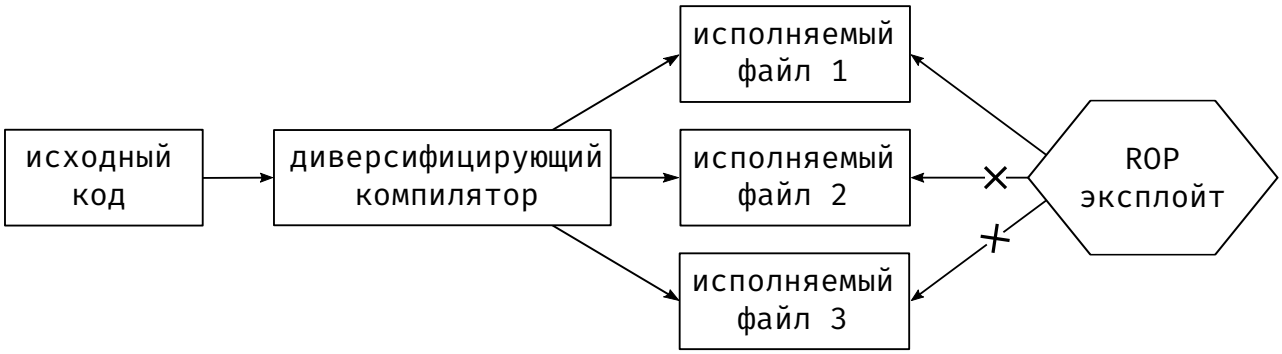

Рисунок 2.1 — Модель атаки на диверсифицированную популяцию исполняемых файлов.

успешно срабатывать сразу на нескольких копиях. В любом случае ему потребуется больше времени и ресурсов для планирования атаки и разработки эксплойта.

В современном мире большое количество программного обеспечения распространяется через магазины приложений различных платформ (Apple's App Store, Android Marketplace, Windows Market). В такой схеме распространения теоретически возможно осуществить автоматическую генерацию уникальной копии бинарного кода приложения для каждого пользователя предварительно или по запросу. В такой ситуации после загрузки из магазина приложений одной копии бинарного пакета атакующий может создать для неё эксплойт. Однако, этот эксплойт не будет работать для других копий бинарного пакета, которые предоставлены другим пользователям. Попытки атаки полученным эксплойтом на другие копии данного приложения будут неудачными, тем самым удастся избежать широкомасштабной эксплуатации. Схематично данная ситуация изображена на рис. 2.1.

Предлагаемый подход генерации диверсифицированной популяции приложений увеличивает стоимость распространения программного обеспечения конечным пользователям. Кроме того, в случае необходимости обязательной сертификации распространяемого программного обеспечения могут возникнуть проблемы из-за того, что все версии программного обеспечения будут требовать сертификации, что увеличивает и удлиняет ее стоимость. Данные факты могут серьезно ограничивать применимость данного подхода.

2.3 Компиляторная диверсификация

2.3.1 Источник случайности

Для генерации разнообразных программ компилятор должен иметь внутри себя источник случайности. При реализации описываемых методов использовался линейный конгруэнтный генератор случайных чисел, в котором последовательность псевдослучайных чисел генерируется рекуррентной формулой:

$$r_n = (Ar_c + C)\%M \tag{2.1}$$

где A, C, M — некоторые специальным образом подобранные константы, r_c — случайное число на текущей итерации генератора случайных чисел, r_n — случайное число на следующей итерации генератора случайных чисел. Последовательность генерируемых псевдослучайных чисел однозначно задается по начальной затравке, которая задается флагом компилятора. Таким образом, реализуется воспроизводимая случайность процесса компиляции, которая необходима для воспроизведения процесса компиляции и последующей отладки программ.

Такой генератор случайных чисел не предоставляет истинно-случаной последовательности чисел. Для того, чтобы избежать псевдослучайности, можно использовать истинный генератор случайных чисел, который бы был достаточно производительным, чтобы не замедлять значительным образом процесс компиляции. Современные технологии создания аппартных генераторов случайных чисел позволяют достигать скорости генерированния истинно-случайного потока данных в 21 Гбит/с, например фотонный генератор из работы Ugajin [59]. Кроме того, генератор случайных чисел или компилятор должен запоминать в таком случае последовательность сгенерированных чисел для того, чтобы была возможность воспроизводить компиляцию определённого бинарного файла для отладки.

2.3.2 Возможности для диверсификации исполняемых файлов в компиляторе

Целевыми языками при рассмотрении возможных направлений для диверсификации являются Си/Си++. В данном разделе приводится описание тех вещей в данных языках программирования, которые при недетерминированной трансляции в машинный код обладают диверсифицирующей мощью. Приводится оценка диверсифицирующей мощи каждого из преобразований.

В первую очередь остановимся на функциях данных языков программирования. Исходный код не определяет их местоположение в исполняемых файлах формата ELF. Тем самым, можно в ходе трансляции переставлять функции местами, получая различные исполняемые файлы. Мощность такой диверсификации вычисляется следующим образом:

$$\prod_{i=1}^{K} n_i!, \tag{2.2}$$

где n_i — количество функций в i единице трансляции исполняемого файла, состоящего из K модулей, при это общее количество функций равно:

$$N = \sum_{i=1}^{K} n_i. \tag{2.3}$$

Выполнение перестановки функций местами на этапе компоновки, когда доступны функции из всех единиц трансляций, обладает большей диверсифицирующей мощью, но может быть реализовано только на этапе компоновки. Диверсифицирующая мощь перестановки функций на этапе компоновки:

$$N!. \tag{2.4}$$

Дополнительного увеличения диверсифицирущей мощи можно добиться путем клонирования функций. Клонирование функций разрабатывалось и рассматривалось как один из способов обфускации программного кода в разработанном в ИСП РАН инструменте «ИСП Обфускатор» [60]. Наработки по преобразованию, создающему клоны функций может быть использовано в целях ди-

версификации. Особенно это актуально для исполняемых файлов с небольшим количеством функций.

Перестановка функций может негативно сказываться на производительности программ в том случае, если часто вызываемые функции разносятся далеко, нарушая тем самым локальность распределения кода.

Физический порядок следования базовых блоков в машинном коде может быть изменен, поскольку он никаким образом не специфицируется исходным текстом функции на языках Си/Си++. Данное преобразование негативно сказывается на производительности по причине добавления лишних инструкций передачи управления, поскольку до перемешивания базовых блоков они, как правило, распределяются компилятором таким образом, чтобы максимально задействовать механизм передачи управления без использования отдельной инструкции при последовательном расположении базовых блоков.

Диверсифицирующая мощь преобразования перестановки базовых блоков:

$$\prod_{i=1}^{n} b_i!, \tag{2.5}$$

где b_i — количество базовых блоков в функций с номером i.

Дополнительного увеличения диверсифицирующей мощи можно добиться путем размножения базовых блоков. Размножение базовых блоков рассматривалось в работах по обфусцирующему компилятору «ИСП Обфускатор» для увеличения сложности преобразования диспетчер и при проведении преобразования приведения графа потока управления к плоскому виду [60]. Особенно это актуально для функций, состоящих из небольшого количества базовых блоков.

Порядок следования локальных переменных на стеке функции может быть изменен, поскольку по стандарту программист не должен на него полагаться. Изменение следования локальных переменных на стеке функции, например, позволит изменить размер входных данных требуемый для перехвата потока управления при эксплуатации уязвимости переполнения буфера на стеке.

Диверсифицирующая мощь данного преобразования: $N!$, где N — количество локальных переменных функции. Для увеличения этого значения можно добавлять дополнительные локальные переменные в функцию.

Некоторые машинные инструкции следующие в программе друг за другом можно переставлять местами, если они не зависят по данным и управлению. Пу-

тем внесения изменений в планировщик команд можно добиться дополнительной диверсификации за счет перестановки местами инструкций, однако это отрицательно скажется на производительности программы, поскольку планировщик учитывает особенности производительности целевой машины. Диверсифицирующую мощь данного преобразования сложно оценить численно по причине того, что возможность перестановки инструкций сильно зависит от размера и характера кода конкретной функции. Для увеличения диверсифицирующей мощи преобразования перестановки инструкций можно добавлять в код функции бесполезные инструкции. Кроме того данное преобразование может и само по себе изменять относительные смещения внутри кода транслируемой программы. Под бесполезным кодом понимаются такие инструкции, которые выполняются, но не влияют на результат вычисляемый функцией.

Кроме изменения относительного положения имеющихся инструкций можно изменять сами инструкции. Основные подходы к изменению самих инструкций:

– замена инструкции на эквивалентную, например `mov eax, 0` на `xor eax, eax`;
– замена последовательности инструкций на эквивалентную, например `mov eax, ebx` на `push ebx; pop eax`;
– переименование регистров, компилятор распределяет внутри функции используемые регистры некоторым оптимальным образом, в общем случае возможно перераспределить регистры другим образом, тем самым получив другой код с другими смещениями;

2.4 Компиляторная инфраструктура GCC

Компилятор GCC является де-факто стандартом для сборки операционных систем таких как: основанные на Debian дистрибутивы, основанные на Red Hat дистрибутивы, BSD семейство и другие. Компиляторная инфраструктура GCC транслирует код, написанный на языках высокого уровня (Си/Си++ и другие), в машинный код. При этом поддерживается большое количество целевых архитек-

тур набора команд (более 40), включая все популярные и широко используемые (x86_64, ARM, MIPS, SPARC, PowerPC, AVR и многие другие).

Компиляторная инфраструктура логически делится на три части, так называемые: компилятор переднего плана, компилятор среднего плана, компилятор заднего плана. Компилятор переднего плана производит разбор текста исходного кода на токены и лексемы и строит абстрактное синтаксическое дерево. Данное дерево зависит от исходного языка программирования. По абстрактному синтаксическому дереву после некоторых анализов и преобразований строится машинно-независимое промежуточное представление, называемое GIMPLE. Над этим представлением производятся машинно-независимые оптимизации. Каждая оптимизация устроена как отдельный компиляторный проход, принимающий на вход промежуточное представление GIMPLE и генерирующий измененное промежуточное представление. После того, как все машинно-независимые оптимизации произведены, GIMPLE преобразуется в машинно-зависимое промежуточное представление RTL. Это представление более конкретно описывает транслируемую программу в терминах (типах и инструкциях), соответствующих целевой архитектуре набора команд. Машинно-зависимые оптимизации и преобразования производятся на промежуточном представлении RTL. В самом конце процесса трансляции инструкции RTL заменяются на машинные инструкции целевой архитектуры.

Большинство описываемых в данном разделе диверсифицирующих преобразований было реализовано на машинно-независимом промежуточном представлении GIMPLE. Оно представляет собой трехадресный код в форме единственного присваивания. Диверсифицирующие преобразования, реализованные на машинно-независимом уровне, применимы для всех целевых архитектур набора команд компиляторной инфраструктуры GCC. Кроме того, промежуточное представление GIMPLE не зависит от языка программирования, на котором написаны тексты исходных кодов. Поэтому диверсифицирующее преобразование, реализованное на этом уровне, применимо также ко всем языкам программирования, которые поддерживаются компиляторной инфраструктурой GCC, среди них: Си, Си++, Фортран, Ada, Objective C, Go, D. Реализация диверсифицирующих преобразований на GIMPLE расширяет их практическую применимость.

2.5 Компиляторная инфраструктура LLVM

LLVM представляет собой инфраструктуру для создания компиляторов и сопутствующих им инструментов. Данная инфраструктура строится вокруг промежуточного представления (байткод LLVM). Вокруг этого представления построены компиляторы переднего и заднего плана, а также системы оптимизации и интерпретации. Для LLVM существует большое количество компиляторов переднего плана, поддерживающих языки программирования: Си, Си++, D, Delphi, Fortran, Haskell, Julia, Lua, Objective C, Swift, Rust. При этом поддерживается широкий набор целевых архитектур набора команд: ARM, MIPS, x86, PowerPC, SPARC, RISC-V, даже WebAssembly.

Некоторые из преобразований, описываемых в данной статье, были изначально реализованы в разрабатываемом в ИСП РАН обфусцирующем компиляторе (Курмангалеев [60—62]), на базе LLVM, а позднее были перенесены в компилятор GCC. Данные преобразования были реализованы на машинно-независимом промежуточном представлении LLVM. Диверсифицирующие преобразования, реализованные на машинно-независимом уровне, применимы для всех целевых архитектур набора команд компиляторной инфраструктуры LLVM. Кроме того, машинно-независимый байткод LLVM не зависит от языка программирования, на котором написаны тексты исходных кодов. Поэтому диверсифицирующее преобразование, реализованное на этом уровне, применимо также ко всем языкам программирования, которые поддерживаются компиляторной инфраструктурой. Реализация диверсифицирующих преобразований на байткоде LLVM расширяет их практическую применимость и позволяет быстро получить первый работоспособный прототип.

2.6 Реализованные преобразования

Перестановка функций местами

Перестановка функций местами меняет структуру памяти процесса. Изменяется относительный порядок функций, меняются адреса точек входа. Данное преобразование особо полезно для противодействия ROP-атакам. При планировании таких атак важно знать местоположения гаджетов. Изменение местоположения функций в исполняемых файлах приводит к тому, что атакующий знает адреса гаджетов только в своей копии приложения. Разработчик эксплойта, подобрав нужную ему последовательность гаджетов для доступной ему копии приложения, столкнется с той проблемой, что созданный им эксплойт в виде ROP-цепочки не будет работать на других копиях приложения.

При выполнении перестановки функций местами (рис. 2.2) каждой функции в модуле сопоставляется случайное число, полученное от линейного конгруэнтного генератора случайных чисел. Функции переставляются в соответствии с порядком возрастания, присвоенных им случайных чисел. Количество уникальных перестановок функций местами, следовательно, и количество уникальных копий приложения оценивается формулой 2.2. Перестановка функций в компиляторе выполняется перед самым моментом записи ассемблерного кода в выходной файл. Перестановка функций может вступать в противоречие с оптимизациями, которые улучшают локальность распределения функций. Данные оптимизации могут использовать профиль и помогают экономить время при инициализации приложения и располагают рядом горячие функции, которые часто друг друга вызывают. Данные оптимизации выключаются при включении преобразования перестановки функций в модуле. Стоит заметить, что перестановка местами функций не меняет граф вызова программы и никаким образом не сказывается на корректности исполнения программы.

Перестановка функций, реализованная внутри компилятора, имеет недостаток, который заключается в том, что для перемешивания доступны только функции, содержащиеся в обрабатываемой единице трансляции (модуле). Преобразование перестановки функций местами, реализованное в компоновщике или во

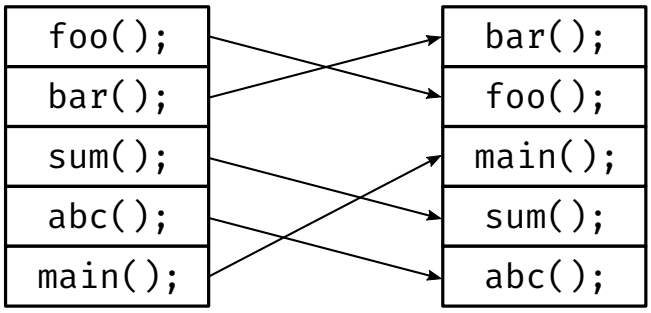

Рисунок 2.2 — Перестановка функций местами

время оптимизации времени связывания, позволяет достичь большей диверсифицирующей мощи для заданного приложения, как показано формулами 2.2, 2.4.

Перестановка функций в компоновщике может быть реализована с помощью скрипта-обертки над драйвером компилятора GCC и скрипта-обертки над статическим компоновщиком LD. В скрипте-обертке над компилятором GCC к оригинальной команде компиляции добавляется ключ командной строки `-ffunction-sections`. Данный ключ заставляет компилятор сохранять каждую функцию в отдельную секцию кода исполняемого файла. Скрипт-обертка над компоновщиком проходит по всем объектным файлам, подаваемым на вход, и переименовывает в них каждую секцию кода с формированием случайного суффикса. В конце концов, скрипт-обертка запускает оригинальный компоновщик с дополнительной опцией командной строки (`--sort-section=name`), которая заставляет отсортировать секции кода в выходном файле по имени.

Перестановка местами базовых блоков

Перестановка базовых блоков внутри функции реализована следующим образом. Компиляторная инфраструктура GCC имеет преобразование, которое оценивает горячие базовые блоки на основе динамического или статического профиля функции. Исходя из профиля и задачи минимизации инструкций передачи управления между базовыми блоками, происходит присваивание базовым блокам относительных весов. При формировании линейной последовательности базовых блоков для их записи в выходной файл данные веса учитываются при расположении базовых блоков. Присвоение относительных весов случайным образом приводит к перестановке базовых блоков в выходном файле местами. Естествен-

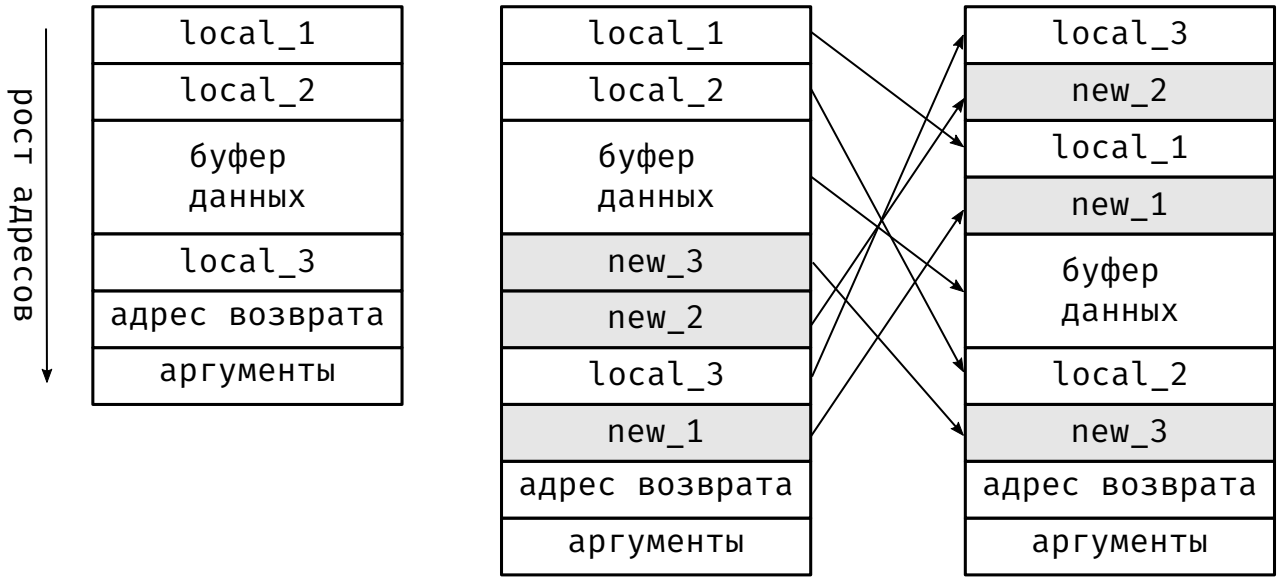

Рисунок 2.3 — Добавление локальных переменных и их перемешивание на стеке

ным образом получается, что оптимизации расположения базовых блоков перестают работать. Их отключение негативно сказывается на производительность программы и увеличивает ее размер за счет некоторого количества дополнительно вставленных инструкций перехода. Стоит заметить, что данное преобразование не изменяет графа потока управления, что обеспечивает консервативность данного преобразования с точки зрения корректности транслируемой программы.

Добавление и перемешивание локальных переменных

Для добавления локальных переменных на стек функции был реализован компиляторный проход над промежуточным представлением GIMPLE. Этот проход добавляет в каждую функцию заданное опцией командной строки количество локальных переменных, хранящихся на стеке рис. 2.3. Добавление локальных переменных происходит путем генерации в промежуточном представлении дополнительной инструкции, генерирующей локальную переменную. Данное преобразование было реализовано как в компиляторной инфраструктуре LLVM, так и в компиляторной инфраструктуре GCC.

Дополнительные инструкции генерируются только в промежуточном представлении для создания локальных переменных. Поскольку инициализации и использования добавленных переменных не генерируется, то и дополнительных ин-

струкций не добавляется. В ассемблерном коде выделение памяти на стеке изменится, и указатель стека будет сдвигаться на другую большую величину, а также изменятся смещения локальных используемых переменных относительно указателя на вершину стека. Из-за увеличения размера стекового увеличится потребление памяти программой. Однако в свете современного количества памяти увеличение потребления из-за нескольких переменных не выглядит значительным.

Перемешивание переменных на кадре стеке функции (рис. 2.3) реализовано аналогично перестановке функций местами. Каждой локальной переменной, находящейся на стеке функции, сопоставляется случайное число, полученное от линейного конгруэнтного генератора случайных чисел. Локальные переменные перемешиваются в соответствии с порядком возрастания присвоенных им случайных чисел. Количество уникальных перестановок кадра стека $N!$, — где N количество локальных переменных.

2.7 Влияние реализованных преобразований на производительность

Реализованные диверсифицирующие преобразования оказывают влияние на производительность. Как было показано в разделе 2.6, они могут влиять отрицательно на скорость выполнения компилируемых программ. Для количественной оценки влияния на производительность используется набор тестов SPEC CPU®. Он представляет собой набор приложений и инфраструктуру для их запуска, данный набор тестов является общепризнанным стандартом для измерения производительности. Для измерения производительности производится запуск каждого теста несколько раз на большом наборе данных для того, чтобы снизить влияние случайных факторов на результат. Время работы разных запусков усредняется, а затем вычисляется отношение полученного времени к времени запуска на референсной машине (для SPEC CPU® 2006 это рабочая станция Sun конца 90-х). Данное отношение и является результатом тестирования для конкретного теста в наборе. Это означает, что чем больше данное значение тем лучше. Суммарный результат по всем тестам вычисляет в SPEC CPU®как среднее геометрическое между результатами отдельных тестов.

