Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский физико-технический институт (государственный университет)»

Факультет радиотехники и кибернетики
Кафедра теоретической и прикладной информатики

Направление подготовки/специальность: 03.04.01 Прикладные математика и физика
Направленность (профиль) подготовки: Математические и информационные технологии
Форма обучения: очная

Допущен к ГИА
Руководитель учебного подразделения
_____/_____
(подпись) *расшифровка подписи*
«____» _____20____г.

Динамическая модификация кода монитора виртуальных машин.

(магистерская диссертация)

Обучающийся: Конычев Валерий Витальевич

_____
(подпись обучающегося)

Научный руководитель: Пашков Руслан Анатольевич, к.ф.-м.н.

_____
(подпись научного руководителя)

Научный консультант: Корякин Алексей Борисович

_____
(подпись научного консультанта)

Москва 2018

Аннотация

В результате данной научно–исследовательской работы была разработана кроссплатформенная библиотека для модификации кода, написанного на языках C/C++ и компилируемого для архитектуры Intel® 64.

Необходимость данной работы обусловлена наличием актуальных сценариев для быстрой модификации кода: скрытие и восстановление редко используемых участков кода, а также расширение функциональности уже скомпилированного кода.

Помимо кроссплатформенности интересной особенностью технологии является также отсутствие прерываний во время работы, что существенно ускоряет работу модифицируемого кода и расширяет области применения технологии.

Разработанная технология внесена в окружение монитора виртуальных машин Parallels Desktop®, где была подтверждена её полная работоспособность. На данный момент начинается её постепенное коммерческое применение.

Данная научно–исследовательская работа будет интересна всем, кто занимается системным программированием, и тем, кто ищет решение похожих проблем. Область применения технологии очень широка это и мониторы виртуальных машин, для которых она была в первую очередь разработана, и ядра операционных систем, и драйвера устройств, и обфускация кода, и, наконец, обычные пользовательские приложения.

Содержание

1 Введение **7**

1.1 Актуальность работы 7

1.2 Цели работы 8

1.3 Задачи работы 9

2 Обзор литературы **10**

2.1 Обзор Parallels Desktop® 10

2.2 Прямое исполнение 11

2.3 Обзор компилятора clang 13

2.4 Бинарные форматы исполняемых файлов 13

2.4.1 ELF 14

2.4.2 Mach-o 17

3 Сравнение с существующими технологиями динамической модификации кода **22**

3.1 Static keys 22

3.2 KProbes 25

3.3 Ftrace 26

3.4 Ptrace 26

4 Описание разработанных технологий динамической модификации кода гипервизора **29**

4.1 MOD_CODE 29

4.2 Перемещение инструкций архитектуры Intel® 64 33

4.2.1 Структура команды архитектуры Intel® 64 33

4.2.2 Режимы адресации в Intel® 64 35

4.2.3 Проблемы при перемещении x86 кода 35

4.2.4 Решение проблем при перемещении x86 кода 36

4.3 Модификация произвольных участков в технологии MOD_CODE 41

4.4 post_process 43

4.5 MOD_SCRIPT 44

5 Решённые проблемы технологии динамической модификации кода **46**
5.1 Неисправность работы линковщика mach-o файлов 46
5.2 Использование технологии MOD_CODE во встраиваемых функциях 50
5.3 Выравнивание 50
5.4 Проблемы с ключевым словом volatile 51

6 Заключение **53**

Список иллюстраций

1 Последовательность событий при работе с Intel® VT-x 12
2 Схема работы Parallels Desktop®. 13
3 ELF-файл с точки зрения линковщика. 14
4 ELF-файл с точки зрения загрузчика. 15
5 Базовая структура mach-o-файла. 18
6 Цепочка вызовов для уведомления strace о системном вызове. . . 28
7 Структура команды архитектуры Intel® 64. 34
8 Пример переноса кода Intel® 64. 36
9 Первый вариант модификации примера переноса кода Intel® 64. 37
10 Второй вариант модификации примера переноса кода Intel® 64. . 37
11 Модификации ModR/M байта RIP-relative инструкции. 40
12 Программа до обработки post_process. 42
13 Программа после обработки post_process. 42
14 Восстановление скрытой функциональности. 43

Список листингов

1 Структура ELF header для 64-битной версии ОС. 15
2 Описатель сегмента ELF-файла для 64-битной версии ОС. 16
3 Описатель секции ELF-файла для 64-битной версии ОС. 16
4 Описатель символа ELF-файла для 64-битной версии ОС. 17
5 Заголовок mach-o-файла для 64-битной ОС. 18

6 Заголовок толстого бинарника mach-o–файла. 19
7 Заголовок команд загрузки mach-o-файла. 19
8 Команда загрузки 64-битного сегмента mach-o-файла. 19
9 Описатель 64-битной секции mach-o-файла. 20
10 Описатель таблицы символов mach-o-файла. 20
11 Запись в таблицы символов mach-o-файла для 64-битной архитектуры. 20
12 Команда загрузки используемая динамическим загрузчиком mach-o–файла. 21
13 Пример использования static keys. 23
14 Пример использования static_key_true. 23
15 Пример использования static_key_false. 23
16 Пример функции getppid. 23
17 Результат компиляции функции getppid с static key. 24
18 Результат компиляции функции getppid без static key. 24
19 Структура kprobe для ядра ОС Linux 2.6.11.7 25
20 Вызов ptrace с флагом PTRACE_TRACEME 27
21 Определение _TIF_WORK_SYSCALL_ENTRY 28
22 Структура описывающая точку модификации 30
23 Внутреннее представление MOD_CODE 31
24 Прыжок в любую точку 64-битного адресного пространства . . . 32
25 Применяемые вызовы функции при переносе кода. 37
26 Все jmp_stub, применяемые при перемещении кода Intel® 64. . . 38
27 Варианты jcc_stub для инструкций *jcc rel8*. 39
28 Call_stub для 64 битного адресного пространства. 39
29 Stub для RIP-relative инструкций Intel® 64. 40
30 Описатель релокации mach-o формата для Intel® 64. 46
31 Программа для воспроизведения неисправности линковщика mach-o файлов. 47
32 Результат компиляции программы для воспроизведения неисправности линковщика mach-o файлов. 47
33 Объектный файл программы для воспроизведения неисправности линковщика mach-o файлов. 48
34 Обход неисправности линковщика mach-o файлов. 49

35 Объектный файл, генерируемый для обхода неисправности линковщика mach-o файлов. 49
36 Исполняемый файл, генерируемый для обхода неисправности линковщика mach-o файлов. 49
37 Ассемблерная метка. 50
38 Ограничение оптимизаций компилятора для участка кода. 52

1 Введение

Технология динамической модификации кода (SMC — *англ.* self-modifying code) нашла широкое применение в программировании: в различных вирусах, загрузчиках, интерпретаторах, бинарных трансляторах, операционных системах (ОС), гипервизорах виртуальных машин [1], отладчиках и в обфускации[1] кода [2].

В данной работе исследовалось применение динамической модификации кода в мониторе виртуальных машин, в результате чего была разработана кроссплатформенная библиотека для SMC, применение которой можно найти не только в гипервизоре, но и в ядрах ОС, драйверах и пользовательских приложениях.

1.1 Актуальность работы

Многие из существующих технологий динамической модификации кода являются отладочными. Так как в отладочных программах быстродействие не является критически важным критерием, то в их реализациях нередко можно встретить использование прерываний. Например, в программах для архитектуры Intel® 64 применяется специально предназначенное для этого прерывание *int 3*, обработчик которой зарезервирован для отладочных целей [3]. Длина *int 3* равна одному байту. Именно размер инструкции позволяет использовать её в качестве отладочного прерывания в архитектуре Intel® 64, потому что позволяет заменять любую команду, длина которой может быть от одного до пятнадцати байтов.

Однако, *int 3* — прерывание, и его обработка — небыстрый процесс. При этом есть сценарии, в которых может быть необходимо быстро модифицировать код. Например, можно скрывать редко используемую функциональность, убирая её из основного кода, и при необходимости как можно быстрее восстанавливать скрытые участки. Очевидно, что прерывание для такой ситуации не сильно подходит. Во-первых, из-за времени обработки. Во-вторых, прерывание будет срабатывать постоянно при достижении места его расположения в коде. Примером технологии, применимой в таком сценарии, может служить static

[1] Обфускация (*англ.* obfuscation) — технология защиты программного продукта от реверс–инжиниринга с помощью изменения кода, при котором сохраняется его корректная работоспособность.

keys ядра ОС Linux [4, 5]. Подробно технология рассмотрена в разделе 3.1.

Второй интересный сценарий применения динамической модификации кода, предназначенный не для отладки, расширение функциональности уже скомпилированной программы. Данная возможность способна улучшить поддержку программного продукта за счёт того, что позволит компаниям быстрее выпускать исправления для конкретного клиента до глобальных обновлений. Также подобная технология способна упростить процесс тестирования программ благодаря дополнительным возможностям автоматизации без внесения новых участков кода в продукт.

Однако, несмотря на существование подобных сценариев применения динамической модификации кода, на данный момент нет кроссплатформенной технологии, позволяющей модифицировать программы, написанные на языках C/C++ для Intel® 64, во время исполнения и не предназначенной для отладки. Что делает данную работу крайне актуальной.

1.2 Цели работы

У данной работы было две цели. Первой целью исследования было сделать возможным исключать участки программы, которые собирают, выводят диагностические данные, или реализуют функциональность специфичную для малого числа аппаратных платформ, из основного кода диспетчера виртуальных машин Parallels Desktop® [6], позволив динамически включать их исполнение с помощью непосредственной модификации исполняемого кода. Это позволит уменьшить объем основной части исполняемого файла и, соответственно, объем используемого кеша процессора, а также избежать избыточного условного ветвления потока исполняемых команд.

Второй целью исследования было расширение функциональности уже скомпилированной программы. Что, как уже упоминалось, позволяет улучшить поддержку гипервизора и упростить процессы тестирования.

Результатом работы должна являться автономная библиотека для языков C/C++ для динамической модификации кода, скомпилированного с помощью компилятора — clang [7] для архитектуры Intel® 64 [3].

1.3 Задачи работы

В процессе работы решались следующие задачи:

1. Изучить существующие методы динамической модификации кода.
2. Проанализировать наличие новых технологий, позволяющих добиться поставленных целей.
3. Выбрать способ и технологии для реализации библиотеки.
4. Разработать простой прототип библиотеки для первичного тестирования и выявления потенциальных проблем.
5. Решение выявленных проблем.
6. Усовершенствование библиотеки.
7. Внесение библиотеки в окружение гипервизора Parallels Desktop®.
8. Исправление возникших ошибок.
9. Выбор частей монитора виртуальных машин, к которым будет применена технология.
10. Полное тестирование гипервизора вместе с разработанной библиотекой.

2 Обзор литературы

В обзоре литературы представлены научные статьи, техническая документация и технологии, которые были исследованы в процессе работы. Часть из них были изучены для ознакомления с текущим опытом в области динамической модификации кода, часть для реализации модулей итоговой библиотеки, другие для понимания того, с какими инструментами нужно работать и как их можно использовать.

2.1 Обзор Parallels Desktop®

Данная работа выполнялась для гипервизора виртуальных машин Parallels Desktop®, который работает, как пользовательское приложение операционной системы, и позволяет работать на Mac OS X с другими операционными системами такими как Windows, Linux, и их приложениями [6].

В своём составе Parallels Desktop® содержит улиты Parallels Tools для улучшения работы гостевых операционных систем. Parallels Tools позволяет улучшить производительность, и делает более комфортным использование гипервизора [6] с различными ОС.

Parallels Desktop® работает как на уровне kernel space[2], так и на user space[3]. На user space работает пользовательский клиент, и происходит эмуляция внешних устройств. На kernel space работает монитор виртуальных машин.

Гипервизор может работать в нескольких режимах. Первый — режим прямого исполнения, в котором монитор старается находится максимально возможное время. Так как Parallels Desktop® предназначен для работы под ОС Mac OS X, которая в свою очередь сейчас поставляется на устройствах, в основе которых находятся 64-битные процессоры семейства x86, то прямое исполнение основано на технологии аппаратной виртуализации Intel® VT-x [3, том 3]. Более подробно прямое исполнение разобрано в секции 2.2.

Второй — бинарная трансляция (БТ). БТ используется на платформах без поддержки аппаратной виртуализации, или с ней, но без возможности эмуляции реального режима процессоров x86, так как на начальном этапе загрузки

[2]Kernel space — уровень работы ядра операционной системы, что соответствует нулевому кольцу защищённого режима работы процессора x86.

[3]User space — уровень работы пользовательских приложений, что соответствует 3 кольцу защищённого режима работы процессора x86.

клиент работает в этом режиме.

Третий — интерпретация отдельных инструкций. Когда гость пытается выполнить команду, недоступную в прямом исполнении, происходит прерывание, и гипервизор просто вызывает функцию-обработчик для неё.

2.2 Прямое исполнение

Прямое исполнение (*англ.* Direct execution) — режим работы монитора виртуальных машин, когда код гостя напрямую исполняется на хозяйской аппаратуре, что возможно, когда их архитектура совпадает, например, x86 на x86.

Однако, для эффективного прямого исполнения архитектура должна удовлетворять достаточным условиям виртуализируемости, которые были сформулированы в классической работе по виртуальным машинам *Попеком* и *Гольдбергом* [8] в 1974 году.

