DOI: 10.15514/ISPRAS-2019-31(6)-6

Обзор методов автоматизированной генерации эксплойтов повторного использования кода

[1,2] *А.В. Вишняков, ORCID: 0000-0003-1819-220X <vishnya@ispras.ru>*
[1] *А.Р. Нурмухаметов, ORCID: 0000-0003-1681-1580 <nurmukhametov@ispras.ru>*
[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25*
[2] *Московский государственный университет имени М.В. Ломоносова, 119991, Россия, Москва, Ленинские горы, д. 1*

Аннотация. В работе приводится обзор существующих методов и инструментов автоматизированной генерации эксплойтов повторного использования кода. Такие эксплойты используют код, уже содержащийся в уязвимом приложении. Подход повторного использования кода (например, возвратно-ориентированное программирование) позволяет эксплуатировать уязвимости программного обеспечения при наличии защитного механизма операционной системы, который запрещает исполнение кода страниц памяти, помеченных в качестве данных. В статье дается определение базовых понятий таких, как гаджет, фрейм гаджета, каталог гаджетов. Кроме того, показывается, что гаджет по своей сути является инструкцией, а их набор задает некоторую виртуальную машину. Задача создания эксплойта сводится к задаче генерации кода для такой виртуальной машины. Набор команд виртуальной машины задается исполняемым кодом конкретной программы. В работе приводится обзор методов поиска гаджетов и определения их семантики (формирования каталога гаджетов). Они позволяют получить набор команд виртуальной машины. Если набор гаджетов в каталоге полон по Тьюрингу, то гаджеты из каталога можно использовать в качестве набора команд целевой архитектуры компилятора. Однако в каталоге гаджетов для конкретного приложения могут отсутствовать некоторые инструкции, поэтому в литературе было предложено несколько способов для замены отсутствующих инструкций несколькими гаджетами. Связывание гаджетов в цепочки может происходить как поиском гаджетов по шаблонам, задаваемым регулярными выражениями, так и с учетом семантики гаджета. Более того, существуют подходы конструирования ROP цепочек с использованием генетических алгоритмов, а также методы с использованием SMT-решателей. В статье проводится сравнение инструментов с открытым исходным кодом. Мы предлагаем тестовую систему rop-benchmark, с помощью которой была проведена экспериментальная проверка работоспособности генерируемых инструментами цепочек на специально сформированном наборе тестов.

Ключевые слова: атаки повторного использования кода; возвратно-ориентированное программирование; ROP; предотвращение выполнения данных; DEP; рандомизация размещения адресного пространства; ASLR; гаджет; фрейм гаджета; каталог гаджетов; ROP цепочка; поиск гаджетов; классификация гаджетов; ROP компилятор; символьная интерпретация; ROP benchmark

Для цитирования: Вишняков А.В., Нурмухаметов А.Р. Обзор методов автоматизированной генерации эксплойтов повторного использования кода. Труды ИСП РАН, том 31, вып. 6, 2019 г., стр. 99–124. DOI: 10.15514/ISPRAS–2019–31(6)–6

Благодарности: Работа поддержана грантом РФФИ № 17-01-00600

Survey of methods for automated code-reuse exploit generation

[1,2] *A.V. Vishnyakov, ORCID: 0000-0003-1819-220X <vishnya@ispras.ru>*
[1] *A.R. Nurmukhametov, ORCID: 0000-0003-1681-1580 <nurmukhametov@ispras.ru>*
[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia*
[2] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russian Federation*

Abstract. This paper provides a survey of methods and tools for automated code-reuse exploit generation. Such exploits use code that already contains in a vulnerable program. The code-reuse approach, e.g., return-oriented programming, allows one to exploit vulnerabilities in the presence of operating system protection that prohibits execution of code in memory pages marked as data. We define fundamental terms such as gadget, gadget frame, gadget catalog. Moreover, we show that a gadget is, in fact, an instruction, and a set of gadgets defines a virtual machine. We can reduce an exploit creation problem to code generation for this virtual machine. Each particular executable file defines a virtual machine instruction set. We provide a survey of methods for gadgets searching and determining their semantics (creating a gadget catalog). These methods allow one to get the virtual machine instruction set. If a set of gadgets is Turing-complete, then a compiler can use a gadget catalog as a target architecture. However, some instructions can be absent. Hence we discuss several approaches to replace missing instructions with multiple gadgets. An exploit generation tool can chain gadgets by pattern searching (regular expressions) or taking gadgets semantics into consideration. Furthermore, some chaining methods use genetic algorithms, while others use SMTsolvers. We compare existing open source tools and propose a testing system rop-benchmark that can be used to verify whether a generated chain successfully opens a shell.

Keywords: code-reuse attacks; return-oriented programming; ROP; data execution prevention; DEP; address space layout randomization; ASLR; gadget; gadget frame; gadget catalog; ROP chain; gadget search; gadget classification; ROP compiler; symbolic execution; ROP benchmark

For citation: Vishnyakov A.V., Nurmukhametov A.R. Survey of methods for automated code-reuse exploit generation. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 6, 2019. pp. 99-124 (in Russian). DOI: 10.15514/ISPRAS-2019-31(6)-6

Acknowledgements. This work was supported by the Russian Foundation for Basic Research, project no. 17-01-00600

1. Введение

Современное программное обеспечение содержит ошибки. Их наличие рассматривается некоторыми исследователями как неизбежность. Однако далеко не все ошибки возможно использовать (*эксплуатировать*) в злонамеренных целях. Эксплуатируемые ошибки называются *уязвимостями*. Эксплуатация уязвимостей приводит к серьезным последствиям: денежным убыткам, деградации средств коммуникации, компрометации криптографических ключей [1] и др. С развитием технологий интернета вещей окружающие нас повседневно предметы (чайники, холодильники, душевые системы и др.) могут быть подвержены эксплуатации. Особенно критичны вопросы безопасности медицинского оборудования. Гальперин (Halperin) и др. [2] показали возможность эксплуатации имлантируемых сердечных дефибрилляторов.

Вместе с развитием безопасного цикла разработки программного обеспечения улучшаются также методы по обнаружению разнообразных программных дефектов. В ответ на усовершенствование методов защиты от эксплуатации уязвимостей разрабатываются новые методы их обхода и эксплуатации. Поэтому необходимо знать и понимать, как устроены методы как защиты, так и атаки на программное обеспечение. Более того, для приоритетного исправления уязвимостей вендоры и разработчики программного обеспечения требуют подтверждающие примеры – *эксплойты*.

Переполнение буфера на стеке является, пожалуй, одним из самых эксплуатируемых программных дефектов [3]. Это объясняется сравнительной легкостью в использовании этого дефекта для перехвата потока управления под контроль атакующего. В простейшем случае полного отсутствия защит эксплуатация происходит следующим образом. Адрес возврата, расположенный на стеке выше локального буфера, перезаписывается контролируемым значением. Данное значение указывает обратно внутрь буфера, где располагается код, который хочет исполнить атакующий.

Для противодействия исполнению кода, расположенного в буфере на стеке, появилась защита DEP. Она позволила запретить исполнять определенные регионы памяти процесса, такие как стек и куча. И, в свою очередь, запретить писать в регионы памяти, помеченные для исполнения. Данная защита положила конец эпохи инъекции кода в память процесса. Атакующие оказались ограничены в своих возможностях исполнять только код, имеющийся в памяти процесса.

В ответ на повсеместное распространение DEP начали активно развиваться атаки повторного использования кода. Первым таким типом атаки являлась атака возврата в библиотеку [4]. На стек помещается адрес функций для вызова на место адреса возврата, а за ним следом размещаются аргументы функции. Обобщением данного метода атаки стало развитие методов возвратно-ориентированного программирования [5–7]. При возвратно-ориентированном программировании вместо функций выступают короткие последовательности инструкций, заканчивающиеся инструкцией возврата и называемые *гаджетами*. Гаджеты связываются в цепочки так, чтобы они последовательно передавали друг другу управление и осуществляли в совокупности некоторую вредоносную нагрузку. Шахем (Shacham) [5] дал определение понятию гаджет и привел первый каталог гаджетов, для которого была показана полнота по Тьюрингу для архитектуры набора команд x86. В дальнейшем была показана применимость возвратно-ориентированного программирования и для других архитектур набора команд: ARM [8–12], SPARC [13], Atmel AVR [14], PowerPC [15], Z80 [16], MIPS [15]. В работах [9, 17–19] было показано, что можно использовать гаджеты, которые оканчиваются не только инструкциями возврата.

Вместе с развитием методов повторного использования кода происходило и развитие инструментов, с помощью которых атакующий конструировал атаки данного типа. Вначале это процесс был практически ручным, но со временем он постепенно автоматизировался. В данный момент в литературе представлен набор подходов к автоматизированному построению эксплойтов повторного использования кода [6, 7, 11–13, 18, 20–26]. Кроме того, для некоторых из них даже доступны инструменты [27–36].

Данная работа ставит своей целью детальное изучение имеющихся методов и инструментов автоматизированной генерации эксплойтов повторного использования кода с целью определения сильных и слабых сторон каждого из них, а также выявления перспективных направлений исследования в данной области.

Кроме практической применимости, рассматриваемые в статье методы и инструменты могут иметь и научный интерес. Задача автоматизированной генерации эксплойтов для атак повторного использования кода является задачей трансляции некоторого описания эксплойта в архитектуру набора команд виртуальной машины, неявно задаваемой состоянием памяти эксплуатируемого процесса. В качестве инструкций набора команд выступают гаджеты, расположенные в памяти процесса. Причем заранее неизвестно, какой набор инструкций предоставляет эксплуатируемый исполняемый файл. Для его определения необходимо найти все гаджеты, а затем произвести процедуру определения их функциональности (семантики). В результате, формируется каталог гаджетов, где описана их семантика. Каталог гаджетов является входными данными для инструмента, который генерирует эксплойт. Генерирующий эксплойт инструмент должен учитывать тот факт, что в наборе гаджетов, в отличии от инструкций процессора, могут

отсутствовать некоторые инструкции, а другие – иметь нетривиальные побочные эффекты. Все это усложняет построение инструментов автоматической генерации эксплойтов возвратно-ориентированного программирования.

