Московский государственный университет имени М.В.Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра системного программирования

Вишняков Алексей Вадимович

Верификация семантики линейной последовательности машинных инструкций

КУРСОВАЯ РАБОТА

Научный руководитель:
чл.-кор. РАН, д.ф.-м.н., профессор
Аветисян Арутюн Ишханович

Москва, 2019

Аннотация

Верификация семантики линейной последовательности
машинных инструкций

Вишняков Алексей Вадимович

Обеспечение безопасности программного обеспечения является на сегодняшний день одной из первостепенных задач. Сбои в работе программного обеспечения могут привести к серьезным последствиям, а злонамеренная эксплуатация уязвимостей может причинить колоссальный ущерб. Крупные корпорации уделяют особое внимание оценке критичности программных дефектов. Атаки повторного использования кода, основанные на возвратно-ориентированном программировании (ROP), приобретают всю большую популярность с каждым годом и могут быть применены даже в условиях работы защитных механизмов современных операционных систем. ROP эксплоит состоит из цепочки блоков инструкций (гаджетов), которые последовательно передают друг другу управление. Метод классификации найденных в программе гаджетов позволяет сделать вывод о применимости техники ROP. Однако классификация гаджетов опирается на анализ эффектов выполнения гаджета на нескольких различных случайных входных данных, что может приводить к неверной классификации. Целью данной работы является уточнение метода классификации гаджетов. В данной работе предлагается метод верификации семантики линейной последовательности машинных инструкций, который позволяет формально доказать семантику ROP гаджета: тип и параметры гаджета, список «испорченных» регистров, размер фрейма и смещение ячейки с адресом следующего гаджета. Метод был реализован в виде программного инструмента и позволил устранить ошибку классификации гаджетов, которая составляла 0.7%.

Содержание

1 Введение

Обеспечение безопасности программного обеспечения является на сегодняшний день одной из первостепенных задач. Программные продукты применяются в повседневно окружающих нас вещах: компьютерах, смартфонах, автомобилях, банкоматах, объектах городской инфраструктуры, медицинском оборудовании жизнеобеспечения и технологиях «интернета вещей». Сбои в работе программного обеспечения могут привести к серьезным последствиям: денежным убыткам, деградации средств коммуникации, задержке в работе экстренных служб, авариям и даже причинению вреда здоровью человека. А злонамеренная эксплуатация уязвимостей[1] может причинить колоссальный ущерб. По данным Национального института стандартов и технологий США ежегодно публикуются тысячи описаний новых уязвимостей CVE (рис. 1) [1, 2]. Крупные корпорации уделяют особое внимание оценке критичности программных дефектов.

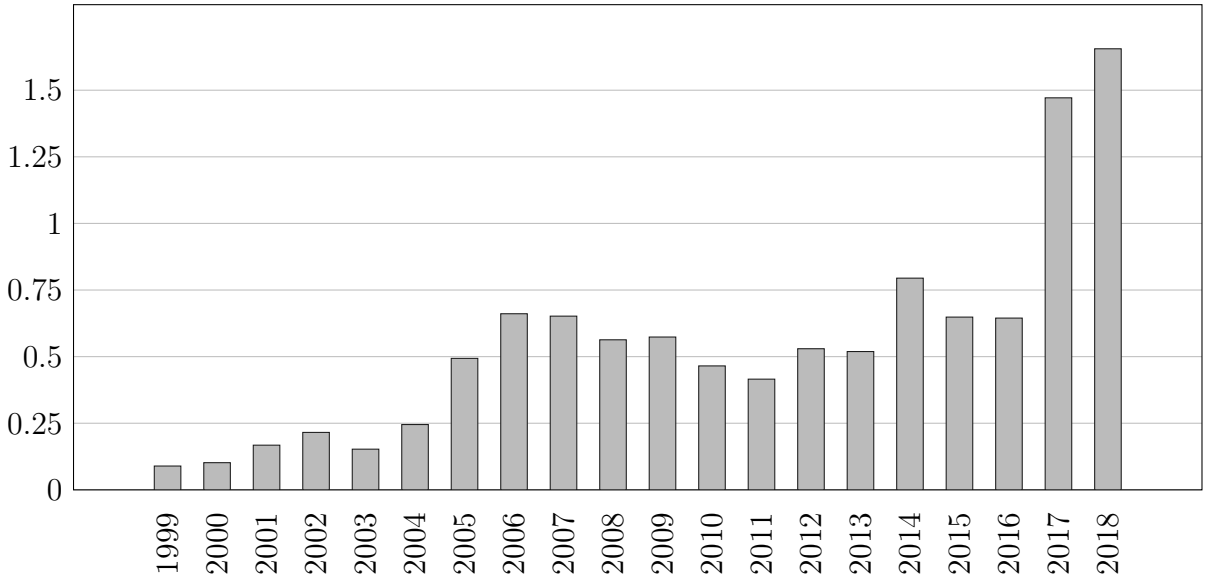


Рис. 1: Количество (десятки тысяч) новых уязвимостей (CVE) в год

Для эксплуатации уязвимостей в условиях работы защитных механизмов современных операционных систем часто применяется техника возвратно-ориентированного программирования (ROP). Это атака повторного использования кода, позволяющая обхо-

[1]Уязвимость – недостаток в системе, используя который можно злонамеренно нарушить конфиденциальность, целостность или доступность информации.

дить защитный механизм, запрещающий региону памяти быть одновременно доступным на запись и исполнение (DEP), и современные реализации рандомизации размещения адресного пространства (ASLR), которые оставляют часть адресного пространства программы нерандомизированной. Так в Linux адрес загрузки кода программы часто остается постоянным, а некоторые динамические библиотеки Windows загружаются по фиксированным адресам. Злоумышленник использует кусочки кода из нерандомизированного адресного пространства программы, которые называются гаджетами. Каждый гаджет выполняет некоторые вычисления (например, складывает значения двух регистров) и передает управление следующему гаджету. Гаджеты связываются в цепочку последовательно выполняемых кусочков кода. Таким образом, с помощью цепочки гаджетов можно выполнить некоторые вредоносные действия.

Метод классификации найденных в программе гаджетов [3] позволяет сделать вывод о применимости техники ROP. Однако классификация гаджетов опирается на анализ эффектов выполнения гаджета на нескольких различных случайных входных данных, что может приводить к неверной классификации. Целью данной работы является уточнение метода классификации гаджетов.

В данной работе предлагается метод верификации семантики линейной последовательности машинных инструкций, который позволяет формально доказать семантику ROP гаджета: тип и параметры гаджета, список «испорченных» регистров, размер фрейма и смещение ячейки с адресом следующего гаджета.

Работа организована следующим образом. В главе 2 приводится обзор атак и защитных механизмов, послуживших предпосылками к появлению ROP (разд. 2.5). В главе 4 проводится обзор существующих методов определения семантики ROP гаджетов. В главе 5 представлен метод верификации семантики линейной последовательности машинных инструкций. В главе 6 описываются детали реализации предлагаемого метода. Результаты практического применения приводятся в главе 7.