Таблица 2 — Результаты влияние реализованных диверсифицирующих преобразований на производительность компилируемых программ

Название теста	base	peak	Время	Размер
400.perlbench	45,9	45,8	-0,2	0
401.bzip2	29,3	29,5	0,7	0
403.gcc	39,0	38,8	-0,5	1,0
429.mcf	41,5	41,2	-0,7	0
445.gobmk	30,8	30,6	-0,6	0,2
456.hmmer	29,6	29,5	-0,3	1,3
458.sjeng	33,4	33,2	-0,6	0
462.libquantum	76,4	77,2	1,0	0
464.h264ref	55,7	55,2	-0,9	0,7
471.omnetpp	23,8	23,6	-0,8	1,0
473.astar	24,1	23,7	-1,7	7,9
483.xalancbmk	43,2	42,5	-1,6	1,6
CINT	37,2	37,0	-0,5	-
410.bwaves	51,1	50,5	-1,2	0
416.gamess	39,3	39,4	0,3	0
433.milc	28,0	27,9	-0,4	2,9
434.zeusmp	32,9	33,1	0,6	0,7
435.gromacs	26,4	26,4	0,	0
436.cactusADM	27,0	26,9	-0,4	1,0
437.leslie3d	38,6	38,4	-0,5	0
444.namd	28,4	28,4	0,	0
447.dealII	52,0	51,0	-1,9	1,8
450.soplex	45,5	45,4	-0,2	1,7
453.povray	50,3	49,6	-1,4	0,7
454.calculix	15,2	15,2	0,	0,5
459.GemsFDTD	37,2	37,2	0,	0
465.tonto	26,2	26,1	-0,4	0
470.lbm	67,9	67,8	-0,1	0
481.wrf	31,7	31,6	-0,3	0
482.sphinx3	49,5	49,2	-0,6	0
CFP	35,9	35,7	-0,6	-

В таблице 2 представлены результаты тестирования на наборе тестов SPEC CPU® 2006. Данный набор тестов состоит из двух больших наборов тестов: тесты на вычисления с целочисленными значениями CINT, тесты на вычисления с числам с плавающей запятой CFP. В таблице представлены как результаты этих наборов, так и результаты отдельных тестов в наборах. Кроме того, приводится две дополнительных колонки. В первой указывается относительное изменение результата, а во второй указывается относительное изменение размера исполняемого файла.

По представленным результатам можно сделать вывод, что среднее падение производительности оценивается в 0,5 %. При этом ни для одного из тестовых приложений падение производительности не превосходило 2 %, тогда как для нескольких тестов наблюдалось незначительное увеличение производительности. 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 %.

Глава 3. Мелкозернистая рандомизация внутренней структуры программы при запуске

Прежде чем приступить к описанию реализации мелкозернистой рандомизации внутренней структуры программы при запуске, предоставим краткое описание формата исполняемого файла (ELF от англ. Executable and Linkable Format — формат исполнимых и компонуемых файлов) операционной системы CentOS. Детальное описание можно найти в следующих ресурсах [63—65]. Характерной особенностью формата ELF является двойственная природа представления файла. С одной стороны файл представлен как набор секций, которые используются компилятором, ассемблером и компоновщиком. С другой стороны файл представлен как набор сегментов, которые загружаются системным загрузчиком при запуске программы в память процесса.

Стандарт формата ELF описывает различные типы файлов формата ELF. Перемещаемый файл — содержит данные и инструкции, которые могут быть скомпонованы с другими перемещаемыми файлами. В результате такой компоновки могут получается разделяемый объектный файл или исполняемый файл. Файлы статических библиотек являются перемещаемыми файлами. Разделяемый объектный файл — содержит данные и инструкции. Он может быть скомпонован с другими перемещаемыми файлами, в результате чего получается разделяемый объектный файл. Кроме того, он может быть динамически связан с другим разделяемым объектным файлом или исполняемым файлом при загрузке. Исполняемый файл — содержит описание, достаточное для создания образа процесса и включающее в себя: инструкции, данные, список необходимых разделяемых объектных файлов и, если необходимо, то отладочную информацию.

Исходный текст программы на языке Си или Си++ транслируется компилятором в низкоуровневый язык машинных команд (ассемблерный код). Ассемблерный код далее переводится специальным транслятором (ассемблером) в набор машинных команд в бинарной форме, который организован как перемещаемый файл формата ELF. Такой файл далее поступает на вход статическому компоновщику, который статически связывает его с другими перемещаемыми файлами и динамически связывает его с разделяемыми объектными файлами библиотек (динамические библиотеки). В результате компоновки получается исполняемый файл или

разделяемый объектный файл библиотеки. Полученный исполняемый файл ELF может быть выполнен в рамках совместимой операционной системы.

При выполнении исполняемого файла формата ELF происходит следующая последовательность шагов:

1. ядро операционной системы создает новое адресное пространство для запускаемого исполняемого файла, первоначально пустое;
2. в него в первую очередь загружается динамический системный загрузчик;
3. управление передается загрузчику и дополнительно передается информация о том, какую программу нужно запустить и с какими параметрами;
4. загрузчик выделяет память для исполняемого файла и отображает туда содержимое файла;
5. затем загрузчик обходит динамические библиотеки, от которых зависит исполняемый файл;
6. для каждой библиотеки выполняется загрузка ее в память и динамическое связывание с исполняемым файлом;
7. после того как все зависимости успешно разрешены, то загрузчик передает управление на точку входа исполняемого файла.

Заметим, что в ходе компоновки происходит статическое связывание, т. е. заполняются конкретными значениями известные на момент компоновки ссылки на символы. Однако значения местоположение некоторых символов неизвестно в момент компоновки. Заполнение ссылок на такие символы производится динамическим загрузчиком в ходе запуска программы, с использованием информации доступной только во время запуска программы.

3.1 Схема предлагаемого метода

Для реализации рандомизации адресного пространства программы с зернистостью до уровня функций необходимо отложить заполнение ряда ссылок на символы с этапа статического связывания до этапа динамического связывания. Для реализации этого был предложен следующий подход:

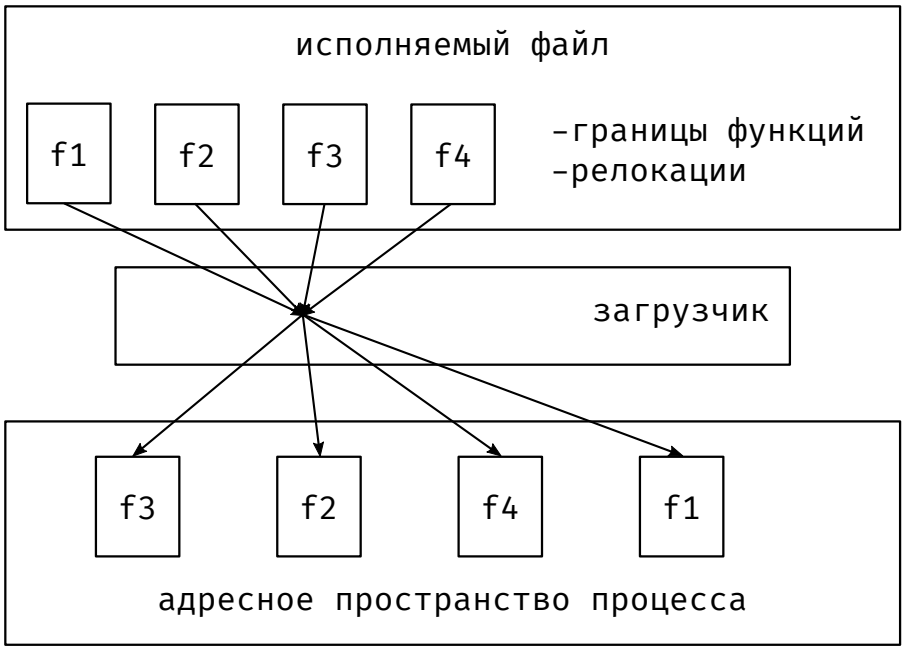

Рисунок 3.1 — Схема работы мелкозернистой рандомизации во время запуска программы. Динамический загрузчик с помощью информации, сохраненной в дополнительной секции, производит перестановку функций местами с целью изменения относительного порядка функций в адресном пространстве процесса.

1. во время трансляции исходного кода программы сохранять информацию о границах функций;
2. во время статической компоновки сохранять информацию о ссылках на символы;
3. во время загрузки и динамической компоновки выполнять последовательно:
   - перемешивание функций местами, используя информацию из первого пункта;
   - исправление значений ссылок на символы, используя информацию из второго пункта.

Концептуальная схема работы предлагаемого метода представлена на рисунке 3.1.

3.2 Реализация предлагаемого метода

Для реализации мелкозернистой рандомизации на этапе запуска программы, необходимо было внести изменения в инструментарий для сборки программ, а также в динамический загрузчик. Реализация предлагаемого метода проводилась для архитектуры x86-64 и среды выполнения операционной системы CentOS 7, использующей ELF как основной формат исполняемых и библиотечных файлов. Для минимизации потенциальных проблем совместимости, в этот формат не было внесено никаких изменений. Вся информация, необходимая для рандомизации, хранится в дополнительной секции, и при использовании стандартного динамического загрузчика просто игнорируется. Модифицированный динамический загрузчик при наличии дополнительной секции производит мелкозернистую рандомизацию адресного пространства процесса. В ее отсутствии он работает в точности как стандартный динамический загрузчик.

Сохранение информации о границах функций было реализовано с помощью ключа командной строки `-ffunction-sections`. Данный ключ поддерживается всеми компиляторами переднего плана набора компиляторов GCC (GNU Compiler Collection). Данный набор является де-факто стандартным для компиляции подавляющего большинства пакетов операционной системы CentOS 7, написанных на языках Си, Си++, Фортран. Этот ключ заставляет компилятор размещать каждую функцию транслируемой программы в отдельную секцию объектного файла, как правило, каждая такая секция именуется следующим образом `.text.<function>`. Данный ключ можно включить по умолчанию для компилятора либо использовать специальный скрипт-обертку над драйвером компилятора для его автоматического подключения. Компоновщик `ld` из пакета `binutils` был изменен таким образом, чтобы во время статического связывания он обходил все исполняемые секции объектных файлов и сохранял информацию об их начальном адресе и размере в дополнительной секции. Таким образом, реализуется сохранение информации о границах функций внутри дополнительной секции.

Сохранение информации о ссылках на символы с процесса статической компоновки было реализовано внутри статического компоновщика `ld`. Информация о ссылках на символы сохраняется вместе с информацией о границах функций в

дополнительной секции. Формат дополнительной секции подробно описывается в главе 3.2.1. Поведение компоновщика, отвечающее за генерацию этой секции, занесено под ключ командной строки `-store-reloc-infos`. Данный ключ отвечает за генерацию дополнительной секции с именем `.reloc_infos`, и наполнении ее информацией о границах функций и ссылках на символы и код. При указании данного ключа при промежуточной сборке нескольких объектных файлов в один, предотвращается слияние секций с кодом и создание дополнительной секции. При сборке статических исполняемых файлов, которые не загружаются динамическим загрузчиком, этот ключ игнорируется. 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. Для удобства использования в случае сборки целой операционной системы CentOS 7 с поддержкой мелкозернистой рандомизации ключ `-store-reloc-infos` может быть включен по умолчанию; или может включаться избирательно для пакетов, требующих повышенной степени защищенности от эксплуатации уязвимостей.

В операционной системе CentOS 7 при запуске исполняемого файла ELF ядро операционной системы загружает в память из этого файла, а также из упомянутого в нем динамического загрузчика все сегменты, помеченные необходимыми для загрузки. Выполнение начинается с точки входа динамического загрузчика, который работает уже в контексте исполняемого процесса. Он загружает требуемые динамические библиотеки, подготавливает программу к запуску и передает управление на точку входа запускаемой программы. Для эффективного выполнения мелкозернистой рандомизации программы при запуске необходимо было внести изменения в динамический загрузчик `ld.so`, который является частью библиотеки `glibc`. В загрузчик была добавлена функция, которая ищет в исполняемом файле формата ELF, дополнительную секцию. В случае, если эта секция отсутствует ничего не происходит и загрузка программы происходит без мелкозернистой рандомизации. В случае, если дополнительная секция присутствует, то на основе ее содержимого производится перестановка функций местами и исправление ссылок. Эта функция вызывается перед обработкой исполняемого файла и каждой динамической библиотеки.

3.2.1 Структура дополнительной секции исполняемого файла

Дополнительная секция `.reloc_infos` генерируется модифицированным статическим компоновщиком. В ней содержится информация о границах функций и ссылках на данные и код. Данная секция содержит заголовок, список границ сегментов, список границ функций, список ссылок на данные и код, требующих исправления после рандомизации. Все функции программы пронумерованы, начиная с единицы, в порядке их размещения в списке функций. Номер ноль не присваивается никакой из функций программы. Нулевой номер, использованный при описании ссылки, означает, что данная ссылка не указывает ни в одну из функций программы.

Заголовок содержит четыре числа: количество сегментов; количество функций (на самом деле секций, но, по сути, из-за ключа `--function-sections` это равносильно количеству функций); количество ссылок, которые необходимо будет исправить при запуске программы; номер функции, которая является точкой входа исполняемого файла. Другими словами заголовок предоставляет достаточную информацию для прочтения из секции всех вышеперечисленных списков. Описание данного заголовка представлено в листинге 3.1.

Листинг 3.1 Описание структуры заголовка дополнительной секции

```
struct Header
{
  unsigned int segsz;
  unsigned int secsz;
  unsigned int relsz;
  unsigned int entry;
};
```

Каждая запись списка границ сегментов состоит из двух чисел (размером достаточным для помещения туда указателя): виртуальный адрес сегмента и его размер в байтах. Каждый элемент списка границ функций состоит из двух чисел (размером достаточным для помещения туда указателя): виртуальный адрес функции при загрузке без рандомизации, и ее длина в байтах. Три старших бита числа для хранения длины используются для записи дополнительной информации о секции. В них хранится минимально необходимая степень выравнивания

адреса начала функции. Описание элементов списка границ сегментов и функций представлено в листинге 3.2.

Листинг 3.2 Описание структур элементов списка границ функций и сегментов

```
struct Segment
{
  bfd_vma address;
  bfd_vma length;
};

struct Section {
  bfd_vma start;
  union {
    struct {
      bfd_vma size;
    };
    struct {
      unsigned int l_size;
      unsigned int h_size : 29;
      unsigned int align  : 3;
    };
  };
};
```

Каждый элемент списка ссылок на данные и код содержит: виртуальный адрес по которому расположено значение, требующее исправления после рандомизации; номер функции, в которой находится ссылка; номер функции, в которую указывает ссылка; размер ссылки в байтах (1, 2, 4 или 8 байт); тип ссылки (абсолютный или относительный). В качестве адреса указан такой виртуальный адрес, по которому при загрузке без рандомизации будет размещено значение, требующее исправление после рандомизации.

Листинг 3.3 Описание структуры элемента списка ссылок

```
struct Relocation
{
  unsigned int from : 29;
  unsigned int size : 2;
  unsigned int abs  : 1;
  unsigned int to;
  bfd_vma address;
};
```

Номера функций сохраняются для упрощения реализации исправления и для его ускорения. Их можно было бы вычислить в ходе рандомизации при запус-

ке программы, используя информацию о границах функций и исходного значения по адресу ссылки. Из этих же соображений, некоторые адреса из структуре самого ELF файла и определённых частей программы тоже записаны в список ссылок для исправления. К ним относятся, например, адреса в секции `.eh_frame` и адреса динамических символов и ссылок на них.

Помимо дополнительной секции `.reloc_infos` в ELF файл добавляется секция `.note.reloc_infos`. Она добавляется в сегмент `NOTE` ELF файла в виде дополнительной записи, содержащей виртуальный адрес дополнительной секции `.reloc_infos`. Сегмент `NOTE` предназначен для хранения произвольной дополнительной информации, и все загрузчики и инструменты просто игнорируют неизвестные им записи. Таким образом, ELF файл, собранный с поддержкой рандомизации, может быть загружен любым стандартным динамическим загрузчиком, и вся дополнительная информация будет просто проигнорирована.

3.2.2 Создание и заполнение дополнительной секции на этапе компоновки

Сбор информации о границах функций происходит следующим образом: обходятся все исполняемые секции файла, для каждой сохраняется ее адрес и размер, а также выравнивание, указанные в заголовке каждой секции.

Сбор информации о ссылках происходит в два этапа:

1. в том месте, где компоновщик уже вычислил для каждой ссылки целевой адрес, происходит сохранение этого адреса и адреса, по которому расположена сама ссылка;
2. после того, как собрана информация о всех ссылках, происходит нумерование функций, а каждый адрес ссылки заменяется номером функции, в которой он находится.

При этом часть ссылок оказывается ненужной для последующей рандомизации перемешивания функций. Такие ссылки удаляются, к ним относятся:

– абсолютные ссылки, указывающие на такой адрес, который не принадлежит ни одной функций;
– относительные ссылки, которые находятся в той же самой функции, в которую и указывают.

Статический компоновщик при указании ключа `--store-reloc-infos` выделяет в создаваемом файле место под дополнительную секцию `.reloc_infos`. Причем эта секция должна располагаться в загружаемом сегменте файла формата ELF, иначе динамический загрузчик не будет иметь к ней доступа. Это приводит к тому, что к моменту обработки ссылок компоновщиком дополнительная секция, все секции кода и данных уже должны быть размещены в конкретном месте в файле. Следовательно, необходимо знать размер дополнительной секции до обхода и обработки всех ссылок. Данная проблема решается использованием верхней границы оценки количества ссылок, что приводит к тому, что в конце дополнительной секции остается пустое место.

Часть ссылок пришлось обрабатывать специальным образом. Например, многие ссылки, имеющие отношение к реализации TLS (локальная память потока — от англ. thread local storage). В данном случае код изменяется на уровне ассемблерных команд, а информация об итоговом размещении и целевом адресе ссылки нигде не сохраняется. Реализация каждого такого случая рассматривалась отдельно для определения формулы для подсчета целевого адреса.

Кроме того особая обработка потребовалась для секции `.eh_frame`, которые используются для размотки стека при обработке исключений, например в языке Си++. Дело в том, что уже после исправления ссылок внутри этой секции происходит еще одна их обработка, поэтому сохраненная о них информация перестает быть действительной. Для получения корректной информации о ссылках в этой секции пришлось реализовать разбор их итогового бинарного содержимого, не используя хранимую компоновщиком информацию.

3.2.3 Перестановка функций и исправление ссылок на этапе загрузки

В динамический загрузчик была добавлена функциональность, отвечающая за перестановку функций при загрузке исполняемых файлов и разделяемых библиотек. При загрузке в файле формата ELF ищется дополнительная секция `.reloc_infos` с помощью записи в секции `.note.reloc_infos`. В случае если файл содержит дополнительную секцию производится перестановка функций, в противном случае загрузка происходит как обычно.

Операция перестановки производится в тот момент, когда файл уже корректно загружен в память и произведена, если необходима, динамическая компоновка. Перестановка производится случайным образом, при этом запоминается на какое расстояние (смещение) в байтах была сдвинута функция. После перестановки производится исправление ссылок. К значению каждой ссылки прибавляется смещение целевой функции. Кроме того, в том случае, если ссылка относительная, то вычитается смещение той функции, в которой содержится исправляемая ссылка.

Перестановка функций при загрузке модифицирует код программы в памяти. Почти всегда из-за соображений безопасности код загружается в память, для которой запрещена запись. Для осуществления перестановки функций на память, содержащую код, вызывается системный вызов `mprotect`, для временного разрешения ее модификации. В случае присутствия в операционной системе механизмов защиты SELinux [66], PaX [14], seccomp [67], которые запрещают одному региону памяти за время жизни процесса быть исполняемым и записываемым, может потребоваться их дополнительная настройка и адаптация для разрешения такого поведения. В свете сказанного, перспективным направлением для развития в будущем данного подхода является создание мелкозернистой рандомизации с разделением памяти.

Область памяти, в которой сохраняются смещения функций и другая временная информация, выделяется системным вызовом `mmap`. После завершения рандомизации, она освобождается вызовом `munmap`, после чего её содержимое невозможно восстановить. Содержимое секции `.reloc_infos` также обнуляется. Таким образом, после завершения перестановки в памяти процесса не остаётся никакой дополнительной информации, утечка которой могла бы раскрыть размещение функций.

3.2.4 Улучшение предложенной реализации

Описанная в предыдущих разделах реализация была подвергнута различным проверкам на качество защиты, как теоретическим, так и практическим. Подробнее об этом написано в главах 4, 5. Теоретические оценки и практические

эксперименты по качеству защиты реализации мелкозернистой рандомизации показали её недостатки:

– недостаточную энтропию случайности при небольшом количестве функций,
– повышенную вероятность остаться на своём месте для кода, находящегося на границах исполняемого файла (функции в начале и конце).

Для преодоления обнаруженных недостатков было предложено улучшение метода мелкозернистой рандомизации. Дополнительно к перестановке функций местами было реализовано смещение базового адреса секции с функциями. Это позволяет сравнять вероятность обнаружения крайних и центральных функций на первоначальных местах и увеличивает энтропию рандомизации.

При реализации предложенного улучшения были внесены следующие исправления в описанные алгоритмы работы мелкозернистой рандомизации:

– во время статической компоновки секция `.reloc_infos` располагается всегда перед секцией кода `.text`,
– во время работы динамического загрузчика перемешанные функции копируются в область памяти, которая на некоторую случайную величину сдвинута внутрь секции `.reloc_infos`.

3.3 Влияние на производительность

С точки зрения практической применимости важно знать, какое влияние на производительность оказывает реализованная мелкозернистая рандомизация при загрузке. Для этого важно измерить следующие величины:

– изменение времени запуска программы,
– изменение времени работы программы,
– изменение потребления оперативной памяти программы,
– изменение размера исполняемых файлов в формате ELF.

Тесты на производительность производились на наборе тестов SPEC CPU® 2017 [68]. Данный набор тестов является де-факто стандартом для измерения изменения производительности при реализации оптимизирующих преобразований компиляторов. Он содержит 4 набора тестов: intspeed, fpspeed,

intrate, fprate. Сводные результаты измерений представлены в таблицах 4, 3. Запуск под названием `base` соответствует запуску без мелкозернистой рандомизации, а запуск с названием `peak` соответствует запуску с включенной мелкозернистой рандомизацией.

Таблицы устроена следующим образом:

- каждому тесту соответствует отдельная строка;
- в столбце с названием «Название теста» указано название теста;
- в столбце с название «N» указано количество функций в исполняемом файле теста;
- в столбце с название «$\Delta t, \%$» указано относительное ускорение (или замедление) в процентах по следующей формуле $(t_{peak} - t_{base})/t_{base}$, соответственно положительная величина показывает замедление, а отрицательная ускорение;
- в столбце с названием «F_b, MB» указывается размер исполняемого файла в мегабайтах для серии измерений `base`;
- в столбце с названием «$\Delta F, \%$» показано изменение размера исполняемого файла для серии измерений `peak` относительно предыдущего столбца в процентах;
- в столбце с названием «S_b, мс» указывается в миллисекундах среднее время запуска теста для серии измерений `base`, измеренное на 1000 запусков;
- в столбце с названием «S^p, мс» указывается в миллисекундах среднее время запуска теста для серии измерений `peak`, измеренное на 1000 запусков;
- в столбце с названием «S_r» указывается замедление запуска теста с мелкозернистой рандомизацией относительно обычного времени запуска, которое посчитано по формуле S^r/S_b, оно показывает во сколько раз замедлился запуск программы с включенной мелкозернистой рандомизацией при запуске.