В исследовании Попека и Гольдберга были введены следующие понятия:

- Служебные инструкции (*англ.* sensitive instructions) — инструкции изменяющие режим работы процессора или положение и размер доступного сегмента памяти.

- Привилегированные инструкции (*англ.* privileged instructions) — их исполнение в пользовательском режиме вызывает прерывание.

С их помощью было сформулировало достаточное условие виртуализируемости — архитектура может быть эффективно подвергнута виртуализации, если множество служебных инструкций является подмножеством привилегированных. Данное условие может быть переформулировано: при попытке исполнения служебных инструкций в режиме, где работает виртуальная машина, аппаратура должна генерировать системное прерывание [9].

Так как многие инструкции x86 можно исполнять в нескольких режимах без вызова прерываний, при этом поведение операций зависит от режима процессора, например, команда POPF [3, том 2], то очевидно, что архитектура Intel® 64 сама по себе не удовлетворяет достаточным условиям виртуализируемости. Потому в 2005 году в архитектуру было добавлено расширение аппаратной виртуализации Intel® VT-x, которое является основой эффективного прямого исполнения x86 на x86.

На рис. 1 изображена временная ось, на которой отображаются события цикла работы VT-x. После выполнения команды VMXON симулятор входит в режим VMX root, где гипервизор выполняет необходимые настройки и с помощью команды VMLAUNCH переходит в режим VMX non-root. В этот момент начинается загрузка состояния виртуальной машины, что происходит не моментально а в течении некоторого времени — Entry time. Исполнение виртуальной машины начинается только после полной загрузки, однако некоторые процессы, связанные с самим VT-x, могут начаться и раньше.

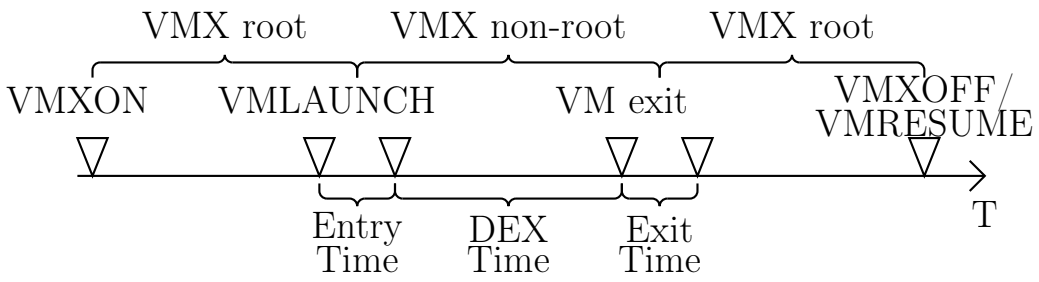

Рисунок 1 – Последовательность событий при работе с Intel® VT-x. Entry/Exit Time - время перехода между режимами исполнения. DEX Time - время прямого исполнения.

Выход из прямого исполнения VM exit может наступить по достаточно большому спектру условий, например, потому что гостевая операционная система решила прочитать MSR[4] [3, Том 3], который ей запрещено читать. Выход, так же как и загрузка, происходит не моментально, а в течении — Exit time.

После этого гипервизор выясняет причину выхода и может либо выйти из режима VMX root с помощью операции VMXOFF, либо вернуться к прямому исполнению с помощью команды VMRESUME. С учётом всего выше сказанного и разобранного прямого исполнения схему работы Parallels Desktop® можно представить в виде рис. 2. Из схемы видно, что технология динамической модификации кода способна существенно ускорить работу в ситуациях, которые вызывают VM Exit.

[4]MSR (model specific register) — специальные регистры архитектуры Intel® 64, предназначенные для доступа к различным особенностям процессора, таким как управление режимом работы процессора, счётчики и т.д.

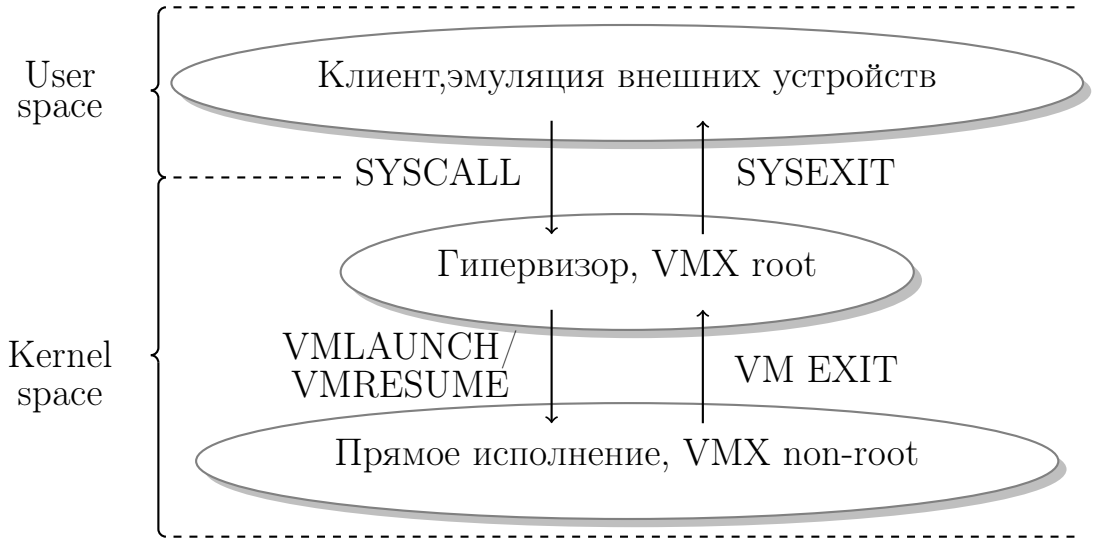

Рисунок 2 – Схема работы Parallels Desktop®.

2.3 Обзор компилятора clang

Clang [7] — фронтенд[5] компилятора языков C, C++, Objective-C, Objective-C++ и их расширений OpenMP, OpenCL и CUDA. Бекендом[6] является llvm [10]. Clang транслирует конструкции языков в команды виртуальной машины llvm, которая потом оптимизирует и компилирует их в машинный код.

В контексте данного исследования компилятор изучался на возможность влияния на процесс компиляции без внесения изменений в сам компилятор, что обусловлено коммерческой стороной разработки библиотеки для динамической модификации кода. Потому были изучены плагины clang plugins [11] для clang, позволяющие применять во время компиляции дополнительные действия, определённые пользователем. Однако, в результате исследования было выяснено, что на основе данного расширения можно построить анализирующие инструменты, но никак нельзя повлиять на процесс компиляции.

2.4 Бинарные форматы исполняемых файлов

Любой исполняемый файл на современных операционных системах Linux, Mac OS X и Windows содержит не только последовательность машинных инструкций, которые процессору необходимо выполнить, но и большое количество служебной информации, которую используют различные программы, такие как

[5]Фронтенд (*англ.* front-end) — часть программного продукта, с которой взаимодействует пользователь.
[6]Бекенд (*англ.* back-end) — часть программного продукта, обеспечивающая работоспособность фронтенда.

динамический загрузчик, линковщик и отладчик. Наиболее распространёнными бинарными форматами сейчас являются ELF, Mach-o и PE.

2.4.1 ELF

ELF (Executable and linkable format) [12] — бинарный формат большинства UNIX подобных операционных систем, включая Linux, FreeBSD и Solaris. ELF является наследником другого бинарного формата COFF [13] (Common object file format) и был представлен в октябре 1988 года в стандарте System V Release 3.

ELF-файл содержит информацию, как для исполнения, так и для создания конечного файла. Потому на ELF-файл можно смотреть с двух сторон. Во-первых, со стороны линковщика[7] – рис. 3. Во-вторых, с точки зрения загрузчика исполняемых файлов – рис. 4. Целью подобных представлений является ускорение и упрощение процессов линковки и загрузки соответственно.

ELF-файл начинается с заголовка — ELF header, структура которого представлена в листинге 1. Заголовок содержит в себе всю информацию об устройстве файла, для какой платформы он скомпилирован и о его типе[8]

ELF header
Program header table
Section 1
...
Section n
...
Section header table

Рисунок 3 – ELF-файл с точки зрения линковщика.

[7]Файлы, с которыми работает линковщик, в стандарте ELF называются релоцируемыми (*англ.* relocatable).

[8]Обычно Elf-файлы бывают исполняемыми (*англ.* executable), релоцируемыми (*англ.* relocatable) и динамически подгружаемыми (*англ.* shared object file).

ELF header
Program header table
Segment 1
...
Segment n
...
Section header table

Рисунок 4 – ELF-файл с точки зрения загрузчика.

После заголовка обычно располагается Program header table — таблица, содержащая информацию для загрузчика о том, как создать процесс для исполнения файла. При этом, таблица может отсутствовать в релоцируемом файле за ненадобностью, тогда поле *e_phoff* заголовка, указывающее на местоположение таблицы в файле, будет равно нулю.

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT]; /* Magic number and other info */
  Elf64_Half    e_type;         /* Object file type */
  Elf64_Half    e_machine;      /* Architecture */
  Elf64_Word    e_version;      /* Object file version */
  Elf64_Addr    e_entry;        /* Entry point virtual address */
  Elf64_Off     e_phoff;        /* Program header table file offset */
  Elf64_Off     e_shoff;        /* Section header table file offset */
  Elf64_Word    e_flags;        /* Processor-specific flags */
  Elf64_Half    e_ehsize;       /* ELF header size in bytes */
  Elf64_Half    e_phentsize;    /* Program header table entry size */
  Elf64_Half    e_phnum;        /* Program header table entry count */
  Elf64_Half    e_shentsize;    /* Section header table entry size */
  Elf64_Half    e_shnum;        /* Section header table entry count */
  Elf64_Half    e_shstrndx;     /* Section header string table index */
} Elf64_Ehdr;
```

Листинг 1 – Структура ELF header для 64-битной версии ОС.

Каждый элемент Program header table описывает свой сегмент (листинг 2): расположение и размер в файле и в виртуальном пространстве, что содержит, какие права доступа.

```
typedef struct
{
  Elf64_Word  p_type;     /* Segment type */
  Elf64_Word  p_flags;    /* Segment flags */
  Elf64_Off   p_offset;   /* Segment file offset */
  Elf64_Addr  p_vaddr;    /* Segment virtual address */
  Elf64_Addr  p_paddr;    /* Segment physical address */
  Elf64_Xword p_filesz;   /* Segment size in file */
  Elf64_Xword p_memsz;    /* Segment size in memory */
  Elf64_Xword p_align;    /* Segment alignment */
} Elf64_Phdr;
```

Листинг 2 – Описатель сегмента ELF-файла для 64-битной версии ОС.

Каждый сегмент является набором одной или более секции. Сами секции описаны в другой таблице — Section header table, местоположение которой хранится в поле *e_shoff* заголовка. Эта таблица может отсутствовать в исполняемом файле, тогда *e_shoff* будет равна нулю. Элементы Section header table представлены в листинге 3.

```
typedef struct
{
  Elf64_Word  sh_name;    /* Section name (string tbl index) */
  Elf64_Word  sh_type;    /* Section type */
  Elf64_Xword sh_flags;   /* Section flags */
  Elf64_Addr  sh_addr;    /* Section virtual addr at execution */
  Elf64_Off   sh_offset;  /* Section file offset */
  Elf64_Xword sh_size;    /* Section size in bytes */
  Elf64_Word  sh_link;    /* Link to another section */
  Elf64_Word  sh_info;    /* Additional section information */
  Elf64_Xword sh_addralign;   /* Section alignment */
  Elf64_Xword sh_entsize; /* Entry size if section holds table */
} Elf64_Shdr;
```

Листинг 3 – Описатель секции ELF-файла для 64-битной версии ОС.

Есть множество различных типов секций, наиболее интересными из которых являются:

- SHT_SYMTAB – содержит таблицу символов (листинг 4). Символы содержат информацию о различных объектах, местоположение которых может понадобиться, например, функции, глобальные переменные, метки в коде и т.д.

- SHT_DYNSYM – содержит таблицу динамических символов, они не определены в данном исполняемом файле и подгружаются на этапе исполне-

ния из динамических библиотек.

- SHT_STRTAB – содержит таблицу строк. Таблица строк организована, как в COFF-формате [13]: последовательность ASCII символов, разделённых символом '\0'.
- SHT_REL/SHT_RELA – содержит структуры данных, описывающие объекты в коде, которые должны быть отреалоцированы.
- SHT_DYNAMIC – содержит информацию, используемую динамическим загрузчиком.