2 Обзор атак и защитных механизмов

В данной главе приводится обзор атак на переполнение буфера на стеке. Описываются защитные механизмы операционной системы: ограничение исполняемых областей (DEP) и рандомизация размещения адресного пространства (ASLR). В разделе 2.5 дается определение возвратно-ориентированного программирования, метода эксплуатации уязвимости переполнения буфера на стеке, позволяющего обойти DEP и современные реализации ASLR.

2.1 Уязвимость переполнения буфера на стеке

Уязвимость переполнения буфера на стеке возникает, когда размер данных, записываемых в буфер на стеке, превышает размер этого буфера [4]. Например, в приведенной на листинге 1 программе на Си уязвимая функция `vul` не проверяет длину сроки `str`, записываемой в буфер фиксированного размера `buf`. Если длина первого аргумента командной строки `argv[1]` окажется большей или равной размеру буфера `buf`, то произойдет переполнение буфера на стеке.

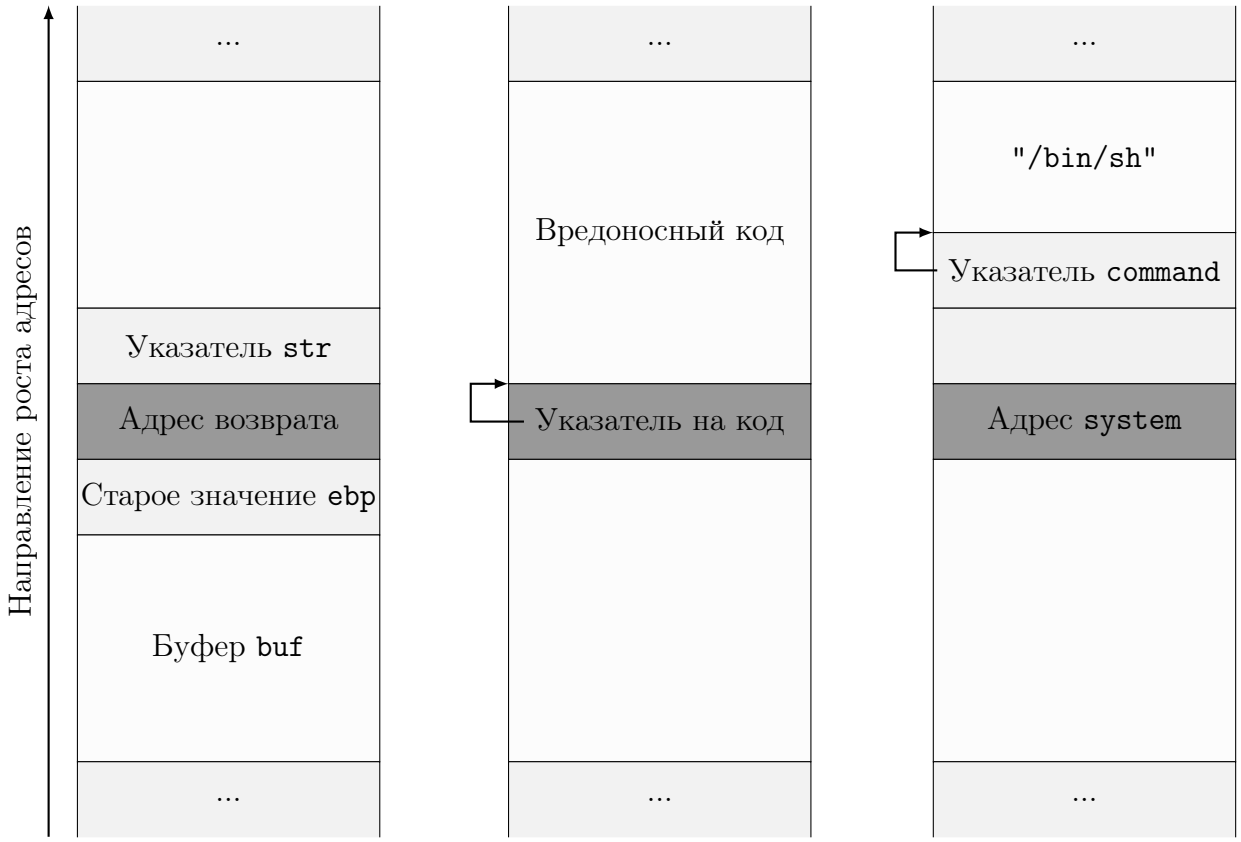
Листинг 1: Переполнение буфера `buf` на стеке в функции `vul`

```
void vul(char *str) {
    char buf[512];
    strcpy(buf, str);
}

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

На рисунке 2а показан стековый фрейм функции `vul` до переполнения. Стек в архитектуре x86 растет от больших адресов к меньшим (на рисунке – сверху вниз). Аргументы функции поочередно кладутся на стек справа налево. При вызове функции адрес возврата кладется на стек, после чего функция может сохранить старое значение регистра `ebp` и выделить на стеке память для локальных переменных, в нашем случае

– для буфера `buf`. Данные в буфер записываются в порядке возрастания адресов (на рисунке – снизу вверх). Переполнение буфера приводит к перезаписи ячеек выше по стеку, в том числе адреса возврата, после чего почти всегда следует аварийное завершение программы.



(a) Стековый фрейм функции `vul` до переполнения

(b) Размещение вредоносного кода на стеке

(c) Атака возврата в библиотеку

Рис. 2: Стековый фрейм функции `vul` и способы эксплуатации уязвимости переполнения буфера `buf` на стеке

Эксплуатация уязвимости переполнения буфера на стеке позволяет выполнять произвольный код. Рассмотрим ситуацию, когда злоумышленник контролирует значение первого аргумента командной строки `argv[1]`, а следовательно, контролирует значения, записываемые в буфер `buf`. В таком случае злоумышленник может добиться перезаписи адреса возврата указателем на размещенный на стеке вредоносный код (рис. 2b). Таким образом, после возврата из функции `vul` управление передается на сформированный

злоумышленником код. Обычно в качестве такого кода используется код, приводящий к вызову командной оболочки операционной системы, который называется шелл-кодом. Чтобы избежать негативных последствий от переполнения буфера на стеке, появились различные защитные механизмы.

2.2 Ограничение исполняемых областей (DEP)

Ограничение исполняемых областей (DEP) – защитный механизм операционной системы, запрещающий исполнение кода из областей памяти, помеченных как «данные». Попытка исполнения кода из помеченных областей вызывает исключение. Таким образом, стек и куча становятся недоступными для выполнения, что предотвращает выполнение размещенного на них вредоносного кода. Механизм успешно применяется в операционных системах Windows, Linux и др.

2.3 Атака возврата в библиотеку

Для обхода DEP используется атака возврата в библиотеку. Атака заключается в подмене адреса возврата адресом некоторой библиотечной функции, например, функции `system` из библиотеки `libc`.

На рисунке 2c показано состояние стека после переполнения. Адрес возврата перезаписан адресом функции `system(const char *command)`. Выше лежит произвольный адрес возврата из функции `system` и ее единственный аргумент `command`, который является указателем на нуль-терминированную строку `"/bin/sh"`, размещенную следом за указателем. Таким образом, после возврата из функции `vul` вызовется библиотечная функция `system("/bin/sh")`, которая в свою очередь вызовет командную оболочку операционной системы.

2.4 Рандомизация размещения адресного пространства (ASLR)

Рандомизации размещения адресного пространства (ASLR) – защитный механизм операционной системы, позволяющий размещать ключевые элементы процесса (образ программы, стек, куча, динамические библиотеки) по различным адресам во время загрузки исполняемого файла. Данная защита затрудняет проведение атаки возврата в

библиотеку, т.к. адрес библиотечной функции неизвестен до загрузки программы и отличается для каждого запуска.

Следует отметить, что рандомизация адресов исполняемых секций программы или библиотеки требует, чтобы они были скомпилированы в позиционно-независимый код, что не всегда выполняется. Так в Linux адрес загрузки кода программы часто остается постоянным, а некоторые динамические библиотеки Windows загружаются по фиксированным адресам. Таким образом, в условиях работы современных реализаций ASLR часть адресного пространства программы остается нерандомизированной.

2.5 Возвратно-ориентированное программирование (ROP)

Возвратно-ориентированное программирование (ROP) [5] – метод эксплуатации уязвимости переполнения буфера на стеке, который по сути является обобщением атаки возврата в библиотеку. Метод так же применим в условиях работы DEP, но представляет большую опасность, т.к. может быть использован для обхода современных реализаций ASLR, когда часть адресного пространства остается нерандомизированной (разд. 2.4).

ROP предполагает использование последовательностей инструкций в нерандомизированных исполняемых областях памяти, которые заканчиваются инструкцией передачи управления (`ret`). Такие последовательности инструкций называются гаджетами. Следует отметить, что архитектура x86 не требует выравнивания адресов инструкций, т.е. позволяет выполнение инструкций, размещенных по произвольным адресам памяти. А значит, некоторая последовательность инструкций в программе может содержать в себе гаджет, отсутствовавший в коде программы. На листинге 2 приводятся бинарный и ассемблерный коды гаджета, который содержится внутри последовательности инструкций оригинальной программы [6]. Гаджеты собираются в цепочки, а их адреса размещаются от адреса возврата на стеке так, чтобы первый гаджет передавал управление второму, второй – третьему и т.д. Таким образом, с помощью цепочки гаджетов можно выполнить некоторые вредоносные действия.

Листинг 2: Гаджет внутри последовательности инструкций

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

На рисунке 3 приводится состояние стека после размещения на нем ROP цепочки, которая производит запись значения `memValue` по адресу `memAddr`. Адрес возврата перезаписан адресом первого гаджета. После возврата из функции управление передастся первому гаджету, который загрузит со стека значение `memValue` на регистр `eax`. При возврате (после выполнения инструкции `ret`) первый гаджет передаст управление второму гаджету, который в свою очередь загрузит значение `memAddr` на регистр `edx`. Потом третий гаджет сохранит значение регистра `eax` (`memValue`) по адресу `edx` (`memAddr`). Далее управление передастся четвертому гаджету и т.д. На листинге 3 приводится эта же ROP цепочка в бинарном виде, которая записывает значение `"/bin"` по адресу `0x0830caa0`. Эта последовательность байтов размещается злоумышленником на стеке от адреса возврата.

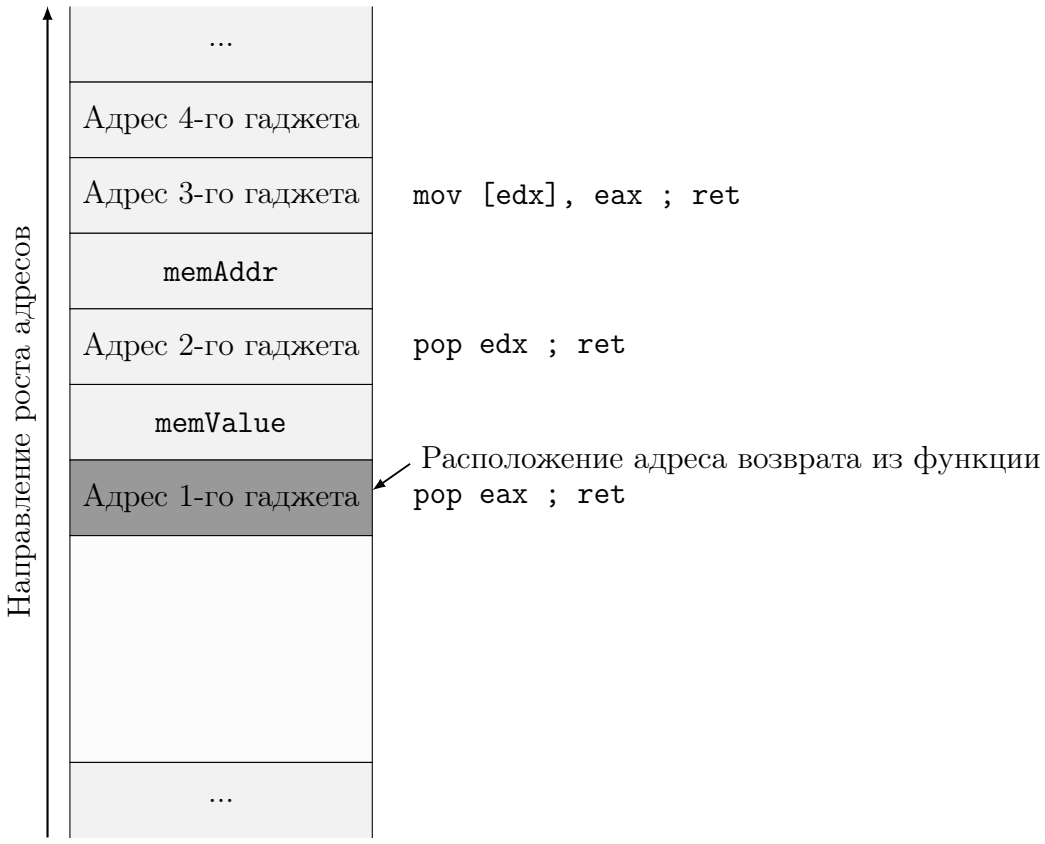


Рис. 3: Состояние стека после размещения на нем ROP цепочки, которая производит запись значения `memValue` по адресу `memAddr`

Листинг 3: Бинарная ROP цепочка, записывающая `"/bin"` по адресу `0x0830caa0`

