САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра системного программирования

Свидерский Павел Юрьевич

Общий подход к восстановлению адресного пространства процесса из образа памяти ОС Android

Дипломная работа

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., проф. Терехов А.Н

Научный руководитель:
ст. преп. Губанов Ю.А.

Рецензент:
ст. преп. Зеленчук И.В.

Санкт-Петербург
2013

SAINT-PETERSBURG STATE UNIVERSITY
Mathematics & Mechanics Faculty

Software Engineering Chair

Pavel Sviderski

General approach to recovery of process address space using Android OS memory image

Graduation Thesis

Admitted for defence.
Head of the chair:
Dr. Sci, Prof. A.N. Terekhov

Scientific supervisor:
Ass. Prof. Yu.A. Gubanov

Reviewer:
Ass. Prof. I.V. Zelenchuk

Saint-Petersburg
2013

Оглавление

Введение 5

1. Постановка задачи 8

2. Обзор 9

2.1. Методы снятия образа памяти 9

2.1.1. LiME 9

2.2. Существующие решения в области анализа образа памяти 10

2.2.1. Volatility Framework 10

2.2.2. Volatilitux 11

2.3. Механизм виртуальной памяти в архитектуре ARMv6/v7 11

2.3.1. Аппаратное управление памятью 11

2.3.2. Страничная организация виртуальной памяти 14

2.3.3. Таблица страниц 1-го уровня 15

2.3.4. Таблица страниц 2-го уровня 18

2.4. Особенности реализации виртуальной памяти в ядре ОС Android 20

2.4.1. Четырёхуровневая иерархия таблиц страниц 22

2.4.2. Существование двух видов таблиц страниц 2-го уровня 22

3. Архитектура решения 24

3.1. Нахождение процессов 24

3.1.1. Аппаратный и программный подходы 25

3.1.2. Алгоритм нахождения потенциальных PGD 25

3.2. Восстановление адресного пространства процесса 28

3.3. Нахождение структур ядра, описывающих процесс 31

3.3.1. Нахождение процесса swapper 32

3.3.2. Нахождение смещения поля tasks структуры task_struct 33

3.3.3. Нахождение смещения поля mm структуры task_struct 34

3.3.4. Нахождение смещения поля pgd структуры mm_struct 35

3.4. Особенности реализации в Volatility Framework 35

3.4.1. Вывод адреса символа ядра (linux_auto_ksymbol) 36

3.4.2. Вывод списка потенциальных PGD (linux_auto_dtblist) 36

3.4.3. Вывод списка процессов (linux_auto_pslist) 37

3.4.4. Восстановление памяти процесса (linux_auto_dump_map) 38

4. Тестирование реализации 40

Заключение 42

Список литературы **43**

Введение

Цифровой криминалистический анализ (Digital Forensic Analysis) — это комплекс экспертных мероприятий, направленных на сбор, хранение и анализ информации, находящейся на компьютерах и прочих цифровых устройствах. Полученная в результате экспертизы устройства цифровая информация может быть предоставлена в суде как потенциальное доказательство, в случае, если данное устройство имело отношение к рассматриваемому судебному делу. Другой немаловажной целью проведения экспертизы является поиск причин отказа или некорректной работы устройства. Одной из частых причин некорректной работы может являться заражение цифрового устройства вредоносной программой.

В цифровом криминалистическом анализе выделяют отдельную область анализа мобильных цифровых устройств. Обычно под мобильным цифровым устройством понимают мобильный телефон, однако это может быть любое цифровое устройство, который имеет внутреннюю память и способность связи, в том числе планшетные компьютеры, MP3-плееры, электронные книги и устройства GPS. С активным ростом доступности этих мощных устройств увеличилась частота использования их преступниками, в том числе при совершении преступлений. Преступники могут использовать мобильные устройства для мошенничества через интернет и электронную почту, вымогательства с помощью SMS-сообщений, коммуникаций, связанных с торговлей наркотиками и др. Данные, которые хранятся внутри мобильного устройства, могут быть очень полезны аналитику в ходе расследования преступления. В самом деле, мобильные устройства содержат большой объем доказательной информации, которая связана с личностью владельца этого устройства: историю звонков, контакты (телефонная книга) и SMS-сообщения. Современные мобильные устройства — к примеру, смартфоны — могут содержать еще больше информации, полезной для анализа, такой как фотографии, видео, электронные календари и заметки, электронная почта (отправленные и принятые письма), история посещения страниц в Web-браузере, история и журналы различных чатов и приложений для обмена быстрыми сообщениями, история сообщений и контакты в социальных сетях. Широкое распространение мобильных телефонов на потребительском рынке вызвало спрос на цифровую криминалистическую экспертизу мобильных устройств, анализ которых не может быть удовлетворен существующими методами компьютерной экспертизы [1].