```
typedef struct
{
  Elf64_Word    st_name;    /* Symbol name (string tbl index) */
  unsigned char st_info;    /* Symbol type and binding */
  unsigned char st_other;   /* Symbol visibility */
  Elf64_Section st_shndx;   /* Section index */
  Elf64_Addr    st_value;   /* Symbol value */
  Elf64_Xword   st_size;    /* Symbol size */
} Elf64_Sym;
```

Листинг 4 – Описатель символа ELF-файла для 64-битной версии ОС.

2.4.2 Mach-o

Mach-o [14] — бинарный формат операционных систем Mac OS X, является наследником другого формата a.out, который впоследствии был заменён на UNIX системах COFF-стандартом. Базовая структура mach-o файла представлена на рис. 5.

Так же как и ELF, mach-o-файл начинается с заголовка (листинг 5). Заголовок mach-o содержит в себе информацию о том, для какой архитектуры бинарник скомпилирован, тип файла[9], количество команд загрузки, с которыми работают линковщик и загрузчик, а также флаги описывающие особенности файла.

[9]Обычно mach-o-файлы бывают исполняемыми (*англ.* executable), объектными (*англ.* object, или релоцируемые в ELF) и динамически подгружаемыми библиотеками (*англ.* dynamic shared libraries).

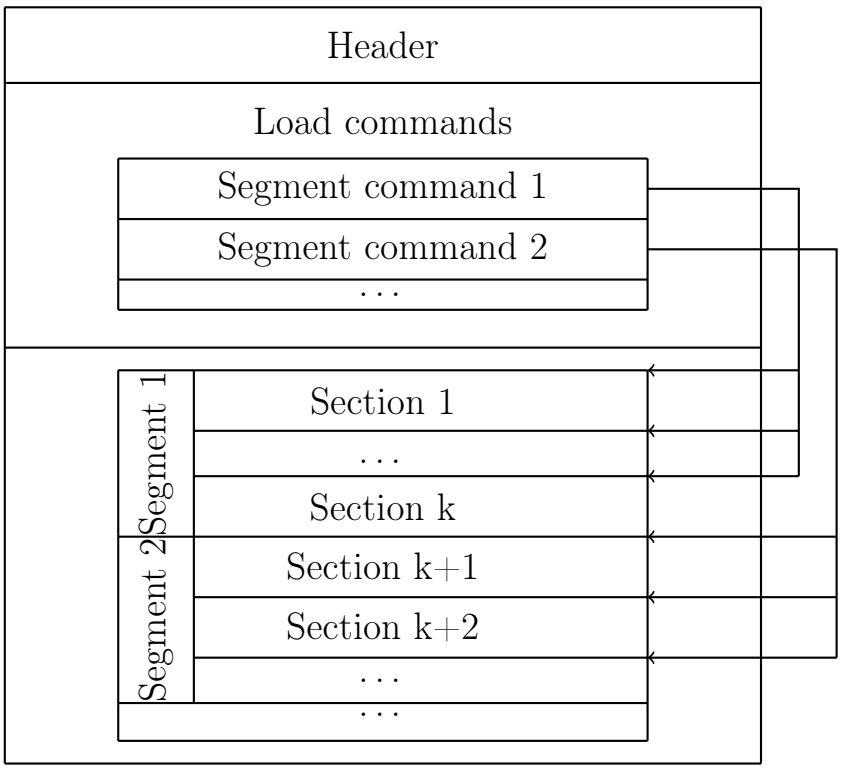

Рисунок 5 – Базовая структура mach-o-файла.

```
struct mach_header_64 {
  uint32_t      magic;      /* mach magic number identifier */
  cpu_type_t    cputype;    /* cpu specifier */
  cpu_subtype_t cpusubtype; /* machine specifier */
  uint32_t      filetype;   /* type of file */
  uint32_t      ncmds;      /* number of load commands */
  uint32_t      sizeofcmds; /* the size of all the load commands */
  uint32_t      flags;      /* flags */
  uint32_t      reserved;   /* reserved */
};
```

Листинг 5 – Заголовок mach-o-файла для 64-битной ОС.

Интересной особенностью mach-o-стандарта является так называемый толстый бинарник (*англ.* fat binary), содержащий в себе более одного двоичного файла, скомпилированных для разных архитектур. Fat binary начинается не с обычного заголовка, а со структуры *fat_header* (листинг 6). Если это толстый бинарник, то начинаться он будет со значения 0xcafebabf, записанного в big-endian[10] порядке. Соответственно на big-endian архитектурах будет FAT_MAGIC_64(0xcafebabf), little-endian[11] – FAT_CIGAM_64(0xbfbafeca).

[10]Big-endian — порядок записи байт, при котором старшие байты записываются по меньшим адресам, а младшие по большим.

[11]Little-endian — порядок записи байт, при котором младшие байты записываются по младшим адресам, а старшие по большим.

```
struct fat_header {
  uint32_t      magic;      /* FAT_MAGIC or FAT_MAGIC_64 */
  uint32_t      nfat_arch;  /* number of structs that follow */
};
```

Листинг 6 – Заголовок толстого бинарника mach-o–файла.

Сразу после заголовка идут *ncmds* команд загрузки, каждая из которых начинается со структуры, представленной на листинге 7 и содержащий тип и размер команды.

```
struct load_command {
  uint32_t      cmd;       /* type of load command */
  uint32_t      cmdsize;   /* total size of command in bytes */
};
```

Листинг 7 – Заголовок команд загрузки mach-o-файла.

Есть множество различных типов команд, все их можно узнать из файла */usr/include/mach-o/loader.h* операционной системы семейства Mac OS X. Далее будут разобраны только три из них: LC_SEGMENT_64, LC_SYMTAB, LC_DYSYMTAB.

LC_SEGMENT_64 — команда описывающая загрузку 64–битного сегмента — набора 64–битных секций. Команда загрузки сегмента представлена в листинге 8. Поле *cmdsize* структуры *segment_command_64* учитывает размер описателей всех секций, которые сегмент содержит (листинге 9), так как они идут сразу после команды загрузки.

```
struct segment_command_64 { /* for 64-bit architectures */
  uint32_t      cmd;        /* LC_SEGMENT_64 */
  uint32_t      cmdsize;    /* includes sizeof section_64 structs */
  char          segname[16];    /* segment name */
  uint64_t      vmaddr;     /* memory address of this segment */
  uint64_t      vmsize;     /* memory size of this segment */
  uint64_t      fileoff;    /* file offset of this segment */
  uint64_t      filesize;   /* amount to map from the file */
  vm_prot_t     maxprot;    /* maximum VM protection */
  vm_prot_t     initprot;   /* initial VM protection */
  uint32_t      nsects;     /* number of sections in segment */
  uint32_t      flags;      /* flags */
};
```

Листинг 8 – Команда загрузки 64-битного сегмента mach-o-файла.

```
struct section_64 { /* for 64-bit architectures */
  char          sectname[16];   /* name of this section */
  char          segname[16];    /* segment this section goes in */
  uint64_t      addr;       /* memory address of this section */
  uint64_t      size;       /* size in bytes of this section */
  uint32_t      offset;     /* file offset of this section */
  uint32_t      align;      /* section alignment (power of 2) */
  uint32_t      reloff;     /* file offset of relocation entries */
  uint32_t      nreloc;     /* number of relocation entries */
  uint32_t      flags;      /* flags (section type and attributes)*/
  uint32_t      reserved1;  /* reserved (for offset or index) */
  uint32_t      reserved2;  /* reserved (for count or sizeof) */
  uint32_t      reserved3;  /* reserved */
};
```

Листинг 9 – Описатель 64-битной секции mach-o-файла.

LC_SYMTAB — команда загрузки, описывающая таблицу символов (листинг 10). Таблица символов используется, как линковщиком, так и динамическим загрузчиком, также она используется отладчиком, так как таблица содержит в себе все символы и статические, и динамические, и отладочные. Элемент таблицы представлен в листинге 11.

```
struct symtab_command {
  uint32_t      cmd;        /* LC_SYMTAB */
  uint32_t      cmdsize;    /* sizeof(struct symtab_command) */
  uint32_t      symoff;     /* symbol table offset */
  uint32_t      nsyms;      /* number of symbol table entries */
  uint32_t      stroff;     /* string table offset */
  uint32_t      strsize;    /* string table size in bytes */
};
```

Листинг 10 – Описатель таблицы символов mach-o-файла.

```
struct nlist_64 {
  union {
      uint32_t  n_strx;     /* index into the string table */
  } n_un;
  uint8_t       n_type;     /* type flag, see below */
  uint8_t       n_sect;     /* section number or NO_SECT */
  uint16_t      n_desc;     /* see <mach-o/stab.h> */
  uint64_t      n_value;    /* value of this symbol (or stab offset) */
};
```

Листинг 11 – Запись в таблицы символов mach-o-файла для 64-битной архитектуры.

LC_DYSYMTAB — команда загрузки, предоставляющая дополнительную информацию для динамического загрузчика (листинг 12).

```
struct dysymtab_command {
  uint32_t  cmd;/* LC_DYSYMTAB */
  uint32_t  cmdsize;/* sizeof(struct dysymtab_command) */
  uint32_t  ilocalsym;/* index to local symbols */
  uint32_t  nlocalsym;/* number of local symbols */
  uint32_t  iextdefsym;/* index to externally defined symbols */
  uint32_t  nextdefsym;/* number of externally defined symbols */
  uint32_t  iundefsym;/* index to undefined symbols */
  uint32_t  nundefsym;/* number of undefined symbols */
  uint32_t  tocoff;/* file offset to table of contents */
  uint32_t  ntoc;/* number of entries in table of contents */
  uint32_t  modtaboff;/* file offset to module table */
  uint32_t  nmodtab;/* number of module table entries */
  uint32_t  extrefsymoff;/* offset to referenced symbol table */
  uint32_t  nextrefsyms;/* number of referenced symbol table entries */
  uint32_t  indirectsymoff;/* file offset to the indirect symbol table */
  uint32_t  nindirectsyms;/* number of indirect symbol table entries */
  uint32_t  extreloff;/* offset to external relocation entries */
  uint32_t  nextrel;/* number of external relocation entries */
  uint32_t  locreloff;/* offset to local relocation entries */
  uint32_t  nlocrel;/* number of local relocation entries */
};
```

Листинг 12 – Команда загрузки используемая динамическим загрузчиком mach-o–файла.

3 Сравнение с существующими технологиями динамической модификации кода

Как упоминалось ранее, технологии динамической модификации кода нашли широкое применение в индустрии информационных технологий, при этом многие из них предназначены для отладки и профилировки. В данном разделе будут рассмотрены только некоторые из них, которые изменяют код на уровне машинных инструкций архитектуры Intel® 64.

3.1 Static keys

Static keys — технология динамической модификации кода, позволяющая исключать и восстанавливать редко используемые участки кода ядра ОС Linux [4, 5], который должен работать так быстро, как только можно.

Первоначально static keys разрабатывалась для уменьшения накладных расходов на работу другой технологии — tracepoint [15]. Сейчас технология static keys доступна во всех частях ядра.

Применительно к tracepoint static keys позволяет избегать избыточного условного ветвления и уменьшает давление на кэш данных, так как при использовании tracepoint проверяется глобальная переменная, которая к тому же может быть разделена между несколькими исполнителями. При этом важно отметить, что почти всегда tracepoint'ы не активны и не представляют из себя какой-либо функциональности ядра ОС Linux, потому накладные расходы на них разработчики хотели свести к минимуму.

Технология static keys является зависимой от компилятора из за того, что основана на структуре *asm goto*, поддерживаемой только GCC. Структура *asm goto* позволяет заменять условные конструкции на безусловные на этапе компиляции, что избавляет от проверки глобальной переменной. На этапе исполнения подобные места могут быть модифицированы, чтобы изменить поток исполнения, включив скрытый код.

В листинге 13 показан пример использования static keys. После компиляции функции *printk* не будет в основном потоке исполнения, она будет скрыта на unlikely пути. На месте проверки условия будет инструкция NOP длинной 5 байт, которая на этапе исполнения может быть заменена безусловным перехо-

дом на *printk*.

```
if (static_key_false(&key))
        printk("I am the true branch\n");
```

Листинг 13 – Пример использования static keys.

Для использования static keys необходимо определить глобальную переменную типа *struct static_key* и проинициализировать через макросы STATIC_KEY_INIT_TRUE или STATIC_KEY_INIT_FALSE, по умолчанию переменная будет инициализирована как FALSE. Для переменных инициализированных с помощью STATIC_KEY_INIT_TRUE нужно использовать static_key_true() (листинг 14), а для STATIC_KEY_INIT_FALSE – static_key_false() (листинг 15)

```
if (static_key_false(&key))
        do unlikely code
else
        do likely code
```

Листинг 14 – Пример использования static_key_true.

```
if (static_key_true(&key))
        do likely code
else
        do unlikely code
```

Листинг 15 – Пример использования static_key_false.

Примеры скомпилированного кода из листинга 16 представлены на листингах 17 и 18. Как видно, static keys также уменьшает размер исполняемого кода: 5 байт для NOP вместо 10 байт для mov, test и jne.

```
SYSCALL_DEFINE0(getppid)
{
        int pid;

        if (static_branch_unlikely(&key))
                printk("I am the true branch\n");

        rcu_read_lock();
        pid = task_tgid_vnr(rcu_dereference(current->real_parent));
        rcu_read_unlock();

        return pid;
}
```

Листинг 16 – Пример функции getppid.

```
ffffffff81044290 <sys_getppid>:
ffffffff81044290:  55                    push   %rbp
ffffffff81044291:  48 89 e5              mov    %rsp,%rbp
ffffffff81044294:  e9 00 00 00 00        jmpq   <sys_getppid+0x9>
ffffffff81044299:  65 48 8b 04 25 c0 b6 mov    %gs:0xb6c0,%rax
ffffffff810442a0:  00 00
ffffffff810442a2:  48 8b 80 80 02 00 00 mov    0x280(%rax),%rax
ffffffff810442a9:  48 8b 80 b0 02 00 00 mov    0x2b0(%rax),%rax
ffffffff810442b0:  48 8b b8 e8 02 00 00 mov    0x2e8(%rax),%rdi
ffffffff810442b7:  e8 f4 d9 00 00        callq  <pid_vnr>
ffffffff810442bc:  5d                    pop    %rbp
ffffffff810442bd:  48 98                 cltq
ffffffff810442bf:  c3                    retq
ffffffff810442c0:  48 c7 c7 e3 54 98 81 mov    $0xffffffff819854e3,%rdi
ffffffff810442c7:  31 c0                 xor    %eax,%eax
ffffffff810442c9:  e8 71 13 6d 00        callq  <printk>
ffffffff810442ce:  eb c9                 jmp    <sys_getppid+0x9>
```

Листинг 17 – Результат компиляции функции getppid с static key.