```
00000000  47 65 06 08 2f 62 69 6e  3d 76 07 08 a0 ca 30 08  |Ge../bin=v....0.|
00000010  b5 8b 08 08                                       |....|
00000014
```

Следует отметить, что применение ROP для эксплуатации уязвимости возможно при условии наличия достаточного набора гаджетов из нерандомизированных областей памяти.

3 Постановка задачи

Необходимо разработать и реализовать метод верификации семантики линейной последовательности машинных инструкций.

Метод должен позволять по заданному постусловию верифицировать семантику ROP гаджета:

- Проверить, удовлетворяет ли гаджет определению семантики его типа с заданными значениями параметров.
- Проверить список «испорченных» регистров.
- Проверить размер фрейма и адрес следующего гаджета.

4 Обзор существующих решений

4.1 Поиск гаджетов

Для поиска гаджетов может быть использован алгоритм Галилео [5], который осуществляет поиск инструкций передачи управления в исполняемых секциях программы. Для каждой найденной инструкции производится дизассемблирование нескольких байтов, предшествующих инструкции. Все корректно дизассемблированные последовательности инструкций добавляются в набор найденных гаджетов.

4.2 Определение семантики гаджета

Schwartz и др. [7] предложили определять функциональность гаджета его принадлежностью типам, которые приведены в таблице 1. Однако не каждый гаджет может быть использован для составления пригодных для эксплуатации ROP цепочек. В работе требуют, чтобы каждый гаджет удовлетворял следующим свойствам:

- **Функциональность.** У каждого гаджета есть тип (табл. 1), который определяет его функциональность. Тип гаджета описывается семантически с помощью булева предиката, который должен быть всегда истинным после выполнения гаджета.

- **Сохранение управления.** Каждый гаджет должен быть способен передать управление другому гаджету, т.е. заканчиваться инструкцией `ret` или семантически эквивалентной последовательностью инструкций (например, `pop eax ; jmp eax`).

- **Известные побочные эффекты.** У гаджета не должно быть неизвестных побочных эффектов. Побочные эффекты выполнения гаджета не должны приводить к неконтролируемому поведению программы. Например, запись значения по произвольному адресу памяти может привести к аварийному завершению программы.

- **Константное смещение стека.** Большинство типов гаджетов требуют, чтобы указатель стека увеличивался на постоянное значение после каждого выполнения.

Тип	**Параметры**	**Семантика**
`NoOpG`	—	Не меняет ничего в памяти и на регистрах
`JumpG`	`AddrReg`	`IP` $\leftarrow$ `AddrReg`
`MoveRegG`	`InReg`, `OutReg`	`OutReg` $\leftarrow$ `InReg`
`LoadConstG`	`OutReg`, `Offset`	`OutReg` $\leftarrow$ `[SP + Offset]`
`ArithmeticG`	`InReg1`, `InReg2`, `OutReg`, $\circ$	`OutReg` $\leftarrow$ `InReg1` $\circ$ `InReg2`
`LoadMemG`	`AddrReg`, `OutReg`, `Offset`	`OutReg` $\leftarrow$ `[AddrReg + Offset]`
`StoreMemG`	`AddrReg`, `InReg`, `Offset`	`[AddrReg + Offset]` $\leftarrow$ `InReg`
`ArithmeticLoadG`	`AddrReg`, `OutReg`, `Offset`, $\circ$	`OutReg` $\circ\leftarrow$ `[AddrReg + Offset]`
`ArithmeticStoreG`	`AddrReg`, `InReg`, `Offset`, $\circ$	`[AddrReg + Offset]` $\circ\leftarrow$ `InReg`

Таблица 1: Типы гаджетов. `[Addr]` означает доступ к памяти по адресу `Addr`, $\circ$ – бинарную операцию. `a` $\leftarrow$ `b` означает, что конечное значение `a` равно начальному значению `b`. `X` $\circ\leftarrow$ `Y` – сокращение для `X` $\leftarrow$ `X` $\circ$ `Y`

4.2.1 Типы гаджетов

Набор типов гаджетов задает новую архитектуру набора команд (ISA), в которой каждый тип гаджета исполняет роль инструкции. Семантика каждого типа гаджета определяется постусловием (булевым предикатом) $\mathcal{B}$, которое должно быть всегда истинно после выполнения гаджета.

Будем говорить, что последовательность инструкций $\mathcal{I}$ удовлетворяет постусловию $\mathcal{B}$, если для любого начального состояния после выполнения $\mathcal{I}$ постусловие $\mathcal{B}$ истинно. Начальное состояние состоит из присваиваний регистрам и памяти некоторых начальных значений.

Список типов гаджетов и определений их семантики (постусловий) приводится в таблице 1. 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` $\leftarrow$ `eax` и `LoadConstG: ecx` $\leftarrow$ `[esp + 0]`.

4.2.2 Семантический анализ

Для того чтобы определить, удовлетворяет ли последовательность инструкций $\mathcal{I}$ постусловию $\mathcal{B}$, Schwartz и др. [7] используют известную технику из формальной верификации – вычисление слабейшего предусловия [8]. Проще говоря, слабейшее предусловие $wp(\mathcal{I}, \mathcal{B})$ для последовательности инструкций $\mathcal{I}$ и постусловия $\mathcal{B}$ – это булево предусловие, которое описывает, когда $\mathcal{I}$ завершается в состоянии, удовлетворяющем $\mathcal{B}$.

Слабейшие предусловия используются, чтобы убедиться, что определение семантики гаджета всегда выполняется после выполнения последовательности инструкций $\mathcal{I}$. Для этого достаточно проверить:

$$wp(\mathcal{I}, \mathcal{B}) \equiv true \tag{1}$$

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 гаджетов показала себя очень медленной на практике. Для ускорения процесса инструкции гаджета предварительно несколько раз выполняются с использованием случайных входных данных, и 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.

Выполнение со случайными входными данными может быть также использовано для выявления возможных значений параметров (табл. 1) гаджетов. Например, посмотрев на значения регистров и на адреса чтения из памяти, можно вычислить набор возможных смещений (`Offset`) для гаджета загрузки из памяти `LoadMemG`.

4.3 Символьная интерпретация

Символьная интерпретация была изначально предложена для тестирования программного обеспечения [9]. При использовании модульного тестирования программист может убедиться в корректности выполнения программы на некотором наборе тестов, однако корректность программы вне тестового набора остается под сомнением. Вместо исследования поведения программы на фиксированных входных данных, символьная интерпретация позволяет анализировать программу на целых классах входных данных.

Символьная интерпретация представляет собой процесс интерпретации программы, где конкретным значениям переменных сопоставляются символьные переменные, которые могут принимать произвольные значения. Преобразования данных в программе описываются в виде формул над символьными переменными и константами. Каждое условное ветвление, условие выполнения которого зависит от символьных переменных, порождает уравнение, описывающее прохождение потока управления по определенной ветке.

Обычно свободные символьные переменные ставятся в соответствие входным данным программы и исследуется конкретный путь выполнения в программе. Система уравнений, построенная в результате символьной интерпретации этого пути, называется предикатом пути. Предикат пути подается на вход SMT-решателю, где символьные переменные выступают в качестве неизвестных. Решением системы уравнений является конкретный набор входных данных, обеспечивающий прохождение потока управления по исследуемому пути.

4.4 Моделирование символьных адресов