Результаты по изменению времени работы приложений следующие. Геометрическое среднее показателя измерения производительности набора тестов SPEC CPU® 2017 не претерпело существенных изменений:

- intspeed, 3,85 для `base` против 3,84 для `peak`, т.е. ухудшилось на 0,3 %.
- fpspeed, 7,80 для `base` против 7,80 для `peak`, т.е. не изменились.

– intrate, 3,09 для `base` против 3,06 для `peak`, т.е. ухудшилось на 1 %.
– fprate, 3,16 для `base` против 3,15 для `peak`, т.е. ухудшилось на 0,3 %.

Это означает, что преобразования, проведенные над программой, незначительно ухудшают производительность.

Более детальное рассмотрение результатов изменения времени работы в таблицах 4, 3 показывает, что производительность отдельных тестов как ухудшалась до 7 % для теста xalancbmk_s, так и увеличивалась до 4,3 % для теста xalancbmk_r. Эти два теста являются крайними точками, тогда как время работы большинства тестов изменялось в пределах 2 % в обе стороны. Ускорение и замедление может быть объяснено двумя причинами: плавающий результат из-за некоторой нестабильности измерения и улучшением (или ухудшением) локальности расположения кода в адресном пространстве программы.

Мелкозернистая рандомизация увеличивает количество действий, производимых системным загрузчиком, при запуске программы. Важно понимать, насколько замедляется запуск программы с включенной мелкозернистой рандомизацией адресного пространства. По результатам соответствующих измерений в таблицах 4, 3 можно сделать следующий вывод:

– время загрузки программ замедлилось в 1,1–33 раз, при среднем относительном замедлении около 6;
– абсолютное время загрузки не превышает 30 мс, что является малой величиной по сравнению со временем работы программы и средним временем инициализации программы до активного взаимодействия со стороны пользователя.

Размер исполняемых файлов увеличился по результатам, приведенным в таблицах 4, 3, в среднем на 31 %. Изменение размера исполняемых файлов происходит из-за сохранения в них в пределах отдельной секции .reloc_infos информации о границах функций и релокациях.

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 исключает возможность утечки размещения адресного пространства данной библиотеки из других процессов операционной системы.

Таблица 3 — Изменение времени работы тестов из набора int SPEC CPU® 2017

Название теста	N	Δt, %	F_b, MB	ΔF, %	S_b, мс	S^p, мс	S_r
perlbench_r	2464	2,4	2,527	45	0,60	5,93	9,8
perlbench_s	2464	1,3	2,527	45	0,96	6,29	6,6
gcc_r	12929	1,3	11,086	67	1,08	20,30	18,9
gcc_s	12929	2,2	11,086	67	1,45	20,83	14,3
mcf_r	46	0,2	0,039	20	0,57	0,63	1,1
mcf_s	46	-3,2	0,039	20	0,82	0,92	1,1
omnetpp_r	7902	0,8	3,262	60	1,65	7,81	3,7
omnetpp_s	7902	3,0	3,262	60	1,89	7,93	4,2
xalancbmk_r	17305	2,0	8,879	39	1,77	14,26	8,1
xalancbmk_s	17305	7,0	8,879	39	1,98	14,32	7,2
x264_r	537	0,2	0,684	18	0,60	1,86	3,1
x264_s	537	0,5	0,684	18	0,87	2,11	2,4
deepsjeng_r	114	0,7	0,117	30	1,44	1,68	1,2
deepsjeng_s	114	-1,2	0,117	33	1,66	1,87	1,1
leela_r	407	0,0	0,308	25	1,48	2,02	1,4
leela_s	407	0,4	0,308	25	1,69	2,31	1,4
exchange2_r	5	0,8	0,129	9	1,48	9,95	5,7
exchange2_s	5	-1,4	0,129	12	1,12	1,38	1,2
xz_r	375	0,5	0,242	26	0,54	0,87	1,6
xz_s	381	-4,3	0,246	25	0,74	1,16	1,6

Таблица 4 — Изменение времени работы тестов из набора fp SPEC CPU® 2017

Название теста	N	Δt, %	F_b, МВ	ΔF, %	S_b, мс	S^p, мс	S_r
bwaves_r	11	0,4	0,051	15	0,91	0,97	1,1
bwaves_s	11	0,0	0,078	15	1,11	1,32	1,2
cactuBSSN_r	2719	0,4	4,828	37	1,46	10,07	6,9
cactuBSSN_s	2797	-1,4	4,867	37	1,76	9,96	5,7
namd_r	137	0,3	0,852	12	1,39	3,08	2,2
parest_r	17917	2,5	12,375	29	1,69	19,55	11,6
povray_r	1627	-0,1	1,324	46	1,45	4,43	3,1
lbm_r	21	-0,6	0,031	13	1,61	7,88	4,9
lbm_s	28	0,1	0,031	13	0,88	0,97	1,1
wrf_r	1246	0,3	21,133	23	0,91	30,31	33,5
wrf_s	1246	0,2	21,301	23	1,17	30,17	25,8
blender_r	38216	1,5	22,371	52	5,84	31,72	5,4
cam4_r	1278	-0,2	53,824	5	0,97	13,40	13,8
cam4_s	1278	0,2	54,090	5	1,14	13,55	11,9
pop2_s	1069	0,0	174,52	2	1,21	11,34	9,4
imagick_r	2192	-0,1	2,434	49	0,67	6,31	9,5
imagick_s	2374	0,5	2,555	48	0,93	6,63	6,6
nab_r	238	-1,5	0,266	41	0,59	1,07	1,8
nab_s	242	0,3	0,273	41	0,85	1,32	1,5
fotonik3d_r	21	-0,5	0,457	40	0,90	1,80	2,0
fotonik3d_s	21	0,5	0,473	39	1,10	2,07	1,9
roms_r	157	-0,2	1,008	26	0,94	2,98	3,2
roms_s	157	-0,7	1,020	26	1,14	3,11	2,7

Таблица 5 — Изменение количества оперативной памяти ОС, расходуемой на хранение исполняемых секций с кодом, при применении мелкозернистой рандомизации

Конфигурация ОС	S_o, MB	S_r, MB	Δ, MB	S_r / S_o
Серверная версия	45	205	160	3,6
Настольная версия	466	2449	1983	4,3

Из-за отказа от разделяемости библиотек увеличивается общее потребления оперативной памяти в операционной системе.

Величину изменения потребления оперативной памяти образом ОС достаточно сложно измерить объективным образом, потому что она сильно зависит от типа операционной системы и характера, а также количества приложений в ней запущенных. В таблице 5 приводится результат экспериментального измерения этой величины для двух конфигураций установленной операционной системы. Были исследованы две конфигурации:

– Серверная версия — установлена с минимального установочного образа ОС; не содержит графического интерфейса; содержит только базовый набор консольных утилит, а также веб-сервер и OpenSSH сервер.
– Настольная версия представляет собой типичную установку для настольного компьютера, которая содержит графическую оболочку пользователя, интернет-браузер, почтовый клиент и мессенджер.

Измерялся только объём памяти, израсходованный на хранение страниц памяти, доступных для выполнения. Для этого обходились все файлы `/proc/pid/maps`. Из этих файлов извлекалась информация о размере секции кода и имени библиотеки. Суммарное потребление памяти подсчитывалось для двух ситуаций: обычная ОС с разделением памяти, соответствует столбцу S_o; ОС с примененной ко всем приложениям мелкозернистой рандомизацией, что исключает разделение библиотек между процессами, соответствует столбцу S_r. В столбце Δ указана величина на которую увеличилось потребления оперативной памяти в обоих случаях. В столбце S_r / S_o указано во сколько раз увеличилось потребление оперативной памяти во втором случае по сравнению с первым. По предоставленным в таблице 5 данным видно, что потребление памяти в полносистемном случае применения мелкозернистой рандомизации увеличивается значительно, однако суммарное количество для настольных операционных систем, требующих повышенного уровня защищенности, выглядит приемлимо.

Глава 4. Оценка эффективности реализованных методов защиты

Для оценки качества реализованных в главе 3 преобразований необходимо провести оценку их эффективности с точки зрения защиты от эксплуатирования методами повторного использования кода. В литературе существует два принципиальных подхода к рассмотрению данного вопроса:

1. теоретико-логическое обоснование эффективности предложенного метода,
2. экспериментальная проверка реализованных методов.

Прежде чем перейти к теоретико-логическому обоснованию эффективности предложенного метода зафиксируем модель атаки на приложение. Модель атаки: атакующий получил в свое распоряжение исполняемый файл и предполагает, что в атакуемой операционной системе среди защит имеется только DEP. Не вдаваясь в подробности поиска и создания входных данных для реализации имеющейся в программе уязвимости, будем считать, что атакующий обладает входными данными на которых реализуется уязвимость. Атакующий создает нагрузку для эксплуатации, используя метод повторного использования кода, при этом он полагает, что внутри модуля относительное местоположения кода не меняется. Созданный таким образом эксплойт подается на вход программе, адресное пространство которой рандомизируется при запуске по предложенной методике в главе 3.

Перейдем теперь к теоретико-логическому обоснованию эффективности метода защиты. В предложенной модели атаки эксплойт на рандомизированные при запуске программы не будет работать из-за изменения относительного положения функций. Куски, повторно используемого в нагрузке кода, изменят свое местоположение. Управление вместо начала гаджета попадет в какое-то произвольное место. На архитектуре x86 из-за плотности набора инструкций зачастую с этого произвольного места начнется исполнение каких-то инструкций, которое закончится на обращении в какую-нибудь невыделенную память ошибкой сегментации или неправильной инструкцией. Таким образом, атака понизит свою опасность с удаленного выполнения кода до отказа в обслуживании. Стоит заметить, что существует вероятность и такая перестановка функций, при которой использованные атакующим гаджеты не изменяют своего местоположения. Такому условию всегда удовлетворяет исходный порядок функций. Оценить вероятность

его реализации несложно. Она равна $1/N!$. Кроме исходной перестановки могут существовать и другие, которые оставляют на месте используемые атакующим гаджеты. Оценить их количество гораздо сложнее.

Вообще стоит заметить, что эффективность предложенного метода защиты сильно зависит от количества функций в программе. В таблице 6 приведена оценка числа среднего числа попыток до успешной реализации эксплойта в предположении, что только исходное расположение функций приводит к успешной атаке.

Таблица 6 — Среднее количество попыток до успешной атаки на рандомизированное при запуске приложение в предположении, что только исходное расположение функций приводит к успешной атаке

Кол-во функций	Число попыток
N	$N!/2$
1	1
2	1
3	3
4	12
5	60
6	360
7	2520
8	20160
9	181440
10	1814400
15	$6{,}5 * 10^{11}$
20	$1{,}2 * 10^{18}$
50	$1{,}5 * 10^{64}$
100	$4{,}7 * 10^{157}$

Методология и результаты экспериментальной проверки эффективности реализованных методов защиты приводятся в следующем разделе.

Приведенная выше модель атаки не подразумевала никакого конкретного типа атаки повторного использования кода. В данном разделе в рамках эксперимента оценивается эффективность защиты от атаки возвратно-ориентированного программирования (ROP). Предполагаемый сценарий атаки на приложение заключается в следующем: атакующий имеет в своем распоряжении исполняемый файл приложения. В этом файле находятся все гаджеты, а затем из подходящих гаджетов строится ROP-цепочка, реализующая нагрузку. Полученную ROP-

цепочку подают на вход запущенному приложению несколько раз, наблюдая за поведением и фиксируя результат.

Экспериментальная проверка состояла из нескольких серий измерений. Исследуемый набор приложений представлял собой исполняемые файлы из стандартной минимальной установки CentOS 7, располагаемые в директориях /usr/bin/, /usr/sbin. 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. Тестовый набор состоит из 487 файлов.

Для исследования и подсчета количества гаджетов необходимо было сохранять состояние адресного пространства процесса после момента его перемешивания. Это было сделано с помощью утилиты сохранения снимков памяти процесса. Снимок памяти сохраняется в ELF формате, где для каждого сегмента создается своя отдельная секция. По умолчанию сохранять секции с кодом в снимок памяти не имеет смысла, поскольку секции с кодом остаются неизменными и всегда доступны в файле на диске. Для записи всех сегментов рабочей памяти процесса, в том числе и исполняемых, были внесены изменения в алгоритм сохранения снимков памяти отладчика gdb. С помощью этой утилиты для каждого из 10 запусков каждого исполняемого файла были получены 4870 снимков памяти. Собранные снимки памяти вместе с оригинальными файлами образовали тестовую популяцию над которой производились эксперименты.

4.1 Поиск и классификация гаджетов

Поиск гаджетов производился с помощью инструмента с открытым исходным кодом ROPgadget [69]. Данный инструмент находит инструкции передачи управления в исполняемых секциях программы и дизассемблирует несколько байт, предшествующих найденным инструкциям. Все успешно дизассемблированные блоки инструкций добавляются в список потенциально полезных гаджетов.

Полученные кандидаты в гаджеты классифицируются согласно семантическим типам. Список семантических типов по которым производится классификация:

1. NoOp — не меняет никаких значений памяти и регистров (за исключением счётчика команд).
2. MovReg — копирует значение из одного регистра в другой.
3. LoadConst — загружает значение со стека, расположенное по заданному смещению от указателя стека, на заданный регистр.
4. Arithmetic — производит арифметическую бинарную операцию над значениями двух регистров с сохранением результата в третьем регистре.
5. LoadMem — загружает значение из памяти, расположенное по заданному смещению от заданного регистра, на заданный регистр.
6. StoreMem — сохраняет значение заданного регистра в память, расположенную по заданному смещению от заданного регистра.

Полный список семантических типов и подробное описание процесса классификации опубликовано в статье [70].

Процесс классификации работает следующим образом:

– инструкции гаджета транслируются в промежуточное представление;
– промежуточное представление интерпретируется;
– в ходе интерпретации отслеживаются все обращения к регистрам и памяти на чтение и запись, начальные считанные значения генерируются случайным образом;
– в результате интерпретации получаются начальные и конечные значения регистров и памяти, которые ограничивают список возможных семантических типов, которым удовлетворяет гаджет;
– процесс интерпретации повторяется несколько раз с различными входными данными;

В результате остаются только те семантические типы, для которых выполнились условия на всех запусках. Классифицированные гаджеты сохраняются в базу данных вместе с дополнительной информацией о типе гаджета, его адресе, о параметрах гаджета и его побочных эффектах. С помощью полученного классифицированного набора гаджетов возможно узнать, существует ли в данном файле на заданном адресе гаджет, какие у него параметры и побочные эффекты и тому подобное.

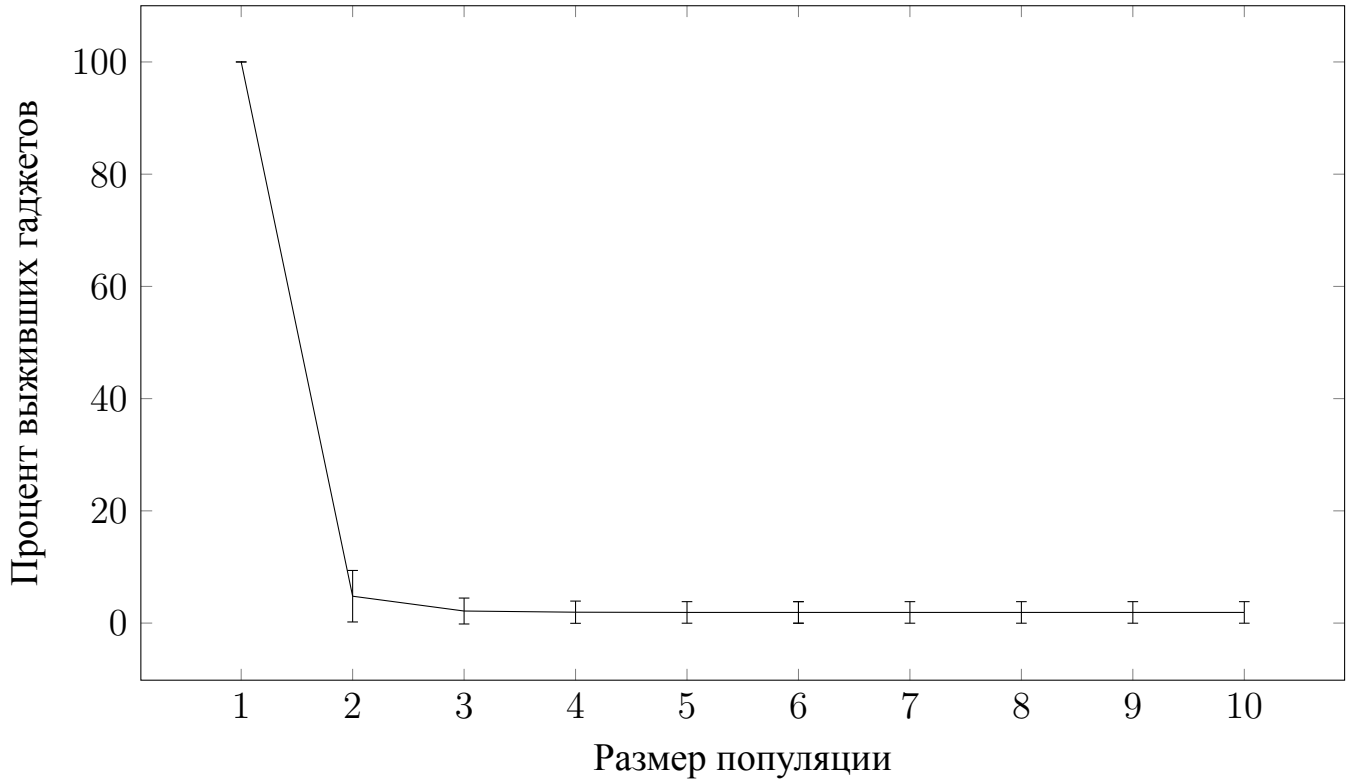

Рисунок 4.1 — Среднее относительное количество выживших гаджетов в зависимости от размера популяции

4.2 Оценка количества выживших гаджетов

Введем 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. Такое определегие было предложено Coffman в работе [71]. Выжившие гаджеты важны для исследования по причине того, что составленная из таких гаджетов ROP-цепочка работоспособна на каждом снимке памяти из популяции.

Измерение количества выживших гаджетов производилось путем обращения к базам данных гаджетов исполняемого файла и снимков памяти процессов. Для каждого гаджета из исполняемого файла проверялось находится ли по его адресу в каждом снимке памяти процесса гаджет с таким же семантическим типом и похожими побочными эффектами. Зависимость количества выживших гаджетов от размера популяции представлена на рисунке 4.1. На данном рисунке пред-

ставлена 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 следующим замечанием. В исполняемом сегменте кроме кода функций, изменяющих свое местоположение, находятся также вспомогательные секции: таблица связывания процедур PLT, секции INIT, FINI и другие исполняемые секции. Их местоположение остается неизменным, а следовательно гаджеты внутри них всегда являются выжившими.

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.

$$\frac{\sum_{j=1}^{m} \left(\frac{\sum_{i=1}^{n_j} k_i^j}{10 n_j} \right)}{m} = 0{,}05 \tag{4.1}$$

Это означает, что вероятность гаджета остаться на своем месте (по тестовому набору файлов) составляет примерно 5 %.

4.3 Оценка работоспособности ROP-цепочек

Кроме метрики количества выживших гаджетов важно понять, насколько велики шансы у реально используемых нагрузок в виде ROP-цепочек успешно отработать на рандомизированном при запуске исполняемом файле. Для этого был проведен следующий эксперимент. Были взяты примеры пяти нагрузок и полуавтоматическими методами собраны в виде ROP-цепочек для каждого файла из тестового набора предыдущей главы.

4.3.1 Пять типов модельной нагрузки

Для эксперимента были взяты пять примеров модельной нагрузки разной сложности.

1. Вызов функции без аргументов: `foo();`.
2. Вызов функции с одним аргументом: `foo(1);`.
3. Вызов функции с двумя аргументами: `foo(1, 2);`.
4. Вызов функции с тремя аргументами: `foo(1, 2, 3);`.
5. Вызов оболочки системного интерпретатора: `system("/bin/sh");`.

Выбор таких примеров нагрузки обусловлен желанием выбрать осмысленные, но между тем плавно нарастающие по сложности ROP-цепочки. Изменение сложности поможет отследить некоторые закономерности в дальнейшем. Цепочки каждого типа полуавтоматически создавались для каждого файла из тестового набора из 487 файлов CentOS 7.

Приведем пример того, как могут быть составлены ROP-цепочки для каждого модельного примера. Все нижеприведенные примеры были составлены для исполняемого файла `as`. Строчки вида `0xdeadbeef -> POP RAX; RET` соответствуют гаджетам в цепочках, слева от стрелочки указан адрес гаджета в исполняемом файле, справа от стрелочки написан ассемблерный код инструкций, которые располагаются по этому адресу. Строчки вида `: 0x68732f6e69622f (/bin/sh)` соответствуют данным, которые гаджет загружает на регистры.