Одной из проблем в области криминалистического анализа мобильных устройств является существование на сегодняшний день огромного количества видов и моделей этих устройств и отсутствие стандартизации аппаратного и программного обеспечения в отрасли производства мобильных устройств. Различаться могут виды носителей, на которых хранится информация, файловые системы, а также операционные системы, под управлением которых работают устройства. Даже различные модели

мобильных телефонов, сделанные одним и тем же производителем, могут потребовать различные кабели передачи данных и программное обеспечение для доступа к информации телефона.

К концу 2012 года около 70% общей доли рынка мобильных устройств занимали устройства под управлением операционной системы Android [2]. Первый выпуск мобильных телефонов под управлением ОС Android состоялся в конце 2008 года [3], и всего за несколько лет ОС Android стала лидером в области операционных систем для мобильных устройств. Массовое распространения ОС Android вызвало необходимость разработки надёжных методов проведения цифрового криминалистического анализа устройств, работающих на базе этой платформы.

До недавнего времени процедура создания точной и достоверной копии данных с мобильного устройства в ходе расследования ограничивалась только устройством хранения информации, таким как встроенная флеш-память или внешняя карта памяти. Это означает, что исследователь полагался только на файловую систему накопителя. У такого подхода есть свои плюсы. Во-первых, даже с выключенным питанием данные на флеш-накопителе долгое время остаются неизменными. Процедура снятия такого образа безопасна с точки зрения подлинности данных, так как не влечет изменения содержимого накопителя. Во-вторых, уже существует огромное количество утилит и продуктов для анализа файловых систем, что значительно облегчает экспертизу. Однако, есть и большой минус: файловая система, снятая с флеш-накопителя мобильного устройства, может быть зашифрована. Если неизвестен секретный ключ, процедура расшифровки файловой системы становится практически невозможной.

Анализ оперативной памяти

Для проведения более полной экспертизы мобильного устройства, помимо флеш-накопителя, необходимо анализировать оперативную память устройства. Анализ оперативной памяти имеет огромное значение для исследования, поскольку она содержит большое количество информации, которую трудно или невозможно получить, располагая только флеш-накопителем. В оперативной памяти хранится актуальная на данный момент информация о работающей на устройстве операционной системе, запущенных приложениях (в том числе вредоносных), открытых файлах, установленных сетевых соединениях с другими устройствами или веб-узлами, расшифрованных данных и многое другое. Более того, в памяти могут храниться данные, относящиеся к прошлому, например, список недавно работающих приложений. Отсутствие средств исследования памяти может сделать некоторые типы экспертиз невозможными, например, анализ современных вредоносных программ, которые не взаимодействуют с энергонезависимой памятью.

Информация, которая содержится в оперативной памяти, будет утеряна после вы-

ключения или перезагрузки устройства. Для того чтобы её сохранить, с работающего мобильного устройства снимается копия оперативной памяти и записывается в файл (образ). Далее анализ ведётся только с образом памяти. Одна из ключевых целей которого — найти и идентифицировать в образе процессы, включая завершённые, потенциально скрытые и вредоносные. Сложность их поиска обусловлена существованием большого количества различных версий ОС Android, а также различных версий и сборок ядра Linux, которое используется в семействе ОС Android.