Данная работа устроена следующим образом. В разд. 2 приводится определение возвратно-ориентированного программирования. В разд. 3 описывается общая схема генерации эксплойтов повторного использования кода. В разд. 4 вводится определение *каталога гаджетов*. В разд. 5 описываются подходы к поиску гаджетов. В разд. 6 приводятся методы определения семантики гаджетов. В разд. 7 проводится обзор методов генерации цепочек гаджетов. В разд. 8 освещается проблема учета запрещенных символов в цепочках. Экспериментальное сравнение инструментов с открытым исходным кодом, выполненное с использованием разработанной нами тестовой системы rop-benchmark [37], описывается в разд. 9.

2. Возвратно-ориентированное программирование

Шахем [5] предложил термин возвратно-ориентированное программирование (return-oriented programming, ROP). ROP является эффективным средством обхода предотвращения выполнения данных (DEP, W^X). В некотором смысле данный подход является обобщением атаки возврата в библиотеку (return-to-libc) [4]. Однако вредоносная нагрузка осуществляется не вызовом одной функции, а формируется из нескольких уже присутствующих в программе кусочков кода, которые называются *гаджетами*. Гаджет – это последовательность инструкций, заканчивающаяся инструкцией передачи управления. Каждый гаджет изменяет состояние регистров и памяти вычислительной машины. Например, складывает значения двух регистров и записывает результат в третий. Атакующий, изучив все имеющиеся в программе гаджеты, связывает их в *цепочки*, в которых гаджеты последовательно передают управление друг другу. Суммарная вредоносная нагрузка реализуется такой цепочкой гаджетов. При достаточном количестве гаджетов атакующим может быть сформирован полный по Тьюрингу набор, который позволит реализовывать произвольные вычисления [5]. Следует отметить, что ROP может также применяться при частичной рандомизации адресного пространства. Тогда используются гаджеты из нерандомизированных областей памяти.

Для наглядности приводится пример нескольких гаджетов для x86 в таблице 1, где представлен ассемблерный код[1] инструкций трех гаджетов. Каждый из гаджетов заканчивается инструкцией передачи управления `ret`, которая позволяет передавать управление следующему гаджету через адрес, размещаемый на стеке.

Табл. 1. Пример гаджетов для x86
Table 1. Example of x86 gadgets

`mov eax, ebx ; ret`	Копирование значения регистра `ebx` в регистр `eax`
`pop ecx ; ret`	Загрузка на регистр `ecx` значения со стека
`add eax, ebx ; ret`	Прибавление к регистру `eax` значения регистра `ebx`

Архитектура x86 является CISC. Инструкции x86 имеют нефиксированную длину, и каждая инструкция может выполнять несколько других низкоуровневых команд. Количество команд настолько велико, и они закодированы так плотно, что практически любая последовательность байтов декодируется в корректную инструкцию. Кроме того, из-за различных длин команд (от 1 байта до 15) архитектура x86 не требует выравнивания инструкций. С точки зрения ROP это означает следующее. Набор гаджетов в программе не ограничивается только инструкциями, которые были сгенерированы компилятором. Этот набор расширяется за счет инструкций, не присутствовавших в исходной программе

[1] Здесь и далее мы будем использовать синтаксис Intel для ассемблера x86.

и полученных при доступе в середину других команд. Пример, иллюстрирующий это, приводится ниже [38]:

```
f7c7070000000f9545c3 →    test edi, 0x7 ; setnz BYTE PTR [ebp-0x3d]
  c7070000000f9545c3 →    mov DWORD PTR [edi], 0xf000000 ;
                        xchg ebp, eax ; inc ebp ; ret
```

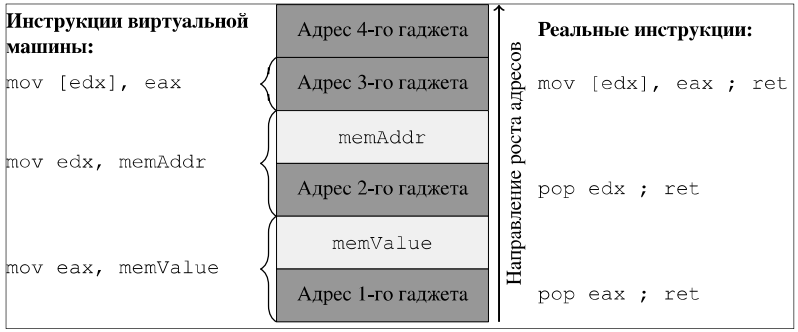

Рис. 1. Цепочка гаджетов, осуществляющая запись значения memValue по адресу memAddr
Fig. 1. A ROP Chain, storing memValue to memAddr

Набор гаджетов, который можно использовать при составлении ROP цепочки по сути задается исполняемым файлом. И для другого исполняемого файла ROP цепочку придется собирать заново. ROP цепочку можно рассматривать как программу для некоторой виртуальной машины, задаваемой исполняемым файлом [39]. Указатель стека выполняет роль счетчика инструкций этой виртуальной машины. Коды операций (адреса гаджетов) и их операнды размещаются на стеке. Грациано (Graziano) и др. [40] даже предложили инструмент для трансляции ROP цепочек в обычную программу x86. На рис. 1 приводится пример размещенной на стеке цепочки гаджетов, осуществляющей запись значения `memValue` по адресу `memAddr`. Коды операций (адреса гаджетов) размещаются от адреса возврата на стеке и закрашены темно-серым. Операнды `memValue` и `memAddr` закрашены светло-серым. Фигурными скобками обозначены инструкции виртуальной машины (код команды и ее операнды). Реальные инструкции гаджета на машине x86 приводятся справа. В начало цепочки, где при нормальном исполнении размещается адрес возврата из функции, помещается код команды – адрес первого гаджета. Затем располагается операнд `memValue`, который первый гаджет загрузит на регистр `eax`. Затем следует адрес второго гаджета, которому передаст управление первый гаджет на инструкции `ret`, и так далее. Даже бинарные файлы (x86) сравнительно небольшого размера содержат практически применимые с точки зрения атакующих наборы гаджетов. Шварц (Schwartz) и др. [6, 41] приводят статистику, что среди программ размером больше 100KB около 80 % содержат наборы гаджетов, которые позволяют вызывать любую функцию из динамически скомпонованной с уязвимым приложением библиотеки [42].

Рис. 2. Фрейм гаджета pop eax ; ret 8
Fig. 2. pop eax ; ret 8 gadget frame

В дальнейшем применение ROP было успешно продемонстрировано для других архитектур набора команд: SPARC [13], Z80 [16], ARM [8–12, 43]. В перечисленных работах было показано, что на RISC архитектурах возможно сконструировать как применимый для эксплуатации, так и полный по Тьюрингу набор гаджетов. RISC архитектуры часто характеризуются фиксированной длиной команд, требованием к выравниванию инструкций по их размеру и упрощенным доступом к памяти (к памяти как правило обращаются только инструкции сохранения и загрузки). Требование к выравниванию инструкций приводит к тому, что по сравнению с x86 остаются только гаджеты, оканчивающимися инструкциями возврата, которые изначально содержались в программе.

2.1 Фрейм гаджета

Для размещения ROP цепочки на стеке удобно ввести понятие *фрейма гаджета* [44, 45] аналогично стековому кадру x86. Цепочка гаджетов собирается из фреймов. Фрейм гаджета содержит в себе значения параметров гаджета (например, значение, загружаемое на регистр со стека) и адрес следующего гаджета. Начало фрейма определяется значением указателя стека перед выполнением первой инструкции гаджета. На рис. 2 фигурной скобкой обозначены границы фрейма гаджета `pop eax ; ret 8`. Гаджет загружает значение со стека в `eax` по смещению 0 от начала фрейма. Гаджет имеет размер фрейма `FrameSize = 16`, а адрес следующего гаджета располагается по смещению 4 от начала фрейма (`NextAddr = [esp + 4]`).

3. Общая схема генерации эксплойтов повторного использования кода

Схематично процесс генерации эксплойтов повторного использования кода делится на четыре этапа.

1. Поиск гаджетов в нерандомизованных исполняемых областях образа памяти процесса (разд. 5).
2. Определение семантики гаджетов (в некоторых методах данный этап может быть пропущен). На этом этапе определяется полезная нагрузка, которую выполняет каждый гаджет (разд. 6).
3. Комбинация гаджетов и их параметров для получения цепочки гаджетов, выполняющей заданную последовательность действий (разд. 7).
4. Автоматизированная генерация эксплойта [46–50] – входных данных, приводящих к эксплуатации программы путем размещения и выполнения ROP цепочки. На этом этапе в результате символьной интерпретации [51–53] машинных инструкций на трассе выполнения программы от точки получения входных данных до точки проявления уязвимости происходит построение предиката пути. Предикат пути объединяется с предикатом безопасности, описывающим размещение ROP цепочки и передачу на нее управления. Решением полученной системы уравнений будет эксплойт. Предикат пути обеспечит выполнение программы по тому же пути до уязвимости. А предикат безопасности – перехват потока управления.

4. Каталог гаджетов

Определим *каталог гаджетов* как список записей со следующим содержанием.

1. **Семантическое описание** последовательности инструкций машинного кода. Каждое описание соответствует, как правило, некоторой базовой вычислительной операции или операции работы с памятью (сложение, вычитание, запись в память, чтение из

памяти, инициализация регистра непосредственным значением, передача управления и т.д.).

2. **Виртуальный адрес** гаджета, обнаруженного в адресном пространстве приложения. Является некоторой аналогией кода операции для архитектуры набора команд, которая задана каталогом гаджетов.
3. **Машинные инструкции** гаджета – конкретная последовательность инструкций, реализующих заданное семантическое описание. Может задаваться вручную при составлении каталога или заполняться при автоматическом анализе двоичного образа приложения.
4. **Параметры гаджета** – параметры семантического описания, а именно: конкретные регистры, константы и т.д.
5. **Побочные эффекты** выполнения гаджета с соответствующей семантикой.