1. Вызов функции без аргументов:

```
0x403050 -> foo
```

2. Вызов функции с одним аргументом:

```
0x43db43 -> POP RDI ; RET
 : 0x1 (\x01)
0x403050 -> foo
```

3. Вызов функции с двумя аргументами:

```
0x43a4d2 -> POP RSI ; RET
 : 0x2 (\x02)
0x43db43 -> POP RDI ; RET
 : 0x1 (\x01)
0x403050 -> foo
```

4. Вызов функции с тремя аргументами:

```
0x4693e0 -> POP RDX ; RET
 : 0x3 (\x03)
0x43a4d2 -> POP RSI ; RET
 : 0x2 (\x02)
0x43db43 -> POP RDI ; RET
 : 0x1 (\x01)
0x403050 -> foo
```

5. Вызов оболочки системного интерпретатора:

```
0x479a30 -> POP RAX ; RET
 : 0x68732f6e69622f (/bin/sh)
0x43db43 -> POP RDI ; RET
 : 0x830c654 (T\xc60\x08)
0x411b98 -> MOV QWORD PTR [RDI], RAX ; RET
0x43db43 -> POP RDI ; RET
 : 0x830c654 (T\xc60\x08)
0x403050 -> system
```

4.3.2 Описание процесса создания ROP-цепочек

Процесс создания ROP-цепочек происходил с помощью инструмента автоматической генерации эксплойтов повторного использования кода

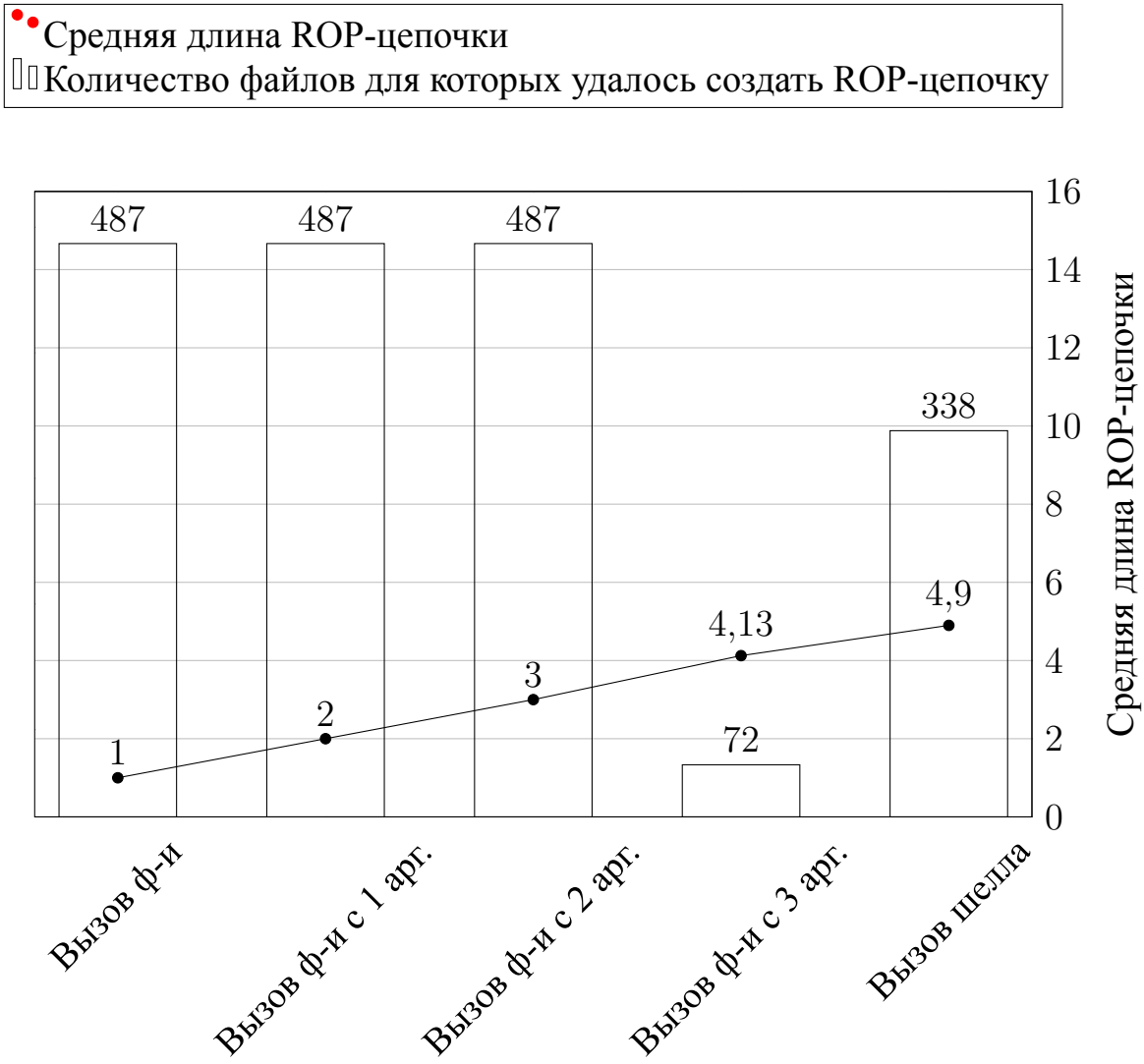

Рисунок 4.2 — Статистика по результатам создания ROP-цепочек для модельных типов нагрузок. По оси абсцисс отложены типы созданных ROP-цепочек. Столбцы показывают количество исполняемых файлов для которых были успешно созданы ROP-цепочки. Сплошной линией показано среднее количество ROP-гаджетов, которые потребовались для создания цепочки соответствующего типа.

MAJORCA [72], разработанного в ИСП РАН. Схематично общий алгоритм составления цепочки можно описать следующим образом:

– 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: `RDI`, `RSI`, `RDX` соответственно для первого, второго и третьего аргумента.
– Производился поиск в базе данных гаджетов для инициализации регистров, определенных на предыдущем шаге. Инициализация производилась путем загрузки значений на регистры со стека, который, как правило, контролируется атакующим при типичном переполнении буфера на стеке.
– Для цепочки номер 4 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.

На рисунке 4.2 столбцами значений показана статистика по количеству созданных цепочек каждого типа для тестового набора файлов размером 487. Надо заметить, что первые три типа цепочек успешно создались для всех файлов тестового набора. Вызов функции с тремя аргументами удалось создать только для 72 файлов. Это объясняется тем, что гаджет загрузки значения со стека на регистр `RDX` (третий аргумент передается через данный регистр) встречается не во всех исполняемых файлах. Цепочки типа вызов оболочки системного интерпретатора удалось создать для 338 файлов. Для оставшихся файлов не удалось создать данную цепочку по причине отсутствия гаджетов записи значения в память с соответствующими гаджетами загрузки значений со стека.

Кроме того на рисунке 4.2 показана зависимость среднего количество гаджетов, необходимых для того, чтобы собрать цепочку указанного типа. Для простейшего примера вызова функции без аргументов требуется один гаджет (сама

вызываемая функция). Для вызова оболочки системного интерпретатора требуется в среднем почти 5 разных гаджетов. Данный график показывает, что подобранные модельные типы ROP-нагрузок возрастают в сложности своей конструкции от вызова функции без аргументов до вызова оболочки системного интерпретатора.

4.3.3 Описание методологии проверки работоспособности ROP-цепочек

Для проверки работоспособности необходимо было проводить многократные запуски тестируемых приложений с включенной рандомизацией адресного пространства при запуске. Для приближения экспериментальных тестов к реальным необходимо было либо найти эксплуатируемую уязвимость внутри тестируемого приложения, либо привнести ее в него искусственно. Поиск реальных уязвимостей является исключительно сложной задачей и находится за рамками рассматриваемой в данной диссертации темы. Кроме того, далеко не в каждом тестируемом файле существует эксплуатируемая уязвимость, что уменьшает количество тестируемых файлов и может негативно сказываться на качестве результатов эксперимента по проверка качества рандомизации.

Разумным компромиссом в данной ситуации является подход, основанный на внедрении в тестируемый исполняемый файл типичной уязвимости, приближенной к реальному переполнению буфера и эксплуатации ее модельными примера нагрузок, описанных выше. Желательно произвести внедрение уязвимости в тестируемый файл с минимальным влиянием на его адресное пространство. Самым разумным решением в данной ситуации оказалась следующая схема, представленная на рисунке 4.3.

В адресное пространство процесса тестируемой программы внедряется при помощи LD_PRELOAD динамическая библиотека, загружаемая по фиксированному адресу (`0xbb000000`). Код данной библиотека доступен в приложении Д. Данная библиотека состоит из ряда функций:

– `__libc_start_main` Точка входа в динамическую библиотеку. При загрузке библиотеки эта функция подменяет собой в адресном пространстве функцию с соответствующим именем из библиотеки `libc.so`. Каж-

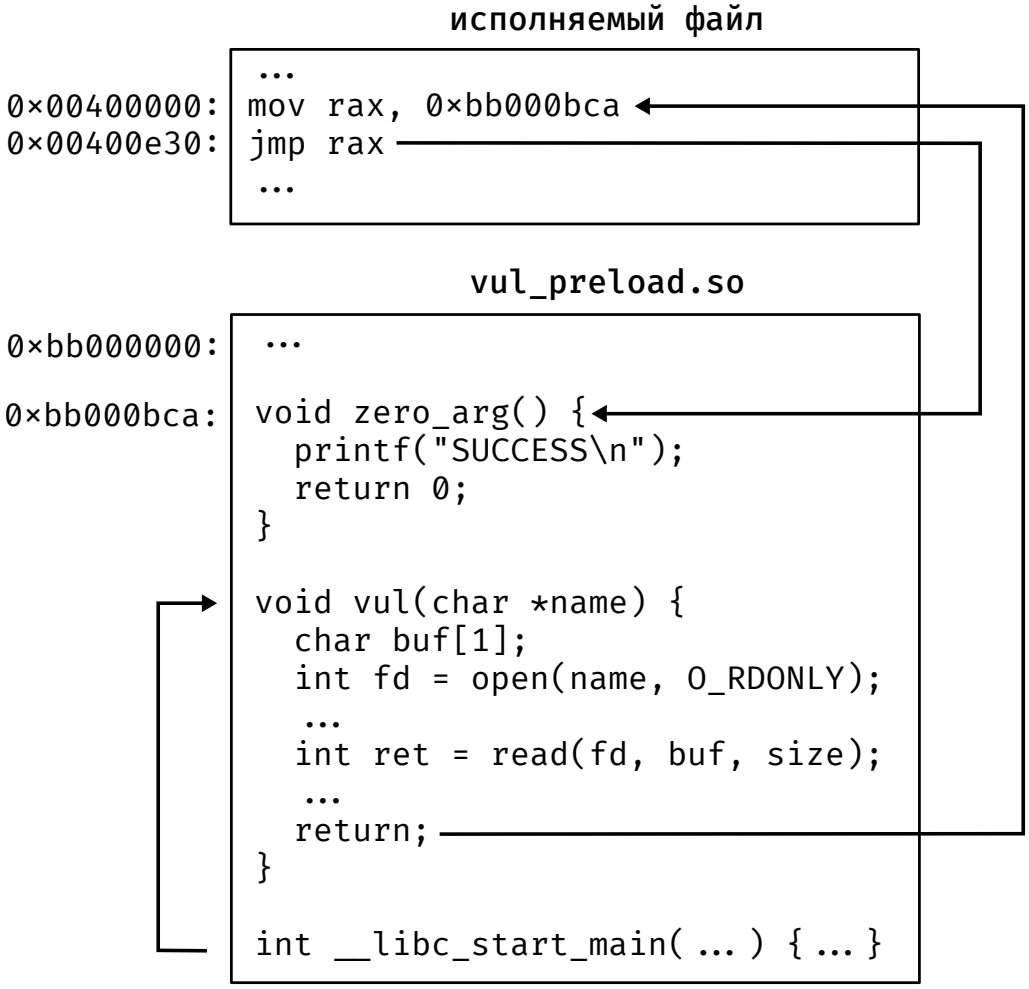

Рисунок 4.3 — Адресное пространство тестируемой программы с динамической библиотекой VUL_PRELOAD.SO, содержащее внедренную модельную уязвимость в функции ’vul’. Стрелками показан путь передачи потока управления во время тестирования успешной ROP-цепочки.

дый раз вместо вызова функции из `libc` будет вызываться функция нашей динамической библиотеки.

– `__so_main` Функция выделяет заведомо достаточное место для размещения эксплойта на стеке.
– `vul` Функция читает данные из файла на диске под названием `"/tmp/vul_input"` и записывает их в локальный буфер `buf`. Она содержит ошибку переполнения буфера на строке:

```
int ret = read(fd, buf, size);
```

При достаточном размере входного файла происходит переполнение буфера и перезапись адреса возврата. Размер данных достаточных для инициализации уязвимости составляет 53 байта. За ними следует полезная нагрузка в формате ROP, где адрес первого гаджета перетирает адрес возврата из функции `vul`.

– `filesize` Вспомогательная функция, вычисляющая размер файла `"/tmp/vul_input"`.
– `zero_arg` Функция для проверки ROP-нагрузки вызова функции без аргументов.
– `one_arg` Функция для проверки ROP-нагрузки вызова функции с одним аргументом.
– `two_arg` Функция для проверки ROP-нагрузки вызова функции с двумя аргументами.
– `three_arg` Функция для проверки ROP-нагрузки вызова функции с тремя аргументами.
– `__system` Функция для проверки ROP-нагрузки вызова оболочки системного интерпретатора.

При запуске программы происходит следующая последовательность действий:

1. До запуска тестируемой программы производится подготовка входных данных для реализации и эксплуатации уязвимости переполнения буфера в библиотеке `vul_preload.so`.
2. Происходит загрузка в адресное пространство библиотеки `vul_preload.so`. Библиотека содержит функцию под названием `__libc_start_main`. Динамический загрузчик запоминает ее рас-

положение и затеняет ей оригинальную функцию из стандартной библиотеки Си.

3. Происходит загрузка в адресное пространство исполняемого файла.
4. Исполняемый файл собран с поддержкой мелкозернистой рандомизации адресного пространства при запуске и содержит специальную секцию под названием `.reloc_infos`. С помощью информации, содержащейся в этой секции динамический загрузчик производит изменение порядка расположения функций в исполняемом сегменте памяти процесса.
5. Динамический загрузчик передает управление на входную точку приложения через вызов функции `__libc_start_main`. Тем самым перед самым началом работы тестируемого приложения управление передается в динамическую библиотеку `vul_reload.so`.
6. В динамической библиотеке `vul_preload.so` начинается выполнение кода, который приводит к уязвимости переполнения буфера на стеке в функции `vul`.
7. При возврате из функции `vul` исполнение передается на первый гаджет из ROP-цепочки, после чего начинается поочередное выполнение ROP-гаджетов из цепочки. В случае успешного выполнения всей цепочки исполнение попадает в функции, проверяющие успешность выполнения ROP-нагрузки и печатающие на экран сообщения вида `SUCCESS`, `PARAMETERS ARE CORRECT`. Наличие или отсутствие этих сообщений позволяет судить об успешности выполнения ROP-нагрузки.

Стоит пояснить механизм передачи управления в функции для проверки ROP-нагрузок из последнего пункта. Дело в том, что все модельные примеры нагрузок оканчиваются вызовами функций. В принятой нами модели атаки, когда само приложение не является позиционно-независимым, кандидатами в такие функции являются все функции тестируемого приложения. Однако, в тестируемом приложении может не быть функции с правильной сигнатурой и подходящим количеством аргументов. Тем более, такие функции не будут содержать единообразную проверку значений своих аргументов на соответствие тем, которые записаны в ROP-цепочке, в заданном нами формате сообщения на стандартном выводе.

Из-за перечисленных причин было принято решение об искусственном внедрении такой функциональности в каждую тестируемую программу следую-

щим образом. В тестируемом приложении выбиралась функция, в которой эпилог переписывался на код, который передавал управление на одну из функций проверки ROP-нагрузки внутри библиотеки `vul_reload.so`. В приведенной на рисунке 4.3 схеме такой функцией оказалась функция, которая загружается без рандомизации на адрес `0x400e30`.

В схеме 4.3 показано адресное пространство тестируемой программы без рандомизации. Когда рандомизация отключена, то поток управления проходит по следующему пути: после загрузок всех модулей и библиотек в память управление передается на функцию `__libc_start_main`, далее управление следует в уязвимую функцию `vul`, в ней происходит переполнение буфера на стеке в момент чтения буфера из файла функцией `read`, возврат из функции `vul` происходит на перезаписанный ROP-нагрузкой адрес, в случае с нагрузкой вызова функции без аргументов этот адрес `0x400e30`, далее управление возвращается обратно в динамическую библиотеку в функцию проверки `zero_arg`. Без рандомизации ROP-нагрузка приводит поток управления в функцию `zero_arg`, в которой успешно выполняется и печатается на экран сообщение `SUCCESS`. В случае работающей рандомизации адресного пространства при запуске программы будет меняться местоположение функции, и с некоторой вероятностью она будет оказываться по адресу отличному от `0x400e30`. В этом случае управление пойдет по некоторому неизвестному пути и не приведет к вызову функции `zero_arg`. Тем самым и осуществляется с помощью многократных запусков статистическая экспериментальная проверка реализованного метода защиты от эксплуатации уязвимостей.

4.3.4 Результаты и их обсуждение

Каждую созданную ROP-цепочку подавали на вход тестируемого приложения по методике, описанной в предыдущей главе. Данная процедура повторялась по 100 раз для каждой цепочки. Кроме того, выбор местоположения целевой функции ROP-эксплойта влияет на процент успешных запусков (глава 5, рисунок 5.1). Из-за этого проводились для каждой модельной нагрузки две серии по 100 запусков. Для первой серии целевой функцией являлась первая функция в секции кода

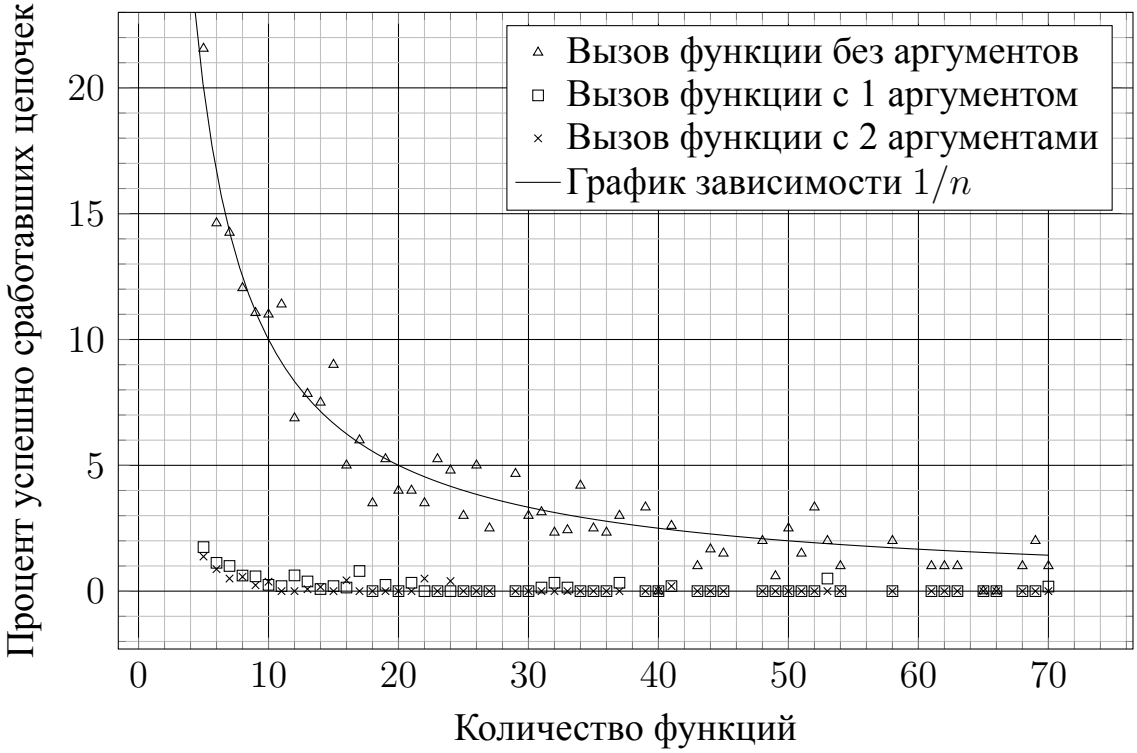

Рисунок 4.4 — Зависимость процента успешно сработавших ROP-цепочек в зависимости от количества функций в тестируемой программе для разных типов модельной нагрузки для серии измерений с краевой целевой функцией (КЦ).

исполняемого файла (КЦ - краевая целевая). Для второй серии целевой функцией являлась средняя функция в секции кода исполняемого файла (СЦ - средняя целевая).

Кроме того, как уже было замечено в таблице 6 вероятность успешной атаки зависит от количества функций в программе. График зависимости количества успешных запусков от количества функций в тестируемом файле приведен на рисунке 4.4 и рисунке 4.5 для серий измерений с краевой целевой функцией и средней целевой функцией соответственно. По этим графикам видно, что с увеличением количества функций процент успешно сработавших цепочек уменьшается. При этом процент успешных срабатываний для модельных нагрузок типа вызова функции с 1 и 2 аргументами больше 1 % только для тестируемых файлов с количеством функций меньше 15. Процент успешных срабатываний для модельной нагрузки типа вызова функции без аргументов на рисунке 4.4 для серии измерений с краевой целевой функцией соответствует зависимости $1/n$, что отвечает посчитанной вероятности остаться на своем месте для краевой функции. Вероятность краевой функции остаться на своем месте получается из формулы 5.11, при подстановке индекса равного 1 или n, и описывается формулой $MX_i = 1/n$.

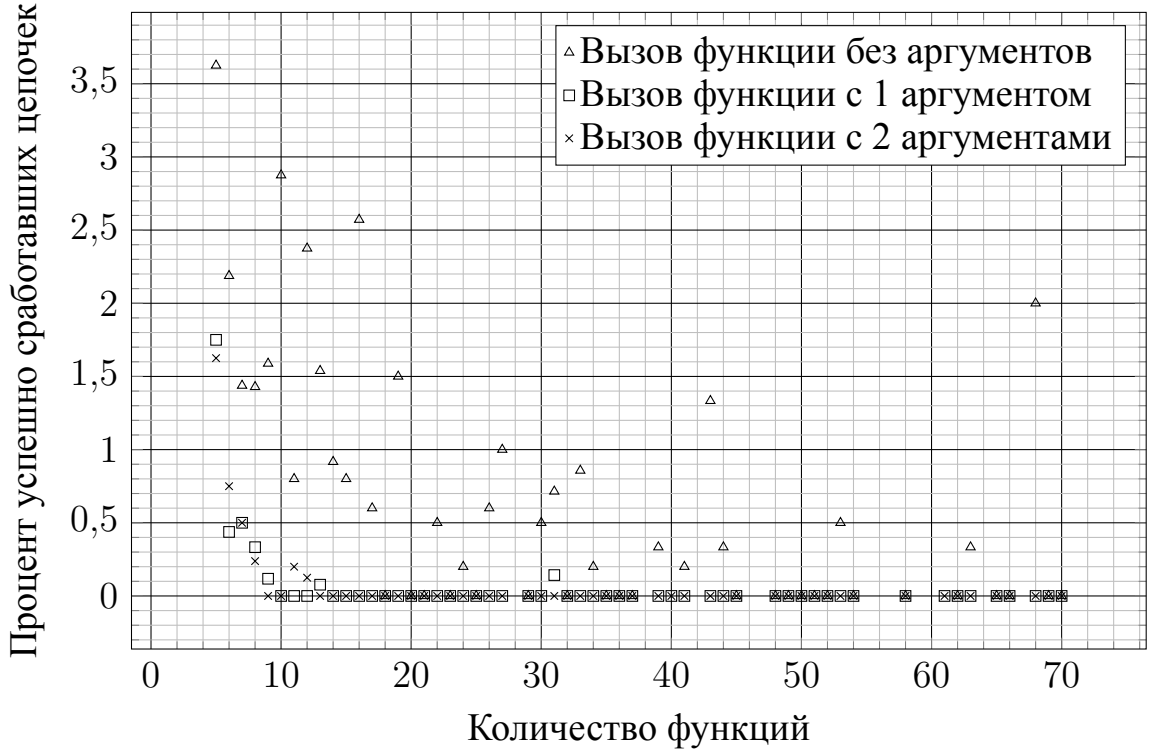

Рисунок 4.5 — Зависимость процента успешно сработавших ROP-цепочек в зависимости от количества функций в тестируемой программе для разных типов модельной нагрузки для серии измерений с средней целевой функцией (СЦ).