Проблема символьных адресов [10, 11] появляется, когда значение адресного выражения зависит от входных данных, что часто встречается в бинарном коде. В частности, символьные адреса возникают в табличных преобразованиях входных данных. Например, выражение языка Си `switch(c)` компилируется в таблицу переходов, где входной символ `c` выступает в качестве индекса. Стандартные функции преобразования строк (конвертация между ASCII и Unicode, `tolower`, `toupper` и т.д.) также используют таблицы.

Обработка символьных адресов является сложной задачей, так как символьный адрес (в худшем случае) может указывать на произвольную ячейку адресуемой памяти. Существуют два основных подхода к обработке символьных адресов:

- Конкретизация адреса.
- Работа с адресом как с полностью символьным.

Первый подход предполагает конкретизацию адреса некоторым фиксированным значением, что позволяет упростить получаемые формулы и уменьшить время решения предиката пути. Однако фиксирование значения символьного адреса может привести к

тому, что некоторые ограничения на символьные переменные будут потеряны, и набор входных данных, полученный в результате решения предиката пути, не проведет программу по тому же самому пути выполнения, для которого строился предикат пути. Конкретизация символьных адресов является распространенным выбором при динамической символьной интерпретации, например, в SAGE [12].

Второй подход предполагает рассмотрение адреса как полностью символьного. Тем самым разрешение символьных адресов ложится на SMT-решатель. Основной недостаток по сравнению с первым подходом – производительность. Формулы, использующие символьную память, более дорогие для вычислений, соответственно, время решения предиката пути обычно больше.

Символьный интерпретатор в инструменте Mayhem [11] использует оба подхода. Символьные адреса, по которым производится чтение, рассматриваются как символьные. Символьный адрес записи при этом всегда конкретизируется.

4.4.1 Индексная модель памяти (Mayhem)

Индексная модель памяти – модель, предложенная в Mayhem [11], для обработки операций чтения по символьным адресам.

Глобальная память представляется как отображение $\mu : I \mapsto E$ из индексов i (адресов памяти) в выражения e (формулы над символьными переменными и константами). Таким образом, $\mu[i]$ представляет содержимое i-ой ячейки памяти.

Для моделирования чтения по символьному адресу используются объекты памяти, представляющие собой аналогично с глобальной памятью отображение $M : I \mapsto E$ из индексов в выражения. Каждый раз, когда во время символьной интерпретации происходит чтение по символьному адресу, создается новый объект M, который содержит все значения, к которым можно обратиться по индексу. M, по сути, – частичный снимок глобальной памяти.

Для создания объекта памяти требуется определить все возможные значения, которые может принимать индекс. Mayhem пытается определить границы $[L, U]$, в которых лежат значения индекса. Чтобы вычислить эти границы, с использованием SMT-решателя производится бинарный поиск по значению индекса в контексте текущего предиката пути. Например, изначально нижняя граница 32-битного индекса i: $L \in [0, 2^{32} - 1]$. Если $i < \frac{2^{32}-1}{2}$ выполняется, тогда $L \in [0, \frac{2^{32}-1}{2} - 1]$, в то время как невы-

полнимость означает, что $L \in [\frac{2^{32}-1}{2}, 2^{32}-1]$. Процесс повторяется, пока границы не будут вычислены. Используя вычисленные границы, можно инициализировать объект памяти следующим образом: $\forall i \in [L, U] : M[i] = \mu[i]$.

Любое значение, загружаемое из объекта памяти M, может быть символьным. Чтобы разрешить уравнения, содержащие загружаемое значение ($M[i]$), SMT-решатель должен найти выражение в M, удовлетворяющее уравнениям, и убедиться, что индекс этого выражения выполним для данной системы уравнений. Чтобы облегчить работу SMT-решателя, Mayhem использует бинарные деревья поиска, которые представляются в виде формулы загрузки выражения $IST(E)$. Так для (отсортированного по адресу) списка выражений E объекта памяти M и индекса i: $IST(E) = ite(i < addr(E_{right}), IST(E_{left}), IST(E_{right}))$, где ite — if-then-else выражение, E_{left} (E_{right}) — левая (правая) часть выражений из E, а $addr(\cdot)$ возвращает наименьший индекс из данного списка выражений. Для списка из одного выражения IST возвращает это выражение без создания ite выражений.

Чтобы уменьшить количество листьев в IST, Mayhem объединяет несколько выражений в один контейнер, который представляет собой выражение, параметризованное индексом и возвращающее значения объекта памяти по индексам объединенных выражений. Выражения объединяются по принципу линейной зависимости значения от индекса. Два выражения с индексами i_i и значениями v_i добавляются в один контейнер, если $v_i = \alpha i_i + \beta$. Таким образом, объект памяти разбивается на контейнеры, где каждый контейнер представляет собой либо линейное выражение, либо изолированную точку. Например, IST для функции `tolower` выглядит следующим образом:

$$ite(n < 91, ite(n < 64, n, n + 32), n)$$

4.4.2 S^{2}E

Символьный интерпретатор инструмента S^{2}E [13] использует свою модель символьной памяти. Во время разыменования символьного указателя S^{2}E определяет страницы памяти, к которым можно обращаться по этому указателю, и передает их содержимое SMT-решателю. Для облегчения работы SMT-решателя S^{2}E разбивает страницы на страницы меньшего размера (например, 128 байт).

Также поддерживается конкретизация символьных адресов. S^{2}E использует бинарный поиск для определения интервала, которому могут принадлежать значения сим-

вольного указателя, и создает n состояний, каждое из которых имеет одно фиксированное значение указателя, удовлетворяющее текущему предикату пути.

5 Метод верификации семантики линейной последовательности машинных инструкций

Предлагаемый в данной работе метод верификации семантики линейной последовательности машинных инструкций позволяет точно определить семантику ROP гаджетов в программе. Метод делится на три этапа: поиск, классификацию и верификацию гаджетов. Сначала в исполняемом файле алгоритмом Галилео (разд. 4.1) производится поиск гаджетов. Найденные гаджеты классифицируются по семантическим типам. Классификация гаджета выявляет набор возможных типов и параметров, которым он соответствует. Семантика гаджета определяется в результате анализа эффектов выполнения гаджета на нескольких различных случайных входных данных. Однако классификация гаджета не гарантирует соответствия семантике результата выполнения гаджета на произвольных входных данных, что может привести к неверной классификации. Для точного определения типов гаджета производится верификация результатов классификации, которая позволяет формально доказать соответствие семантике для произвольных входных данных. В результате верификации удаляются неверно классифицированные гаджеты, и тем самым устраняется ошибка классификации гаджетов.

5.1 Фрейм гаджета

Для связывания гаджетов в цепочки вводится понятие фрейма гаджета аналогичное стековому кадру x86. Цепочка гаджетов разбивается на фреймы. Фрейм гаджета содержит в себе значения параметров гаджета (например, значение, загружаемое на регистр со стека гаджетом `LoadConstG`) и адрес следующего гаджета. Начало фрейма определяется значением указателя стека перед выполнением первой инструкции гаджета.