Табл. 2. Заполненный каталог гаджетов
Table 2. Complete gadget catalogue

Семантическое описание	Виртуальный адрес	Машинные инструкции	Параметры гаджета	Побочные эффекты
r1 += r2	0xdeadbeef	add eax, ebx pop edx ret	r1 = eax r2 = ebx	edx ✗
	0xcafecafe	add eax, ebx pop ecx ret	r1 = eax r2 = ebx	ecx ✗
	0xcafebabe	add edx, ecx ret	r1 = edx r2 = ecx	—
r = M[ESP + Offset]	0x12345678	pop eax ret	r = eax Offset = 0	—
r1 = M[ESP + Off1] r2 = M[ESP + Off2] r3 = M[ESP + Off3]	0x10203040	pop eax pop ebx pop ecx ret	r1 = eax, Off1 = 0 r2 = ebx, Off2 = 4 r3 = ecx, Off3 = 8	—

Побочным эффектом является любое изменение памяти и регистров, не описываемое семантикой гаджета. Побочные эффекты могут задаваться при построении каталога гаджетов вручную или автоматически вычисляться в процессе классификации гаджетов.

Для пояснения данного определения приведем пример. В табл. 2 представлен заполненный каталог гаджетов, состоящий из нескольких семантических описаний. Первое семантическое описание соответствует операции сложения значений двух регистров *r*1 += *r*2. Второе семантическое описание соответствует инструкции загрузки значения со стека в регистр. Последнее семантическое описание определяет гаджет загрузки трех регистров со стека. Первые два гаджета, найденные по адресам 0xdeadbeef и 0xcafecafe, обладают побочными эффектами относительно основного семантического описания, потому что они изменяют значения регистров `edx`, `ecx` соответственно.

4.1 Полный по Тьюрингу каталог гаджетов

Авторы многих работ [5, 7, 18, 19, 54, 55] составляют каталог гаджетов таким образом, чтобы набор семантических описаний являлся полным по Тьюрингу. Такой каталог гаджетов задает некоторую новую вычислительную машину, способную выполнять произвольные вычисления.

В процессе поиска и определения семантики гаджетов происходит заполнение каталога адресами найденных гаджетов. После этого возможны две ситуации:

1. для каждого семантического описания удалось найти конкретный гаджет;

2. для некоторых семантических описаний отсутствуют конкретные гаджеты.

В первом случае получается, что каталог гаджетов реализует на конкретном исполняемом файле вычислительную машину, способную производить произвольные вычисления. Более того, можно использовать этот каталог в качестве описания набора команд целевой архитектуры для компилятора языка Си (llvm [7]).

Во втором случае, когда отсутствуют гаджеты для некоторых семантических описаний полного по Тьюрингу каталога гаджетов, произвольные вычисления уже не выполнить. Нужно пытаться построить эксплойт из имеющихся гаджетов. В худшем случае, задача генерации может свестись к перебору всех возможных комбинаций гаджетов.

Для составления полных по Тьюрингу ROP цепочек необходимо уметь условно изменять указатель стека, который выступает в роли счетчика команд. Ремер (Roemer) и др. [7] предлагают следующий способ реализации условного ветвления для архитектуры x86:

1. выполнить некоторую операцию, которая обновит интересующий флаг;
2. скопировать интересующий флаг из регистра флагов в регистр общего назначения;
3. использовать этот флаг для условного изменения указателя стека на желаемое смещение (например, путем умножения смещения на значение флага 0 или 1).

5. Поиск гаджетов

Независимо от способа построения эксплойта необходимо в первую очередь найти в двоичном образе приложения все доступные гаджеты. К задаче поиска гаджетов существует два принципиальных подхода. Первый из них предлагает осуществлять поиск гаджетов по списку шаблонов. Шаблоны, как правило, задаются регулярными выражениями над бинарными кодами команд гаджетов. Изначально каталог гаджетов содержит семантические описания гаджетов. Для каждого семантического описания производится поиск гаджетов по некоторому шаблону. В результате в каталог гаджетов для семантических описаний будут добавлены конкретные гаджеты: виртуальные адреса, машинные инструкции и параметры гаджетов. Побочные эффекты (например, испорченные регистры [29, 33]) могут быть получены после анализа машинных инструкций найденных гаджетов.

Второй подход заключается в автоматическом поиске всевозможных последовательностей инструкций, заканчивающихся инструкцией передачи управления. Классическим алгоритмом, реализующим поиск всех гаджетов, является алгоритм Галилео [5]. Алгоритм сначала ищет инструкции передачи управления в исполняемых секциях программы. Для каждой найденной инструкции пробует дизассемблировать несколько байтов, предшествующих инструкции. Все корректно дизассемблированные последовательности инструкций добавляются в каталог гаджетов. Таким образом, каталог будет содержать виртуальные адреса и машинные инструкции гаджетов. Данный алгоритм используется во многих инструментах поиска гаджетов с открытым исходным кодом [27–33, 56–58].

6. Определение семантики гаджетов

Не все найденные гаджеты пригодны для составления ROP цепочек. Для использования гаджета при составлении ROP цепочки необходимо понимать, какую полезную нагрузку выполняет этот гаджет. Определение семантики гаджета может производится вручную [5]. При шаблонном поиске гаджетов семантика содержится в описании шаблона [7, 9, 13, 17–20, 54, 59].

6.1 Типы гаджетов

Шварц и др. [6] предложили определять функциональность гаджета его принадлежностью к некоторым параметризованным типам, которые задают новую архитектуру набора команд (ISA). Параметрами типов выступают регистры, константы и бинарные операции. Тип гаджета описывается семантически с помощью постусловия – булева предиката $\mathcal{B}$, который должен быть всегда истинным после выполнения гаджета. Следует отметить, что один гаджет может принадлежать сразу нескольким типам. Например, гаджет `push eax ; pop ebx ; pop ecx ; ret` одновременно перемещает `eax` в `ebx` и загружает значение со стека в `ecx`, что соответствует типам `MoveRegG: ebx ← eax` и `LoadConstG: ecx ← [esp + 0]`.

Для того чтобы определить, удовлетворяет ли последовательность инструкций гаджета $\mathcal{I}$ постусловию $\mathcal{B}$, Шварц и др. [6] используют известную технику из формальной верификации – вычисление слабейшего предусловия [60]. Проще говоря, слабейшее предусловие $wp(\mathcal{I}, \mathcal{B})$ для последовательности инструкций для последовательности инструкций и постусловия $\mathcal{B}$ – это булево предусловие, которое описывает, когда $\mathcal{I}$ завершается в состоянии, удовлетворяющем $\mathcal{B}$. Слабейшее предусловие используется, чтобы убедиться, что определение семантики гаджета всегда выполняется после выполнения последовательности инструкций $\mathcal{I}$. Для этого достаточно проверить: $wp(\mathcal{I}, \mathcal{B}) \equiv true$. Если формула верна, то $\mathcal{B}$ всегда истинно после выполнения $\mathcal{I}$, а значит, $\mathcal{I}$ – гаджет с семантическим типом $\mathcal{B}$.

Однако формальная верификация семантики гаджетов на практике показала себя очень медленной. Для ускорения процесса авторы предложили комбинированный подход. Инструкции гаджета предварительно несколько раз выполняются с использованием случайных входных данных, а затем проверяется истинность $\mathcal{B}$. Если $\mathcal{B}$ окажется ложным хотя бы для одного выполнения, то последовательность инструкций не может быть гаджетом этого типа. Таким образом, более сложное вычисление слабейшего предусловия производится, только если $\mathcal{B}$ истинно для каждого выполнения.

Комбинированный подход можно условно разделить на два этапа: классификацию гаджетов и верификацию гаджетов. На этапе классификации делаются гипотезы о принадлежности гаджетов к некоторым типам и о значениях параметров этих типов. Гипотезы по сути задаются булевыми постусловиями. А на этапе верификации для каждого постусловия формально доказывается его истинность или ложность, и гипотеза принимается или отвергается соответственно.

6.1.1 Классификация гаджетов

В настоящее время существует множество процессорных архитектур с различными инструкциями. Промежуточное представление машинных инструкций (VEX [61], REIL [62], Pivot [63] и др.) позволяет абстрагироваться от конкретной архитектуры и писать универсальные алгоритмы.

В работах [44, 64] классификация гаджета производится на основе интерпретации промежуточного представления инструкций гаджета. Во время интерпретации отслеживаются обращения к регистрам и памяти. Если происходит первое чтение регистра или области памяти, считанное значение генерируется случайным образом. В результате интерпретации будут получены начальные и конечные значения регистров и памяти. На основе этой информации делается вывод о возможной принадлежности гаджета тому или иному типу. Например, для принадлежности типу `MoveRegG` [6] должна существовать такая пара регистров, что начальное значение первого регистра равно конечному значению второго. В результате анализа составляется список всех удовлетворяющих гаджету типов и их параметров (список кандидатов). Затем

производится еще несколько запусков процесса интерпретации с отличными входными данными, в результате которых из списка кандидатов удаляются ошибочно определенные типы.

Более того, в результате классификации гаджета могут быть получены [44]:

- список «испорченных» регистров, значения которых изменились в результате выполнения гаджета;
- информация о фрейме гаджета (разд. 2.1): размер фрейма и смещение ячейки с адресом следующего гаджета относительно начала фрейма.

Следует отметить, что число неверно классифицированных гаджетов можно уменьшить, если добавить запуски процесса интерпретации с граничными входными данными 0 и -1. Доля неверно классифицированных гаджетов в таком случае незначительна и составляет 0.7 % [65].

6.1.2 Верификация гаджетов

Классификация гаджета предоставляет набор постусловий, описывающих возможную семантику гаджета. Верификация гаджета позволяет формально доказать истинность этих постусловий для произвольных входных данных. Верификация гаджета может производиться как на основе построения слабейшего предусловия [6, 23], как было описано выше, так и с использованием символьной интерпретации инструкций гаджета [64, 65].