Для наглядности зависимость $1/n$ изображена на этом рисунке сплошной линией. Кроме того, по данным графикам можно сделать вывод, что предположение, сделанное при подсчете таблицы 6, которое формулировалось как — только исходное расположение функций приводит к успешной атаке является слишком строгим и завышает количество попыток, необходимых для успешного запуска ROP-цепочки, особенно для случаев, когда цепочка состоит из небольшого количества гаджетов.

На графиках 4.4, 4.5 изображены только зависимости для модельных нагрузок первых трех типов по сложности по причине того, что остальные два типа нагрузок (вызов функции с тремя аргументами и вызов оболочки системного интерпретатора) нет смысла изображать на этих графиках по причине отсутствия успешных запусков ROP-цепочек на этих типах. Такие тривиальные зависимости для обоих графиков просто бы лежали на оси абсцисс и затрудняли чтение других графиков.

На рисунке 4.6 изображена усредненная зависимость статистики эффективности противодействия. По оси абсцисс отложены пять точек, соответствующих модельным нагрузкам: вызов функции без аргумента, вызов функции с одним

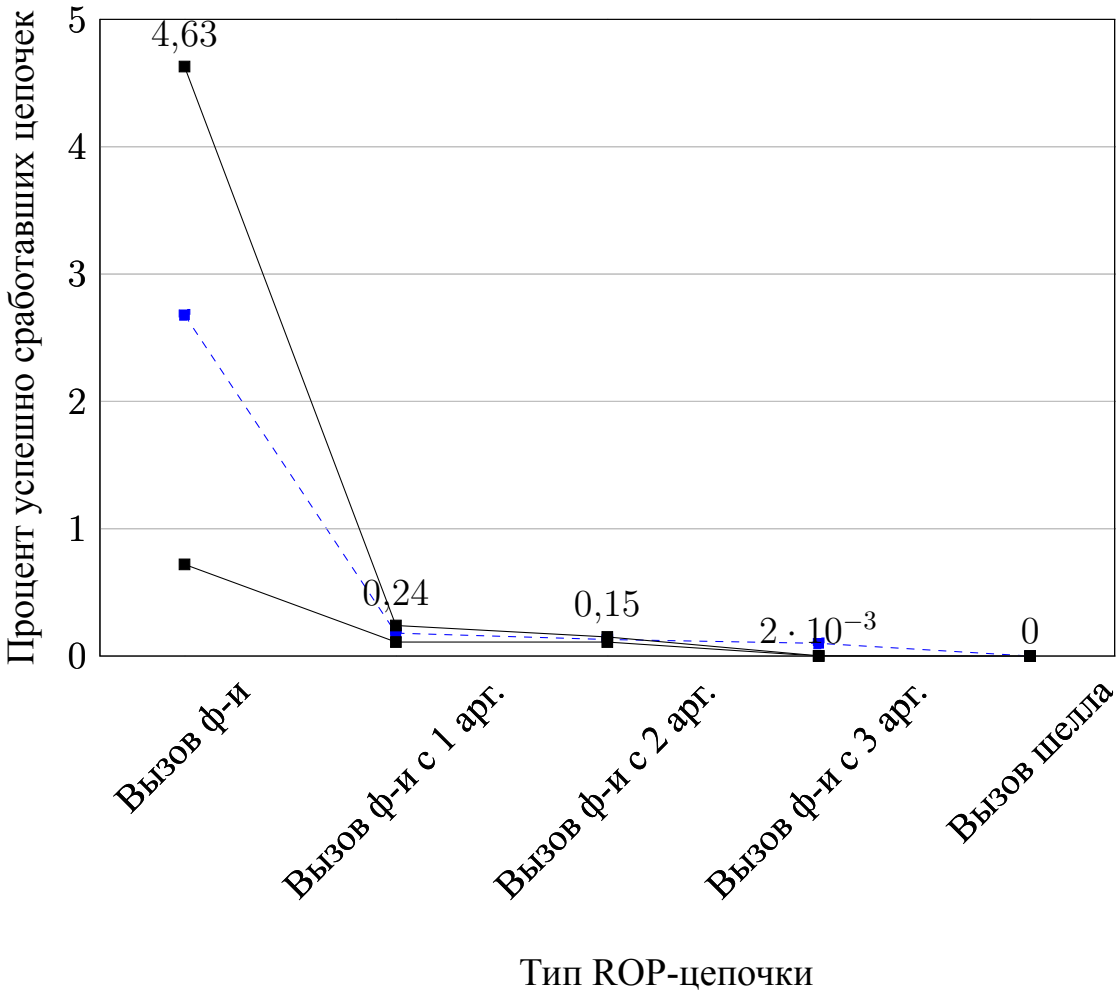

Рисунок 4.6 — Статистика по эффективности противодействия атакам повторного использования кода в виде ROP-цепочек. По оси абсцисс отложены типы созданных ROP-цепочек. Прерывистая линия изображает усредненный по обеим сериям тестов процент успешно запущенных ROP-цепочек. Верхняя сплошная линия показывает процент успешно отработавших цепочек для серии тестов с краевой целевой функцией (КЦ). Нижняя сплошная линия показывает процент успешно отработавших цепочек для серии тестов со средней целевой функцией (СЦ).

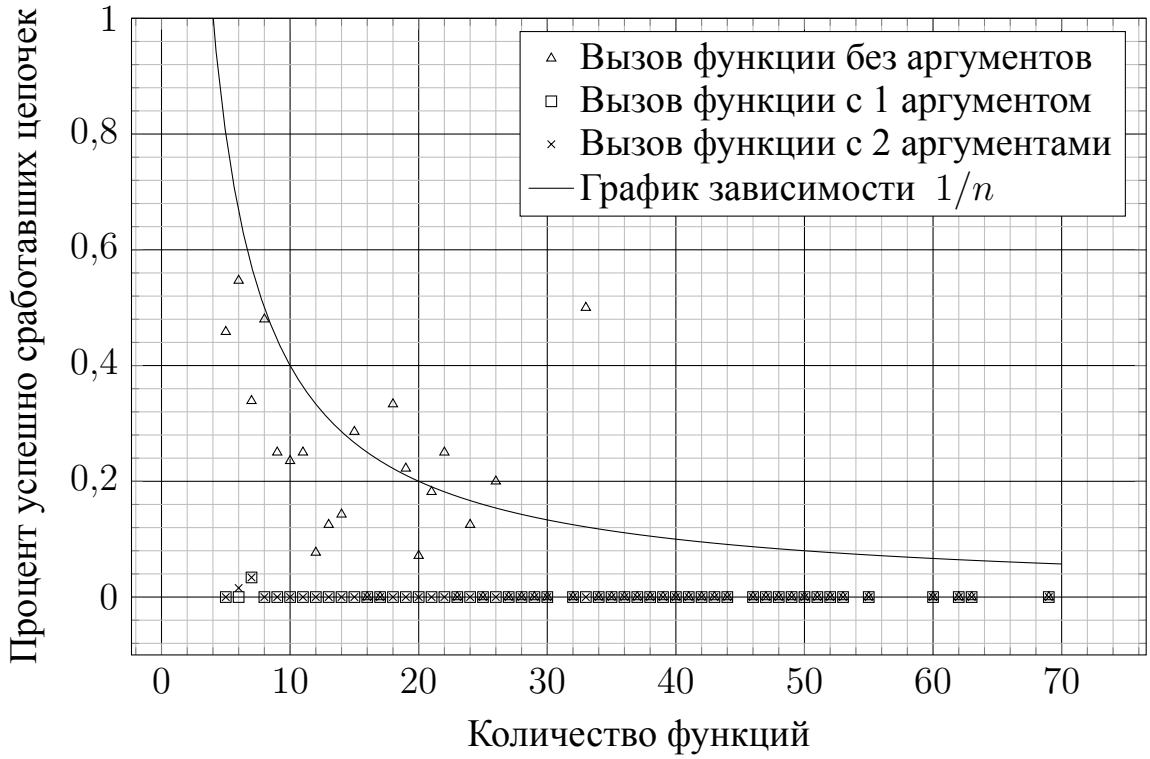

Рисунок 4.7 — Зависимость процента успешно сработавших ROP-цепочек в зависимости от количества функций в тестируемой программе для разных типов модельной нагрузки для серии измерений с краевой целевой функцией (КЦ).

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.

По приведенным результатам экспериментов видно, однако, что процент успешно сработавших цепочек (КЦ) для вызова функции без аргумента (рис. 4.4) остается значительным в том числе для исполняемых файлов, содержащих значительное количество функций (десятки). Кроме того, из рис. 4.6 и рис. 4.4 – 4.5 хорошо видно, что статистика эффективности и процент успешности сильно раз-

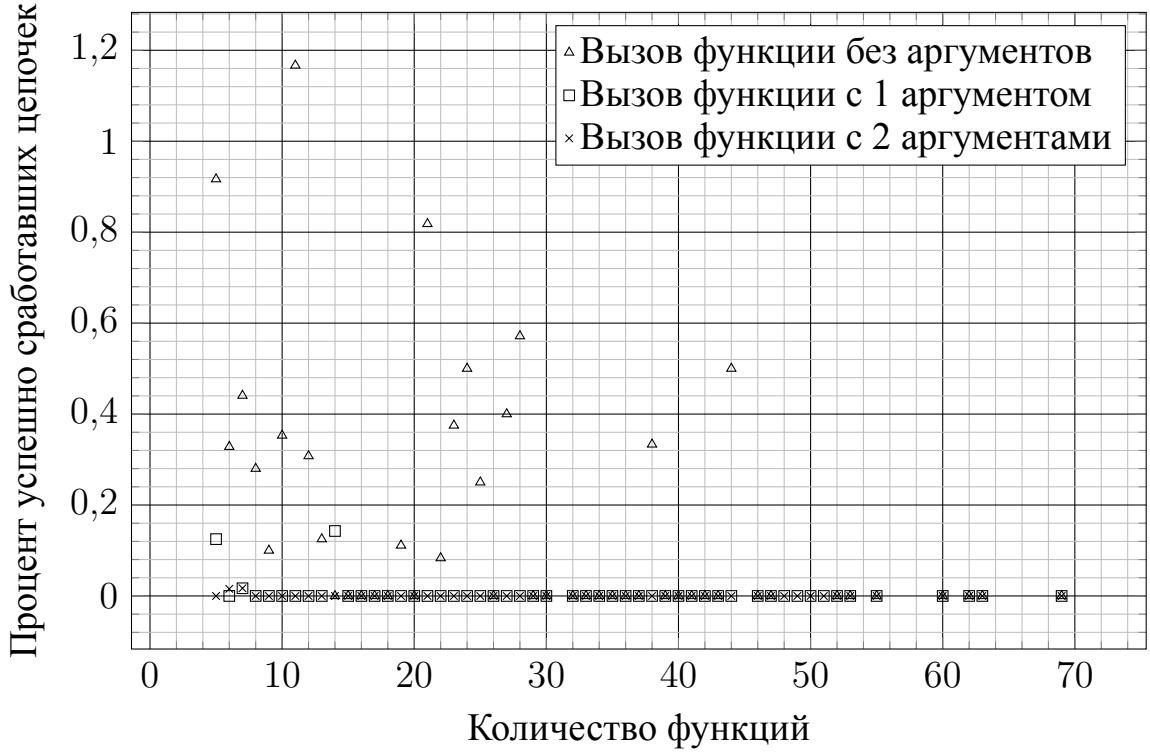

Рисунок 4.8 — Зависимость процента успешно сработавших ROP-цепочек в зависимости от количества функций в тестируемой программе для разных типов модельной нагрузки для серии измерений с средней целевой функцией (СЦ).

нятся для разных серий испытаний. Результаты успешности по серии испытания с крайней целевой (КЦ) функцией сильно превосходит результаты успешности по серии испытаний с средней целевой функцией (СЦ). Данная неоднородность вместе с немалыми величинами успешности (даже СЦ серия для вызова функции без аргументов превосходит 1 %) подтолкнули к идее улучшения предложенного метода рандомизации адресного пространства программы при запуске.

Идея улучшения заключается в следующем — добавить к перемешиванию порядка сдвиг базового адреса. В главе 3.2.4 описывается модернизация самого метода. На рис. 4.7, 4.8, 4.9 приводятся результаты тех же самых процедур экспериментальной проверки для модернизированного метода рандомизации адресного пространства. По результатам экспериментов хорошо видно, что результаты по проценту успешно сработавших ROP-цепочек значительно уменьшились по сравнению с рис. 4.4, 4.5 (все результаты даже для самых маленьких исполняемых файлов менее 1 %). Также средние значения успешности, приведенные на рис. 4.9, заметно меньше соответствующих значений, приведенных на рис. 4.6, а их абсолютное значение не превышает 0,3 %. По рис. 4.9 и рис. 4.7, 4.8 видно, что пропала неоднородность между сериями измерений с крайней целевой и сред-

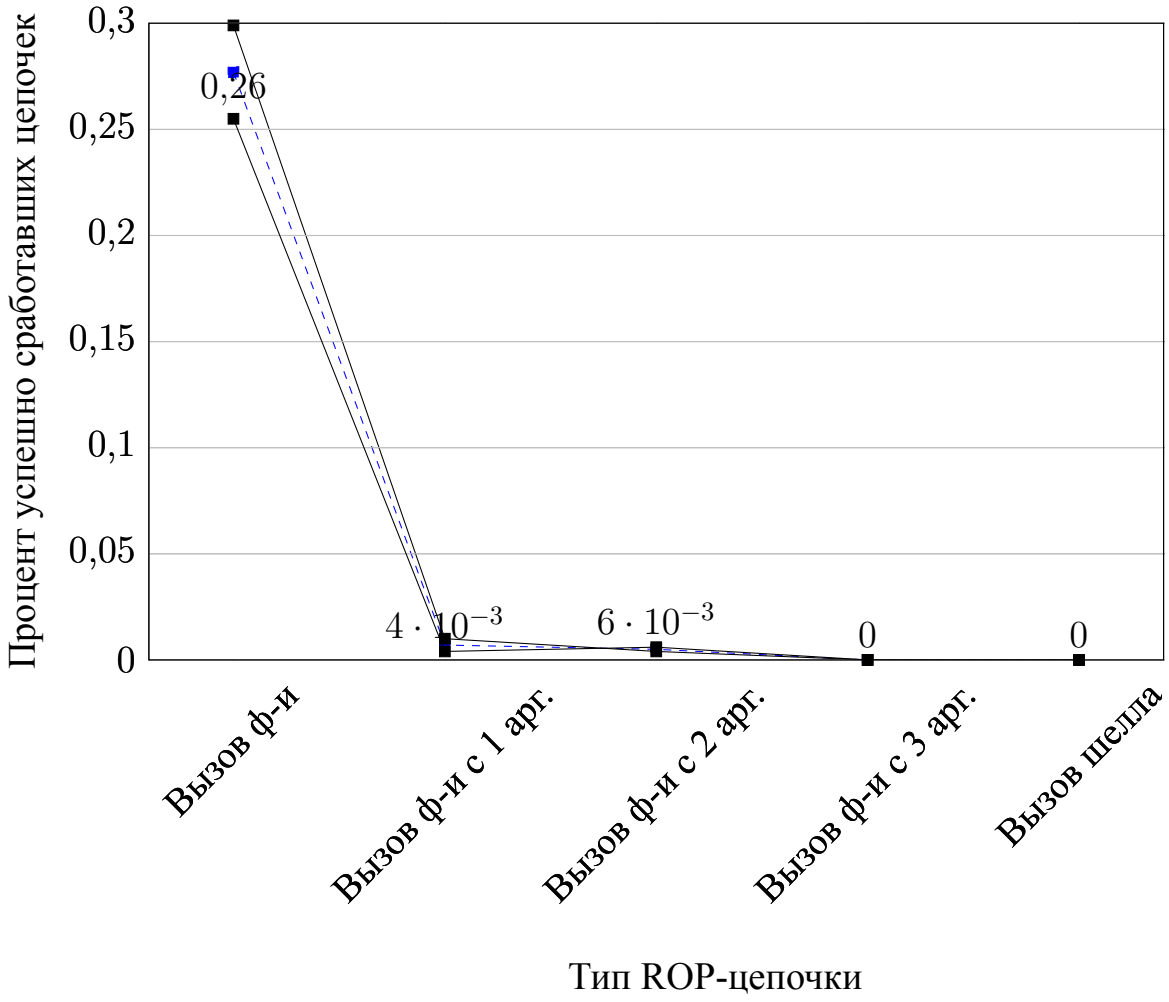

Рисунок 4.9 — Статистика по эффективности противодействия атакам повторного использования кода в виде ROP-цепочек для улучшенного метода. По оси абсцисс отложены типы созданных ROP-цепочек. Прерывистая линия изображает усредненный по обеим сериям тестов процент успешно запущенных ROP-цепочек. Верхняя сплошная линия показывает процент успешно отработавших цепочек для серии тестов с краевой целевой функцией (КЦ). Нижняя сплошная линия показывает процент успешно отработавших цепочек для серии тестов со средней целевой функцией (СЦ).

ней целевой функцией. После модернизации метода результаты успешности для серий КЦ и СЦ не отличаются, и все показатели значительно снизились. Только благодаря математическому анализу и экспериментальной проверке удалось выявить и устранить нетривиальные свойства и недостатки первичной реализации мелкозернистой рандомизации.

4.4 Работоспособность метода на реальных примерах уязвимостей

Эффективность предложенного метода была протестирована на реальных уязвимостях из базы данных CVE:

- gv CVE-2004-1717,
- rsync CVE-2004-2093,
- proftpd CVE-2006-6563,
- opendchub CVE-2010-1147,
- nginx CVE-2013-2028,
- torque CVE-2014-0749.

Для каждого примера был создан эксплойт в формате возвратно-ориентированного программирования. Было показано, что мелкозернистая рандомизация позволяла предотвращать эксплуатацию уязвимых приложений.

Глава 5. Случайные перестановки функций местами

В данном разделе проведем исследование задачи о случайных перестановках функций внутри исполняемого файла с точки зрения математической теории. Начнем рассмотрение с двух важных частных случаев с накладыванием изначального ограничения на длины функций. Затем можно будет перейти от них к общему случаю различных по длине функций.

Будем считать, что исполняемый файл состоит из n функций. Пространство элементарных исходов Ω состоит из всевозможных перестановок σ функций местами, общее количество которых $n!$. Будем считать, что все перестановки равновероятны, т.е.

$$\forall \sigma \in \Omega \quad P(\sigma) = \frac{1}{n!} \tag{5.1}$$

Будем называть перестановку σ, случайно взятую из Ω, случайной перестановкой. Перестановку можно записывать в следующем виде:

$$\begin{pmatrix} f_1 & f_2 & \dots & f_n \\ f'_1 & f'_2 & \dots & f'_n \end{pmatrix} \tag{5.2}$$

Тогда неподвижной точкой будет называться такая точка под номером i, для которой справедливо $f_i = f'_i$.

Задача 1. Пусть в исполняемом файле имеется n функций одинаковой длины. Найдем математическое ожидание числа неподвижных точек при случайной перестановке функций местами.

Не ограничивая общности можно считать, что их длины равны 1. Обозначим случайную величину, равную количеству неподвижных точек, как X. Ее можно выразить следующим образом:

$$X = \sum_{i=1}^{n} X_i \quad , \tag{5.3}$$

где X_i — случайная величина, которая определяется формулой:

$$X_i(\sigma) = \begin{cases} 1, & \text{если} \quad f_i = f'_i \\ 0, & \text{если} \quad f_i \neq f'_i \end{cases} \tag{5.4}$$

Откуда следует, принимая во внимание свойство линейности математического ожидания, следующее:

$$MX = M(\sum_{i=1}^{n} X_i) = \sum_{i=1}^{n} MX_i \tag{5.5}$$

Вычислим математическое ожидание дискретной случайной величины X_i на основе ее определения 5.4.

$$MX_i = \sum_{\sigma \in \Omega} X_i(\sigma) P(\sigma) = \frac{1}{n!} \sum_{\sigma \in \Omega} X_i(\sigma) = \frac{(n-1)!}{n!} = \frac{1}{n} \tag{5.6}$$

Заметим, что при вычислении $\sum X_i(\sigma)$ достаточно посчитать количество перестановок, которые оставляют функцию f_i на месте, потому что значение случайной величины для других элементарных исходов равно 0. Количество таких перестановок равно $(n-1)!$. Подставляя полученное значение в формулу 5.5 и суммируя от 1 до n, получаем ответ.

$$MX = \sum_{i=1}^{n} \frac{1}{n} = n \cdot \frac{1}{n} = 1 \tag{5.7}$$

Удивительно, но факт заключается в том, что математическое ожидание количества неподвижных функций при случайной перестановке в предположении о равенстве длин функций не зависит от числа функций n. Эта величина всегда равна 1.

Однако, в реальной жизни, за исключением синтетических примеров, не встречаются исполняемые файлы с функциями равной длины. Поэтому необходимо ослабить введенное ограничение на равенство длин функций. Пусть для исполняемого файла задана дискретная функция $\mathcal{L}$, определяющая для данной функции f ее длину L, так что $L = \mathcal{L}(f)$. Значениями данной функции могут быть только положительные целые числа. Тогда для перестановки 5.2 определение неподвижности функции под номером i будет требовать соблюдения двух условий:

$$f_k = f'_m, \quad \sum_{i=1}^{k-1} \mathcal{L}(f_i) = \sum_{i=1}^{m-1} \mathcal{L}(f'_i) \tag{5.8}$$

В исполняемом файле не существует двух функций стоящих на одном и том же месте, т.е.:

$$\forall k, m \in [1,n] : k \neq m \rightarrow \sum_{i=1}^{k-1} \mathcal{L}(f_i) \neq \sum_{i=1}^{m-1} \mathcal{L}(f_i) \tag{5.9}$$

Обозначим множество всех функций исполняемого файла $\mathfrak{F} = \{f_1, f_2, \ldots, f_n\}$ Пусть длины всех функций в исполняемом файле существенно разные, т.е.

$$\forall \mathfrak{A} \subseteq \mathfrak{F}, \forall \mathfrak{B} \subseteq \mathfrak{F} \setminus \mathfrak{A} \quad \rightarrow \quad \sum_{g \in \mathfrak{A}} \mathcal{L}(g) \neq \sum_{r \in \mathfrak{B}} \mathcal{L}(r) \tag{5.10}$$

С введенными определениями рассмотрим следующую задачу.