Рис. 4: Фрейм гаджета `pop eax ; ret 8`

На рисунке 4 фигурной скобкой обозначены границы фрейма гаджета `pop eax ; ret 8`. Гаджет загружает значение со стека в `eax`, что соответствует типу загрузки константы `LoadConstG: eax ← [esp + 0]`. Гаджет имеет размер фрейма `FrameSize = 16`, а адрес следующего гаджета располагается по смещению 4 от начала фрейма (`NextAddr = [esp + 4]`).

5.2 Классификация гаджетов

Метод классификации гаджетов, описанный в статье [3], позволяет определить семантику гаджетов. Семантика гаджета определяется набором булевых постусловий (типов гаджета) и значениями их параметров, которым удовлетворяют инструкции гаджета (разд. 4.2.1). Набор типов гаджетов, предложенный Schwartz и др. [7], (табл. 1) был расширен дополнительными типами, которые приводятся в таблице 2. Более того, были добавлены типы гаджетов, которые не гарантируют сохранения управления (внизу таблицы 2).

Классификация гаджета производится на основе анализа эффектов выполнения гаджета на случайных входных данных. Инструкции гаджета транслируются в промежуточное представление. Далее запускается процесс интерпретации промежуточного представления. Во время интерпретации отслеживаются обращения к регистрам и памяти. Если происходит первое чтение регистра или области памяти, считанное зна-

Тип	**Параметры**	**Семантика**
JumpMemG	AddrReg, Offset	IP ← [AddrReg + Offset]
GetSPG	OutReg	OutReg ← SP
InitConstG	OutReg, Value	OutReg ← Value
InitMemG	AddrReg, Value, Offset, Size	[AddrReg + Offset] ← Value
NegG	InReg, OutReg	OutReg ← −InReg
ArithmeticConstG	InReg, OutReg, Value, ∘ $(+/\oplus)$	OutReg ← InReg ∘ Value
ShiftStackG	Offset, ∘ $(+/-)$	SP ∘← Offset
PushAllG	—	([ESP - 4] ← EAX) ∧ ([ESP - 8] ← ECX) ∧ ([ESP - 12] ← EDX) ∧ ([ESP - 16] ← EBX) ∧ ([ESP - 20] ← ESP) ∧ ([ESP - 24] ← EBP) ∧ ([ESP - 28] ← ESI) ∧ (EIP ← EDI)
Не сохраняют управление		
JumpSPG	—	IP ← SP
CallG	AddrReg	IP ← AddrReg
CallMemG	AddrReg, Offset	IP ← [AddrReg + Offset]
IntG	Value	Вызвать прерывание Value
SyscallG	—	Системный вызов

Таблица 2: Дополнительные типы гаджетов. [Addr] означает доступ к памяти по адресу Addr, ∘ – бинарную операцию. a ← b означает, что конечное значение a равно начальному значению b. X ∘← Y – сокращение для X ← X ∘ Y

чение 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 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.

В результате классификации гаджета будут получены семантические типы гаджета и их параметры, а также информация о фрейме гаджета (разд. 5.1) – размер фрейма и смещение ячейки с адресом следующего гаджета относительно начала фрейма.

Следует отметить, что предложенный метод основывается на результатах выполнения гаджета на ограниченном количестве наборов конкретных входных данных, что в общем случае не гарантирует соответствия семантике результата выполнения гаджета на произвольных входных данных. Для точной классификации необходимо производить формальную верификацию семантики гаджета. Таким образом, возможна неверная классификация гаджета.

5.3 Верификация гаджетов

Классификация гаджета предоставляет набор постусловий, описывающих возможную семантику гаджета. Верификация гаджета позволяет формально доказать истинность этих постусловий для произвольных входных данных.

Выполняется символьная интерпретация инструкций гаджета. Изначально всем регистрам присваиваются свободные символьные переменные. Символьная память в начале представляет из себя пустой байтовый массив SMT [14] Array, где $< addrSize >$ размерность адресного слова архитектуры:

$$(Array\ (\_\ BitVec\ < addrSize >)\ (\_\ BitVec\ 8)).$$

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 реализована через операции select и store над Array. Функция $(select\ a\ i)$ возвращает i-ый элемент массива a и моделирует чтение байта по адресу i. Функция $(store\ a\ i\ e)$ возвращает массив, полученный из массива a сохранением элемента e по индексу i, что моделирует запись байта e по адресу i.

Шаг	**Символьное состояние**	**Инструкция**	**Множество формул**
initial	M, $rax = \phi_1$, $rbx = \phi_2$, $rcx = \phi_3$, $rsp = \phi_4$, $rip = \phi_5$	—	$S_0 = \emptyset$
1	$rcx = \phi_6$	`mov rcx, [rax]`	$S_1 = S_0 \cup \{\phi_6 = M[\phi_1]\}$
2	$rbx = \phi_7$	`add rbx, rcx`	$S_2 = S_1 \cup \{\phi_7 = \phi_2 + \phi_6\}$
final	$rip = \phi_8$, $rsp = \phi_9$	`ret`	$S_3 = S_2 \cup \{\phi_8 = M[\phi_4], \phi_9 = \phi_4 + 8\}$
	Определение семантики		**Верификация**
verify	$(final(rbx) = initial(rbx) + initial(M[rax])) \wedge (final(rip) = initial(M[rsp])) \wedge (final(rsp) = initial(rsp) + 8)$		$\neg((\phi_7 = \phi_2 + M[\phi_1]) \wedge (\phi_8 = M[\phi_4]) \wedge (\phi_9 = \phi_4 + 8))$ is **UNSAT**

Таблица 3: Пример верификации гаджета $ArithmeticLoadG : rbx \leftarrow rbx + [rax]$

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 таблице 3 приводится пример верификации гаджета $ArithmeticLoadG : rbx \leftarrow rbx + [rax]$ с адресом следующего гаджета `NextAddr = [rsp]` и размером фрейма `FrameSize = 8`. 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 $M[\phi_1]$. В символьном состоянии регистру rcx ставится в соответствие символьная переменная ϕ_6. На втором шаге результат сложения присваивается переменной $\phi_7 = \phi_2 + \phi_6$, которая в свою очередь ставится в соответствие результирующему операнду инструкции – регистру rbx в символьном

состоянии. На конечном шаге символьное состояние обновляется согласно операционной семантике инструкции возврата, т.е. указатель инструкций загружается со стека $rip = \phi_8 = M[\phi_4]$, а указатель стека увеличивается на 8: $rsp = \phi_9 = \phi_4 + 8$. Постусловие для верификации представляет из себя конъюнкцию определения параметризованного типа гаджета, условие на расположение адреса следующего гаджета и условие на размер фрейма. В полученную формулу подставляются символьные переменные из начального и конечного символьных состояний. При помощи SMT-решателя проверяется выполнимость отрицания формулы. Отрицание формулы невыполнимо, значит, гаджет удовлетворяет заявленным типу с параметрами, адресу следующего гаджета и размеру фрейма.

6 Программная реализация

6.1 Поиск гаджетов

Набор гаджетов для классификации получается при помощи инструмента с открытым исходным кодом ROPgadget [15], в котором реализован алгоритм Галилео поиска гаджетов. Инструмент получает на вход исполняемый файл и выводит список найденных в нем гаджетов.

6.2 Классификация гаджетов

Классификация гаджетов была реализована в виде модуля расширения среды анализа бинарного кода, разрабатываемой в ИСП РАН [16]. Классификатор гаджетов получает на вход исполняемый файл и список виртуальных адресов найденных гаджетов. Поддерживаются следующие форматы исполняемых файлов: ELF32, ELF64, PE32 и PE32+. Результаты классификации (типы и параметры гаджета, список «испорченных» регистров, размер фрейма и смещение ячейки с адресом следующего гаджета) сохраняются в базу данных и дублируются выводом в файл. Пример вывода классификатора гаджетов приводится на листинге 4.

Листинг 4: Пример вывода классификатора гаджетов