Стоит объяснить, зачем вообще использовать технику анализа образа оперативной памяти, если существуют утилиты, такие как, например, ps[1] для вывода списка запущенных процессов в работающей системе. Дело в том, что целевая система может быть заражена вредоносными программами, которое могут нарушать корректную работу системных утилит, например, изменять вывод списка процессов с целью скрытия себя. Кроме того, современные вредоносные программы способны делать вставку вредоносного кода непосредственно в выполняемые процессы так, что никакие новые процессы при этом не создаются. Исследование же образа оперативной памяти позволяет напрямую анализировать структуры ядра ОС и находить такой "обман".

Существует большое количество исследовательских работ, в которых обсуждается анализ файловой системы ОС Android. Также успешно развиваются техники снятия и исследования оперативной памяти мобильных устройств на базе ОС Android. В данной дипломной работе проводится обзор существующих методов снятия образа, описаны механизмы работы виртуальной памяти в ОС Android, а также разработаны и реализованы общие, независимые от версий ОС и моделей устройств, методы для восстановления виртуального адресного пространства процессов из образа памяти.

[1] ps — утилита командной оболочки ОС Android для вывода списка запущенных процессов в системе.

1. Постановка задачи

В рамках данной дипломной работы была поставлена задача разработать общий, независимый от версии ОС Android, версии ядра и модели мобильного устройства, алгоритм нахождения всех запущенных процессов в образе памяти. Для разработки и тестирования алгоритма рассматривались и использовались только наиболее распространённые на данный момент архитектуры мобильных устройств — ARMv6 и ARMv7.

Каждый процесс в системе имеет своё виртуальное адресное пространство, которое отображается в оперативную память, не обязательно прямолинейно. Необходимо разработать алгоритм восстановления (дефрагментации) виртуального адресного пространства каждого найденного процесса.

Для любого запущенного процесса в ОС Android в ядре ОС создаются специальные структуры, содержащие информацию о нём, такую, как его идентификатор, имя, состояние, список дочерних процессов и многое другое. Необходимо разработать общий метод нахождения структур ядра ОС, описывающих найденный процесс, для восстановления его имени. Общность метода заключается в его применимости к любой версии ядра ОС Android.

Необходимо реализовать разработанные алгоритмы и методы и провести тестирование реализации на образах памяти, снятых с различных мобильных устройств, работающих под управлением различных версий ОС Android.

2. Обзор

2.1. Методы снятия образа памяти

К сожалению, мобильные устройства на базе ОС Android не предоставляют ни аппаратного, ни программного интерфейса для получение доступа к их физической памяти. Поэтому, для того чтобы снять образ памяти с устройства, его необходимо определённым образом подготовить. Во-первых, на устройстве требуется получить привилегии суперпользователя, чтобы можно было загрузить программный код в ядро ОС и считать физическую память. Процесс получения прав суперпользователя и возможность его применения с точки зрения криминалистического анализа описаны в работе [4]. Естественно, этап получения прав суперпользователя может быть пропущен для устройства, на котором необходимые права уже были получены ранее, например, владельцем устройства.

2.1.1. LiME

В работе [4] был разработан загружаемый модуль LiME (Linux Memory Extractor, ранее DMD) для ядра ОС Android, позволяющий производить считывание всей физической памяти мобильного устройства. Это единственный на данный момент рабочий способ получения образа памяти с мобильного устройства на базе ОС Android. В этой же работе описаны проблемы, из-за которых другие существующие решения снятия памяти оказались неприменимыми.

Одной из трудностей использования загружаемого модуля ядра LiME — это отсутствие переносимости. Таким образом, для каждой конкретной версии ядра ОС Android, используемой на устройстве, необходимо выполнять отдельную сборку модуля LiME. Сборка модуля также требует файл конфигурации соответствующего ядра (`.config`). Обычно файл конфигурации на работающем устройстве можно извлечь из `/proc/config.gz`. Однако на некоторых устройствах этот файл может отсутствовать [4].

Модуль LiME считывает все страницы оперативной памяти и записывает их в выходной файл на внешнюю карту памяти. Также модуль поддерживает скачивание считываемой памяти через TCP соединение. LiME может записывать выходные данные как в простом “сыром” формате (объединённые вместе все непрерывные диапазоны системной памяти), так и в собственном “lime” формате, в котором каждый диапазон начинается с заголовка фиксированного размера, содержащего информацию о начальном и конечном физических адресах считанного диапазона. Подробнее формат образов памяти LiME описан в документации [5].