```
ffffffff810441f0 <sys_getppid>:
ffffffff810441f0:  8b 05 8a 52 d8 00     mov    <key>,%eax
ffffffff810441f6:  55                    push   %rbp
ffffffff810441f7:  48 89 e5              mov    %rsp,%rbp
ffffffff810441fa:  85 c0                 test   %eax,%eax
ffffffff810441fc:  75 27                 jne    <sys_getppid+0x35>
ffffffff810441fe:  65 48 8b 04 25 c0 b6 mov    %gs:0xb6c0,%rax
ffffffff81044205:  00 00
ffffffff81044207:  48 8b 80 80 02 00 00 mov    0x280(%rax),%rax
ffffffff8104420e:  48 8b 80 b0 02 00 00 mov    0x2b0(%rax),%rax
ffffffff81044215:  48 8b b8 e8 02 00 00 mov    0x2e8(%rax),%rdi
ffffffff8104421c:  e8 2f da 00 00        callq  <pid_vnr>
ffffffff81044221:  5d                    pop    %rbp
ffffffff81044222:  48 98                 cltq
ffffffff81044224:  c3                    retq
ffffffff81044225:  48 c7 c7 13 53 98 81 mov    $0xffffffff81985313,%rdi
ffffffff8104422c:  31 c0                 xor    %eax,%eax
ffffffff8104422e:  e8 60 0f 6d 00        callq  <printk>
ffffffff81044233:  eb c9                 jmp    <sys_getppid+0xe>
ffffffff81044235:  66 66 2e 0f 1f 84 00 data32 nopw %cs:0x0(%rax,%rax,1)
ffffffff8104423c:  00 00 00 00
```

Листинг 18 – Результат компиляции функции getppid без static key.

Технология static keys похожа на разработанную в результате данного исследования MOD_CODE. При этом у MOD_CODE есть несомненные преимущества. Во-первых, разработанная технология не требует наличие глобальных переменных. Во-вторых, MOD_CODE в отличие от static keys является незави-

симой от компилятора. Наконец, при использовании технологии MOD_CODE неисполняемые инструкции переносятся в отдельную секцию, что исключает их попадание в КЭШ инструкций при неактивных скрытых участках кода. Static keys же оставляет их в той же секции, а потому неисполняемые инструкции будут занимать место в КЭШе.

3.2 KProbes

KProbes — технология, разработанная компанией IBM® и предназначенная для отладки программ и сбора информации об их производительности. Данная технология позволяет указать любой адрес в виртуальном пространстве ядра, до или после которого будет собираться статистика или выполняться отладочные действия [16, 17]. Структура *kprobe* для ядра ОС Linux 2.6.11.7 представлена в листинге 19

```
struct kprobe {
    struct hlist_node hlist;
    kprobe_opcode_t addr;              /* Address of probe */
    kprobe_pre_handler_t pre_handler;  /* Address of pre-handler */
    kprobe_post_handler_t post_handler;/* Address of post-handler */
    kprobe_fault_handler_t fault_handler;
    kprobe_break_handler_t break_handler;
    kprobe_opcode_t opcode;
    kprobe_opcode_t insn[MAX_INSN_SIZE];
};
```

Листинг 19 – Структура kprobe для ядра ОС Linux 2.6.11.7

На основе KProbes была разработана другая технология DProbes, интересной особенностью, которой является возможность исполнения скриптов по указанным адресам. Таким образом, DProbes от части похоже на разработанную в результате данного исследования технологию MOD_SCRIPT, за исключением принципов работы.

KProbes работает через прерывания. Когда регистрируют новый адрес, по нему помещается прерывание int3. При достижении потоком исполнения данного места срабатывает прерывание и его обработчик проверяет зарегистрированные KProbes для данного адреса, после чего передаёт управление на определённые пользователем функции.

MOD_SCRIPT, как и MOD_CODE работают без прерываний по достижении необходимых адресов, потому если необходимо выполнение дополнитель-

ных скриптов или скрытых участков кода, то необходимо вызвать функцию-обработчик точек модификации. За счёт этого достигается максимальная производительность, когда точки не модифицированы и после модификации, потому что нет прерываний. В то же время есть необходимость выполнения дополнительных действий, что отлично подходит для гипервизора, так как скрытая функциональность необходима только в определённые моменты-события, которые известны гипервизору, когда они происходят, выполнение всё равно прерывается, а потому затраты на расширение функциональности менее критичны.

3.3 Ftrace

Ftrace — трассировщик, предназначенный для помощи разработчикам в изучении происходящего внутри ядра. Ftrace можно использовать как для отладки, так и для получения информации о latency[12] [18].

Для того чтобы ftrace работал необходимо компилировать проект с флагом *-pg*, который разместит вызов функции *mcount()* вначале каждой функции. Именно в этой функции и будут осуществляться все действия по трассировке функций.

Очевидными недостатками такого подхода являются зависимость от компилятора и постоянное исполнение инструкции *call* в начале каждой функции вне зависимости от необходимости. Разработанные технологии MOD_CODE и MOD_SCRIPT лишены подобных недостатков.

3.4 Ptrace

Ptrace — системный вызов Unix-подобных операционных систем, предназначенный для изучения и модификации одним процессом состояния другого [19].

На ОС Gnu/Linux (ядро v4.16.12) разрешить отслеживание и изменение состояние одного процесса другим можно двумя способами. Первый — вызвать со стороны отслеживаемого процесса *ptrace* с флагом PTRACE_TRACEME, что разрешит родителю следить за ребёнком. После этого обычно следует системный вызов *execve*. Если процесс является отслеживаемым, то после исполнения *execve* он получит сигнал SIGTRAP, что позволит процессу–родителю с

[12]latency — задержка, время между событиями.

помощью вызова *wait* начать безопасно управлять состоянием ребёнка. Простой пример показан в листинге 20.

```
int main(int argc, char* argv[])
{
    int wait_status;
    pid_t child_pid = fork();
    if (child_pid == 0) {
        ptrace(PTRACE_TRACEME, 0, 0, 0);

        char *newargv[] = { NULL, NULL };
        char *newenviron[] = { NULL };
        execve(argv[1], newargv, newenviron);
    } else if (child_pid > 0) {
        wait(&wait_status);

        do_trace(wait_status);
    }

    return 0;
}
```

Листинг 20 – Вызов ptrace с флагом PTRACE_TRACEME

Второй способ — со стороны следящего процесса вызвать *ptrace* с одним из двух флагов PTRACE_ATTACH или PTRACE_SEIZE.

Ptrace лежит в основе некоторых программ, предназначенных для отладки, таких как gdb [20, 21], strace и ltrace [22].

GDB (The GNU Project Debuger) [20, 21] — полновесный отладчик, позволяющий изучать и модифицировать всё внутреннее состояние исследуемой программы, включая регистры, стек, исполняемый код и т.д.

Strace (system calls and signals tracer) — трассировщик ОС Linux, используемый для отслеживания системных вызовов, сигналов и состояния программ. В основе strace текущей версии 4.22 также лежит ptrace, что не сложно проверить, исследовав исходный код в официальном хранилище `https://github.com/strace/strace`. Дальнейшее описание работы strace сделано на основе изучения исходного кода ярда Linux v4.16.12.

Для того чтобы можно было отслеживать системные вызовы сначала необходимо установить связь между наблюдателем и наблюдаемым, как описывалось раньше. После, наблюдатель должен вызвать *ptrace* с флагом PTRACE_SYSCALL, что приведёт к выставлению флага TIF_SYSCALL_TRACE поля *flags* структуры *struct thread_info*, которая хра-

нится в описателе процесса *struct task_struct*.

Любой системный вызов в ядре Linux проходит через прослойку entry.S, написанную на ассемблере; для 64–битных процессоров x86 – */arch/x86/entry/entry_64.S*. Из неё управление передаётся в функцию *do_syscall_64()*, определённую в */arch/x86/entry/common.c*. В функции *do_syscall_64()* проверяется условие _TIF_WORK_SYSCALL_ENTRY представленное в листинге 21.

```
#define _TIF_WORK_SYSCALL_ENTRY \
    (_TIF_SYSCALL_TRACE | _TIF_SYSCALL_EMU | _TIF_SYSCALL_AUDIT |   \
     _TIF_SECCOMP | _TIF_SYSCALL_TRACEPOINT |   \
     _TIF_NOHZ)
```

Листинг 21 – Определение _TIF_WORK_SYSCALL_ENTRY

Если условие выполняется, то происходит длинная цепочка вызовов, представленная на рис. 6. В результате работы этой цепочки отслеживаемый процесс получит сигнал SIGTRAP и будет остановлен, а strace будет уведомлен о системном вызове отслеживаемого процесса.

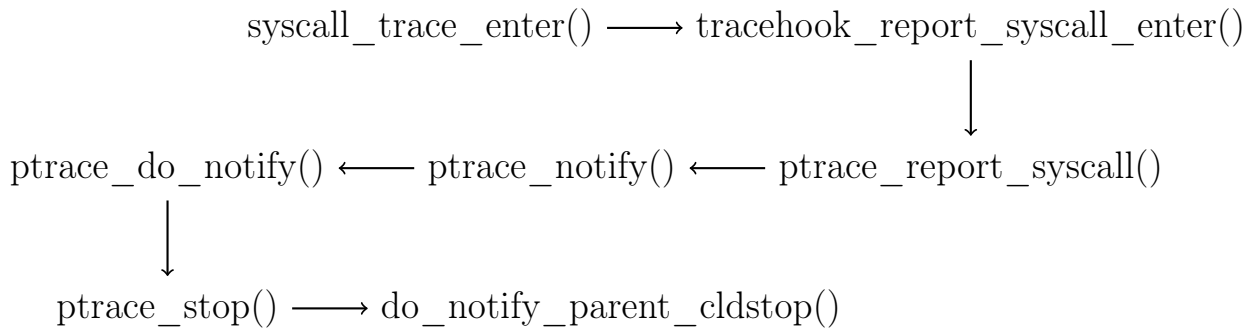

Рисунок 6 – Цепочка вызовов для уведомления strace о системном вызове.

Таким образом, системный вызов *ptrace()* позволяет реализовывать очень сложные технологии для отладки и трассировки, но требуется поддержка операционной системы. Разработанные технологии MOD_CODE и MOD_SCRIPT не требуют подобной поддержки и реализованы без использования прерываний.

4 Описание разработанных технологий динамической модификации кода гипервизора

В данной секции представлено полное описание разработанных технологий MOD_CODE и MOD_SCRIPT, как они устроены и работают.

4.1 MOD_CODE

MOD_CODE (modification code) — технология, предназначенная для модификации компилируемого кода языков C/C++. Цель MOD_CODE — увеличение производительности скомпилированного кода, что достигается за счёт возможности исключения во время компиляции редко используемых участков кода из основного тела программы и восстановления их при необходимости во время исполнения. В частности, для гипервизоров библиотека позволяет исключить код, отвечающий за сбор и вывод диагностики, или реализующий функциональность специфичную для малого числа аппаратных платформ. Это приводит к уменьшению объема основной части исполняемого файла[13], и, как следствие, к уменьшению давления на КЭШ инструкций за счёт того, что неисполняемые инструкции в него не попадают.

Так как большинство таких участков кода находятся внутри условных конструкций, то технология MOD_CODE также позволяет избежать избыточного условного ветвления потока исполняемых команд, что должно улучшить работу предсказателя условных переходов.

Структура, описывающая точку модификации представлена в листинге 22, а реализация макроса _MOD_CODE для исключения участков кода и сохранения информации о них на этапе компиляции в листинге 23 . Описание структуры будет разбито на две части: сначала будут разобраны основные поля структуры, а потом те, что были добавлены для модификации технологии.

Итак, технология исключает часть кода из основной программы, потому необходимо сохранить информацию об адресе для модификации, за это отвечает первое поле — *insert_point*. Поля *start* и *end* содержат виртуальные адреса начала и конца перенесённых ассемблерных инструкций.

[13]Для Linux – секция .text, для Mac OS X – __TEXT,__text.