```
0x08048061 : Asm : XCHG EBX, EAX ; XCHG EBX, EAX ; RET
0x08048061 : NoOpG
0x08048064 : Asm : MOV EAX, EBX ; RET
0x08048064 : MoveRegG : EAX <- EBX, AX <- BX, AH <- BH, AL <- BL
0x08048075 : Asm : POP EAX ; RET
0x08048075 : LoadConstG : EAX <- [ESP], AX <- [ESP], AH <- [ESP+1],
             AL <- [ESP] : NextAddr=[ESP+4], FrameSize=8
0x08048075 : ShiftStackG : ESP +<- 4
0x08048088 : Asm : ADD EAX, EBX ; RET
0x08048088 : ArithmeticG : EAX <- EAX + EBX, AX <- AX + BX, AL <- AL + BL
0x080480cb : Asm : MOV DWORD PTR [EAX - 00000456h], EBX ; RET
0x080480cb : StoreMemG : [EAX-0x456] <- EBX
0x080480da : Asm : ADD EAX, DWORD PTR [EBX] ; RET
```

```
0x080480da : ArithmeticLoadG : EAX +<- [EBX]
```

6.2.1 Промежуточное представление машинных инструкций

В настоящее время существует множество процессорных архитектур с различными инструкциями. Для того, чтобы абстрагироваться от специфики конкретной архитектуры для написания универсальных алгоритмов, традиционно используется промежуточное представление машинных инструкций. В этом случае алгоритмы анализа бинарного кода работают с более простым промежуточным представлением, а не с архитектурой целевого процессора.

Для классификации гаджетов используется разработанное в ИСП РАН промежуточное представление инструкций [17], удовлетворяющее SSA-форме и имеющее трехадресный код, что упрощает разработку алгоритмов анализа. Основные операторы модельной архитектуры приводятся ниже:

- **NOP.** Не имеет никакого эффекта.
- **INIT.** Инициализирует локальную переменную константным значением.
- **APPLY.** Применяет одну из модельных операций. В качестве параметров и результата используются локальные переменные.
- **BRANCH.** Осуществляет передачу управления (условную или безусловную).
- **LOAD.** Производит загрузку на локальную переменную значения из одного из адресных пространств.
- **STORE.** Записывает значение локальной переменной в одно из адресных пространств.

Адресные пространства памяти и регистров представляются в виде двух байтовых массивов. Адресное пространство регистров состоит из всех регистров машины с учетом наложений и пересечений. Для учета побочных эффектов используется слово состояния, аналогичное регистру флагов x86. На листинге 5 приводится промежуточное представление инструкции `ADD EAX, EDX` архитектуры x86-64.