2.2. Существующие решения в области анализа образа памяти

Образ памяти представляет собой мегабайты или даже гигабайты "сырых" данных. Самый простой способ анализа этих данных — это поиск строковых шаблонов. Данный подход относительно надёжный, но, к сожалению, очень медленный при работе с большими образами. Для некоторых задач использование поиска строковых шаблонов в образе памяти может быть очень сложным, например, в задаче вывода списка запущенных процессов операционной системы. Если версия и производитель операционной системы известны, можно составить шаблоны, которые будут подходить конкретно к данной версии ОС. Однако это невозможно в некоторых случаях, например, для самостоятельно скомпилированного ядра ОС Android. В данной работе будут описаны альтернативные подходы анализа образа оперативной памяти.

2.2.1. Volatility Framework

Volatility Framework [6] представляет собой набор утилит для разностороннего анализа образа физической памяти. Сейчас Volatility Framework поддерживается компанией Volatile Systems. Это один из самых функциональных открытых инструментов для анализа образов памяти. Его также можно использовать как платформу для проведения дальнейших исследований в этой области. Volatility Framework обладает следующими возможностями: извлечение информации о запущенных процессах, открытых сетевых соединениях, открытых файлах и загруженных библиотеках для каждого процесса, извлечение виртуального адресного пространства процесса и исполняемого файла из него, и многое другое. На данный момент предоставляется поддержка образов памяти операционных систем семейства Windows (XP, 2003 Server, Vista, 2008 Server, 7).

В октябре 2012 года была объявлена официальная поддержка ОС Linux, данное направление сейчас активно развивается. Возможность анализа различных версий ядра Linux достигается с помощью использования специальных профилей. Профиль — это набор файлов, описывающих содержимое и расположение различных структур ядра в памяти. Так как для поддержки конкретного ядра требуется иметь соответствующий ему профиль, необходимо поддерживать достаточно большую базу профилей. Естественно, для самостоятельно скомпилированного ядра невозможно заранее создать соответствующий ему профиль.

В последней версии Volatility Framework 2.3 (на данный момент еще официально не выпущенной) появилась поддержка виртуальной памяти архитектуры ARM и, соответственно, поддержка ОС Android. Также, как и для ядра Linux, для каждой версии и сборки ядра ОС Android требуется создание соответствующего профиля.

2.2.2. Volatilitux

Volatilitux [7] — это инструмент, написанный на языке Python, предназначенный для анализа образов физической (оперативной) памяти операционных систем на базе ядра Linux. Его создатель, Emilien Girault, был вдохновлён заданием, предложенным на конференции SSTIC летом 2010 года, целью которого было провести анализ образа физической памяти телефона на базе ОС Android. Попытка решить предложенное задание привела к созданию этого инструмента.

На данный момент Volatilitux поддерживает архитектуры ARM, x86 и x86 с расширением PAE и поддерживает выполнение следующих команд: вывод списка запущенных процессов, вывод отображения виртуальной памяти определённого процесса, чтение адресного пространства процесса, вывод списка открытых файлов и чтение их содержимого из образа памяти.

В сравнении с инструментом Volatility Framework, преимущество Volatilitux в том, что он не использует специальные профили, а необходимую информация о структурах ядра (их размеры, смещения полей внутри структур и т.д.) определяет автоматически исследуя образ памяти. К сожалению, для анализа многих тестовых образов мобильных устройств с ОС Android данный инструмент оказался неработоспособным. Одной из причин является отсутствие поддержки устройств, оперативная память которых начинается не с нулевого физического адреса (см. 3.2).

Последние изменения в коде проекта датированы декабрём 2010 года, и на данный момент он не развивается.

2.3. Механизм виртуальной памяти в архитектуре ARMv6/v7

Архитектура ARM (Advanced RISC Machine, усовершенствованная RISC-машина) — это семейство микропроцессоров, разрабатываемых компанией ARM Limited. Эта компания занимается только разработкой ядер и прикладные инструментов для них и не занимается производством или продажей процессоров, но даёт лицензии на их производство сторонним организациям. На сегодняшний день основная часть всех 32-разрядных мобильных процессоров использует архитектуру ARMv6 или ARMv7.