Табл. 3. Пример верификации гаджета `ArithmeticLoadG: ebx ← ebx + [eax]`
Table 3. Verification of gadget `ArithmeticLoadG: ebx ← ebx + [eax]`

Шаг	Символьное состояние	Инструкция	Множество формул
initial	$M, eax = \phi_1, ebx = \phi_2, ecx = \phi_3, esp = \phi_4, eip = \phi_5$	—	$S_0 = \emptyset$
1	$ecx = \phi_6$	`mov ecx, [eax]`	$S_1 = S_0 \cup \{\phi_6 = (concat\ (select\ M\ \phi_1)\ (select\ M\ \phi_1 + 1)\ (select\ M\ \phi_1 + 2)\ (select\ M\ \phi_1 + 3))\}$
2	$ebx = \phi_7$	`add ebx, ecx`	$S_2 = S_1 \cup \{\phi_7 = \phi_2 + \phi_6\}$
final	$eip = \phi_8, esp = \phi_9$	`ret`	$S_3 = S_2 \cup \{\phi_8 = (concat\ (select\ M\ \phi_4),\ (select\ M\ \phi_4 + 1),\ (select\ M\ \phi_4 + 2),\ (select\ M\ \phi_4 + 3)),\ \phi_9 = \phi_4 + 4\}$
	Определение семантики		**Верификация**
verify	$final(ebx) = initial(ebx) + initial(M[eax])$		$\phi_7 \neq \phi_2 + (concat\ (select\ M\ \phi_1)\ (select\ M\ \phi_1 + 1)\ (select\ M\ \phi_1 + 2)\ (select\ M\ \phi_1 + 3))$ is **UNSAT**

Рассмотрим подробнее способ классификации гаджета с использованием символьной интерпретации [51–53]. Во время символьной интерпретации моделируется семантика гаджета с использованием SMT выражений. Изначально всем регистрам присваиваются свободные символьные переменные. Символьная память в начале представляет из себя пустой байтовый массив M битовых векторов:

$$M = (Array\ (\_\ BitVec\ \langle addrSize \rangle)\ (\_\ BitVec\ 8)),$$

где ⟨*addrSize*⟩ – размерность адресного слова архитектуры. Символьное состояние содержит отображение регистров в символьные переменные и текущее состояние символьной памяти. Символьная интерпретация инструкций гаджета порождает SMT формулы над переменными и константами, а также обновляет символьное состояние регистров и памяти в соответствии с операционной семантикой инструкции. Работа с символьной памятью реализуется через операции *select* и *store* над *Array*. Функция (*select M i*) возвращает *i*-ый элемент массива *M* и моделирует чтение байта по адресу *i*. Функция (*store M i b*) возвращает массив, полученный из массива *M* сохранением элемента *b* по индексу *i*, что моделирует запись байта *b* по адресу *i*.

Хейтман (Heitman) и др. [64] сначала транслируют инструкции гаджета в промежуточное представление REIL [62]. А после уже производится символьная интерпретация REIL инструкций.

Постусловие для верификации семантики гаджета представляет из себя булевый предикат над начальными и конечными значениями регистров и памяти. В предикат подставляются регистры и память из соответствующих символьных состояний. Общезначимость формулы постусловия проверяется через невыполнимость ее отрицания с использованием SMT-решателя.

В табл. 3 приводится пример верификации гаджета `ArithmeticLoadG: ebx ← ebx + [eax]`. Изначально регистрам сопоставляются свободные символьные переменные ϕ_i, а память представляется массивом *M*. Множество формул пусто. Новые формулы добавляются в соответствии с операционной семантикой интерпретируемой инструкции. Для множества формул поддерживается SSA форма – при добавлении формулы создается новая символьная переменная, которой присваивается эта формула. На первом шаге создается новая символьная переменная ϕ_6, которая равна загруженному из памяти значению второго операнда инструкции `[eax]`. В символьном состоянии регистру *ecx* ставится в соответствие символьная переменная ϕ_6. Аналогично интерпретируются оставшиеся шаги. В постусловие, описывающее тип гаджета, подставляются символьные переменные из начального и конечного символьных состояний. При помощи SMT-решателя проверяется выполнимость отрицания формулы. Отрицание формулы невыполнимо, значит, гаджет удовлетворяет заявленному типу с параметрами.

Например, гаджет `neg eax ; sbb eax, eax ; and eax, ecx ; pop ebp ; ret` может быть неверно классифицирован, а верификация позволит устранить эту ошибку. Во время классификации гаджет был отнесен к типу `MoveRegG: EAX ← ECX`. Для отличного от нуля начального значения регистра `eax` гаджет действительно скопирует значение регистра `ecx` в `eax`. Однако, если начальное значение `eax` будет нулевым, то и его конечное значение будет нулевым, что не является копией значения регистра `ecx`, отличного от нуля.

6.2 Резюме гаджетов

Резюме гаджета [8, 24] представляет собой описание семантики гаджета в виде компактной спецификации. Резюме гаджета содержит предусловия и постусловия над значениями регистров и памяти. В частности, резюме гаджета может содержать:

- регистры, загруженные со стека (`eax = [esp + 4]`);
- регистры, считанные из памяти (`ecx = [edx + 2]`);
- регистры, значение которых было изменено (`ecx = eax + ebx`);
- диапазоны адресов памяти, по которым производились чтение или запись (`[rsp] <-> [rsp + 0x20]`).

Фоллнер (Follner) и др. [24] предложили следующий метод составления резюме гаджета. Сначала инструкции гаджета поднимаются до уровня промежуточного представления VEX [61]. Потом продвигаются все присваивания, таким образом, чтобы сформировать в результате одно выражение, называемое постусловием, которое описывает все операции, с помощью которых получилось конечное значение в рассматриваемом регистре. Анализ поддерживает модель памяти, которая позволяет корректно моделировать ситуацию передачи значения через стек. Также этот анализ позволяет получить предусловия, которые описывают диапазоны доступа к памяти по регистру со смещением (`[rax] <-> [rax + 0x20]`). Предусловия указывают на то, что регистры из этих диапазонов памяти должны указывать на память, доступную для чтения/записи.

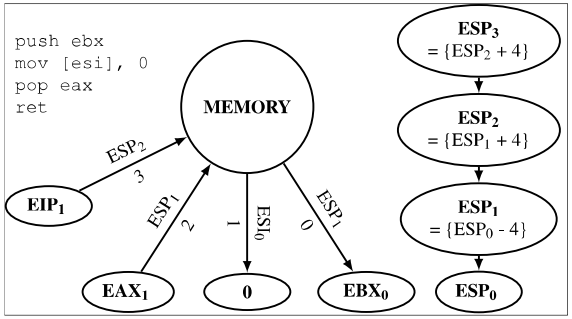

Рис. 3. Граф зависимостей гаджета

Fig. 3. Gadget dependency graph

6.3 Граф зависимостей гаджета

Миланов (Milanov) [25] предложил представлять гаджет в виде ориентированного графа зависимостей (рис. 3). Вершины соответствуют регистрам, памяти и константам. Вся память представляется единственной вершиной. Регистр же может соответствовать нескольким вершинам: каждая модификация регистра порождает новую вершину (reg_0, reg_1, reg_2 и т.д.). Направленные ребра отражают зависимости по данным (присваивание регистров, доступ к памяти и др.). Инструкции гаджета порождают новые ребра на графе. Ребра, соединенные с памятью, также содержат метки с адресом доступа к памяти и пронумерованы в хронологическом порядке.

Каталог гаджетов (разд. 4) заполняется следующим образом. Изначально каталог содержит виртуальные адреса и инструкции гаджетов. Инструкции каждого гаджета транслируются в промежуточное представление REIL [62], для которого после строится граф зависимостей. В результате обхода графа вычисляется семантическое описание гаджета: конечные значения регистров и памяти выражаются через начальные. Выражение для некоторого конечного значения может иметь условие, при котором это выражение истинно. Далее гаджеты классифицируются по типам (6.1). Автор мотивировал такой метод определения семантики гаджета меньшим временем работы по сравнению с методами, использующими SMT-решатели. Для примера на рис. 3 будет получено следующее семантическое описание:

$$EIP_1 = MEM[ESP_0]$$
$$ESP_3 = ESP_0 + 4$$
$$EAX_1 = \begin{cases} 0 & \text{if } ESP_0 - 4 = ESI_0 \\ EBX_0 & \text{if } ESP_0 - 4 \neq ESI_0 \end{cases}$$
$$MEM[ESP_0 - 4] = \begin{cases} 0 & \text{if } ESP_0 - 4 = ESI_0 \\ EBX_0 & \text{if } ESP_0 - 4 \neq ESI_0 \end{cases}$$
$$MEM[ESI_0] = 0$$

7. Генерация цепочек гаджетов

В данном разделе описываются различные методы генерации ROP цепочек. Следует отметить, что комбинирование гаджетов в цепочки является переборной задачей, поэтому для уменьшения числа итераций перебора можно предварительно отфильтровывать ненужные гаджеты и сортировать их по качеству [66]. Процесс генерации ROP цепочек отличается от обычной компиляции следующим.

- Чаще всего у ROP цепочки нет возможности сохранять значения регистров в память для их последующего восстановления из-за нехватки соответствующих гаджетов.
- У ROP гаджетов могут быть побочные эффекты. Например, гаджет может «портить» регистры. Значения «испорченных» регистров не сохранятся после выполнения гаджета. Побочные эффекты необходимо учитывать при составлении расписания гаджетов из цепочки [6].
- Некоторые типы гаджетов (6.1), которые выступают в качестве инструкций виртуальной машины, могут отсутствовать. В таком случае необходимо заменять недостающие гаджеты последовательностью других [6].

Во время генерации следует учитывать запрещенные символы, которые нельзя использовать в ROP цепочке. Такая потребность возникает, когда, например, переполнение происходит при помощи функции `strcpy`, что не позволяет цепочке содержать нулевые байты. Однако только немногие [22, 31] полноценно решают проблему запрещенных символов. Большинство просто удаляют гаджеты, чьи адреса содержат запрещенные символы, но не следят за значениями параметров гаджетов на стеке.

ROP нагрузка часто может быть разбита на установку значений регистров в заданные значения и выполнение еще одного гаджета [32]. Таким образом, метод генерации ROP цепочек можно базировать на установке регистров, а остальные нагрузки осуществлять путем добавления к полученной цепочке одного гаджета записи в память, вызова функции, системного вызова и др.

7.1 Полная по Тьюрингу компиляция с фиксированным каталогом гаджетов

Рассмотрим построение компилятора на основе фиксированного каталога гаджетов. Бьюкенен (Buchanan) и др. [7, 13] вручную сформировали полный по Тьюрингу каталог гаджетов из машинного кода стандартной библиотеки `libc` ОС Solaris для архитектуры набора команд SPARC. Каждому семантическому описанию сопоставляется единственная последовательность инструкций из машинного кода библиотеки. Архитектура SPARC допускает только выровненный доступ к инструкциям, поэтому все примеры гаджетов являются легитимными эпилогами функций библиотеки.