```
typedef struct mod_code {
    uint8_t         *insert_point;
    uint8_t         *start;
    uint8_t         *end;
    uint64_t        flag;
    MOD_STATE       state;
    MOD_CODE_TYPE   type;
    uint8_t         ins_code_size;  /* size of code from @insert_point, before
   relocation */
    uint8_t         stub_size;      /* size of relocated code from
   @insert_point */
} PACKED mod_code;
```

Листинг 22 – Структура описывающая точку модификации

Макрос _MOD_CODE оставляет на месте перенесённого кода команды *NOP* с общей длинной 14 байт, чтобы уместился прыжок в любую точку 64-битного адресного пространства (листинг 24). После чего сохраняет информацию о точке модификации, формируя структуру *mod_code*. Все структуры *mod_code* размещаются в отдельной секции, что достигается за счёт директив этапа компиляции *.pushsection*[14] и *.popsection*[15]. Затем с помощью тех же инструкций *.pushsection* и *.popsection* код переносится в отдельную секцию.

Названия секций и флаги для них спрятаны внутри макросов: MC_STRUCT_SEGSECT, MC_STRUCT_FLAGS, MOD_CODE_SEGSECT и MOD_CODE_SECT_FLAGS, потому что у разных бинарных форматов поразному представляются названия секций и выглядят их флаги.

Для контроля секции, в которой компилятор разместит код, можно также использовать директиву *.section* вместо пары *.pushsection* и *.popsection*. Но этот подход менее универсален, потому что команда *.section* требует указания названия новой секции и информацию о предыдущей нигде не сохраняет. Таким образом, *.pushsection* и *.popsection* позволяют использовать технологию в произвольной секции, в том числе созданной пользователем.

На этом, основная идея технологии MOD_CODE заканчивается и начинается описание улучшений, которые технология претерпела в процессе разработки, и соответствующих полей структуры *mod_code*.

[14].pushsection — ассемблерная директива, которая говорит компилятору сохранить текущую секцию на стеке, а все последующие команды располагать в новой секции, указанной после директивы.

[15].popsection — ассемблерная директива, которая говорит компилятору достать из стека сохранённую секцию и все последующие инструкции поместить в неё.

```
#define _MOD_CODE(flag, code)                                          \
do {                                                                   \
    asm volatile(                                                      \
        "0:\n"                                                         \
        /* 8 bytes NOP */                                              \
        ".byte 0x0f,0x1f,0x84,0x00,0x00,0x00,0x00,0x00\n"              \
        /* 6 bytes NOP */                                              \
        ".byte 0x66,0x0f,0x1f,0x44,0x00,0x00\n"                        \
        ".pushsection " MC_STRUCT_SEGSECT ", " MC_STRUCT_FLAGS "\n" \
        /* mod_code structure */                                       \
        /* .insert_point */                                            \
        ".quad 0b\n"                                                   \
        /* .start */                                                   \
        ".quad 1f\n"                                                   \
        /* .end */                                                     \
        ".quad 2f\n"                                                   \
        /* .flag */                                                    \
        ".quad "_STRINGIZE(flag)"\n"                                   \
        /* .state */                                                   \
        ".byte 0x00\n"                                                 \
        /* .type */                                                    \
        ".byte 0x00\n"                                                 \
        /* .ins_code_size */                                           \
        ".byte 0x00\n"                                                 \
        /* .stub_size */                                               \
        ".byte 0x00\n"                                                 \
        ".popsection\n"                                                \
        ".pushsection" MOD_CODE_SEGSECT", "MOD_CODE_SECT_FLAGS "\n" \
        "1:\n"                                                         \
        ::: "memory");                                                 \
    do {                                                               \
        code                                                           \
    } while(0);                                                        \
    asm volatile(                                                      \
        "2:\n"                                                         \
        /* jmp QWORD PTR [rip] */                                      \
        ".byte 0xFF,0x25,0x00,0x00,0x00,0x00\n"                        \
        /* (need to update in process_mod_code function) */            \
        /* address - need to place absolute address for jmp */         \
        ".byte 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00\n"              \
        ".popsection\n"::: "memory");                                  \
} while(0);
```

Листинг 23 – Внутреннее представление MOD_CODE

```
asm volatile(
    /* jmp QWORD PTR [RIP] */
    ".byte 0xFF, 0x25, 0x00, 0x00, 0x00, 0x00\n"
    /* addr */
    ".byte 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00\n"
    ::: "memory");
```

Листинг 24 – Прыжок в любую точку 64-битного адресного пространства

Во-первых, все участки кода, к которым может быть применена технология, могут быть поделены на отдельные группы, например: отладочная информация, вывод логов, платформозависимая функциональность и т.д. Очевидно, что при необходимости получить отладочную информацию, скорее всего, нет необходимости восстанавливать платформозависимую функциональность и наоборот. Для того чтобы можно было обрабатывать точки модификации по группам, а не все сразу, было добавлено 64–битное поле *flags*.

Во-вторых, одна и та же точка может относиться одновременно к нескольким различным группам, потому может возникнуть ситуация её повторной модификации. В данной ситуации разработанная библиотека отрабатывает корректно, и это не приводит к экстренному прекращению исполнения программы. Однако, чтобы ускорить обработку точек модификации, было добавлено поле state в структуру *mod_code*, которая содержит информацию о том модифицирована точка или нет. Соответственно, если точка уже модифицирована, то никаких действий библиотека не предпринимает.

В-третьих, первоначальная версия макроса _MOD_CODE, как видно из листинга 23, с помощью *NOP* резервировала место под инструкции для восстановления кода, а это лишние инструкции для исполнения, которые к тому же занимают место в секции .text. Соответственно, последнее улучшение, которое было внесено в технологию — возможность модификации произвольного участка кода без *NOP*. Данная модификация полностью описана в секции 4.3, здесь же будет описание последних трёх полей структуры *mod_code*: *type*, *ins_code_size* и *stub_size*.

Поле *type* собственно определяет _MOD_CODE с *NOP*-ами или без. Если без *NOP*, то обрабатывать точку модификации нужно по другому алгоритму: нужно вычислить сколько и какой длины инструкции нужно перенести из адреса, на который указывает поле *insert_point*, чтобы было достаточно места для прыжка до перенесённого кода.

Размер переносимого кода сохраняется в поле *ins_code_size*, а размер данного отреалоцированного кода будет лежать в *stub_size*. Это позволяет на этапе исполнения не вызывать декодер, а всю работу свести к копированию заранее подготовленного кода известного размера.

Для ускорения исполнения был добавлен новый этап сборки исполняемого файлу — post_process. Он обрабатывает итоговый объектный и исполняемый файл, заранее вычисляя и модифицируя поля *ins_code_size* и *stub_size*, а так же сохраняя отреалоцированный код из точки модификации и код без изменений в секции с перенесённым кодом. Иными словами post_process переформировывается секцию. Благодаря этому модификация на этапе исполнения представляет из себя простое копирование без необходимости вызывать декодер иструкций. Более подробное описание post_process находится в секции 4.4.

4.2 Перемещение инструкций архитектуры Intel® 64

Для работы технологий MOD_CODE и MOD_SCRIPT необходима возможность перемещения инструкций архитектуры Intel® 64 с одного адреса виртуального пространства на другой. Перемещение участков кода содержит в себе некоторые сложности, которые делают данную работу нетривиальной, потому для этой задачи была написана отдельная подбиблиотека. Однако, сначала стоит разобрать структуру команд архитектуры Intel® 64 для дальнейшего понимания проблем и их решения.

4.2.1 Структура команды архитектуры Intel® 64

Общий вид структуры команды архитектуры Intel® 64 [3] представлен на рис. 7. Максимальная длина инструкций в любом режиме и для любого расширения ограничена 15 байтами.

Сначала опционально идут префиксы. Есть несколько видов префиксов. Во-первых, так называемые legacy префиксы, длина каждого из них 1 байт. Это префиксы существующие с самого начала истории архитектуры Intel® IA-32. У инструкции может быть до 4 legacy префиксов, каждый из которых опционален:

- Lock и Repeat префикс.
- Префикс изменения сегмента.

- Префикс смены режима адресации.
- Префикс изменения размера операнда.

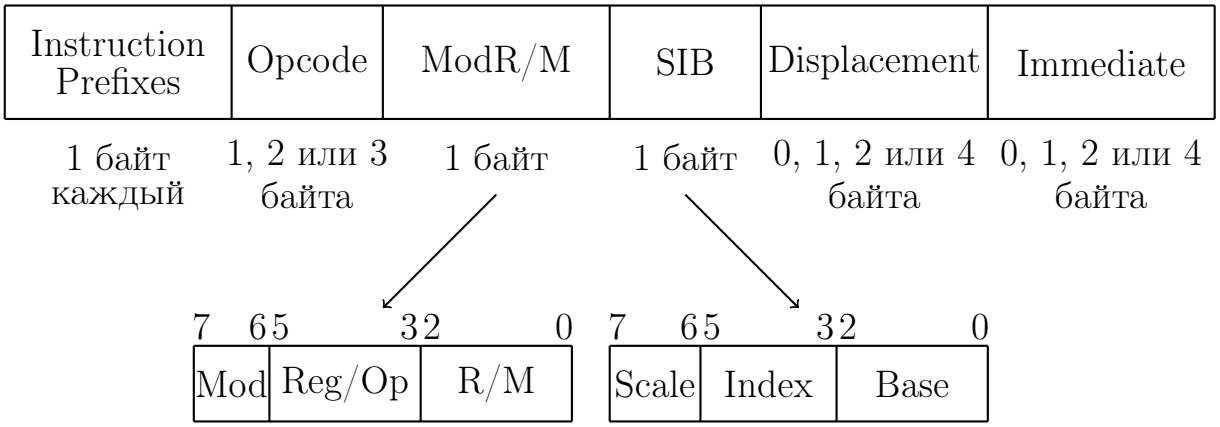

Рисунок 7 – Структура команды архитектуры Intel® 64.

Во-вторых, REX префикс, предназначенный для работы в 64 битном режиме. Он расширяет операнды или поля Opcode, ModR/M, SIB. REX префикс всегда должен идти перед полем Opcode инструкции, а его длина составляет 1 байт.

В-третьих, VEX — префикс векторного расширения архитектуры Intel® 64. Длина VEX префикса может быть от 2 до 3 байт, размещается так же, как и REX префикс перед полем Opcode. С введением векторного расширения Intel® AVX-512 также появился новый префикс — EVEX длинной 4 байта.

Opcode — поле, определяющее тип команды, может быть длинной 1, 2 или 3 байта. Иногда дополнительные 3 бита содержаться в поле Reg/Opcode ModR/M байта. Длина Opcode равна 3 байтас, когда первый байт равен 0x0F, а второй 0x38 или 0x3A. 2 байта, когда первый байт равен 0x0F.

ModR/M — байт описывающий операнды инструкции. Поле Mod вместе с r/m определяют адресацию. Поле Reg/Opcode содержит либо дополнительные 3 бита для Opcode, либо номер одного из восьми регистров общего назначения — операнд инструкции. Некоторым комбинациям полей Mod и r/m необходим ешё один байт для адресации — SIB.

Последние два поля Displacement и Immediate — опциональные константы размером 1,2 или 4 байта, наличие и размер которых определяют Opcode, ModR/M и SIB.

4.2.2 Режимы адресации в Intel® 64

В Intel® IA-32 было два режима адресации: абсолютный и относительный. Как упоминалось ранее, это определялось по полям Opcode, ModR/M и SIB. Относительная адресация была доступна различным инструкциям управления потоком команд, таким как *call* и *jmp*. Итоговый адрес вычисляется как *EIP + disp*, где *EIP* — адрес следующей инструкции, при этом *EIP* участвует в вычислении адреса неявно.

В Intel® 64 добавился новый режим адресации — RIP-relative. Если mod = 0 и r/m = 101, то итоговый адрес операнда инструкции вычисляется, как *RIP + disp32*, где *RIP* — адрес следующей инструкции, а disp32 — 32-битное смещение. В 64-битной архитектуре максимальный размер Displacement остаётся 32 бита. RIP-relative адресация может быть практически в любой инструкции, а не только в командах управления потоком команд, при этом регистр *RIP* участвует в вычислении адреса явно.

4.2.3 Проблемы при перемещении x86 кода

После разбора структуры команд и типов адресации можно приступить к разбору сложностей при переносе x86 кода.

Первая сложность — относительная адресация. Во-первых, при переносе кода внутри него могут быть инструкции с относительной, или RIP-relative адресацией. Во-вторых, внутри него могут быть инструкции — цели для команд с относительной адресацией. Надо заметить, что есть команды передачи управления по абсолютным адресам, но сейчас на Intel® 64 большинство программ по умолчанию компилируются в так называемом PIC (position independent code) коде, в котором применяется только относительная адресация, чтобы можно было применять метод рандомизации адресного пространства (*англ.* address space layout randomization ASLR). Также на Intel® 64 PIC код получается меньше в размере и менее зависимым от регистров, чем код с абсолютной адресацией.

Вторая сложность — переменная длина инструкций. Одной и той по смыслу инструкции может соответствовать несколько разных Opcode и, как следствие, у них будут разные длины. Например, Opcode равный 0xE9 соответствует инструкции *JMP rel32*, а 0xEB *JMP rel8*.

Третья сложность — ограниченный размер поля Displacement. Это проблема, можно сказать, является комбинацией предыдущих двух проблем. Во-первых, у инструкций с RIP-relative адресацией смещение всегда 32-битное, а адресное пространство 64-битное. Во-вторых, какая-нибудь из переносимых инструкций может быть целью для инструкции с относительной адресацией, например, для *JNE rel8*. Длина инструкции — 2 байт: 1 байт — Opcode, и 1 байт — Displacement. Таким образом, при переносе кода в 64-битном адресном пространстве очень легко выйти за пределы Displacement.

4.2.4 Решение проблем при перемещении x86 кода

Для решения проблем, возникающих при перемещении кода архитектуры Intel® 64, рассмотрим пример переносимого кода, представленного на рис. 8. Как видно, *instr1* и *instr2* являются целями для инструкций передачи управления. Задача — перенести инструкции *instr0*, *instr1* и *instr2* с сохранением работоспособности всего кода.

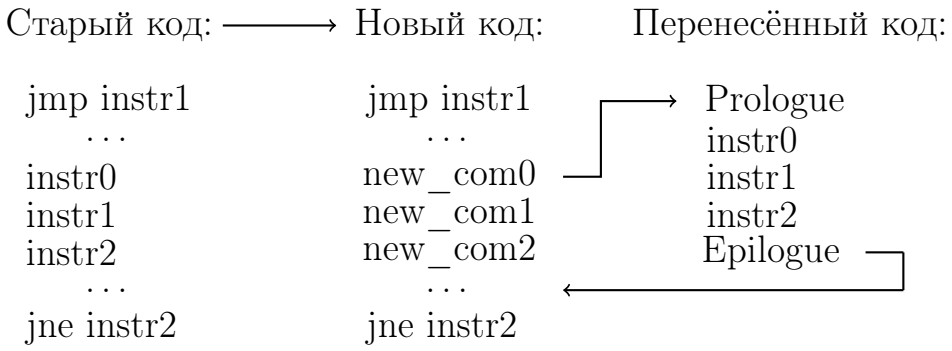

Рисунок 8 – Пример переноса кода Intel® 64.

Для выполнения этой задачи необходимо сформировать последовательность инструкций, которая передаст управление в перенесённый код вместе с информацией о том, какая инструкция должна быть выполнена. Это можно осуществить с помощью помещения на стек необходимой информации и инструкции прыжка, или команды вызова функции, так как она осуществляет все необходимые действия, при этом итоговый размер последовательности получается меньшего размера.

В работе при переносе кода применялось два способа вызвать функцию (листинг 25). Первый — 5-байтовый вызов с относительной адресацией. Opcode такой инструкции равен 0xE8, после него идёт 32-битное смещение. Такой вызов

применяется, если 32-битного смещения достаточно, что в 64-битном адресном пространстве не всегда верно.