Задача 2. Найти математическое ожидание числа неподвижных точек для исполняемого файла, в котором все функции имеют существенно разные длины, при случайной перестановке функций местами.

Для решения данной задачи воспользуемся формулой 5.5, однако выражение 5.6 необходимо пересчитать для изменившихся условий. Наличие условия 5.10 определяет единственно возможный набор функций, стоящих слева от рассматриваемой, которой бы удовлетворял условию 5.8. Этот набор не что иное как начальный набор функций $\{f_1, f_2, \ldots, f_{i-1}\}$. Соответственно справа от функции f_i будут располагаться функции $\{f_{i+1}, f_{i+2}, \ldots, f_n\}$. Число способов которыми можно разместить соответствующие множества функции слева и справа соответственно равны $(i-1)!$ и $(n-i)!$. Всего количество перестановок, при которых функция с номером i неподвижна, равно $(i-1)!(n-i)!$. Откуда следует, что:

$$MX_i = \frac{(i-1)!(n-i)!}{n!} \tag{5.11}$$

Подставляя это выражение в 5.5 имеем:

$$MX = \sum_{i=1}^{n} \frac{(i-1)!(n-i)!}{n!} \tag{5.12}$$

Покажем, что для любого n математическое ожидание количества неподвижных функций при условии существенно разных длин функций MX_d 5.12 не больше, чем математическое ожидание в случае равенства длин функций MX_c 5.7, т.е. :

$$\forall n \rightarrow MX_d \leq MX_c \tag{5.13}$$

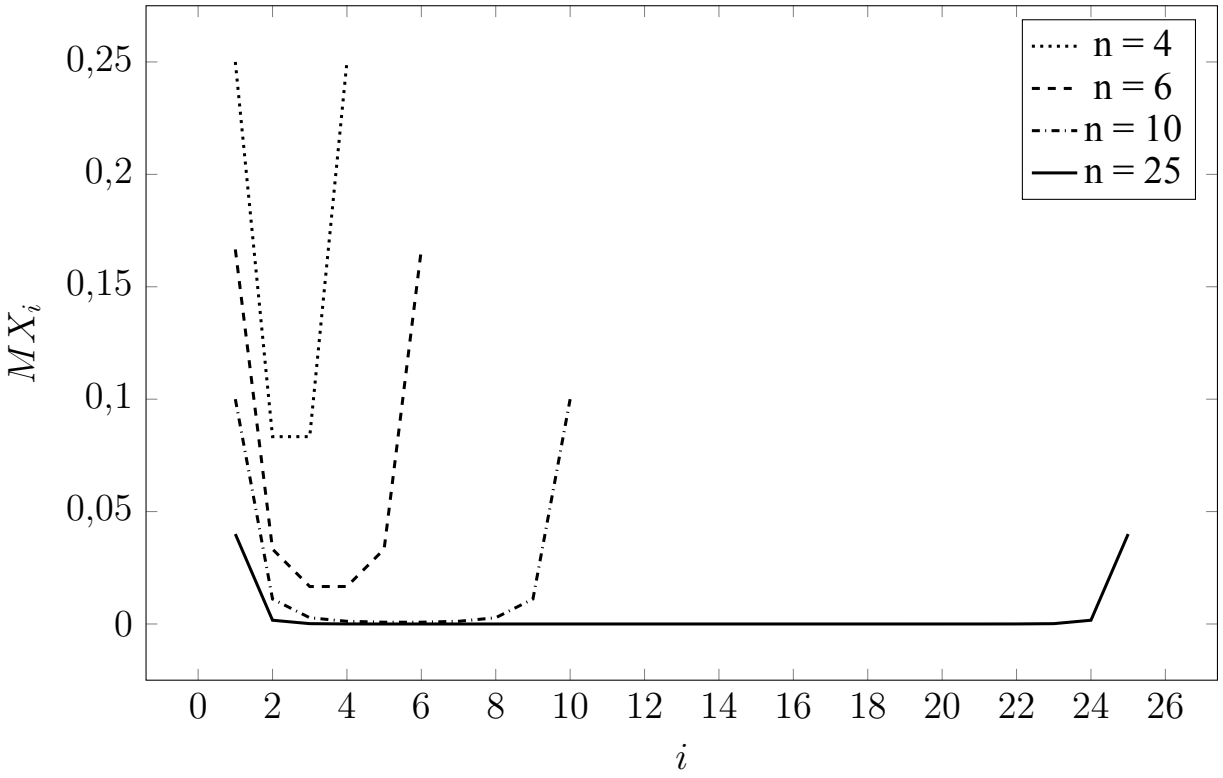

Рисунок 5.1 — Графики зависимости вероятности функции под номером i остаться на своем месте в предположении о существенном различии их длин для $n \in [4{,}6{,}10{,}25]$

Покажем справедливость этого неравенства:

$$\begin{aligned}
\sum_{i=1}^{n} \frac{(i-1)!(n-i)!}{n!} &\;?\; \sum_{i=1}^{n} \frac{(n-1)!}{n!} \\
\frac{(i-1)!(n-i)!}{n!} &\;?\; \frac{(n-1)!}{n!} \\
(i-1)!(n-i)! &\;?\; (n-1)! \\
(i-1)!(n-i)! &\;?\; (n-i)!(n-i+1)\dots(n-1) \\
(i-1)! &\;?\; !(n-(i-1))\dots(n-1) \\
\prod_{k=1}^{i-1}(i-k) &\;?\; \prod_{k=1}^{i-1}(n-k) \\
i-k &\;?\; n-k \\
i &\;\leq\; n \qquad (5.14)
\end{aligned}$$

(5.15)

Неравенство 5.14 верно, потому что $i \in [1,n]$, следовательно верно и неравенство 5.13.

На рисунке 5.1 изображена зависимость вероятности функции под номером i остаться на своем месте для значений $n \in [4,6,10,25]$. На данном графие хорошо видно, что вероятность крайних функций остаться на своем месте значительно выше, чем функций, находящихся в середине исполянемого файла. Для преодоления этого недостатка необходимо комбинировать перестановки функций с добавлениями промежутков.

Обобщим полученные знания для случая, когда никакого дополнительного ограничения на длины функций в исполняемом файле не наложено.

Задача 3. Найти математическое ожидание числа неподвижных точек для исполняемого файла при случайной перестановке функций местами.

Заметим, что в отличии от задачи 2 из-за ослабления условия 5.10, может существовать несколько возможных наборов функций, стоящих слева от рассматриваемой, которые бы удовлетворяли условию 5.8. Причем количество этих функций может быть, вообще говоря, любым от 0 до $n-1$. Необходимо перебрать всевозможные такие наборы, каждый из которых проверить на соответствие условию 5.8. В случае если набор размера $k-1$ подходит, то перестановок с таким набором слева от рассматриваемой функции будет $(k-1)!(n-k)!$. Соответственно выражение 5.11 принимает вид:

$$MX_i = \sum_{k=1}^{n} \frac{\zeta_i(k)(k-1)!(n-k)!}{n!} \tag{5.16}$$

Чтобы определить $\zeta_i(k)$ обозначим через $\mathfrak{C}_n^k$ множество всех способов неупорядоченно выбрать из n функций k штук. Количество его элементов соответственно равно $|\mathfrak{C}_n^k| = C_n^k$. Тогда

$$\zeta_i(k) = \sum_{\mathfrak{a} \in \mathfrak{C}_{n-1}^{k-1}} \varphi_i(\mathfrak{a}) \tag{5.17}$$

$$\varphi_i(\mathfrak{a}) = \begin{cases} 1, & \text{если} \quad \sum\limits_{g \in \mathfrak{a}} \mathcal{L}(g) = \sum\limits_{i=1}^{k-1} \mathcal{L}(f_i) \\ 0, & \text{иначе} \end{cases}$$

Подставляя выражение для MX_i в 5.5, получаем:

$$MX = \frac{1}{n!}\sum_{i=1}^{n}\sum_{k=1}^{n}\zeta_i(k)(k-1)!(n-k)! = \frac{1}{n!}\sum_{k=1}^{n}(k-1)!(n-k)!\sum_{i=1}^{n}\zeta_i(k) \quad (5.18)$$

Заметим следующее свойство дискретной функции $\zeta_i(k)$:

$$1 \leq \sum_{i=1}^{n}\zeta_i(k) \leq C_{n-1}^{k-1} \quad (5.19)$$

Левое неравенство справедливо в силу того, что всегда имеется как минимум один набор функций стоящих слева от рассматриваемой под номером i, а именно начальный $\mathfrak{a}$ для которого $\varphi(\mathfrak{a}) = 1$.

Для того чтобы показать справедливость правой части неравенства 5.19, заметим в силу свойства 5.9 следующее:

$$\forall \mathfrak{a}, \forall i, j : i \neq j \rightarrow \varphi_i(\mathfrak{a}) \neq \varphi_j(\mathfrak{a}) \quad (5.20)$$

Это означает, что для фиксированного набора функций $\mathfrak{a}$ существует не более одной функции, слева от которой может разместить $\mathfrak{a}$ с соблюдением условия $\sum\limits_{g\in\mathfrak{a}}\mathcal{L}(g) = \sum\limits_{i=1}^{k-1}\mathcal{L}(f_i)$, т.е. $\sum\limits_{i=1}^{n}\varphi_i(\mathfrak{a}) \leq 1$, откуда следует:

$$\sum_{i=1}^{n}\zeta_i(k) = \sum_{i=1}^{n}\sum_{\mathfrak{a}\in\mathfrak{C}_{n-1}^{k-1}}\varphi_i(\mathfrak{a}) = \sum_{\mathfrak{a}\in\mathfrak{C}_{n-1}^{k-1}}\sum_{i=1}^{n}\varphi_i(\mathfrak{a}) \leq \sum_{\mathfrak{a}\in\mathfrak{C}_{n-1}^{k-1}} 1 = C_{n-1}^{k-1} \quad (5.21)$$

Что и доказывает правую часть неравенства 5.19.

Покажем, что для любого n математическое ожидание количества неподвижных функций MX_r 5.18 не меньше, чем математическое ожидание количества неподвижных функций при условии существенно разных длин функций MX_d 5.12, т.е. :

$$\forall n \rightarrow MX_d \leq MX_r \quad (5.22)$$

Подставляя выражения (5.12, 5.18), получаем:

$$\begin{aligned}
\sum_{i=1}^{n}\frac{(i-1)!(n-i)!}{n!} \quad &? \quad \frac{1}{n!}\sum_{k=1}^{n}(k-1)!(n-k)!\sum_{i=1}^{n}\zeta_i(k)\\
\sum_{k=1}^{n}(k-1)!(n-k)! \quad &? \quad \sum_{k=1}^{n}(k-1)!(n-k)!\sum_{i=1}^{n}\zeta_i(k)\\
\forall k:(k-1)!(n-k)! \quad &? \quad (k-1)!(n-k)!\sum_{i=1}^{n}\zeta_i(k)\\
\forall k:1 \quad &\leq \quad \sum_{i=1}^{n}\zeta_i(k)
\end{aligned} \tag{5.23}$$

Заметим, что при доказательстве данного неравенства в левой части использовалось изменение имени счетчика суммы, которое очевидно не меняет саму сумму. А затем каждый член суммы слева сравнивался с соответствующим ему по номеру членом суммы справа. Очевидно, что если каждый член суммы меньше каждого соответствующего члена другой суммы, то и вся сумма меньше. Данные замечания приводят нас к неравенству 5.23, справедливость которого доказана ранее в выражении 5.19, которое и доказывает искомое 5.22.

Покажем также, что для любого n математическое ожидание количества неподвижных функций MX_r 5.18 не больше, чем математическое ожидание числа неподвижных функций при условии равенства длин функций MX_c 5.7, т.е.:

$$\forall n \to MX_r \leq MX_c \tag{5.24}$$

Подставляя выражения (5.7, 5.18), получаем:

$$\begin{aligned}
\frac{1}{n!}\sum_{k=1}^{n}(k-1)!(n-k)!\sum_{i=1}^{n}\zeta_i(k) \quad &? \quad \sum_{i=1}^{n}\frac{(n-1)!}{n!}\\
\forall k:(k-1)!(n-k)!\sum_{i=1}^{n}\zeta_i(k) \quad &? \quad (n-1)!\\
\forall k:\sum_{i=1}^{n}\zeta_i(k) \quad &? \quad \frac{(n-1)!}{(k-1)!(n-k)!}\\
\forall k:\sum_{i=1}^{n}\zeta_i(k) \quad &\leq \quad C_{n-1}^{k-1}
\end{aligned} \tag{5.25}$$

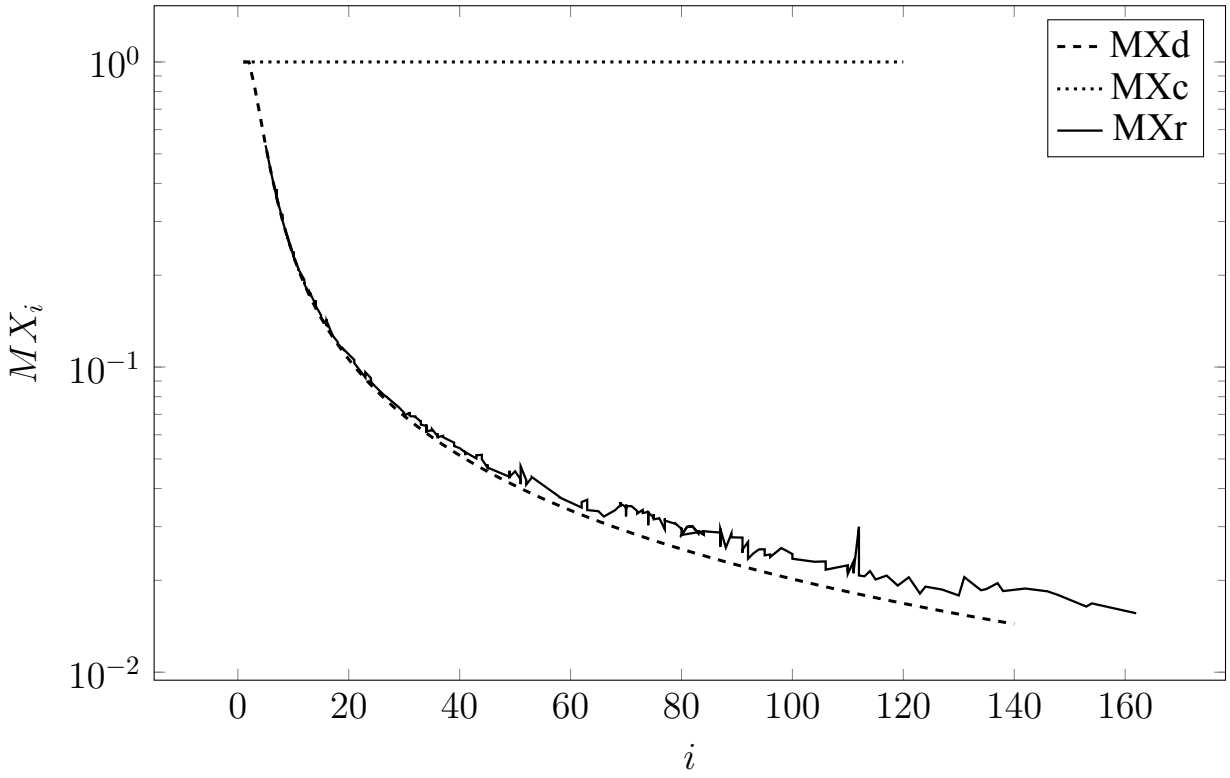

Рисунок 5.2 — Математические ожидания количества неподвижных точек для реальных программ в сравнении с двумя крайними ситуациями

Справедливость неравенства 5.25 прямо следует из 5.19, что и доказывает утверждение 5.24. Суммируя полученный результат с доказанным ранее утверждением 5.23, получаем:

$$\forall n \to MX_d \leq MX_r \leq MX_c \tag{5.26}$$

Это означает, что математическое ожидание числа функций, остающихся на своем месте в реальных исполняемых файлах при перестановке их местами находится всегда между значением математического ожидания для случая совершенно разных длин функций и математического ожидания для случая равных по длине функций. Для того, чтобы понять насколько реальные файлы близки к тому или иному случая обратимся к графику этих величин, изображенному на рисунке 5.2. На данном графике представлены три кривые. Кривой MXc представлен крайний случай с функциями равной длины. Кривая MXd представляет собой случай с функциями в предположениии о существенном различии их длин. Кривая MXr представляет собой кривую построенную по формуле 5.18 на основе реальных данных из приложений, которая вычислялась программным путём с помощью

перебора всех возможных перестановок функций. По представленному графику действительно видно, что неравенство 5.26 справедливо.

Глава 6. Программная реализация

Предлагаемые методы диверсификации программного кода были реализованы в рамках семейства операционных систем с открытым исходным кодом семейства Linux на базе аппаратной платформы `x86_64`. В рамках работы по коммерческому контракту с компанией ЗАО «МВП «СВЕМЕЛ» производилась адаптация реализованных методов под ОС CentOS 7 и Debian 10. Акт о внедрении результатов данной работы находится в приложении В. На разработки получены сертификаты о государственной регистрации программ для ЭВМ:

– обфусцирующий компилятор для затруднения эксплуатации уязвимостей № 2016661393 (приложение А)
– инструмент «Faslr» для усиления системной защиты при запуске программ в ОС Linux № 2017660041 (приложение Б).

Требования практической применимости реализации и совместимости с системными средствами защиты, а также необходимость поддерживать полносистемный сценарий защиты, при котором защите подвергаются все приложения операционной системы, накладывали ограничения на выбор средств и путей реализации. Предлагаемые в данной работе методы были реализованы в качестве дополнительной функциональности системных утилит, средств разработки, элементов среды времени исполнения, и а также ядра ОС. Архитектура реализованной системы дополнительной защиты представлена на рисунке 6.1.

Дополнительная функциональность компилятора описывается в разделе 6.1. Дополнительная функциональность компоновщика описывается в разделе 6.2. Дополнительная функциональность загрузчика описывается в разделе 6.3. Дополнительная функциональность ядра ОС описывается в разделе 6.4.

6.1 Дополнительная функциональность компилятора

Базовым компилятором, осуществляющим сборку полной операционной системы, является компилятор GCC. Для реализации описанных и разработанных в главе 2 *методов диверсификации программного кода на уровне промежуточно-*

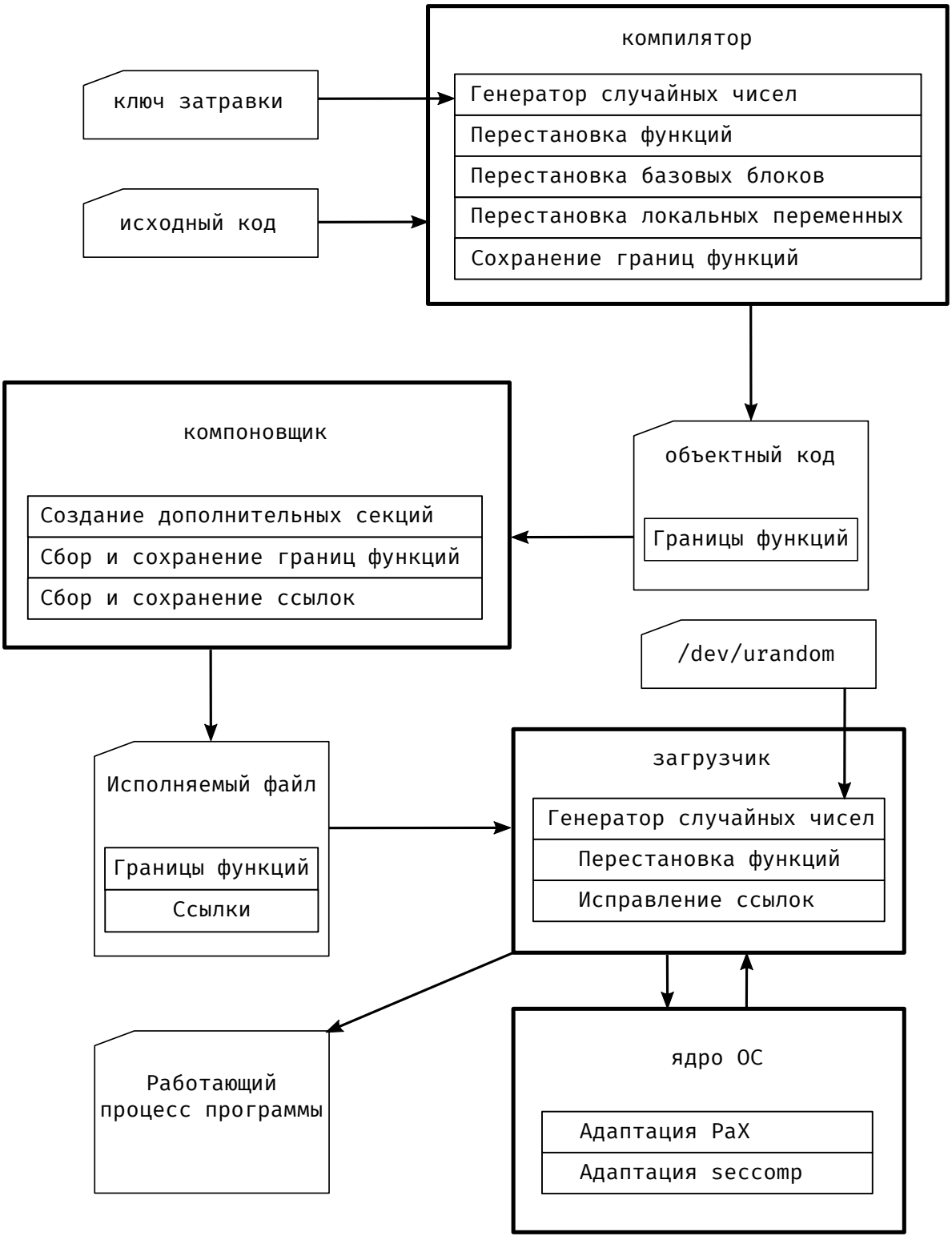

Рисунок 6.1 — Архитектура реализованной системы дополнительной защиты.