Одна из особенностей архитектуры SPARC – использование регистровых окон. Регистровое окно состоит из регистров, предназначенных для входных параметров, возвращаемых значений, временных значений внутри процедуры. При вызове функции

происходит сдвиг регистрового окна вперед, а при возврате – в обратную сторону. При большой вложенности стека вызовов в программе происходит нехватка регистровых окон, что приводит к необходимости их сохранения на стеке. В таком случае, при возврате из функции значения регистров восстанавливаются из сохраненных на стеке значений, что приводит к нежелательному изменению значений регистров при передаче управления от одного гаджета к другому. Таким образом, архитектура SPARC и ее соглашение о вызовах накладывает ограничения на способ передачи вычисленных значений между гаджетами – только через память. Каталог гаджетов Бьюкенена и др. [13] реализует набор гаджетов, использующих модель память-память, которая позволяет использовать регистры только внутри гаджетов, а хранение и передача значений от одного гаджета к другому происходит через память. Каждой переменной в ROP цепочке сопоставляется адрес ячейки памяти, который используется как операнд гаджета.

После полного заполнения каталога гаджетов существует две опции для автоматического создания ROP цепочек. Во-первых, у каталога гаджетов существует программный интерфейс на языке Си. В нем содержатся 13 функций, которые позволяют создавать переменные, присваивать им значения и вызывать функции (или делать системные вызовы). С помощью данного программного интерфейса можно написать программу, которая будет автоматически генерировать ROP цепочку по заполненному каталогу гаджетов. Во-вторых, Бьюкенен и др. [13] написали транслятор из некоторого псевдо-языка описания эксплойтов (урезанного Си) в последовательность вызовов функций программного интерфейса каталога гаджетов на языке Си. Компилятор реализует большую часть базовой арифметики, логических операций, операций работы с указателями и памятью, и операций условной и безусловной передачи управления. Авторами ряда работ [13, 23] отмечается возможность написания для компиляторной инфраструктуры LLVM расширения, позволяющего генерировать код для виртуальной машины, задаваемой каталогом гаджетов.

Инструмент, представленный Мосьером (Mosier) [26, 57], опирается на ROPC-IR, ассемблерный язык описания эксплойтов, который задает полную по Тьюрингу архитектуру набора команд. Он содержит три регистра: `ACC(eax)`, `SP(rbp)`, `PC(rsp)`, и набор базовых команд: арифметические операции, инструкции передачи управления, инструкции для работы с памятью и стеком. В качестве счетчика команд `PC` выступает регистр `rsp`. Кроме того, выделяется отдельный стек для функций ROP цепочки, указателем на который `SP` выступает `rpb`. Поддержка второго стека позволяет реализовывать вызовы функций внутри ROP цепочки, которые реализуют полноценные подпрограммы.

Каталог гаджетов в данном инструменте представлен описанием языка ROPC-IR. Процесс поиска гаджетов выводит всевозможные гаджеты из целевой программы. Затем необходимо вручную найти и сопоставить каждому семантическому описанию конкретный найденный в целевой программе гаджет и вручную заполнить следующие поля каталога гаджетов: виртуальный адрес, параметры гаджета. По определению языка ROPC-IR гаджеты не имеют побочных эффектов (не принимая во внимание выходные регистры). Теоретически ассемблерный язык ROPC-IR может выступать в качестве целевого языка компилятора Си. Однако практическое применение для построения эксплойтов для реальных программ может быть существенно ограничено наличием необходимых гаджетов в программе и размером генерируемых ROP цепочек. Размер цепочек из-за неоптимальности процесса трансляции языка ROPC-IR значительно больше типичных размеров эксплойтов.

Подход к построению автоматизированного инструмента генерации ROP цепочек, предложенный в работах [7, 13, 26], опирается на фиксированный каталог гаджетов. Он единожды сформирован авторами и не изменяется. Кроме того, семантические описания жестко привязаны к конкретным регистрам, используемым в гаджетах. В случае, если в

какой-то версии стандартной библиотеки `libc` отсутствует какой-то из гаджетов, то компиляция ROP цепочки может завершиться неудачно. При этом можно было бы использовать другие гаджеты, имеющие другие операнды, но схожий функционал, и добиться успешной компиляции. Другими словами, данный подход обладает ограниченной практической применимостью, особенно в ситуациях небольшого количества гаджетов в исследуемом коде библиотеки.

7.2 Генерация на основе шаблонов гаджетов

Генерация на основе шаблонов гаджетов заключается в поиске по регулярным выражениям определенной последовательности гаджетов, выполняющей некоторую вредоносную нагрузку: системный вызов `execve` [30, 33], вызов функции `VirtualProtect` с последующим выполнением обычного шелл-кода на стеке [29] и др. Запрещенные символы при таком подходе могут учитываться путем отрицания загруженного со стека значения, многочисленным повторным инкрементом до желаемого значения или же другими арифметическими операциями. Следует отметить, что Ropper [33] поддерживает поиск с использованием SMT-решателей гаджетов, удовлетворяющих семантике, которая задается постусловием над регистрами, памятью и константами. Однако на момент написания статьи для генерации ROP цепочек инструмент использует только регулярные выражения.

В работе Хуана (Huang) и др. [11] для архитектуры набора команд ARM применяется подход, основанный на использовании специального гаджета, который одновременно устанавливает значения всех регистровсо стека. Алгоритм поиска и одновременной проверки гаджета на соответствие заданной семантике производится путем анализа инструкций ассемблерного кода. Генерация цепочки из одного гаджета является тривиальной задачей и требует только правильного расположения значений регистров на стеке.

Другой подход к построению каталога гаджетов и последующей компиляции представлен Хундом (Hund) и др. [20]. На этапе поиска ищутся только гаджеты, состоящие из одной инструкции, не считая саму инструкцию возврата. Скорее всего, так сделано ради упрощения алгоритмов анализа параметров гаджета и побочных эффектов. Такие гаджеты добавляются в каталог гаджетов. На следующем шаге каталог гаджетов дополняется гаджетами, которые можно скомбинировать из имеющихся. Это можно проиллюстрировать следующим примером:

1. `pop eax ; ret` – гаджет загрузки значения со стека в регистр `eax`,
2. `mov ebx, eax ; ret` – гаджет перемещения значения из регистра `eax` в регистр `ebx`.

Эти два гаджета, вызванные последовательно, образуют гаджет загрузки значения со стека в регистр `ebx`. Хунд и др. [20] приводят алгоритм поиска всех возможных комбинаций гаджетов, перемещающих значение из одного регистра в другой регистр. Данная задача сводится к поиску пути от одной вершины к другой в специальном графе. В качестве вершин в нем выступают регистры, а в качестве ребер выступают первичные гаджеты, осуществляющие перемещение одного значения регистра в другой.

Данный подход позволяет расширить каталог гаджетов, что особенно полезно для эксплуатируемых программ небольшого размера. Однако использование комбинированных гаджетов требует обязательного учета побочных эффектов на стадии объединения гаджетов в ROP цепочку.

Нгуен Ань Куинь (Nguyen Anh Quynh) [21] предложил похожую идею объединения нескольких гаджетов в один, выполняющий желаемое поведение. Например, последовательность гаджетов `push 0x1234 ; pop ebp ; ret ; xchg ebp, eax ; ret` может рассматриваться как один гаджет загрузки константы 0x1234 в регистр `eax`.

7.3 Связывание гаджетов в цепочки с использованием семантических запросов

Миланов [25] получает семантическое описание каждого гаджета путем построения его графа зависимостей (разд. 6.3). Связывание гаджетов в цепочки осуществляется последовательными семантическими запросами к каталогу гаджетов. Данный метод реализован в виде инструмента с открытым исходным кодом ROPGenerator [31].

Семантический запрос по сути является выражением над константами, конечными/начальными значениями регистров и памяти. Сначала в каталоге гаджетов ищутся гаджеты с семантическим описанием, удовлетворяющим семантическому запросу. Если такие гаджеты отсутствуют, то семантический запрос разбивается на несколько по некоторым стратегиям. Например, первый регистр можно переместить во второй через некоторый промежуточный третий регистр.

Примечательной особенностью инструмента ROPGenerator является поддержка использования при построении ROP цепочки гаджетов, оканчивающихся на инструкции `call Reg` и `jmp Reg`. Для этого перед такими гаджетами добавляется гаджет загрузки регистра `Reg`, который загружает значение адреса гаджета, которому необходимо передать управление после выполнения гаджета с `call` или `jmp`. В случае с `call` может также потребоваться передача управления на специальный гаджет, убирающий со стека адрес возврата, размещаемый `call`.

7.4 Генетический алгоритм

Фрейзер (Fraser) и др. [12] предлагают иной подход к конструированию ROP цепочки. Авторы предлагают использовать генетические алгоритмы для этого. Инструмент ROPER [28] позволяет генерировать для ARM архитектуры ROP цепочку, устанавливающую значения регистров в заданные значения.

Вначале в исполняемом файле находятся гаджеты. Для каждого из них вычисляется размер фрейма гаджета и смещение адреса следующего гаджета в нем (2.1). Затем исполняемый целевой файл загружается в адресное пространство виртуальной машины для повторяемого выполнения ROP цепочек-кандидатов. Виртуальная машина предоставляет удобный интерфейс для выполнения инструкций гостевой архитектуры.

В процессе генетических мутаций роль генов выполняют адреса гаджетов и случайные значения, размещаемые на стеке в качестве данных и адреса следующего гаджета. Функцией приспособленности является разность текущего и желаемого вектора значений регистров виртуальной машины. Каждый элемент популяции изменяется методами генетических мутаций. Среди всех кандидатов отбирается набор потенциально лучших, для которых процесс мутации повторяется вновь.

Следует отметить, что сформированные генетическим алгоритмом ROP цепочки сильно отличаются от тех, что создаются людьми. Например, они могут писать значения себе на стек или передавать управление на гаджеты, которые не были изначально обнаружены в процессе поиска. Кроме того, размер цепочки из-за неоптимального выбора гаджетов может быть большим. Описанные недостатки могут быть следствием отсутствия у генетического алгоритма информации о семантике инструкций гаджета. Возможно, если в каком-то виде ее учитывать и использовать более современные методы машинного обучения, то можно развить концепцию данного подхода в практически применимый инструмент.