```
 1. I  o.0:i16  = 0000h                  ; адрес регистра EAX в пространстве
                                         ; регистров
 2. I  o.1:i16  = 0010h                  ; адрес регистра EDX
 3. L  t.0:i32  = r[o.0]                 ; загрузка значения EAX
 4. L  t.1:i32  = r[o.1]                 ; загрузка значения EDX
 5. A  t.2      = add.i32(t.0, t.1)      ; сложение
 6. I  t.3:i16  = 0088h                  ; адрес регистра EFLAGS
 7. L  t.4:i16  = r[t.3]                 ; загрузка значения EFLAGS
 8. I  t.5:i16  = 08D5h                  ; подготовка маски для EFLAGS
 9. A  t.6      = x86.uf(t.4, t.5)       ; обновление EFLAGS с учетом маски
10. S  r[t.3]   = t.6                    ; сохранение EFLAGS
11. A  t.7      = zx.i32.i64(t.2)        ; беззнаковое расширение EAX до RAX
12. S  r[o.0]   = t.7                    ; сохранение RAX
```

Листинг 5: Промежуточное представление инструкции `ADD EAX, EDX`

6.2.2 Интерпретация промежуточного представления инструкций гаджета

Инструкции гаджета транслируются в промежуточное представление, интерпретация которого позволяет получить начальные и конечные значения регистров и памяти. Изначально для каждого адресного пространства карты считанных и сохраненных значений пусты. Инструкции промежуточного представления заменяются эквивалентными блоками инструкций x86-64. При этом инструкции сохранения (STORE) заменяются на вызовы функции, обновляющей карту сохраненных значений. А инструкции чтения (LOAD) заменяются вызовом функции, возвращающей актуальное значение, которая выполняет одно из следующих действий:

- считывает значение из карты сохраненных значений, если оно там присутствует;
- считавает значение из карты считанных значений, если оно там присутствует и отсутствует в карте сохраненных значений;
- добавляет в карту считанных значений случайно сгенерированное значение при первом обращении по адресу.

Далее производится выполнение полученного x86-64 кода. В результате будут получены начальное и конечное состояния адресных пространств.

6.3 Верификация гаджетов

Верификация гаджетов была реализована на движке символьной интерпретации Triton [18]. Triton использует конкретизацию значений символьных адресов. Поэтому для поддержки полностью символьных адресов в Triton был добавлен специальный режим SYMBOLIZED_POINTERS [19], который моделирует работу с памятью через операции select и store над SMT [14] Array. В результате верификации гаджетов из базы данных, полученной на предыдущем этапе, удаляются неверно классифицированные гаджеты.

Пример удаленного гаджета приводится на листинге 6. Для отличного от нуля начального значения регистра `eax` гаджет действительно скопирует значение регистра `ecx` в `eax`. Однако, если начальное значение `eax` будет нулевое, то и его конечное значение будет нулевое, что не является копией значения регистра `ecx`, отличного от нуля.

Листинг 6: Пример удаленного в результате верификации гаджета