```
/* 32-bit relative addr call */
asm volatile(
    /* call rel32 */
    ".byte 0xE8, 0x00, 0x00, 0x00, 0x00\n"
    ::: "memory");

/* 64-bit absolute addr call */
asm volatile(
    /* call QWORD PTR [RIP] */
    ".byte 0xFF, 0x15, 0x00, 0x00, 0x00, 0x00\n"
    /* addr */
    ".byte 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00\n"
    ::: "memory");
```

Листинг 25 – Применяемые вызовы функции при переносе кода.

В случаях, когда 32-битного смещения недостаточно используется второй способ вызвать функцию. Это вызов функции по абсолютному адресу, расположенному в памяти. Таким образом, можно преобразовать рис. 8 в рис. 9 или в рис. 10.

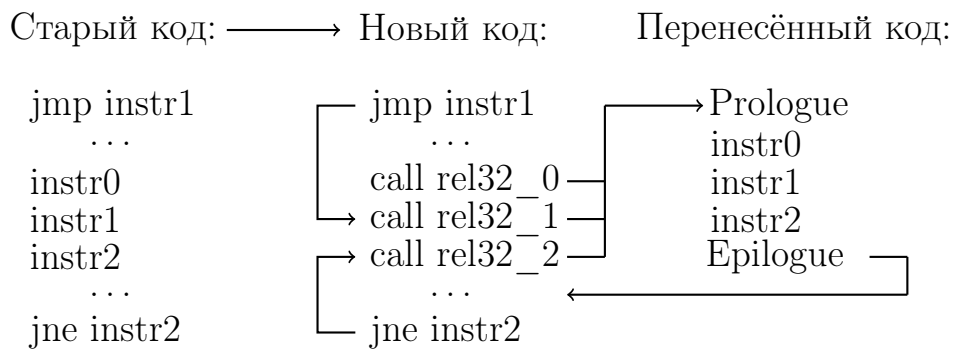

Рисунок 9 – Первый вариант модификации примера переноса кода Intel® 64.

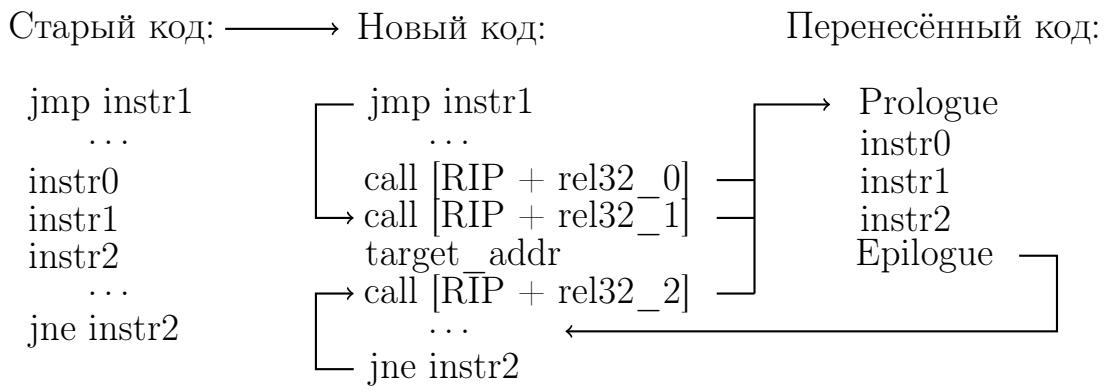

Рисунок 10 – Второй вариант модификации примера переноса кода Intel® 64.

На рис. 9 показана замена переносимых инструкций на вызовы функций с относительной адресацией. Инструкцию *jmp instr1* необходимо модифицировать, чтобы она передавала управление на *call rel32_1*, а *jne instr2 – call rel32_2*. Тогда при передачи управления на перемещённые инструкции на вершине стека будет расположен адрес возврата, который будет проанализирован в прологе перед исполнением перенесённого кода. По адресу, расположенному на стеке можно понять с какой инструкции начинать исполнение в перенесённом коде.

На рис. 10 изображена замена переносимых инструкций на вызов функций по абсолютному адресу, записанному в памяти процесса. В данном случае инструкция *call [RIP + rel32]* — пример RIP-relative команды. Различные rel32_0, rel32_1 и rel32_2 равны смещению от конца соответствующих инструкций до абсолютного адреса перенесённых команд — *target_addr*.

В разделе 4.2.3 упоминалось, что перемещаемые инструкции могут быть либо с относительной адресацией, либо RIP-relative. Значит, пре переносе их нужно модифицировать.

Как упоминалось ранее, инструкции с относительной адресацией это различные *jmp*, *jcc*, и *call*. Если при переносе команды новое смещение не умещается в поле Displacement, то она заменяется на соответствующий *stub* — новая последовательность инструкций.

В Intel® 64 доступно два варианта инструкции *jmp* с относительной адресацией: *jmp rel8* и *jmp rel32*. Для *jmp rel8* роль *stub* может выполнять *jmp rel32*, если 32-битного смещения достаточно, если нет, то для каждого из них подойдёт *stub* показанный в листинге 24. Все jmp_stub представлены в листинге 26.

```
asm volatile(
    /* jmp rel32 */
    ".byte 0xE9, 0x00, 0x00, 0x00, 0x00\n"
    ::: "memory");

asm volatile(
    /* jmp QWORD PTR [RIP] */
    ".byte 0xFF, 0x25, 0x00, 0x00, 0x00, 0x00\n"
    /* addr */
    ".byte 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00\n"
    ::: "memory");
```

Листинг 26 – Все jmp_stub, применяемые при перемещении кода Intel® 64.

Примерно такая же ситуации в Intel® 64 с условными переходами: доступны *jcc rel8* и *jcc rel32*. Однако, инструкций вида *jcc rel8* больше, чем *jcc rel32*. Иными словами, не для каждой команды *jcc rel8* существует соответствующая *jcc rel32*. Потому, если для инструкции *jcc rel8* есть *jcc rel32* и 32–битного смещения достаточно, то *jcc rel32* будет *stub*. Если не существует соответствующей команды *jcc rel32* и 32-битного смещения достаточно, то можно использовать второй *jcc_stub*, представленный в листинге 27. И, наконец, если недостаточно 32-битного смещения, то используется третий вариант *jcc_stub*, так же представленный в листинге 27.

```
/* first variant */
        jcc rel32


/* second variant */
        jcc IF_CLAUSE
        jmp ELSE_CLAUSE
IF_CLAUSE:
        jmp rel32
ELSE_CLAUSE:


/* third variant */
        jcc IF_CLAUSE
        jmp ELSE_CLAUSE
IF_CLAUSE:
        jmp QWORD PTR [RIP]
        jmp_addr
ELSE_CLAUSE:
```

Листинг 27 – Варианты jcc_stub для инструкций *jcc rel8*.

В архитектуре Intel® 64 валидной является только одна версия команды *call* с относительной адресацией — *call rel32*. Если 32-битного смещения недостаточно, то применяется call_stub, показанный в листинге 28.

```
/* 64-bit absolute addr call */
asm volatile(
    /* call QWORD PTR [RIP] */
    ".byte 0xFF, 0x15, 0x00, 0x00, 0x00, 0x00\n"
    /* absolute addr */
    ".byte 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00\n"
    ::: "memory");
```

Листинг 28 – Call_stub для 64 битного адресного пространства.

В разделе 4.2.2 также упоминалось о новом типе адресации — RIP-relative, добавленном при переходе от Intel® IA-32 к Intel® 64. Такая адресация может применяться практически в любой машинной команде, а определяется она ModR/M байтом [3]. Итоговый адрес вычисляется, *RIP + disp32*, где *RIP* — адрес следующей инструкции, а disp32 — 32–битное смещение. Соответственно, при перемещении инструкции, если 32–битного смещения достаточно, то просто изменяется поле Displacement. Если нет, то в библиотеке используется специальный riprel_stub (листинг 29).