Алгоритм 1. Алгоритм поиска кратчайших цепочек инициализации регистров
Algorithm 1. Search algorithm of shortest chain initializing registers

```
regset_to_chain ← empty register set mapping to shortest chains
queue ← empty queue
queue.push(empty chain)
while queue is not empty do
    chain ← queue.pop()
    for all gadget ∈ gadgets do
        new_chain ← chain + gadget
        regset ← controlled_registers(new_chain)
        if regset not in regset_to_chain or new_chain is shorter than regset_to_chain[regset] then
            regset_to_chain[regset] ← new_chain
            queue.push(new_chain)
        end if
    end for
end while
```

7.5 Генерация цепочек с использованием SMT-решателей

Фоллнер и др. [24] предложили метод генерации на основе резюме гаджетов (6.2), который имеет доступный исходный код [27]. Метод позволяет получать последовательность гаджетов, которая запишет в m запрошенных регистров заданные значения. Следует отметить, что метод не вычисляет загружаемые со стека параметры гаджетов, а только предоставляет последовательность адресов гаджетов. Изначально для всех гаджетов составляется резюме. Для каждого запрошенного регистра выбираются n наиболее подходящих гаджетов [66], которые загружают его значение со стека или из памяти, контролируемой атакующим. Далее для всевозможных цепочек гаджетов ($n^m * m!$ комбинаций) вычисляются предусловия и постусловия, но уже для всей цепочки. Если постусловия удовлетворяют ситуации, когда атакующий контролирует значения всех запрошенных регистров, то метод переходит к финальной стадии – разрешению предусловий. К цепочке дополнительно в начало добавляются гаджеты, которые проинициализируют регистры из предусловий, так чтобы они указывали на доступную для чтения и записи память.

Солс (Salls) [32] развил описанный выше метод. Ниже будет описан метод генерации цепочки, устанавливающей значения регистров в заданные значения. Остальные цепочки, такие как запись в память и вызов функции, могут быть получены добавлением всего лишь одного гаджета к цепочке, инициализирующей регистры. Метод можно разделить на три шага:

1. **Составление резюме гаджетов.** Происходит символьная интерпретация [51–53] инструкций каждого гаджета. Резюме гаджетов составляется с использованием статического анализа полученных в результате символьной интерпретации SMT выражений и запросов к SMT-решателю.
2. **Связывание гаджетов в цепочку.** На этом шаге происходит поиск кратчайших цепочек для инициализации произвольных наборов регистров. Предложенный алгоритм 1 был вдохновлен алгоритмом Дейкстры [67] поиска кратчайших путей от одной из вершин графа до всех остальных. Создается пустое отображение из наборов регистров в кратчайшие цепочки, инициализирующие эти регистры. В очередь добавляется пустая цепочка. Алгоритм достает цепочки из очереди. Для каждого гаджета создается новая цепочка, полученная добавлением этого гаджета к взятой из очереди цепочки. Вычисляется набор инициализируемых регистров (*controlled_registers*) новой цепочкой. Если такого набора нет в отображении, или полученная цепочка короче той, что в отображении, то в отображение для этого набора добавляется новая цепочка. Также эта же цепочка добавляется в очередь.

Таким образом, будет получено отображение из наборов регистров в кратчайшие цепочки, которые эти регистры инициализируют.

3. **Размещение ROP цепочки на стеке.** Запускается процесс символьной интерпретации инструкций всей ROP цепочки. Для значений, загружаемых со стека, создаются свободные символьные переменные. В конце процесса интерпретации составляется конъюнкция равенств запрошенных регистров заданным значениям. В результате решения этой конъюнкции SMT-решателем будут получены байты, которые необходимо разместить на стеке.

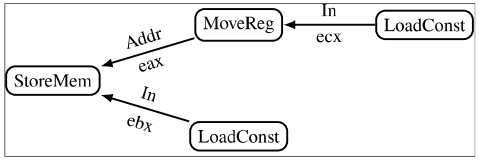

Рис. 4. Дерево гаджетов, которое записывает произвольное значение по произвольному адресу памяти
Fig. 4. Gadget tree that stores arbitrary value at arbitrary memory address

Описанный метод, в отличие от предыдущего, позволяет использовать в цепочках гаджеты, которые инициализируют сразу несколько регистров, а также гаджеты, которые выполняют арифметическую операцию над регистрами, загруженными другими гаджетами (правильные значение на стеке при этом вычислит SMT-решатель). Более того, данный метод позволяет выбирать самые короткие цепочки.

7.6 Генерация на основе семантических деревьев

Шварц и др. [6] предлагают подход к генерации ROP цепочек на основе семантических деревьев. Авторы создали свой язык QooL для написания ROP цепочек, который не обладает полнотой по Тьюрингу, но позволяет выражать применяемые на практике ROP цепочки (вызов библиотечной функции, системный вызов и запись в память). Процесс трансляции программы на языке QooL в ROP цепочку состоит из следующих этапов.

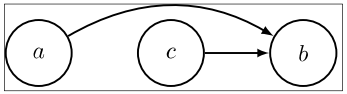

Рис. 5. Построение расписания для дерева гаджетов
Fig. 5. Scheduling gadget tree

1. Генерация семантических деревьев путем замощения [68] абстрактного синтаксического дерева исходной программы на языке QooL. Семантическое дерево состоит из абстрактных гаджетов (типов гаджетов), которые задают архитектуру набора команд и описаны в разделе 6.1.
2. Присвоение абстрактным гаджетам из семантического дерева реальных гаджетов, найденных в программе. Пример дерева реальных гаджетов приводится на рисунке 4. В вершинах дерева записаны типы гаджетов. На ребрах – имена параметров типов и их значения (конкретные регистры). Дерево гаджетов производит запись произвольного значения по произвольному адресу памяти. Записываемое значение и адрес загружаются со стека в регистры *ebx* и *ecx* соответственно. Адрес из регистра *ecx* перемещается в регистр *eax*. После чего уже происходит запись значения регистра *ebx* по адресу *eax*.
3. Построение расписания по дереву гаджетов и генерация ROP цепочки.

На первом шаге происходит ленивая генерация всех возможных семантических деревьев из абстрактных гаджетов. Это необходимо делать поскольку некоторые гаджеты могут отсутствовать в конкретной программе. На втором шаге для каждого семантического дерева применяется присвоение гаджетов. В случае, если не удается присвоить каждому абстрактному гаджету конкретный, то семантическое дерево отбрасывается и берется следующее. В случае успешного присваивания на третий шаг передается дерево реальных гаджетов. Для генерации ROP цепочки его необходимо линеаризовать, т.е. построить расписание. Построение расписания для дерева гаджетов должно учитывать: зависимости между регистрами гаджетов по данным и «испорченные» регистры. Это означает следующее (рис. 5):

1. расписание должно удовлетворять топологической сортировке дерева;
2. если выходной регистр гаджета a используется гаджетом b, то этот регистр не должен быть «испорчен» ни одним гаджетом в расписании между a и b.

При генерации семантических деревьев учитывается возможность отсутствия некоторых типов гаджетов и применяются последовательно все имеющиеся правила выражения вершины абстрактного синтаксического дерева через семантические деревья из абстрактных гаджетов. Например, авторы заметили, что успешность генерации ROP цепочки возрастает, если добавить следующее правило выражения вершины сохранения значения в память:

1. `mov [eax], 0 ; ret`
2. `pop ebx ; ret`
3. `add [eax], ebx ; ret`

Оуян (Ouyang) и др. [23] расширили набор инструкций языка QooL до полного по Тьюрингу набора. В целом они повторяют подход Шварца и др. [6] с построением семантических деревьев, используя при учете побочных эффектов анализ жизни значений. Следует отметить, что существуют попытки реализации метода Шварца и др., имеющие открытый исходный код [34, 36, 69].

8. Учет запрещенных символов

Автоматические инструменты генерации ROP цепочек должны учитывать особенности санитизации входных данных для конкретного эксплойта. Например, данные, копируемые через функцию `strcpy`, не могут содержать нулевые байты.

Байты, требующие санитизации, могут содержаться как в адресах гаждетов, так и в данных, загружаемых этими гаджетами на регистры. В простейшем случае санитизация адресов гаджетов производится путем отбрасывания гаджетов, содержащих запрещенные символы в адресе. Так поступают многие инструменты. Однако данный подход неизбежно приходит к уменьшению каталога гаджетов, что приводит к нехватке гаджетов и необходимости их комбинирования для моделирования недостающих гаджетов.

Гораздо сложнее обстоит ситуация, когда запрещенные символы содержатся в данных, предназначенных для загрузки на регистры (значения аргументов функций и необходимые для записи в память значения). Для решения данной проблемы можно использовать всевозможные арифметические операции для получения значений, содержащих запрещенные символы.

Подробное описание способов борьбы с запрещенными символами описано в статье Дина (Ding) и др. [22]. Стоит отметить, что авторы статьи борются со всеми непечатаемыми символами в цепочках, что может быть избыточно в некоторых случаях. Однако их методы применимы и в более общем случае произвольного заданного множества запрещенных символов. Для каждого найденного гаджета авторы строят семантическое

дерево, описывающее функциональность гаджета и содержащее явные зависимости между регистрами и памятью относительно арифметических операций и операций взаимодействия с памятью.

Построенные семантические деревья используются при построении конечного автомата (рис. 6), используемого для поиска инструкций, загружающих значение на регистр. Вершинами в конечном автомате являются следующие состояния, отвечающие разным загружаемым значениям:

- Z – ноль,
- SI – небольшое число,
- GC – число, не содержащее запрещенных символов,
- BC – число, содержащее запрещенные символы,
- T – конечное состояние.