го представления компилятора для генерации диверсифицированной популяции исполняемых файлов в компилятор GCC были добавлены дополнительные модули:

1. Модуль генератора случайных чисел.
2. Модуль преобразования для перестановки местами функций в пределах единицы трансляции.
3. Модуль преобразования для добавления и перестановки локальных переменных.
4. Модуль преобразования для перестановки местами базовых блоков в пределах функции.

Модуль генератора случайных чисел предоставляет следующий программный интерфейс:

1. функцию для установки затравки генератора случайных чисел, которая используется драйвером компилятора для обеспечения задания затравки пользователем через опцию командной строки,
2. функцию для получения случайного числа, которая используется модулями преобразований.

Модуль генератора случайных чисел реализует псевдо-генератор случайных чисел, подобный используемому в стандартной библиотеке Си. Он был вынесен в отдельный модуль из-за требования его возможной заменяемости по желанию заказчика на криптографический стойкий и/или поддерживаемый со стороны аппаратуры быстрым истинным источником случайности.

Модуль преобразования перестановки местами функций в пределах единицы трансляции осуществляет:

1. перестановку местами функций, определённых в данной единице трансляции;
2. перестановку местами глобальных переменных, определённых в данной единице трансляции.

Данный модуль предоставляет пользователю две независимые настройки через ключи командной строки. Одна управляет тем включена ли перестановка функций, а другая управляет перестановкой глобальных символов.

Перестановка объектов осуществляется следующим образом:

– Производится обход всех объектов, требующих перестановки.

– Для каждого объекта создаётся структура, содержащая два поля указатель на объект (или идентификатор) и случайной число.
– Такие структуры накапливаются в вспомогательном массив.
– Этот массив сортируется с помощью `qsort` с использованием специального компаратора, сравнивающего только второе поле структуры (случайное число).
– По вспомогательному массиву изменяется соответствующим образом порядок объектов в исходной коллекции.

Модуль преобразования для добавления и перестановки локальных переменных осуществляет для каждой функции:

1. Добавление локальных переменных, количество которых случайно и не превышает некоторого порогового значения заданного пользователем через опцию командной строки.
2. Перемешивание имеющихся в функции и добавленных локальных переменных в порядке следования на стеке.

Перестановка осуществляется аналогичным предыдущему преобразованию способом.

Каждый блок в компиляторе GCC имеет специальную характеристику, используемую оптимизацией, которая раскладывает блоки по функции оптимальным образом, чтобы поместить рядом часто используемые блоки для уменьшения промахов кэша. Изменение этой характеристики случайным образом позволяет добиться эффекта перестановки базовых блоков местами. Модуль преобразования для перестановки базовых блоков в пределах функции осуществляет для каждой функции:

– обход всех базовых блоков функции,
– присваивание описанной выше специальной характеристике случайное значение.

Такое преобразование должно быть встроено в последовательность преобразований в компиляторе GCC в правильном месте, т.е. после компиляторного прохода, который присваивает веса базовым блокам функций, но перед тем как производится оптимальная их раскладка в линейную последовательность.

Кроме того, для реализации сохранения информации о границах функций в объектный файл с этапа трансляции был использована имеющаяся функциональность компилятора, включаемая ключом командной строки

`-ffunction-sections`. Данная функциональность заставляет компилятор разместить каждую функцию программы в отдельную секцию объектного файла. Таким образом информация о границах функций может быть получена в компоновщике из информации о секция исполняемого файла, которая ему всегда доступна. Сохранение информации о границах функций необходимо для реализации *метода диверсификации внутренней структуры исполняемого файла при загрузке его в память во время запуска приложения*, который описывается в главе 3.

6.2 Дополнительная функциональность компоновщика

Стандартным компоновщиком ОС семейства Linux является ld из пакета binutils. Компоновщик осуществляет статическую компоновку перемещаемого объектного кода, созданного компилятором как результат процесса трансляции.

Для реализации описанного в главе 3 *метода диверсификации внутренней структуры исполняемого файла при загрузке его в память во время запуска приложения* в компоновщик была добавлена дополнительная функциональность, которая позволяет сохранить в исполняемом файле формата ELF информацию о границах функций и ссылках. Для этого компоновщик:

1. создаёт дополнительную секцию `.reloc_infos`;
2. создаёт дополнительную секцию `.note.reloc_infos`;
3. собирает и сохраняет информацию о границах функций в секцию `.reloc_infos`;
4. собирает и сохраняет информацию о ссылках на символы в секцию `.reloc_infos`;

Создание дополнительных секций в выходном исполняемом файле происходит с помощью программного интерфейса библиотеки bfd, которая используется в binutils для абстрагирования бинарных форматов выходных исполняемых файлов. Секция `.note.reloc_infos` добавляется в сегмент `NOTE`, имеет фиксированный размер, и специализированный тип (по которому её можно найти в загрузчике 6.3. В этой секции содержатся:

– виртуальный адрес секции `.reloc_infos` во время выполнения программы,

– размер секции `.reloc_infos`.

Формат секции `.reloc_infos` описан в главе 3.2.1. Для создания секции `.reloc_infos` необходимо проделать следующую последовательность действий:

1. Посчитать количество функций в выходном исполняемом файле.
2. Посчитать количество ссылок, требующих исправления:
   – 1 слот для динамических символов, `DT_INI`, `DT_FINI`;
   – 2 слота для динамических ссылок (адрес и поправка);
   – 1 слот для ссылки PLT (поправка для `R_*_IRELATIVE`);
   – 1 слот для ссылки `(REX_)GOTPCRELX`.
3. Найти наименьшее общее кратное выравнивания всех функций, и уравнять по нему выравнивание для всех функций для избежания проблем при перестановке.

Секция `.reloc_infos` создаётся в тот момент, когда точно неизвестно количество ссылок на символы, которые реально требуют исправления во время загрузки. По этой причине используется оценка сверху от этого количества (количество всех ссылок). Это приводит к образованию свободного места в конце секции, которое используется в загрузчике при реализации сдвига базового адреса всех функций. Это требует также фиксации позиции секции `.reloc_infos` перед секцией `.text`, что сделано путём изменения стандартного скрипты компоновщика.

После создания секции происходит её заполнение. Сбор информации о границах функций происходит путём обхода каждой исполняемой секции входного объектного файла, для каждой секции компоновщик сохраняет:

1. адрес,
2. размер,
3. выравнивание.

Сбор информации о ссылках распределён по разным местам компоновщика, соответствующим обработчикам разных типов ссылок. В момент обработки компоновщиком ссылки для неё вычисляется целевой адрес. Этот адрес и адрес по которому расположена сама ссылка сохраняются в секции `.reloc_infos`. Особой обработки потребовали ссылки, которые имеют отношение к реализации TLS и `.eh_frame`.

После того, как собрана вся информация о ссылках и функциях, производится нумерация функций. Затем оба виртуальных адреса, характеризующих ссылку,

заменяются на номера функций, в которых они находятся и на которые они указывают. В ходе этой процедуры выясняется, что некоторые ссылки оказываются ненужными. Такие ссылки удаляются, к ним относятся:

1. относительные ссылки, которые указывают в ту же самую функцию, в которой они и находятся;
2. абсолютные ссылки, которые не указывают ни в одну из функций.

Кроме того, для удобства разработчиков в утилиту `readelf` добавлена функциональность, позволяющая печатать содержимое дополнительных секций.

6.3 Дополнительная функциональность загрузчика

Стандартным загрузчиком ОС семейства Linux является `ld.so` из пакета стандартной библиотеки языка Си `glibc`. Загрузчик осуществляет загрузку в память запускаемого процесса необходимые динамические библиотеки и осуществляет с ними динамическую компоновку запускаемого исполняемого файла.

Для реализации описанного в главе 3 *метода диверсификации внутренней структуры исполняемого файла при загрузке его в память во время запуска приложения* в загрузчик была добавлена дополнительная функциональность, которая:

1. находит в адресном пространстве процесса секцию `.reloc_infos`,
2. перемешивает функции местами и исправляет значения ссылок.

Кроме того, по аналогии с компилятором был добавлен модуль генератора случайных чисел (см. 6.1, который предоставляет схожий программный интерфейс для функциональности перемешивания функций. По умолчанию, затравка берётся из псевдо-файла `/dev/urandom`. Стоит заметить, что скорость чтения затравки из этого файла является критичной с точки зрения времени запуска программы. При замене генератора на криптографически стойкий нужно учитывать этот момент.

Функциональность поиска виртуального адреса секции `.reloc_infos` состоит из следующих шагов:

1. Из заголовка программы берётся сегмент типа `PT_NOTE`.

2. В этом сегменте происходит обход всех записей до момента нахождения записи, соответствующей созданной в компоновщике 6.2 секции `.note.reloc_infos`. При отсутствии искомой секции функциональность перестановки функций местами отключается. За счёт этого достигается совместимость модифицированного загрузчика с программами, собранными стандартным компилятором и компоновщиком без поддержки возможности рандомизации во время загрузки.
3. Из найденной секции извлекается виртуальный адрес секции `.reloc_infos` и её размер.

Функциональность перемешивания функций местами состоит из следующей последовательности действий:

1. Подсчёт свободного места в секции `.reloc_infos` и выбор значения случайного сдвига для всей секции кода как целого. Сдвиг осуществляется внутрь секции `.reloc_infos` за счёт имеющегося в ней свободного места.
2. Выделение временного буфера для хранения перемещаемых функций.
3. Изменение разрешения для секции с кодом, таким образом, чтобы она стала доступна для записи. Это осуществляется с помощью вызова `mprotect` и может вступать в конфликт с дополнительными мерами защиты ядра ОС. В таком случае требуется их адаптация для бесконфликтной работы, что описывается в главе 6.4.
4. Копирование всех функций в временный буфер.
5. Перестановка функций местами путём выбора из временного буфера функции со случайным индексом и копировании её обратно в секцию кода.
6. Обход всех ссылок, содержащихся в секции `.reloc_infos` и исправление их значений.
7. Исправление виртуального адреса точки входа в программу.
8. Обратное изменение разрешения на доступ к памяти для секции с кодом таким образом, чтобы были восстановлены исходные разрешения. Как правило это доступ только на выполнение и чтение.
9. Обнуление и освобождение памяти, занимаемой секцией `.reloc_infos`.

6.4 Дополнительная функциональность ядра ОС

В операционных системах, под которые производилась адаптация CentOS 7 и Debian 10 в ядре Linux имелись дополнительные средства защиты. Для реализации описанного в главе 3 *метода диверсификации внутренней структуры исполняемого файла при загрузке его в память во время запуска приложения* требовалась их адаптация.

Адаптации потребовали следующие защитные механизмы ядра Linux:

1. PaX [14],
2. seccomp [67].

PaX представляет собой большой набор дополнительных средств защиты ядра ОС Linux, который не включён в основную ветвь разработки. До недавнего прошлого этот набор изменений находился в открытом доступе. Среди прочего PaX содержит предотвращение внедрения нового кода в адресное пространство процесса. Это достигается за счёт дополнительных ограничения на системный вызов изменения разрешения на доступ к регионам памяти `mprotect`. PaX ограничивает возможность добавления доступа на запись к региону памяти с кодом (т.е. к региону доступному только на чтение и исполнение).

Адаптация механизма ограничения PaX `mprotect` состоит из следующих модификаций функциональности PaX:

– Добавлена функциональность поиска в сегменте `NOTE` записи из секции `.note.reloc_infos`, по содержанию которой происходит идентификация работающей мелкозернистой рандомизации при запуске.
– Добавлена функциональность, которая при наличии мелкозернистой рандомизации разрешает процессу один раз добавлять разрешение на запись для секции с кодом. По сути, для этого была полностью переиспользована функциональность PaX, которая обеспечивает работу перемещаемых библиотек, которые собраны не в формате позиционно-независимого кода (т.н. `CONFIG_PAX_ETEXECRELOCS`).

Seccomp — механизм безопасности ядра ОС Linux, который обеспечивает возможность одностороннего перехода процесса в более безопасное состояние за счёт ограничения доступных для приложения системных вызов. Фильтрация системных вызов может проводиться как просто по их номерам, так и более гибко

с помощью предоставляемых ядру BPF фильтров. BPF фильтры позволяют осуществлять фильтрацию вызовов в том числе по значениям их аргументов.

В качестве подсистемы инициализации и управления службами в Debian 10 используется `systemd`. С недавнего времени она поддерживает настройку для запускаемых служб под названием `DenyWriteExecuteMemory`, которая устанавливает `seccomp` фильтры таким образом, чтобы запрещать выполнение `mprotect` с `PAGE_EXEC`. По сути эта функциональность аналогична PaX `mprotect` ограничению.

Адаптации механизма `seccomp` состоит из следующих изменений функциональности ядра ОС Linux:

– Добавлена функциональность поиска в сегменте `NOTE` записи из секции `.note.reloc_infos`, по содержанию которой происходит идентификация работающей мелкозернистой рандомизации при запуске.
– Добавлены в структуру, описывающую процесс, два поля для хранения виртуальных адресов начала и конца динамического загрузчика.
– Добавлена функциональность, которая при наличии мелкозернистой рандомизации заполняет в структуре текущего процесса адрес начала и конца загрузчика.
– Изменена функциональность `seccomp` таким образом, чтобы разрешать вызов `mprotect` с аргументом `PAGE_EXEC` для для процессов с поддержкой мелкозернистой рандомизации. Вызов разрешается только из области кода загрузчика, что проверяется с помощью сохранённых границ загрузчика.

Заключение

Основные результаты работы заключаются в следующем:

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.

В качестве дальнейших направлений работ по улучшению реализованного метода может рассматриваться поддержка отладочной информации во время рандомизации и оптимизаций, основанных на сборе профиля работающей программы. В качестве дальнейших направлений исследования по данной тематике перспективным направлением является разработка и реализация методов мелкозернистой рандомизации адресного пространства совместимых с механизмом разделения и совместного использования кода библиотек между конктекстами различных процессов операционной системы.

Список литературы

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.
8. Описание классификатора уязвимости CWE-119. — URL: https://cwe.mitre.org/data/definitions/119.html (дата обр. 15.08.2018).

9. Статистика количества уязвимостей классификатора CWE-119 по годам. — URL: https://nvd.nist.gov/vuln/search/statistics?results_type=statistics&cwe_id=CWE-119 (дата обр. 29.02.2020).
10. *Deckard J.* Buffer overflow attacks: detect, exploit, prevent. — Elsevier, 2005.
11. *Newsham T.* Format string attacks. — 2000. — URL: http://forum.ouah.org/FormatString.PDF (дата обр. 22.01.2021).
12. Understanding integer overflow in C/C++ / W. Dietz [и др.] // ACM Transactions on Software Engineering and Methodology (TOSEM). — 2015. — Т. 25, № 1. — С. 1—29.
13. A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003. — URL: https://support.microsoft.com/kb/875352/EN-US/.
14. PaX address space layout randomization (ASLR). — URL: https://pax.grsecurity.net/docs/aslr.txt (дата обр. 26.08.2018).
15. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. / C. Cowan [и др.] // USENIX security symposium. Т. 98. — San Antonio, TX. 1998. — С. 63—78.
16. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks / C. Cowan [и др.] // Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7. — San Antonio, Texas : USENIX Association, 1998. — (SSYM'98).
17. SoK: Eternal War in Memory / L. Szekeres [и др.] // Proceedings of the 2013 IEEE Symposium on Security and Privacy. — Washington, DC, USA : IEEE Computer Society, 2013. — С. 48—62. — (SP '13). — DOI: 10.1109/SP.2013.13.
18. Hacking Blind / A. Bittau [и др.] // Proceedings of the 2014 IEEE Symposium on Security and Privacy. — Washington, DC, USA : IEEE Computer Society, 2014. — С. 227—242. — (SP '14). — DOI: 10.1109/SP.2014.22.
19. *Shacham H.* The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86) // Proceedings of the 14th ACM Conference on Computer and Communications Security. — Alexandria, Virginia, USA : ACM, 2007. — С. 552—561. — (CCS '07). — DOI: 10.1145/1315245.1315313.

20. Описание атаки возврата в библиотеку в почтовой рассылке Bugtraq. — 1997. — URL: http://seclists.org/bugtraq/1997/Aug/63 (дата обр. 17.08.2018).

21. On the Effectiveness of Address-space Randomization / H. Shacham [и др.] // Proceedings of the 11th ACM Conference on Computer and Communications Security. — Washington DC, USA : ACM, 2004. — С. 298—307. — (CCS '04). — DOI: 10.1145/1030083.1030124.

22. The advanced return-into-lib(c) exploits. — 2001. — URL: http://phrack.org/issues/58/4.html (дата обр. 18.08.2018).

23. *Dullien T. F.* Weird machines, exploitability, and provable unexploitability // IEEE Transactions on Emerging Topics in Computing. — 2017.

24. X86-64 Buffer Overflow Exploits and the Borrowed Code Chunks Exploitation Technique. — 2005. — URL: http://users.suse.com/%7Ekrahmer/no-nx.pdf (дата обр. 18.08.2018).

25. Jump-oriented Programming: A New Class of Code-reuse Attack / T. Bletsch [и др.] // Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. — Hong Kong, China : ACM, 2011. — С. 30—40. — (ASIACCS '11). — DOI: 10.1145/1966913.1966919.

26. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications / F. Schuster [и др.] // 2015 IEEE Symposium on Security and Privacy. — 05.2015. — С. 745—762. — DOI: 10.1109/SP.2015.51.

27. It's a TRaP: Table Randomization and Protection Against Function-Reuse Attacks / S. J. Crane [и др.] // Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security. — Denver, Colorado, USA : ACM, 2015. — С. 243—255. — (CCS '15). — DOI: 10.1145/2810103.2813682.

28. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks / H. Hu [и др.] // 2016 IEEE Symposium on Security and Privacy (SP). — 05.2016. — С. 969—986. — DOI: 10.1109/SP.2016.62.

29. Dynamic Loader Oriented Programming on Linux / J. Kirsch [и др.] // Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium. — Vienna, Austria : ACM, 2017. — 5:1—5:13. — (ROOTS). — DOI: 10.1145/3150376.3150381.

30. *Sadeghi A.*, *Niksefat S.*, *Rostamipour M.* Pure-Call Oriented Programming (PCOP): chaining the gadgets using call instructions // Journal of Computer Virology and Hacking Techniques. — 2018. — Май. — Т. 14, № 2. — С. 139—156. — DOI: 10.1007/s11416-017-0299-1.

31. *Guo Y.*, *Chen L.*, *Shi G.* Function-Oriented Programming: A New Class of Code Reuse Attack in C Applications // 2018 IEEE Conference on Communications and Network Security (CNS). — 05.2018. — С. 1—9. — DOI: 10.1109/CNS.2018.8433189.

32. Position-Independent Code Reuse: On the Effectiveness of ASLR in the Absence of Information Disclosure / E. Göktas [и др.] // 2018 IEEE European Symposium on Security and Privacy (EuroS P). — 04.2018. — С. 227—242. — DOI: 10.1109/EuroSP.2018.00024.

33. *Schwartz E. J.*, *Avgerinos T.*, *Brumley D.* Q: Exploit Hardening Made Easy // Proceedings of the 20th USENIX Conference on Security. — San Francisco, CA : USENIX Association, 2011. — С. 25—25. — (SEC'11).

34. *Vishnyakov A. V.*, *Nurmukhametov A. R.* Survey of methods for automated code-reuse exploit generation // Proceedings of the Institute for System Programming of the RAS. — 2019. — Т. 31, № 6. — С. 99—124. — DOI: 10.15514/ISPRAS-2019-31(6)-6.

35. Результаты сравнения ASLR для различных Linux ядер. — URL: https://wiki.archlinux.org/index.php/Security#Userspace_ASLR_comparison (дата обр. 15.03.2020).

36. *Busser P.* Paxtest. — 2006. — URL: https://github.com/opntr/paxtest-freebsd (дата обр. 22.01.2021).

37. Exploiting Format String Vulnerabilities. — URL: https://crypto.stanford.edu/cs155old/cs155-spring08/papers/formatstring-1.2.pdf (дата обр. 30.08.2018).

38. Launching Return-Oriented Programming Attacks Against Randomized Relocatable Executables / L. Liu [и др.] // Proceedings of the 2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications. — Washington, DC, USA : IEEE Computer Society, 2011. — С. 37—44. — (TRUSTCOM '11). — DOI: 10.1109/TrustCom.2011.9.

39. Bypassing PaX ASLR protection. — URL: http://phrack.org/issues/59/9.html (дата обр. 26.08.2018).

40. *Evtyushkin D.*, *Ponomarev D.*, *Abu-Ghazaleh N.* Jump over ASLR: Attacking Branch Predictors to Bypass ASLR // The 49th Annual IEEE/ACM International Symposium on Microarchitecture. — Taipei, Taiwan : IEEE Press, 2016. — 40:1—40:13. — (MICRO-49).

41. Kernel address space layout randomization. — URL: https://lwn.net/Articles/569635/ (дата обр. 31.08.2018).

42. *Hund R.*, *Willems C.*, *Holz T.* Practical Timing Side Channel Attacks Against Kernel Space ASLR // Proceedings of the 2013 IEEE Symposium on Security and Privacy. — Washington, DC, USA : IEEE Computer Society, 2013. — С. 191—205. — (SP '13). — DOI: 10.1109/SP.2013.23.

43. *Gu Y.*, *Lin Z.* Derandomizing Kernel Address Space Layout for Memory Introspection and Forensics // Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy. — New Orleans, Louisiana, USA : ACM, 2016. — С. 62—72. — (CODASPY '16). — DOI: 10.1145/2857705.2857707.

44. Surgically Returning to Randomized Lib(C) / G. F. Roglia [и др.] // Proceedings of the 2009 Annual Computer Security Applications Conference. — Washington, DC, USA : IEEE Computer Society, 2009. — С. 60—69. — (ACSAC '09). — DOI: 10.1109/ACSAC.2009.16.

45. Оценка критичности программных дефектов в условиях работы современных защитных механизмов / А. Н. Федотов [и др.] // Труды ИСП РАН. — 2016. — Т. 28, № 5. — С. 73—92. — DOI: 10.15514/ISPRAS-2016-28(5)-4.

46. Mathias Payer. Too much PIE is bad for performance. — URL: https://nebelwelt.net/publications/files/12TRpie.pdf (дата обр. 29.08.2018).

47. Набор тестов на производительность компьютерных систем SPEC CPU® 2006. — URL: https://www.spec.org/cpu2006/ (дата обр. 25.02.2019).

48. CWE-123: Write-what-where Condition. — URL: https://cwe.mitre.org/data/definitions/123.html (дата обр. 15.03.2020).

49. The Leakage-Resilience Dilemma / B. C. Ward [и др.] // Computer Security – ESORICS 2019. — Springer International Publishing, 2019. — С. 87—106. — DOI: 10.1007/978-3-030-29959-0_5.