```
ROPverifier - INFO - Remove id: 11341,
(0x45cb8a: neg eax ; sbb eax, eax ; and eax, ecx ;
           pop ebp ; ret),
type: move_reg, params: (in_reg: ECX, out_reg: EAX),
clobbered: (EAX, EBP, AX, AH, AL, BP),
next_addr: 4, frame_size: 8
```

7 Результаты

В рамках данной работы был разработан и реализован метод верификации семантики линейной последовательности машинных инструкций. Метод позволяет по заданному постусловию верифицировать семантику ROP гаджета, а именно доказать истинность постусловия для произвольных входных данных.

Классификация гаджета предоставляет набор постусловий, описывающих возможную семантику гаджета. Верификация гаджета позволяет формально доказать, что:

- гаджет удовлетворяет определению семантики его типа с заданными значениями параметров;
- изменяются только регистры из списка «испорченных» регистров;
- размер фрейма и адрес следующего гаджета были правильно определены на этапе классификации.

На этапе верификации удаляются неверно классифицированные гаджеты. Таким образом, верификация гаджетов устраняет ошибки классификации и дает уверенность, что тип гаджета и другие его свойства определены точно.

Средняя скорость классификации составила около 200 гаджетов в секунду, а верификации – 2 гаджета в секунду. В таблице 4 приводится количество гаджетов на каждом этапе (поиска, классификации и верификации) на наборе из 664 программ для 64-разрядной Windows 7. Всего было найдено $14 * 10^6$ гаджетов. В результате классификации для них было получено $9 * 10^6$ постусловий, описывающих семантики гаджетов. В результате верификации $65 * 10^3$ из $9 * 10^6$ постусловий оказались ложными. Ошибка классификации гаджетов составила 0.7% от всех постусловий. Верификация гаджетов позволила избавиться от этой ошибки.

Найдено	Классифицировано	**НЕ** верифицировано
$14 * 10^6$	$9 * 10^6$	$65 * 10^3$

Таблица 4: Количество гаджетов на этапах поиска, классификации и верификации гаджетов

Также разработка верификатора гаджетов и результаты его работы позволили улучшить и отладить алгоритмы классификатора гаджетов. Например, были добавлены запуски классификации на граничных условиях 0 и -1.

8 Заключение

В данной работе был предложен метод верификации семантики линейной последовательности машинных инструкций, который был реализован в виде программного инструмента. Разработанный метод позволяет точно определить семантику ROP гаджета:

- тип и параметры гаджета;
- список «испорченных» регистров;
- размер фрейма и смещение ячейки с адресом следующего гаджета.

Метод делится на три этапа: поиск, классификацию и верификацию гаджетов. Поиск гаджетов в исполняемом файле осуществляется алгоритмом Галилео. Для каждой инструкции передачи управления в исполняемых секциях программы производится дизассемблирование нескольких байтов, предшествующих инструкции. Все корректно дизассемблированные последовательности инструкций добавляются в набор найденных гаджетов.

Тип гаджета описывается семантически с помощью булевого предиката, который должен быть всегда истинным после выполнения гаджета. Функциональность гаджета описывается набором параметризованных типов, которым принадлежит гаджет.

Классификация гаджета выявляет набор возможных типов и параметров, которым он соответствует. Классификация гаджетов основывается на динамической интерпретации промежуточного представления инструкций ROP цепочки. Семантика гаджета определяется в результате анализа эффектов выполнения гаджета на нескольких различных случайных входных данных. Однако классификация гаджета не гарантирует соответствия семантике результата выполнения гаджета на произвольных входных данных, что может привести к неверной классификации. Для точного определения типов гаджета производится верификация результатов классификации.

Таким образом, классификация гаджета предоставляет набор постусловий, описывающих возможную семантику гаджета. Верификация гаджета позволяет формально доказать истинность этих постусловий для произвольных входных данных. Выполняется символьная интерпретация инструкций гаджета, а общезначимость формулы постусловия проверяется через невыполнимость ее отрицания. В результате верификации удаляются неверно классифицированные гаджеты, и тем самым устраняется ошибка

классификации гаджетов. Верификация гаджетов дает уверенность, что тип гаджета и другие его свойства определены точно.

Список литературы

[1] Common Vulnerabilities and Exposures (CVE). https://cve.mitre.org.

[2] Статистика уязвимостей (CVE) по годам. https://www.cvedetails.com/browse-by-date.php.

[3] *Вишняков, А. В.* Классификация ROP гаджетов / А. В. Вишняков // *Труды Института системного программирования РАН.* — 2016. — Т. 28, № 6. — С. 27–36.

[4] CWE-121: Stack-based Buffer Overflow. https://cwe.mitre.org/data/definitions/121.html.

[5] *Shacham, Hovav.* The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86) / Hovav Shacham // Proceedings of the 14th ACM Conference on Computer and Communications Security. — CCS '07. — ACM, 2007. — Pp. 552–561.

[6] *Salwan, Jonathan.* An introduction to the Return Oriented Programming and ROP-chain generation. — 2014. http://shell-storm.org/talks/ROP_course_lecture_jonathan_salwan_2014.pdf.

[7] *Schwartz, Edward J.* Q: Exploit Hardening Made Easy / Edward J. Schwartz, Thanassis Avgerinos, David Brumley // Proceedings of the 20th USENIX Conference on Security. — SEC'11. — USENIX Association, 2011. — P. 25.

[8] *Jager, Ivan.* Efficient Directionless Weakest Preconditions / Ivan Jager, David Brumley. — Technical Report CMU-CyLab-10-002, 2010.

[9] *King, James C.* Symbolic Execution and Program Testing / James C. King // *Communications of the ACM.* — 1976. — Vol. 19, no. 7. — Pp. 385–394.

[10] *Schwartz, Edward J.* All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask) / Edward J. Schwartz, Thanassis Avgerinos, David Brumley // Proceedings of the IEEE Symposium on Security and Privacy. — SP '10. — 2010. — Pp. 317–331.

[11] *Cha, Sang Kil.* Unleashing Mayhem on Binary Code / Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, David Brumley // Proceedings of the IEEE Symposium on Security and Privacy. — SP '12. — 2012. — Pp. 380–394.

[12] *Godefroid, Patrice.* Automated whitebox fuzz testing / Patrice Godefroid, Michael Y. Levin, David Molnar // Proceedings of the Network and Distributed System Security Symposium. — NDSS '08. — 2008. — Pp. 151–166.

[13] *Chipounov, Vitaly.* The S2E platform: design, implementation, and applications / Vitaly Chipounov, Volodymyr Kuznetsov, George Candea // *ACM Transactions on Computer Systems.* — 2012. — Vol. 30, no. 1. — Pp. 2:1–2:49.

[14] *Barrett, Clark.* The SMT-LIB Standard: Version 2.6 / Clark Barrett, Pascal Fontaine, Cesare Tinelli. — 2017.

[15] Инструмент ROPgadget. `https://github.com/JonathanSalwan/ROPgadget`.

[16] *Падарян, В. А.* Методы и программные средства, поддерживающие комбинированный анализ бинарного кода / В. А. Падарян, А. И. Гетьман, М. А. Соловьев, М. Г. Бакулин, А. И. Борзилов, В. В. Каушан, И. Н. Ледовских, Ю. В. Маркин, С. С. Панасенко // *Труды Института системного программирования РАН.* — 2014. — Т. 26, № 1. — С. 251–276.

[17] *Падарян, В. А.* Моделирование операционной семантики машинных инструкций / В. А. Падарян, М. А. Соловьев, А. И. Кононов // *Труды Института системного программирования РАН.* — 2010. — Т. 19. — С. 165–186.

[18] *Saudel, Florent.* Triton: A Dynamic Symbolic Execution Framework / Florent Saudel, Jonathan Salwan // Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015. — SSTIC, 2015. — Pp. 31–54.

[19] *Вишняков, А.В.* Symbolized pointers mode. `https://github.com/JonathanSalwan/Triton/pull/723`.