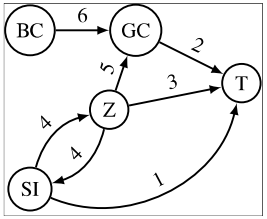

Рис. 6. Конечный автомат, описывающий алгоритм санитизации значений загрузки на регистр
Fig. 6. State machine describing the input sanitizing algorithm

Между этими вершинами проводятся ребра, соответствующие определенным гаджетам, при условии их наличия в каталоге гаджетов. Возможные варианты перехода между состояниями конечного автомата:

1. $SI \longrightarrow T$, из вершины с небольшим числом в конечное состояние ведет ребро, соответствующее гаджету с инструкцией, непосредственно устанавливающей это значение в регистре.
2. $GC \longrightarrow T$, из вершины с числом, не содержащим запрещенных символов, в конечное состояние ведет ребро с гаджетом `pop`.
3. $Z \longrightarrow T$, из вершины с нулем в конечное состояние ведет ребро с инструкцией `xor`.
4. $SI \longleftrightarrow Z$, из вершины с небольшим числом в нуль и обратно ведут ребра с инструкциями `inc, dec`.
5. $Z \longrightarrow GC$, из вершины с нулем в состояние с числом, не содержащим запрещенных символов, ведет ребро с арифметическими инструкциями `and, or, sal, shl, shr, sar`.
6. $BC \longrightarrow GC$, из вершины с числом, содержащим запрещенные символы, в вершину, не содержащую запрещенных символов, ведут ребра, состоящие из комбинации двух арифметических операций, например, $a + b - c$.

Работа алгоритма начинается из состояния, соответствующего значению, которое нужно установить в определенном регистре. Путем обхода состояний данного автомата решается вопрос возможной санитизации данных ROP цепочки. Алгоритм прерывается, если достигнуто конечное состояние, что соответствует успешному нахождению комбинации гаджетов, решающих поставленную задачу, или в случае отсутствия переходов в другие состояния из текущего.

9. Экспериментальное сравнение инструментов

Экспериментальная проверка инструментов с доступным исходным кодом проводилась с помощью тестовой системы rop-benchmark [37]. Данная система позволяет проверять работоспособность ROP цепочек, генерируемых инструментами. Система предоставляет воспроизводимое окружение для проверки факта успешной генерации и работоспособности ROP цепочек, осуществляющих системный вызов `execve("/bin/sh", 0, 0)`. Система тестирования поддерживает платформу Linux x86-64. В качестве тестовых наборов взяты исполняемые файлы и библиотеки минимальных установок нескольких популярных дистрибутивов: CentOS 7, Debian 10, OpenBSD 6.2, OpenBSD 6.4. Дистрибутивы OpenBSD 6.2 и 6.4 взяты по причине того, что авторы этой операционной системы ведут целенаправленную борьбу с ROP гаджетами [70].

Табл. 4. Экспериментальное сравнение инструментов автоматической генерации ROP цепочек
Table 4. The experimental evaluation of automatic ROP chain generating tools

Тестовый набор	Debian 10			CentOS 7			OpenBSD 6.2			OpenBSD 6.4		
Кол-во файлов	689			649			397			410		
Инструмент	OK	F	TL	OK	F	TL	OK	F	TL	OK	F	TL
ROPgadget [1]	7	0	0	8	0	0	4	0	0	2	0	0
angrop [2]	87	8	4	53	6	1	24	1	5	7	1	5
ROPGenerator [3]	83	4	20	66	5	3	24	3	13	1	0	12
Ropper [4]	53	33	0	31	37	0	14	14	1	1	0	2

Результаты экспериментальной проверки приводятся в таблице 4. Четыре столбца соответствуют четырем наборам тестовых файлов. В первой строке указано общее количество тестовых файлов в каждом из наборов. Ниже находятся строки с инструментами, и напротив каждого инструмента указана следующая информация:

- OK – количество тестовых файлов, для которых созданная ROP цепочка работоспособна, т.е. приводит к открытию оболочки системного интерпретатора.
- F – количество тестовых файлов, для которых созданная ROP цепочка не является работоспособной, т.е. по каким-то причинам не приводит к открытию оболочки системного интерпретатора.
- TL – количество тестовых файлов, на которых время работы инструмента превысило установленный лимит в 300 секунд.

В экспериментальное сравнение попали только находящиеся в открытом доступе инструменты, которые способны в полностью автоматическом режиме генерировать ROP цепочку, осуществляющую системный вызов для архитектуры x86-64 с операционной системой семейства Linux. Из-за операционной системы не попал в рассмотрение инструмент mona.py [29]. Другие могут работать только с архитектурой x86 (32-битной) [58], ARM [28]. Некоторые доступные инструменты не удалось успешно встроить в автоматизированную систему запуска [34].

В набор тестовой системы не вошли тесты с запрещенными символами (например, 0x00 в случае переполнения при копировании с `strcpy`) по причине того, что только ROPGenerator [31] поддерживает их в полном объеме, т.е. проверяет на наличие запрещенных символов не только адреса гаджетов, но и значения параметров гаджетов на стеке.

10. Заключение

В данной статье был проведен подробный обзор методов автоматизированной генерации эксплойтов для атак повторного использования кода. Атаки повторного использования кода предполагают использование кусочков кода из адресного пространства программы,

называемых *гаджетами*. Гаджеты связываются в цепочку, выполняющую вредоносную нагрузку. Схематично процесс генерации эксплойтов повторного использования кода делится на четыре этапа: поиск гаджетов в эксплуатируемой программе, определение семантики гаджетов, комбинация гаджетов в цепочки и генерация входных данных, эксплуатирующих уязвимость. Найденные в программе гаджеты добавляются в каталог гаджетов. После этого происходит определение семантики гаджетов: классификация гаджетов по параметризованным семантическим типам, составление резюме гаджетов или построение графов зависимостей гаджетов. Если набор гаджетов в каталоге полон по Тьюрингу, то гаджеты из каталога можно использовать в качестве целевой архитектуры набора команд компилятора. Связывание гаджетов в цепочки может происходить как поиском гаджетов по шаблонам, задаваемых регулярными выражениями, так и с учетом семантики гаджета. Также существуют подходы конструирования ROP цепочек с использованием генетических алгоритмов, а также методы с использованием SMT-решателей.

Мы предложили набор тестов ROP Benchmark [37] для экспериментального сравнения инструментов генерации ROP цепочек. С помощью него было проведено сравнение инструментов генерации ROP цепочек с открытым исходным кодом для платформы Linux x86-64. В том числе сравнение производилось на дистрибутивах операционной системы OpenBSD, авторы которой целенаправленно ведут борьбу с ROP гаджетами [70].

Список литературы / References

[1]. The Heartbleed bug. URL: http://heartbleed.com.

[2]. D. Halperin, T. S. Heydt-Benjamin, B. Ransford, S. S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W. H. Maisel. Pacemakers and implantable cardiac defibrillators: software radio attacks and zero-power defenses. In Proc. of the IEEE Symposium on Security and Privacy, 2008, pp. 129–142.

[3]. 2019 CWE top 25 most dangerous software errors. URL: https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html.

[4]. A. Peslyak. Getting around non-executable stack (and fix). Bugtraq mailing list archives, Aug. 1997. URL: https://seclists.org/bugtraq/1997/Aug/63.

[5]. H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In Proc. of the 14th ACM Conference on Computer and Communications Security, 2007, pp. 552–561.

[6]. E.J. Schwartz, T. Avgerinos, and D. Brumley. Q: exploit hardening made easy. In Proc, of the 20th USENIX Conference on Security, 2011, 16 p.

[7]. R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: systems, languages, and applications. ACM Transactions on Information and System Security, vol. 15, no. 1, 2012, pp. 2:1–2:34.

[8]. T. Kornau. Return oriented programming for the ARM Architecture. Master's thesis, Ruhr-University, Bochum, Germany, 2009.

[9]. S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In Proc. of the 17th ACM Conference on Computer and Communications Security, 2010, pp. 559–572.

[10]. L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Return-oriented programming without returns on ARM. Technical Report HGI-TR2010-002, Ruhr-University, Bochum, Germany, 2010.

[11]. Z.-S. Huang and I. G. Harris. Return-oriented vulnerabilities in ARM executables. In Proc. of the IEEE Conference on Technologies for Homeland Security, 2012, pp. 1–6.

[12]. O.L. Fraser, N. Zincir-Heywood, M. Heywood, and J.T. Jacobs. Return-oriented programme evolution with ROPER: a proof of concept. In Proc. of the Genetic and Evolutionary Computation Conference Companion, 2017, pp. 1447–1454.

[13]. E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: generalizing return-oriented programming to RISC. In Proc. of the 15th ACM Conference on Computer and Communications Security, 2008, pp. 27–38.

[14]. A. Francillon and C. Castelluccia. Code injection attacks on harvard-architecture devices. In Proc. of the 15th ACM Conference on Computer and Communications Security, 2008, pp. 15–26.