50. Position-Independent Code Reuse: On the Effectiveness of ASLR in the Absence of Information Disclosure / E. Göktas [и др.] // 2018 IEEE European Symposium on Security and Privacy (EuroS P). — 04.2018. — С. 227—242. — DOI: 10.1109/EuroSP.2018.00024.

51. Selfrando : Securing the Tor Browser against De-anonymization Exploits // Proceedings on Privacy Enhancing Technologies. — 2016. — Т. 2016, № 4. — С. 1—16. — DOI: 10.1515/popets-2016-0050.

52. Gadge Me if You Can: Secure and Efficient Ad-hoc Instruction-level Randomization for x86 and ARM / L. V. Davi [и др.] // Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security. — Hangzhou, China : ACM, 2013. — С. 299—310. — (ASIA CCS '13). — DOI: 10.1145/2484313.2484351.

53. *Backes M.*, *Nürnberger S.* Oxymoron: Making Fine-grained Memory Randomization Practical by Allowing Code Sharing // Proceedings of the 23rd USENIX Conference on Security Symposium. — San Diego, CA : USENIX Association, 2014. — С. 433—447. — (SEC'14).

54. *Crane S.*, *Homescu A.*, *Larsen P.* Code Randomization: Haven't We Solved This Problem Yet? // 2016 IEEE Cybersecurity Development (SecDev). — 11.2016. — С. 124—129. — DOI: 10.1109/SecDev.2016.036.

55. Timely Rerandomization for Mitigating Memory Disclosures / D. Bigelow [и др.] // Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security. — Denver, Colorado, USA : ACM, 2015. — С. 268—279. — (CCS '15). — DOI: 10.1145/2810103.2813691.

56. Shuffler: Fast and Deployable Continuous Code Re-randomization / D. Williams-King [и др.] // Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. — Savannah, GA, USA : USENIX Association, 2016. — С. 367—382. — (OSDI'16).

57. Страница приложения Google Chrome в магазине приложений Google Play, содержащая статистику по количеству загрузок. — URL: https://play.google.com/store/apps/details?id=com.android.chrome (дата обр. 08.10.2018).

58. Общее число людей, живущих на планете Земля. — URL: http://www.worldometers.info/world-population/ (дата обр. 08.10.2018).

59. Real-time fast physical random number generator with a photonic integrated circuit / K. Ugajin [и др.] // Opt. Express. — 2017. — Март. — Т. 25, № 6. — С. 6511—6523. — DOI: 10.1364/OE.25.006511.

60. Построение обфусцирующего компилятора на основе инфраструктуры LLVM / Ш. Ф. Курмангалеев [и др.] // Труды ИСП РАН. — 2012. — Т. 23. — С. 77—92. — DOI: 10.15514/ISPRAS-2012-23-5.

61. *Курмангалеев Ш.*, *Корчагин В.*, *Матевосян Р.* Описание подхода к разработке обфусцирующего компилятора // Труды Института системного программирования РАН. — 2012. — Т. 23. — DOI: 10.15514/ISPRAS-2012-23-4.

62. 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.

63. *TIS Committee* Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification. — 05.1995. — URL: http://refspecs.linuxbase.org/elf/elf.pdf (дата обр. 15.03.2019).

64. System V Application Binary Interface / M. Matz [и др.]. — 07.2012. — URL: http://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf (дата обр. 15.03.2019).

65. *Levine J. R.* Linkers and Loaders. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1999.

66. What is SELinux. — URL: https://www.redhat.com/en/topics/linux/what-is-selinux (visited on 08/06/2020).

67. Seccomp BPF. — URL: https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html (visited on 08/06/2020).

68. Набор тестов на производительность компьютерных систем SPEC CPU® 2017. — URL: https://www.spec.org/cpu2017/ (дата обр. 25.02.2019).

69. Инструмент поиска гаджетов ROPgadget. — URL: https : / / github . com / JonathanSalwan/ROPgadget (дата обр. 20.08.2018).

70. *Вишняков А. В.* Классификация ROP гаджетов // Труды ИСП РАН. — 2016. — Т. 28, № 6. — С. 27—36. — DOI: 10.15514/ISPRAS-2016-28(6)-2.

71. ROP Gadget Prevalence and Survival under Compiler-Based Binary Diversification Schemes / J. Coffman [и др.] // Proceedings of the 2016 ACM Workshop on Software PROtection. — Vienna, Austria : Association for Computing Machinery, 2016. — С. 15—26. — (SPRO '16). — DOI: 10.1145/2995306.2995309.

72. *Вишняков А. В.* Разработка и реализация метода генерации цепочек возвратно-ориентированного программирования: маг. дис. / Вишняков А. В. — Москва : Московский государственный университет имени М. В. Ломоносова, 2020. — С. 108. — URL: https://vishnya.xyz/vishnyakov-master-thesis2020.pdf.

Список рисунков

Список таблиц

Приложение А

Обфусцирующий компилятор для затруднения эксплуатации уязвимостей

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2016661393

«Обфусцирующий компилятор для затруднения эксплуатации уязвимостей»

Правообладатель: ***Федеральное государственное бюджетное учреждение науки Институт системного программирования Российской академии наук (RU)***

Авторы: ***Нурмухаметов Алексей Раисович (RU), Курмангалеев Шамиль Фаимович (RU), Гайсарян Сергей Суренович (RU), Чукляев Илья Игоревич (RU)***

Заявка № **2016616567**

Дата поступления **22 июня 2016 г.**

Дата государственной регистрации в Реестре программ для ЭВМ ***07 октября 2016 г.***

Руководитель Федеральной службы по интеллектуальной собственности

Г.П. Ивлиев

Приложение Б

Инструмент «Faslr» для усиления системной защиты при запуске программ в ОС Linux

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2017660041

«Инструмент «Faslr» для усиления системной защиты при запуске программ в ОС Linux»

Правообладатель: *Федеральное государственное бюджетное учреждение науки Институт системного программирования Российской академии наук (RU)*

Авторы: *Курмангалеев Шамиль Фаимович (RU), Нурмухаметов Алексей Раисович (RU)*

Заявка № 2017616967
Дата поступления 17 июля 2017 г.
Дата государственной регистрации
в Реестре программ для ЭВМ *13 сентября 2017 г.*

Руководитель Федеральной службы по интеллектуальной собственности

Г.П. Ивлиев

Приложение В

Акт о внедрении результатов диссертационной работы

ЗАКРЫТОЕ АКЦИОНЕРНОЕ ОБЩЕСТВО
"МНОГОПРОФИЛЬНОЕ ВНЕДРЕНЧЕСКОЕ ПРЕДПРИЯТИЕ "СВЕМЕЛ"
Фактический адрес: 127254, Москва, Огородный пр., д. 5, стр.5
Юридический адрес: 127254, Москва, Огородный пр., д. 5, стр.5
Тел/Факс: +7(495) 926-7187, +7(499) 750-7065
E-mail: post@swemel. ru

АКТ

о внедрении результатов кандидатской диссертационной работы

Нурмухаметова Алексея Раисовича

Результаты диссертационного исследования Нурмухаметова Алексея Раисовича на тему «Применение диверсифицирующих преобразований для защиты от эксплуатации уязвимостей» использованы в разрабатываемой ЗАО «МВП «СВЕМЕЛ» операционной системе семейства Linux. В частности, в качестве одного из компонентов защиты от эксплуатации уязвимостей применяется созданная Нурмухаметовым А.Р. мелкозернистая рандомизация размещения функций в секции кода исполняемого файла во время загрузки его в память процесса при запуске программы. Реализованная система защиты обладает приемлемыми характеристиками по ухудшению производительности приложений, совместима с имеющимися средствами защиты для достижения взаимноусиливающего эффекта, применима в рамках целой ОС, а также обеспечивает дополнительную защиту от атак повторного использования кода.

Генеральный директор А.В.Неверко

Приложение Г

Результаты изменения производительности мелкозернистой рандомизации адресного пространства программы при запуске на наборе тестов SPEC CPU® 2017

SPEC(R) CPU2017 Integer Speed Result
Gygabyte Technology Co. GA–970A–D3
Test Sponsor: ISP RAS

CPU2017 License: 3855
Test sponsor: ISP RAS
Tested by: ISP RAS

Test date: Jun–2020
Hardware availability: Jan–2015
Software availability: May–2020

Benchmarks	Base Threads	Base Run Time	Base Ratio	Peak Threads	Peak Run Time	Peak Ratio
600.perlbench_s	4	547	3.25	4	556	3.19
600.perlbench_s	4	549	3.23	4	553	3.21
602.gcc_s	4	822	4.84	4	827	4.82
602.gcc_s	4	824	4.83	4	842	4.73
605.mcf_s	4	986	4.79	4	955	4.94
605.mcf_s	4	961	4.91	4	951	4.96
620.omnetpp_s	4	530	3.08	4	550	2.97
620.omnetpp_s	4	534	3.05	4	537	3.04
623.xalancbmk_s	4	400	3.54	4	431	3.28
623.xalancbmk_s	4	403	3.52	4	406	3.49
625.x264_s	4	566	3.12	4	567	3.11
625.x264_s	4	569	3.10	4	572	3.09
631.deepsjeng_s	4	504	2.84	4	498	2.88
631.deepsjeng_s	4	498	2.88	4	494	2.90
641.leela_s	4	556	3.07	4	554	3.08
641.leela_s	4	555	3.08	4	553	3.09
648.exchange2_s	4	512	5.74	4	505	5.82
648.exchange2_s	4	503	5.84	4	501	5.87
657.xz_s	4	1082	5.71	4	1035	5.97
657.xz_s	4	1057	5.85	4	1020	6.06
600.perlbench_s	4	549	3.23	4	556	3.19
602.gcc_s	4	824	4.83	4	842	4.73
605.mcf_s	4	986	4.79	4	955	4.94
620.omnetpp_s	4	534	3.05	4	550	2.97
623.xalancbmk_s	4	403	3.52	4	431	3.28
625.x264_s	4	569	3.10	4	572	3.09
631.deepsjeng_s	4	504	2.84	4	498	2.88

641.leela_s	4	556	3.07	4	554	3.08
648.exchange2_s	4	512	5.74	4	505	5.82
657.xz_s	4	1082	5.71	4	1035	5.97
SPECspeed2017_int_base			3.85			
SPECspeed2017_int_peak						3.84

SPEC(R) CPU2017 Floating Point Speed Result
Gygabyte Technology Co. GA–970A–D3
Test Sponsor: ISP RAS

CPU2017 License: 3855 Test date: Jun –2020
Test sponsor: ISP RAS Hardware availability: Jan –2015
Tested by: ISP RAS Software availability: May –2020

Benchmarks	Base Threads	Base Run Time	Base Ratio	Peak Threads	Peak Run Time	Peak Ratio
603.bwaves_s	4	2281	25.9	4	2285	25.8
603.bwaves_s	4	2285	25.8	4	2285	25.8
607.cactuBSSN_s	4	1220	13.7	4	1200	13.9
607.cactuBSSN_s	4	1203	13.9	4	1203	13.9
619.lbm_s	4	1644	3.19	4	1651	3.17
619.lbm_s	4	1650	3.17	4	1651	3.17
621.wrf_s	4	2810	4.71	4	2817	4.70
621.wrf_s	4	2812	4.70	4	2811	4.70
627.cam4_s	4	1339	6.62	4	1341	6.61
627.cam4_s	4	1341	6.61	4	1343	6.60
628.pop2_s	4	1485	8.00	4	1486	7.99
628.pop2_s	4	1486	7.99	4	1485	7.99
638.imagick_s	4	2763	5.22	4	2778	5.19
638.imagick_s	4	2761	5.23	4	2760	5.23
644.nab_s	4	1565	11.2	4	1570	11.1
644.nab_s	4	1565	11.2	4	1566	11.2
649.fotonik3d_s	4	1206	7.56	4	1206	7.56
649.fotonik3d_s	4	1215	7.50	4	1221	7.47
654.roms_s	4	2289	6.88	4	2269	6.94
654.roms_s	4	2293	6.87	4	2277	6.92
603.bwaves_s	4	2285	25.8	4	2285	25.8
607.cactuBSSN_s	4	1220	13.7	4	1203	13.9
619.lbm_s	4	1650	3.17	4	1651	3.17
621.wrf_s	4	2812	4.70	4	2817	4.70
627.cam4_s	4	1341	6.61	4	1343	6.60
628.pop2_s	4	1486	7.99	4	1486	7.99

638.imagick_s	4	2763	5.22	4	2778	5.19
644.nab_s	4	1565	11.2	4	1570	11.1
649.fotonik3d_s	4	1215	7.50	4	1221	7.47
654.roms_s	4	2293	6.87	4	2277	6.92
SPECspeed2017_fp_base			7.80			
SPECspeed2017_fp_peak						7.80

SPEC(R) CPU2017 Integer Rate Result
Gygabyte Technology Co. GA–970A–D3
Test Sponsor: ISP RAS

CPU2017 License: 3855
Test sponsor: ISP RAS
Tested by: ISP RAS

Test date: Jun–2020
Hardware availability: Jan–2015
Software availability: May–2020

Benchmarks	Base Copies	Base Run Time	Base Rate	Peak Copies	Peak Run Time	Peak Rate
500.perlbench_r	1	540	2.95	1	565	2.82
500.perlbench_r	1	553	2.88	1	566	2.81
502.gcc_r	1	386	3.67	1	392	3.62
502.gcc_r	1	387	3.65	1	392	3.61
505.mcf_r	1	452	3.57	1	457	3.54
505.mcf_r	1	456	3.54	1	450	3.59
520.omnetpp_r	1	533	2.46	1	537	2.45
520.omnetpp_r	1	530	2.48	1	533	2.46
523.xalancbmk_r	1	399	2.65	1	407	2.60
523.xalancbmk_r	1	398	2.66	1	405	2.61
525.x264_r	1	566	3.09	1	566	3.09
525.x264_r	1	567	3.09	1	568	3.08
531.deepsjeng_r	1	407	2.81	1	410	2.80
531.deepsjeng_r	1	407	2.81	1	407	2.82
541.leela_r	1	555	2.99	1	555	2.98
541.leela_r	1	552	3.00	1	553	2.99
548.exchange2_r	1	507	5.17	1	511	5.13
548.exchange2_r	1	503	5.21	1	509	5.15
557.xz_r	1	444	2.43	1	444	2.43
557.xz_r	1	444	2.43	1	446	2.42
500.perlbench_r	1	553	2.88	1	566	2.81
502.gcc_r	1	387	3.65	1	392	3.61
505.mcf_r	1	456	3.54	1	457	3.54
520.omnetpp_r	1	533	2.46	1	537	2.45
523.xalancbmk_r	1	399	2.65	1	407	2.60
525.x264_r	1	567	3.09	1	568	3.08
531.deepsjeng_r	1	407	2.81	1	410	2.80
541.leela_r	1	555	2.99	1	555	2.98

548.exchange2_r	1	507	5.17	1	511	5.13
557.xz_r	1	444	2.43	1	446	2.42
SPECrate2017_int_base			3.09			
SPECrate2017_int_peak						3.06

SPEC(R) CPU2017 Floating Point Rate Result
Gygabyte Technology Co. GA–970A–D3
Test Sponsor: ISP RAS

CPU2017 License: 3855
Test sponsor: ISP RAS
Tested by: ISP RAS

Test date: Jun–2020
Hardware availability: Jan–2015
Software availability: May–2020

Benchmarks	Base Copies	Base Run Time	Base Rate	Peak Copies	Peak Run Time	Peak Rate
503.bwaves_r	1	961	10.4	1	966	10.4
503.bwaves_r	1	962	10.4	1	963	10.4
507.cactuBSSN_r	1	478	2.65	1	482	2.63
507.cactuBSSN_r	1	480	2.64	1	479	2.64
508.namd_r	1	310	3.06	1	310	3.07
508.namd_r	1	309	3.08	1	311	3.05
510.parest_r	1	731	3.58	1	749	3.49
510.parest_r	1	732	3.57	1	750	3.49
511.povray_r	1	800	2.92	1	799	2.92
511.povray_r	1	793	2.94	1	799	2.92
519.lbm_r	1	352	2.99	1	350	3.01
519.lbm_r	1	347	3.03	1	349	3.02
521.wrf_r	1	1424	1.57	1	1433	1.56
521.wrf_r	1	1429	1.57	1	1427	1.57
526.blender_r	1	555	2.74	1	565	2.70
526.blender_r	1	557	2.74	1	558	2.73
527.cam4_r	1	614	2.85	1	613	2.85
527.cam4_r	1	612	2.86	1	611	2.86
538.imagick_r	1	1234	2.02	1	1235	2.01
538.imagick_r	1	1236	2.01	1	1233	2.02
544.nab_r	1	543	3.10	1	541	3.11
544.nab_r	1	549	3.07	1	541	3.11
549.fotonik3d_r	1	592	6.58	1	592	6.58
549.fotonik3d_r	1	596	6.54	1	593	6.57
554.roms_r	1	658	2.42	1	659	2.41
554.roms_r	1	660	2.41	1	656	2.42
503.bwaves_r	1	962	10.4	1	966	10.4
507.cactuBSSN_r	1	480	2.64	1	482	2.63
508.namd_r	1	310	3.06	1	311	3.05
510.parest_r	1	732	3.57	1	750	3.49

511.povray_r	1	800	2.92	1	799	2.92
519.lbm_r	1	352	2.99	1	350	3.01
521.wrf_r	1	1429	1.57	1	1433	1.56
526.blender_r	1	557	2.74	1	565	2.70
527.cam4_r	1	614	2.85	1	613	2.85
538.imagick_r	1	1236	2.01	1	1235	2.01
544.nab_r	1	549	3.07	1	541	3.11
549.fotonik3d_r	1	596	6.54	1	593	6.57
554.roms_r	1	660	2.41	1	659	2.41
SPECrate2017_fp_base			3.16			
SPECrate2017_fp_peak						3.15

HARDWARE

CPU Name: AMD FX-8150
Max MHz.: 4200
Nominal: 3600
Enabled: 8 cores, 1 chip
Orderable: 1 chip
Cache L1: 384 KB I+D on chip per chip
L2: 8 MB I+D on chip per chip
L3: 8 MB I+D on chip per chip
Other: None
Memory: 32151 MB
'N GB (N x N GB nRxn PC4-nnnnX-X)'
Storage: 108 GB add more disk info here
Other: None

SOFTWARE

OS: Debian
10.3
Compiler: C/C++/Fortran: Version 7.2.1 of GCC, the
GNU Compiler Collection
Parallel: Yes
Firmware: Award Software International, Inc. Version F10
released May-2012
File System: ext4
System State: Run level 5 (add definition here)
Base Pointers: 64-bit
Peak Pointers: 64-bit
Other: ASLP

Приложение Д

Исходный код модуля vul_preload.so

```
// gcc -fPIC -shared vul_preload.c -o vul_preload.so -mcmodel=large  \
// -Wl,-Ttext-segment=0xbb000000 -fno-stack-protector -D_FORTIFY_SOURCE=0
// LD_PRELOAD=./vul_preload.so <command>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <stdint.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>

#define STACK_LINES 10
char *debug = 0;

void zero_arg() {
  printf("   SUCCESS\n");
  return;
}

void one_arg(int64_t i1) {
  printf("   SUCCESS\n");
  printf(" * Parameters: i64 '%li'\n", i1);
  if (i1 == 1)
    printf("   PARAMETERS ARE CORRECT\n");
  else
    printf("   !!!INCORRECT PARAMETERS!!!\n");
  return;
}

void two_arg(int64_t i1, int64_t i2) {
  printf("   SUCCESS\n");
  printf(" * Parameters: i64 '%li', i64 '%li'\n", i1, i2);
  if (i1 == 1 && i2 == 2)
    printf("   PARAMETERS ARE CORRECT\n");
  else
    printf("   !!!INCORRECT PARAMETERS!!!\n");
  return;
}

void three_arg(int64_t i1, int64_t i2, int64_t i3) {
  printf("   SUCCESS\n");
```

```
  printf(" * Parameters: i64 '%li', i64 '%li', i64 '%li'\n", i1, i2, i3);
  if (i1 == 1 && i2 == 2 && i3 == 3)
    printf("   PARAMETERS ARE CORRECT\n");
  else
    printf("   !!!INCORRECT PARAMETERS!!!\n");
  return;
}

void __system(char *s) {
  printf("   SUCCESS\n");
  printf(" * Parameters: char* '%s',\n", s);
  if (!strcmp("/bin/zh", s))
    printf("   PARAMETERS ARE CORRECT\n");
  else
    printf("   !!!INCORRECT PARAMETERS!!!\n");
  return;
}

int filesize(int fd) {
  struct stat st;
  int ret = fstat(fd, &st);
  assert(ret == 0);
  return st.st_size;
}

void vul(char *filename) {
  char buf[1];
  int fd = open(filename, O_RDONLY);
  assert(fd != -1);
  size_t size = filesize(fd);

  if (debug)
    printf("\t- File '%s' with size '%li'\n", filename, size);

  int i = 0, j = 0;
  for (j = 0; j < STACK_LINES; j++) {
    for (i = j * 8; i < (j + 1)*8; i++)
      if (debug)
        printf("%02x ", (unsigned char)buf[i]);
    if (debug)
      printf("\n");
  }

  if (debug)
    printf("\nBOF\n");

  int ret = read(fd, buf, size);
  assert(ret != -1);
```

```
  close(fd);
  for (j = 0; j < STACK_LINES; j++) {
    for (i = j * 8; i < (j + 1)*8; i++)
      if (debug)
        printf("%02x ", (unsigned char)buf[i]);
    if (debug)
      printf("\n");
  }
  return;
}

int _so_main() {
  setbuf(stdout, NULL);

  int var[1000];
  var[999] = 1;
  char *filename = "/tmp/vul_input";

  char str[] = "Start of _so_main of vul_preload.so\n";
  write(1, str, sizeof(str));

  if (debug)
    printf("_so_main: %p\n", _so_main);

  vul(filename);
  printf("   FAIL\n");
  printf(" * Control flow wasn't be hijacked\n");
  return 0;
}

int __libc_start_main(void *func_ptr, int argc, char* argv[], void
    (*init_func)(void), void (*fini_func)(void), void (*rtld_fini_func)(void),
    void *stack_end)
{
  debug = getenv("VUL_PRELOAD_DEBUG");

  if (debug) {
    printf("Fake __libc_start_main\n");
    printf("zero_arg: %p\n", zero_arg);
    printf("one_arg: %p\n", one_arg);
    printf("two_arg: %p\n", two_arg);
    printf("three_arg: %p\n", three_arg);
    printf("system: %p\n", __system);
  }

  _so_main();
}
```