```
push rsi/rdi
movabs rsi/rdi, abs_addr
RIP-rel_instruction
pop rsi/rdi
```

Листинг 29 – Stub для RIP-relative инструкций Intel® 64.

В *riprel_stub* на месте RIP-rel_instruction располагается специально модифицированная RIP-relative инструкция. RIP-relative адресация, как упоминалась ранее, применяется если в ModR/M байте поле mod равно 00, а r/m равно 101 в двоичной системе счисления. Идея для модификации машинной команды проста — изменить ModR/M байт так, чтобы инструкция брала абсолютный адрес операнда в регистре, который к тому же не может использоваться неявно. Такими регистрами могут быть *RSI* и *RDI*. Соответственно, если RIP-relative инструкция не использует регистр *RSI*, что можно узначить проанализировав поле reg ModR/M байта, то абсолютный адрес операнда помещается в него, в противном случае в регистр *RDI* (рис. 11).

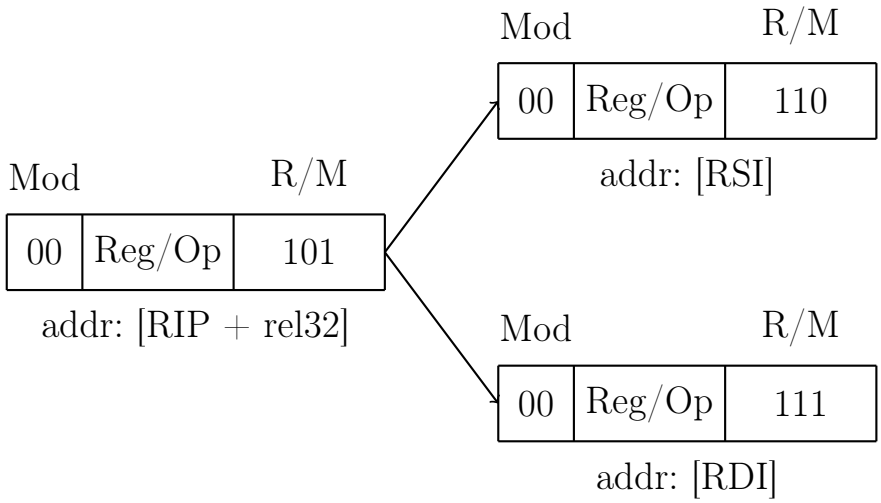

Рисунок 11 – Модификации ModR/M байта RIP-relative инструкции.

После изменения ModR/M байта остаётся только убрать поле Displacement из инструкции, а остальные поля сохранить. У RIP-relative инструкций нет SIB байта, потому его модифицировать или удалять не надо [3].

4.3 Модификация произвольных участков в технологии MOD_CODE

В разделе 4.1 упоминалось, что текущая версия технологии MOD_CODE не резервирует место для прыжка из точки модификации до перенесённого кода с помощью инструкций NOP. Это было достигнуто за счёт библиотеки по перемещению x86 кода, описанной в секции 4.2, и библиотеки для обработки скомпилированной программы post_process, представленной в секции 4.4.

Непосредственно после компиляции всех исходных файлов и линковки их в один большой объектный (релоцируемый) файл можно оценить сколько дополнительного места необходимо для сохранения дополнительной информации. Post_process делает оценку и расширяет секцию, в которой расположен перенесённый код, потому что расширение секции возможно только на этапе объектного файла.

После этого происходит окончательная сборка: итоговый объектный файл линкуется со всеми необходимыми библиотеками, и формируется исполняемый файл. В исполняемом файле уже есть вся необходимая для модификации информация: начало и конец функций и цели для всех инструкций передачи управления. При этом так же известно расположение каждой секции относительно друг друга, потому что их взаимное размещение не должно быть в будущем изменено, чтобы работоспособность PIC кода не была нарушена. А все программы для архитектуры Intel® 64 компилируются в PIC код, что так же ранее обсуждалось в секции 4.2.3.

До работы post_process (рис. 12) секция .mc_struct хранит структуры *struct mod_code*, а .mod_code – последовательность участков кода, перенесённых с помощью макроса _MOD_CODE кодов, и инструкций для возврата управления. Поле *insert_point* хранит виртуальный адрес точки модификации, *start* – начало скрытого кода, а *end* – конец.

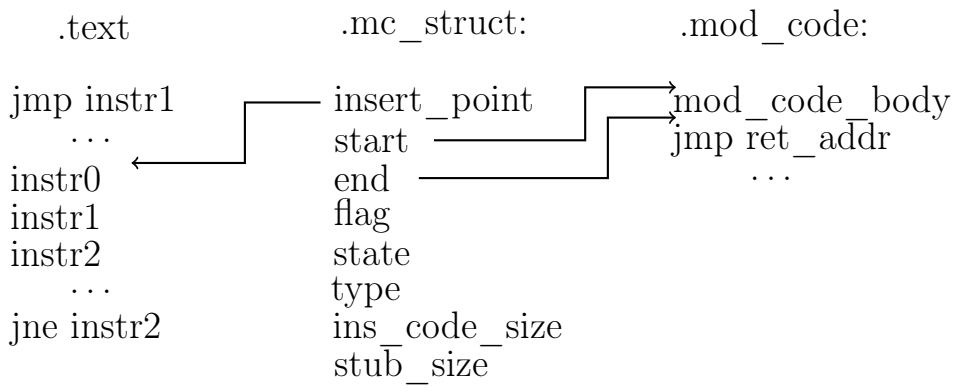

Рисунок 12 – Программа до обработки post_process.

По информации, которую можно получить из исполняемого файла, post_process переформировывает секцию .mod_code и модифицирует соответствующие поля структуры в секции .mc_struct (рис. 13). Post_process оценивает сколько инструкций необходимо перенести с из модифицируемой точки, чтобы уместился stub, передающий управление на скрытую в секции .mod_code функциональность. После этого заново формирует секцию: сначала располагает пролог, потом скрытую функциональность, далее отреалоцированные инструкции, эпилог, команду возврата, сформированный stub и неизменённые инструкции.

Библиотека сохраняет оригинальные инструкции и stub, чтобы на этапе исполнения не вызывать декодер. Когда необходимо восстановить скрытую функциональность, происходит копирование stub по адресу модификации (рис. 14), когда нужно скрыть, копируются оригинальные инструкции. Размер stub сохраняется в поле *stub_size*, неизменённых команд в *ins_code_size*.

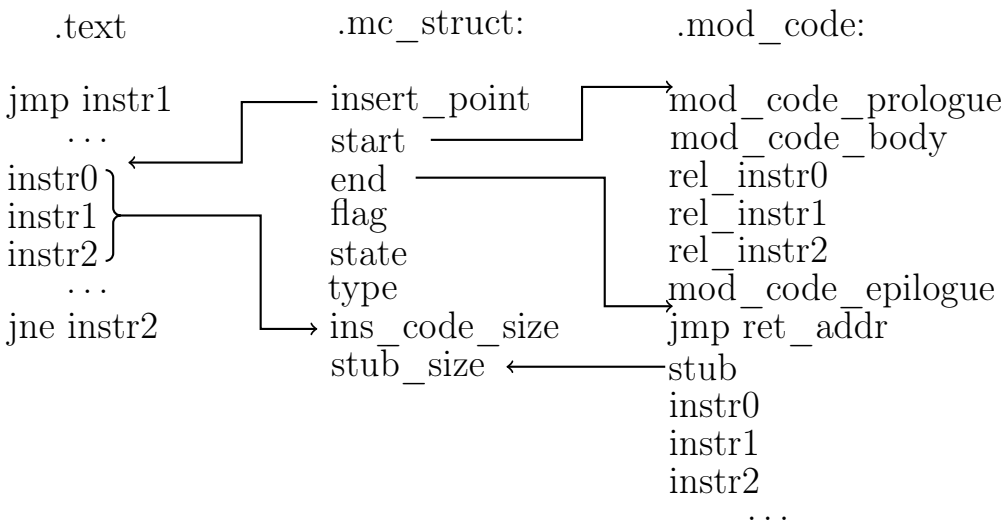

Рисунок 13 – Программа после обработки post_process.

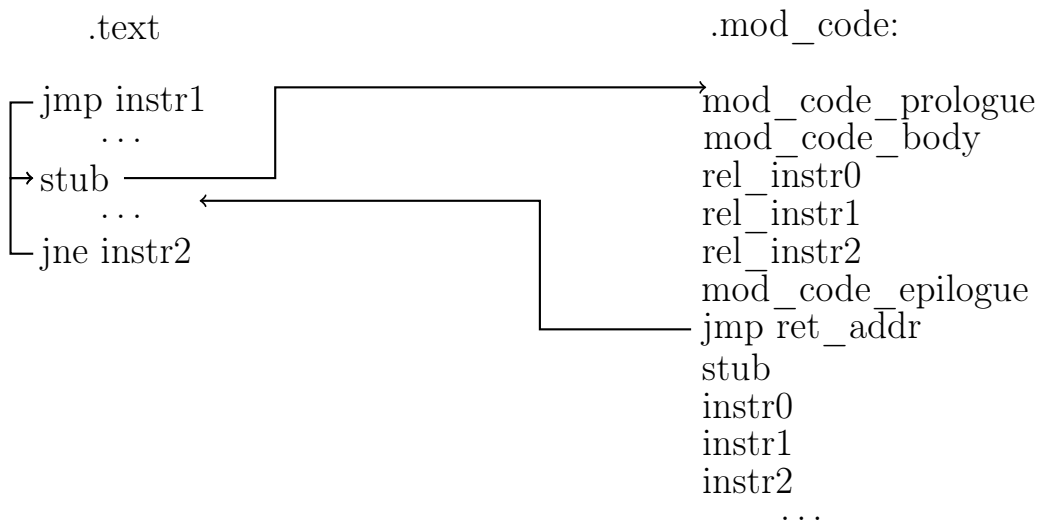

Рисунок 14 – Восстановление скрытой функциональности.

Если среди инструкций, которые заменяются на stub, были цели для команд передачи управления, таких как *jmp* или *call*, то роль stub выполняют последовательности вызовов функций, которые были разобраны в секции 4.2.4 и представлены на рис. 9 и рис. 10. При этом, необходимо изменить инструкции, чтобы они передавали управление на соответствующие команды *call*, ведущие к восстанавливаемому коду и сохраняющие информацию (адрес возврата) на стеке, которая будет проанализирована в прологе.

4.4 post_process

Post_process — утилита, разработанная для модификации бинарных файлов. Цели данных оптимизаций — ускорение работы технологии MOD_CODE и подготовка файла к исполнению в изолированном пространстве. Надо отметить, что основную работу по обеспечению функционирования разработанной технологии осуществляет компилятор, поэтому и без применения post_process всё будет работать.

Первый этап работы post_process — обработка итогового объектного (релоцируемый, ELF) файла. В итоговом объектном файле есть информация обо всех модифицируемых точка. Если в точке применяется технология с резервированием места с помощью команд *NOP* под инструкции перехода, то ничего дополнительного для точки не происходит. Если же без резервирования пространства, то post_process оценивает, сколько и какой длины инструкции нужно перенести, нужен ли пролог и эпилог для перенесённого кода, если да, то вы-

числяет их размер. После оценки необходимого дополнительного пространства post_process расширяет секцию, хранящую скрытый код, на данное значение.

Надо заметить, что только в объектном файле можно без проблем расширить секцию, потому данное действие выполняется на этом этапе.

После модификации объектного файла, происходит линковка со всеми необходимыми библиотеками и создание исполняемого файла. Исполняемый файл уже хранит всю информацию необходимую для конечной модификации секций .mod_code и .mc_struct, которые хранят скрытый код и информацию для модификации соответственно.

4.5 MOD_SCRIPT

MOD_ SCRIPT — вторая технология динамической модификации кода, разработанная в ходе данного исследования. MOD_ SCRIPT создана для расширения функциональности скомпилированного кода языков C/C++. Она позволяет во время работы вставить по определённому адресу исполнение скриптов, написанных на языке lua.

MOD_SCRIPT разрабатывался с целью улучшения поддержки монитора виртуальных машин и упрощения процесса тестирования. Возможность исполнения скриптов по произвольным позволяет достигнуть обе цели. Во-первых, благодаря этому можно выпускать быстрые исправления для конкретного клиента до глобальных обновлений продукта. Во-вторых, технология упрощает процесс автоматизации тестирования, благодаря дополнительным возможностям модификации исполнения программы без внесения новых участков кода в продукт.

MOD_SCRIPT работает за счёт тех же библиотек по перемещению (секция 4.2) и модификации(секция 4.3) кода, что и MOD_CODE. Однако, в отличие от MOD_CODE технология MOD_ SCRIPT работает только на этапе исполнения, поэтому стадия post_process (секция 4.4) недоступна. Соответственно отсутствует доступ к информации о структуре исполняемого файла и, в частности, к информации о начале и конце функции, которой принадлежит модифицируемый виртуальный адрес. Поэтому функция, модифицирующая исполняемый процесс, помимо виртуального адреса и скрипта принимает также начало и размер модифицируемой функции. Данные о модифицируемой функции используются для определения инструкций передачи управления, целью

которых являются команды, которые переносятся для размещения на их месте stub, как и в MOD_CODE.

Скриптовым языком для MOD_SCRIPT был выбран lua, потому что он является мощным и легковесным языком, который к тому же имеет API для взаимодействия с программами, написанными на языке C [23]. Это позволяет скриптам, написанным на lua, использовать функции и структуры данных монитора.

5 Решённые проблемы технологии динамической модификации кода

Технологии, представленные в секции 4, являются результатом доработки изначально созданных прототипов. По мере совершенствования первых вариантов решались различные проблемы, некоторые из них описаны ниже.

5.1 Неисправность работы линковщика mach-o файлов

В результате применения технологии MOD_CODE была обнаружена неисправность работы линковщиков mach-o файлов. Как стандартная версия линковщика ld для операционной системы Mac OS X (PROGRAM:ld PROJECT:ld64-302.3), так и его реализация lld от проекта llvm неправильно обрабатывают один из типов релокаций, доступный в mach-o файлах.

Бинарный формат mach-o описан в разделе 2.4.2. В листинге 9 представлен описатель секции для 64–битной архитектуры, в нём есть поле *reloff*, которое указывает на расположение релокаций для данной секции относительно начала файла.

Релокации mach-o формата могут описываться структурами двух типов: *relocation_info* и *scattered_relocation_info*. Для архитектуры Intel® 64 используется только первый *relocation_info* (листинг 30).

```
struct relocation_info {
  int32_t   r_address;  /* offset in the section to what is being relocated */
  uint32_t  r_symbolnum:24, /* symbol index if r_extern == 1 or section
                             ordinal if r_extern == 0 */
            r_pcrel:1,  /* was relocated pc relative already */
            r_length:2, /* 0=byte, 1=word, 2=long, 3=quad */
            r_extern:1, /* does not include value of sym referenced */
            r_type:4;   /* if not 0, machine specific relocation type */
};
```

Листинг 30 – Описатель релокации mach-o формата для Intel® 64.

Поле *r_address* хранит смещение относительно начала секции до значения, которое необходимо отреалоцировать. В *r_symbolnum* содержится либо номер секции, если *r_extern* равно нулю, либо символа, если *r_extern* – единице, относительно которого релокация осуществляется. *R_pcrel* указывается на то применяется ли относительная адресация или нет, а *r_length* на размер.

Последнее поле *r_type* описывает тип релокации. Например, для прыжка или вызова функции с 4 байтовым смещением тип равен X86_64_RELOC_BRANCH. Именно для такого типа и была найдена неисправность работы линковщика. Программа для воспроизведения ошибки представлена в листинге 31. Результат компиляции показан в листинге 32.

```
int main () {
    asm volatile(
        "jmp 1f\n"
        ".pushsection __TEXT,__my_sect, regular,pure_instructions\n"
        "1: jmp 2f\n"
        ".popsection\n"
        "2: nop\n":::"memory");

    return 0;
}
```

Листинг 31 – Программа для воспроизведения неисправности линковщика mach-o файлов.

```
_main:
0000000100000f90  55                    pushq   %rbp
0000000100000f91  48 89 e5              movq    %rsp, %rbp
0000000100000f94  c7 45 fc 00 00 00 00  movl    $0x0, -0x4(%rbp)
0000000100000f9b  e9 01 00 00 00        jmp 0x100000fa1
0000000100000fa0  90                    nop
0000000100000fa1  31 c0                 xorl    %eax, %eax
0000000100000fa3  5d                    popq    %rbp
0000000100000fa4  c3                    retq
Contents of (__TEXT,__my_sect) section
0000000100000fa5  e9 f6 ff ff ff        jmp 0x100000fa0
}
```

Листинг 32 – Результат компиляции программы для воспроизведения неисправности линковщика mach-o файлов.

Как видно из листинга в итоговом исполняемом файле прыжок по адресу 0x100000f9b неправильно отреалоцирован, при этом в объектном файле с учётом релокации всё верно (листинг 33). Для этой инструкции используется релокация с полем *r_extern* равным нулю, что говорит о том, что поле *r_symbolnum* содержит номер секции __my_sect, в которой находится цель для прыжка.

Если попробовать использовать линковщик из репозитория проекта llvm ревизии r332669, то линковщик выдаст следующее предупреждение: *"bad relocation (unknown type) in section __TEXT/__text (r_address=c, r_type=2, r_extern=0,*

r_length=2, r_pcrel=1, r_symbolnum=2)”. Таким образом, такой тип релокации считается неправильным.