[15]. F. Lindner. Cisco IOS router exploitation. In Black Hat USA, 2009. URL: https://www.blackhat.com/presentations/bh-usa-09/LINDNER/BHUSA09-Lindner-RouterExploit-PAPER.pdf.
[16]. S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, and H. Shacham. Can DREs provide long-lasting security? The case of return-oriented programming and the AVC advantage. In Proc. of the 2009 Conference on Electronic Voting Technology/Workshop on Trustworthy Elections, 2009, 16 p.
[17]. T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In Proc. of the 6th ACM Symposium on Information, Computer and Communications Security, 2011, pp. 30–40.
[18]. P. Chen, X. Xing, B. Mao, L. Xie, X. Shen, and X. Yin. Automatic construction of jump-oriented programming shellcode (on the x86). In Proc. of the 6th ACM Symposium on Information, Computer and Communications Security, 2011, pp. 20–29.
[19]. A. Sadeghi, S. Niksefat, and M. Rostamipour. Pure-call oriented programming (PCOP): chaining the gadgets using call instructions. Journal of Computer Virology and Hacking Techniques, vol. 14, no. 2, pp. 139–156.
[20]. R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: bypassing kernel code integrity protection mechanisms. In Proc. of the 18th Conference on USENIX Security Symposium, 2009, pp. 383–398.
[21]. N.A. Quynh. OptiROP: hunting for ROP gadgets in style. URL: https://media.blackhat.com/us-13/US-13-Quynh-OptiROP-Hunting-for-ROP-Gadgets-in-Style-Slides.pdf.
[22]. W. Ding, X. Xing, P. Chen, Z. Xin, and B. Mao. Automatic construction of printable return-oriented programming payload. In Proc. of the 9th International Conference on Malicious and Unwanted Software: The Americas, 2014, pp. 18–25.
[23]. Y. Ouyang, Q. Wang, J. Peng, and J. Zeng. An advanced automatic construction method of ROP. Wuhan University Journal of Natural Sciences, vol. 20, no. 2, 2015, pp. 119–128.
[24]. A. Follner, A. Bartel, H. Peng, Y.-C. Chang, K. Ispoglou, M. Payer, and E. Bodden. PSHAPE: automatically combining gadgets for arbitrary method execution. Lecture Notes in Computer Science, vol. 9871, 2016, pp. 212–228.
[25]. B. Milanov. ROPGenerator: practical automated ROP-chain generation, 2018. URL: https://youtu.be/rz7Z9fBLVs0.
[26]. N. Mosier and P. Johnson. ROP with a 2nd stack, 2019. URL: http://www.cs.middlebury.edu/~nmosier/portfolio/rsrc/ropc-slides.pdf.
[27]. A. Follner, A. Bartel, H. Peng, Y.-C. Chang, K. Ispoglou, M. Payer, and E. Bodden. PSHAPE - practical support for half-automated program exploitation. URL: https://github.com/Alexandre-Bartel/inspector-gadget.
[28]. O.L. Fraser. ROPER: a genetic ROP-chain development tool. URL: https: //github.com/oblivia-simplex/roper.
[29]. mona.py, Corelan Consulting BVBA. URL: https://github.com/corelan/mona.
[30]. J.Salwan. ROPgadgettool. URL: https://github.com/JonathanSalwan/ROPgadget.
[31]. B. Milanov. ROPGenerator. URL: https://github.com/Boyan-MILANOV/ropgenerator.
[32]. C. Salls. angrop. URL: https://github.com/salls/angrop.
[33]. S. Schirra. Ropper. URL: https://github.com/sashs/ropper.
[34]. Paul. ROPC. URL: https://github.com/pakt/ropc.
[35]. ropc-llvm, Programa STIC. URL: https://github.com/programa-stic/ropc-llvm.
[36]. J. Stewart. An open source, multi-architecture ROP compiler. URL: https:// github.com/jeffball55/rop_compiler.
[37]. A. Nurmukhametov. ROP Benchmark. URL: https://github.com/ispras/rop-benchmark.
[38]. J. Salwan. An introduction to the return oriented programming and ROP-chain generation, 2014. URL: http://shell-storm.org/talks/ROP_ course_lecture_jonathan_salwan_2014.pdf.
[39]. T. F. Dullien. Weird machines, exploitability, and provable unexploitability. IEEE Transactions on Emerging Topics in Computing (Early Access), 2017, 15 p.
[40]. M. Graziano, D. Balzarotti, and A. Zidouemba. ROPMEMU: a framework for the analysis of complex code-reuse attacks. In Proc. of the 11th ACM on Asia Conference on Computer and Communications Security, 2016, pp. 47–58.
[41]. E.J. Schwartz, T. Avgerinos, and D. Brumley. Update on Q: exploit hardening made easy, 2012. URL: https://edmcman.github.io/papers/usenix11-update.pdf.

[42]. G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib(c). In Proc. of the Annual Computer Security Applications Conference, 2009, pp. 60–69.
[43]. N.R. Weidler, D. Brown, S.A. Mitchell, J. Anderson, J.R. Williams, A. Costley, C. Kunz, C. Wilkinson, R. Wehbe, and R. Gerdes. Return-oriented programming on a resource constrained device. Sustainable Computing: Informatics and Systems, vol. 22, 2019, pp. 244-256.
[44]. Вишняков А.В. Классификация ROP гаджетов. Труды ИСП РАН, том 28, вып. 6, 2016, стр. 27-36 / Vishnyakov A.V. Classification of ROP gadgets. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 6, 2016, pp. 27-36 (in Russian). DOI: 10.15514/ISPRAS-2016-28(6)-2.
[45]. Вишняков А.В, Нурмухаметов А.Р., Курмангалеев Ш.Ф., Гайсарян С.С. Метод анализа атак повторного использования кода. Труды ИСП РАН, том 30, вып. 5, 2018 г., стр. 31-54 / Vishnyakov A.V., Nurmukhametov A.R., Kurmangaleev Sh.F., Gaisaryan S.S. Method for analysis of code-reuse attacks. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 5, 2018, pp. 31-54 (in Russian). DOI: 10.15514/ISPRAS-2018-30(5)-2.
[46]. T. Avgerinos, S.K. Cha, B.L.T. Hao, and D. Brumley. AEG: automatic exploit generation. In Proc. of the Network and Distributed System Security Symposium, 2011, pp. 283– 300.
[47]. S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on binary code. In Proc. of the IEEE Symposium on Security and Privacy, 2012, pp. 380–394.
[48]. V.A. Padaryan, V.V. Kaushan, and A.N. Fedotov. Automated exploit generation for stack buffer overflow vulnerabilities. Programming and Computer Software, vol. 41, no. 6, 2015, pp. 373–380.
[49]. Федотов А.Н., Падарян В.А., Каушан В.В., Курмангалеев Ш.Ф., Вишняков А.В., Нурмухаметов А.Р. Оценка критичности программных дефектов в рамках работы современных защитных механизмов. Труды ИСП РАН, том 28, вып. 5, 2016 г., стр. 73-92 / Fedotov A.N., Padaryan V.A., Kaushan V.V., Kurmangaleev Sh.F., Vishnyakov A.V., Nurmukhametov A.R. Software defect severity estimation in presence of modern defense mechanisms. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 5, 2016. pp. 73-92 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-4.
[50]. Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SOK: (state of) the art of war: offensive techniques in binary analysis. In Proc. of the IEEE Symposium on Security and Privacy, 2016, pp. 138–157.
[51]. J.C. King. Symbolic execution and program testing. Communications of the ACM, vol. 19, no. 7, 1976, pp. 385– 394.
[52]. E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In Proc. of the IEEE Symposium on Security and Privacy, 2019, pp. 317– 331.
[53]. P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In Proc. of the 16th Annual Network & Distributed *System* Security Symposium, 2008, pp. 151–166.
[54]. A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz. Microgadgets: size does matter in turing-complete return-oriented programming. In Proc. of the 6th USENIX Workshop on Offensive Technologies*,* 2012, 13 p.
[55]. M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning. On the expressiveness of return-into-libc attacks. Lecture Notes in Computer Science, vol. 6961, 2011, pp. 121–141.
[56]. BARF: binary analysis and reverse engineering framework. URL: https://github.com/programa-stic/barf-project.
[57]. N. Mosier. A pair of return-oriented programming utilities: a gadget finder and ROP compiler. URL: https://github.com/nmosier/rop-tools.
[58]. SQLab ROP payload generation. URL: https://github.com/SQLab/ropchain.
[59]. E. Göktas, B. Kollenda, P. Koppe, E. Bosman, G. Portokalidis, T. Holz, H. Bos, and C. Giuffrida. Position-independent code reuse: on the effectiveness of ASLR in the absence of information disclosure. In Proc. of the IEEE European Symposium on Security and Privacy*, 2018,* pp. 227–242.
[60]. I. Jager and D. Brumley. Efficient Directionless Weakest Preconditions. Technical Report CMU-CyLab-10-002, Carnegie Mellon University, CyLab, 2010.
[61]. N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. SIGPLAN Notice, vol. 42, no. 6, 2007, pp. 89–100.
[62]. T. Dullien and S. Porst. REIL: a platform-independent intermediate representation of disassembled code for static code analysis, 2009.
[63]. В.А. Падарян, М.А. Соловьев, А.И. Кононов. Моделирование операционной семантики машинных инструкций. Труды ИСП РАН, том 19, 2010 г., стр. 165–186 / V.A. Padaryan, M.A.

Soloviev, and A.I. Kononov. Modeling operational semantics of machine instructions. Trudy ISP RAN/Proc. ISP RAS, vol. 19, 2010, pp. 165–186 (in Russian).
[64]. C. Heitman and I. Arce. BARF: a multiplatform open source binary analysis and reverse engineering framework. En los Materiales del XX Congreso Argentino de Ciencias de la Computación, 2014, 10 p.
[65]. А.В. Вишняков. Верификация семантики линейной последовательности машинных инструкций. Курсовая работа, МГУ им. М.В. Ломоносова, ф-т ВМК, 2019 г. / A. V. Vishnyakov. Semantic verification of linear machine instruction sequence. Course Paper, M.V. Lomonosov Moscow State University, faculty of CMC, 2019 (in Russian).
[66]. A. Follner, A. Bartel, and E. Bodden. Analyzing the gadgets: towards a metric to measure gadget quality. Lecture Notes in Computer Science, vol. 9639, 2016, pp. 155– 172.
[67]. E.W. Dijkstra. A note on two problems in connexion with graphs. Numerische Mathematik, vol. 1, no. 1, 1959, pp. 269–271.
[68]. A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. Code Generation by Tiling an Input Tree. In Compilers: principles, technologies, and tools. Addison Wesley, 2th edition, 2006, pp. 560–563.
[69]. J. Stewart and V. Dedhia. ROP compiler. URL: https://css.csail.mit.edu/6.858/2015/projects/je25365-ve25411.pdf.
[70]. T. Mortimer. Removing ROP gadgets from OpenBSD. In Proc. of the AsiaBSDCon, 2019, pp.13–21.

Информация об авторах / Information about authors

Алексей Вадимович ВИШНЯКОВ работает в отделе компиляторных технологий ИСП РАН, закончил бакалавриат ВМК МГУ, сейчас там же получает степень магистра. Сфера научных интересов: компьютерная безопасность, возвратно-ориентированное программирование, анализ бинарного кода, символьная интерпретация, обратная инженерия и компиляторы.

Alexey Vadimovich VISHNYAKOV works for the Compiler Technology Department at ISP RAS, obtained BSc degree in the Faculty of Computational Mathematics and Cybernetics at Lomonosov Moscow State University. He is a M.D. student in the same faculty now. Research interests: computer security, return-oriented programming, binary analysis, symbolic execution, reverse engineering, and compilers.

Алексей Раисович НУРМУХАМЕТОВ – младший научный сотрудник отдела компиляторных технологий ИСП РАН, закончил МФТИ по специальности прикладные математика и физика в 2013 году. Сферой научных интересов являются: компиляторы, компьютерная безопасность, возратно-ориентированное программирование.

Aleksei Raisovich NURMUKHAMETOV is a research fellow at the ISP RAS, the Compiler Technology Department. He obtained both his BSc and MSc degrees in applied mathematics and physics at the Moscow Institute of Physics and Technology in 2011 and 2013, respectively. He has a strong interest in compilers, computer security, return-oriented programming.