```
_main:
0000000000000000  55                    pushq   %rbp
0000000000000001  48 89 e5              movq    %rsp, %rbp
0000000000000004  c7 45 fc 00 00 00 00  movl    $_main, -0x4(%rbp)
000000000000000b  e9 05 00 00 00        jmp 0x15
0000000000000010  90  nop
0000000000000011  31 c0                 xorl    %eax, %eax
0000000000000013  5d                    popq    %rbp
0000000000000014  c3                    retq
Contents of (__TEXT,__my_sect) section
0000000000000015  e9 10 00 00 00        jmp _main+16

RELOCATION RECORDS FOR [__text]:
000000000000000c X86_64_RELOC_BRANCH __my_sect

RELOCATION RECORDS FOR [__my_sect]:
0000000000000001 X86_64_RELOC_BRANCH _main
}
```

Листинг 33 – Объектный файл программы для воспроизведения неисправности линковщика mach-o файлов.

При этом исходя из информации, представленной в системных файлах */usr/include/mach-o/reloc.h* и */usr/include/mach-o/x86_64/reloc.h*, такой тип релокации используется только если до целевого адреса в секции не было нелокальных символов. Если есть хотя бы один нелокальный символ, то используется релокация относительно ближайшего такого символа, и *r_extern* будет равен единице. Данный факт очень легко проверяется с помощью небольшой модификации программы из листинга 31. Модификация представляет из себя добавление метки — MY_LABEL в ассемблерный код (листинг 34), которая будет преобразована в нелокальный символ.

```
int main () {
    asm volatile(
        "jmp 1f\n"
        ".pushsection __TEXT,__my_sect, regular,pure_instructions\n"
        "MY_LABEL:"
        "1: jmp 2f\n"
        ".popsection\n"
        "2: nop\n":::"memory");

    return 0;
}
```

Листинг 34 – Обход неисправности линковщика mach-o файлов.

При размещении нелокального символа в секции созданной пользователем, как и было написано в стандарте, линковщик использует релокацию относительно ближайшего подобного символа, что видно из листинга 35. При этом такой тип релокации разрешается линковщиками mach-o файлов корректно, что подтверждает сгенерированный исполняемый файл (листинг 36).

```
_main:
0000000000000000  55                    pushq   %rbp
0000000000000001  48 89 e5              movq    %rsp, %rbp
0000000000000004  c7 45 fc 00 00 00 00  movl    $_main, -0x4(%rbp)
000000000000000b  e9 00 00 00 00        jmp MY_LABEL
0000000000000010  90                    nop
0000000000000011  31 c0                 xorl    %eax, %eax
0000000000000013  5d                    popq    %rbp
0000000000000014  c3                    retq
Contents of (__TEXT,__my_sect) section
MY_LABEL:
0000000000000015  e9 10 00 00 00        jmp _main+16

RELOCATION RECORDS FOR [__text]:
000000000000000c X86_64_RELOC_BRANCH MY_LABEL

RELOCATION RECORDS FOR [__my_sect]:
0000000000000001 X86_64_RELOC_BRANCH _main
```

Листинг 35 – Объектный файл, генерируемый для обхода неисправности линковщика mach-o файлов.

Таким же образом данная проблема решается в технологии MOD_CODE, перед переносимым кодом ставится метка, чтобы линковщик использовал тип релокации относительно нелокального символа, при этом метка генерируется уникальной, чтобы не было ошибки переопределения символов.

```
_main:
0000000100000f90  55                        pushq   %rbp
0000000100000f91  48 89 e5                  movq    %rsp, %rbp
0000000100000f94  c7 45 fc 00 00 00 00      movl    $0x0, -0x4(%rbp)
0000000100000f9b  e9 05 00 00 00            jmp MY_LABEL
0000000100000fa0  90                        nop
0000000100000fa1  31 c0                     xorl    %eax, %eax
0000000100000fa3  5d                        popq    %rbp
0000000100000fa4  c3                        retq
Contents of (__TEXT,__my_sect) section
MY_LABEL:
0000000100000fa5  e9 f6 ff ff ff            jmp 0x100000fa0
```

Листинг 36 – Исполняемый файл, генерируемый для обхода неисправности линковщика mach-o файлов.

5.2 Использование технологии MOD_CODE во встраиваемых функциях

В первоначальных версиях технологии MOD_CODE использовались ассемблерные метки (листинг 37) для сохранения информации о точке модификации, начале и конце переносимого кода.

```
asm volatile ("label:\n" ::: "memory");
```

Листинг 37 – Ассемблерная метка.

У такого подхода есть однозначный минус — невозможность применения технологии во встраиваемых функциях из-за того, что это приводит к ошибке переопределения нелокального символа, потому итоговый вариант MOD_CODE использует локальные метки (листинг 23).

5.3 Выравнивание

Есть множество различных ситуаций, когда компилятор выравнивает адрес инструкции или данных. Это делается для увеличения производительности итоговых исполняемых файлов. Для архитектуры Intel® 64 все подобные моменты описаны в *Intel® 64 and IA-32 Architectures Optimization Reference Manual* [24].

Выравниваются, к примеру, целевые адреса инструкций перехода и вызова функции. Для выравнивания используется ассемблерная директива *.align*,

однако результат её обработки ассемблером зависит от секции, в которой она расположена. Если *.align* встречается в секции кода, например, в .text, то адрес будет выравнен с помощью команды *NOP*. Если в секции данных, например, в .data, то с помощью нулей.

Первые версии MOD_CODE скрывали код в секции .data ELF-файлов или __DATA,__data mach-o-файлов. Что приводило к неправильной генерации кода, когда среди переносимых инструкций были выравниваемые, так как вместо *NOP* появлялись нули.

Для решения данной проблемы было сделано две вещи. Во-первых, создаются новые секции: для ELF-файлов .mod_code, для mach-o-файлов __TEXT,__mod_code. Во вторых при их создании с помощью флагов явно указывается, что секции должны хранить исполняемые инструкции: для ELF – "ax", для mach-o – regular,pure_instructions. Надо заметить, что без флага "a"для ELF-файлов созданная секция не будет загружена в виртуальное пространство процесса. После выполнения данных условий, адреса выравниваются с помощью инструкции *NOP* вместо нулей.

5.4 Проблемы с ключевым словом volatile

В языках C/C++ есть ключевое слово — volatile. Обычно оно применяется к переменным, говоря компилятору, что их значение может меняться, даже если в коде нет явных команд для их модификации. Таким образом, volatile ограничивает модификации, которые компилятор может применить к коду.

Volatile так же может применяться и к ассемблерным вставкам. В этом случае ключевое слово тоже ограничивает оптимизации компилятора, таких как изменение последовательности ассемблерных блоков. Однако, при определённых условиях блоки могут дуплицироваться.

В технологии MOD_CODE (листинг 23) идёт последовательность из ассемблерного и переносимого кода. Второй ассемблерный блок запросто может быть продублирован компилятором, что явно нарушит логику. В этой ситуации программа не будет скомпилирована, а компилятор выдаст сообщение о том, что обнаружена директива *.popsection* без соответствующей *.pushsection*. Чтобы избежать дублирования второго ассемблерного блока необходимо ещё больше ограничить оптимизации для переносимого кода, поместив его в цикл *while*, зависящий от volatile переменной. Пример кода представлен в листине 38.

Так как переменная guard_of_code_sequence является volatile, то компилятор не знает будет ли исполняться участок кода внутри цикла *while*, а значит и не может свободно применять оптимизации. Соответсвенно, если встречается дублирование ассемблерных блоков при компиляции, то надо поместить скрываемый код в макрос LOCK_CODE_SEQUENCE.

```
extern volatile int guard_of_code_sequence;

#define LOCK_CODE_SEQUENCE(code) do {code} while (guard_of_code_sequence);
```

Листинг 38 – Ограничение оптимизаций компилятора для участка кода.

6 Заключение

В результате данной научно–исследовательской работы была разработана кроссплатформенная библиотека для динамической модификации программ, написанных на языках C/C++ и компилируемых для архитектуры Intel® 64. Библиотека позволяет осуществлять модификацию без прерываний во время работы, что существенно ускоряет работу модифицированного кода и расширяет области применения технологии.

Итоговая библиотека содержит в себе две технологии. Первая технология — MOD_CODE, с помощью которой можно на этапе компиляции скрыть редко используемые участки кода, а во время исполнения при необходимости их восстановить. Технология позволяет увеличить производительность за счёт уменьшения давления на КЭШ инструкций, благодаря тому, что неисполняемые инструкции не занимают в нём место. Также технология избавляет код от избыточного условного ветвления.

Важно отметить что основную работу по обеспечению функционирования MOD_CODE осуществляет компилятор, при это сама технология не зависит от компилятора, которым собирают программу.

Вторая технология — MOD_SCRIPT, которая позволяет расширять функциональность готового исполняемого файла, причём без использования прерываний. Расширение функциональности достигается за счёт возможности модификации любого адреса виртуального пространства для исполнения скриптов на языке lua. Это позволяет улучшить поддержку продукта, так как можно выпускать исправления для конкретного пользователя, и улучшить автоматизирование тестов без модификации исходного кода. За счёт встроенного в язык lua интерфейса для работы с языком C, достигается взаимодействие с методами и структурами программы, написанной на C/C++.

Разработанная технология внесена в окружение монитора виртуальных машин Parallels Desktop®, где была подтверждена её полная работоспособность, на данный момент начинается её постепенное коммерческое применение. При этом, хотя технология и разрабатывалась в первую очередь для гипервизора, применять её можно и за его пределами, например, в ядрах операционных систем.

Для работоспособности технологий динамической модификации кода было написано несколько подбиблиотек, каждая из которых может работать са-

мостоятельно. Например, библиотека для разбора бинарных форматов может использоваться в любом проекте отдельно, так как реализует единый интерфейс для самых распространённых бинарных форматов: ELF, mach-o, PE.

Библиотека для перемещения инструкций архитектуры Intel® 64 предоставляет удобный интерфейс и по сути является простой бинарной трансляцией x86 кода в x86. Библиотека также может найти своё применение в обфускации кода.

В ходе исследования было решено множество инженерных задач, которые позволяют динамически и без прерываний быстро модифицировать код. Одной из таких является обнаружение неисправности работы линковщиков бинарных форматов mach-o, а также нахождение простого решения для её обхода.

Литература

[1] *Hongxu Cai, Zhong Shao, Alexander Vaynberg.* Certified Self-Modifying Code. 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, June 2007. С. 66–77.

[2] *Yuichiro Kanzaki, Akito Monden, Masahide Nakamura, Ken-ichi Matsumoto.* Exploiting Self-Modification Mechanism for Program Protection. Computer Software and Applications Conference, 2003. COMPSAC 2003. ISBN 0-7695-2020-0.

[3] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual, 2017.

[4] *Quintero Alejandro, Dagenais Michel, Boyer Francois-Raymond, Gagnon Etienne M.* LOW-IMPACT SYSTEM PERFORMANCE ANALYSIS USING HARDWARE ASSISTED TRACING TECHNIQUES.

[5] *Christoffer Dall Shih-Wei Li Jason Nieh.* Optimizing the Design and Implementation of the Linux ARM Hypervisor, 2017

[6] Parallels International GmbH. Parallels Desktop® 13 for Mac User's Guide, 2017.

[7] *B.Sc. Dominic Fandrey, Prof. Dr. rer. nat. Thomas Fuchß.* Clang/LLVM Maturity Evaluation Report, 2010.

[8] *Popek G.J., Goldberg R.P.* Formal requirements for virtualizable third generation architectures, Communications of the ACM. Vol. 17. 1974, P 412-421.

[9] *Г.С. Речистов, Е.А. Юлюгин, А.А. Иванов, П.А. Шишпор, Н.Н. Щелкунов, Д.А. Гаврилов* Основы программного моделирование ЭВМ 2-е издание, 2013 – 222 с. ISBN 978-5-7417-0444-8

[10] *Chris Lattner, Vikram Adve*LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation.

[11] Clang 7 documentation. URL: `https://clang.llvm.org/docs/index.html`

[12] ELF-64 Object File Format, Version 1.5 Draft 2, 1998.

[13] Common Object File Format

[14] OS X ABI Mach-O File Format Reference, 2009.

[15] *Jason Baron.* Static Keys.

[16] *Sudhanshu Goswami.* An introduction to KProbes. // LWN.net. 2005. URL: `https://lwn.net/Articles/132196/`

[17] *Jim Keniston, Prasanna S Panchamukhi, Masami Hiramatsu.* Kernel Probes (Kprobes). // kernel.org. URL: `https://www.kernel.org/doc/Documentation/kprobes.txt`

[18] *Steven Rostedt.* ftrace - Function Tracer.

[19] Linux man.

[20] *John Gilmore, Cygnus Solutions, Second Edition, Stan Shebs, Cygnus Solutions.* GDB Internals.

[21] *Richard Stallman, Roland Pesch, Stan Shebs, et al.* Debugging with GDB.

[22] *Rodrigo Rubira Branco* Ltrace Internals. Proceedings of the Linux Symposium Volume One. June 2007. C. 41–52.

[23] *Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes.* Lua 5.0 Reference Manual, 2003.

[24] Intel® 64 and IA-32 Architectures Optimization Reference Manual, 2017.