2.3.1. Аппаратное управление памятью

Каждый процесс в операционной системе нуждается в собственном программном коде, стеке, куче (структура данных для динамического выделения памяти), данных и переменных, которые должны находиться в памяти. Процессор читает инструкции и данные из памяти и записывает данные в память. Современные процессоры, в том числе процессоры семейства ARM, включают в себя набор аппаратных компонентов, которые увеличивают эффективность и надёжность работы с памятью, а

также предотвращают недопустимое обращение к памяти за пределами программы из-за программных ошибок.

Благодаря аппаратному блоку управления памятью (страницами), или MMU (Memory Management Unit), каждый процесс может иметь своё собственное адресное пространство. В архитектуре ARM задача разделения адресного пространства процессов решается использованием механизма виртуальной памяти, предоставляемого аппаратным обеспечением (блоком MMU). В таком случае каждый процесс работает в виртуальном, дополнительном адресном пространстве, которое не зависит от физической памяти. Блок управления страницами преобразует виртуальные адреса каждого процесса в физические (адреса на физической микросхеме памяти).

Блок управления памятью считает, что вся оперативная память разбита на страничные кадры фиксированной длины (физические страницы). Каждый страничный кадр содержит одну страницу, т.е. его длина равна длине страницы. Страничный кадр является составной частью оперативной памяти и, следовательно, областью для хранения данных. Необходимо проводить чёткое различие между страницей и страничным кадром. Страница — это всего лишь блок данных, который может храниться в любом страничном кадре или на диске.

Структуры данных, отображающие виртуальные адреса в физические, называются таблицами страниц. Они также хранятся в памяти и должны быть должным образом проинициализированы ядром операционной системы до включения блока управления страницами [8].

Один и тот же виртуальный адрес может отображать два разных страничных кадра (физические ячейки памяти) для двух разных процессов, так как каждый процесс имеет свою собственную таблицу страниц. Это позволяет разным процессам использовать одинаковые виртуальные адреса и одновременно находиться в физической памяти, тем самым быть любое время немедленно доступными для выполнения на процессоре.

За управление таблицами страниц отвечает операционная система. При создании нового процесса система создаёт для него новую таблицу страниц, выделяя таким образом ему виртуальное адресное пространство. Во время переключения контекста[2] операционная система должна проинформировать процессорный блок управления памятью о том, что он должен использовать другую таблицу страниц. Для этого система изменяет значение специального регистра процессора `TTBR` (Translation Table Base Register), записывая в него физический адрес начала таблицы страниц.

В версиях архитектуры до ARMv6 имеется только один такой регистр, который содержит физический адрес начала таблицы страниц, используемой для отображения

[2]Переключение контекста — процесс прекращения выполнения процессором одной задачи (с сохранением информации о состоянии, необходимой для последующего продолжения с прерванного места) и переход к выполнению другой задачи (с восстановлением и загрузкой её состояния).

всех возможных виртуальных адресов в физические. В версиях, начиная с ARMv6, имеются два регистра: TTBR0 и TTBR1. Физический адрес начала таблицы страниц, используемой для отображения младших виртуальных адресов, содержится в регистре TTBR0, а физический адрес начала таблицы страниц, используемой для отображения старших виртуальных адресов — в регистре TTBR1, причём размер таблицы, адресуемой через TTBR0, зависит от значением регистра TTBCR (Translation Table Base Control Register).

Этот механизм позволяет разделить виртуальное адресное пространство на две части и использовать для каждой из частей разные таблицы страниц. Предполагается, что через TTBR1 адресуется таблица страниц, которая не меняется при переключении контекста и используется для трансляции виртуальных адресов, расположенных в верхней части адресного пространства (виртуальные адреса операционной системы и отображённые в память регистры устройств ввода/вывода). Предполагается, что таблица страниц, адресуемая через TTBR0, используется для трансляции виртуальных адресов пользовательских процессов, расположенных в нижней части адресного пространства.

При переключение контекста операционная система должна изменить значение только регистра TTBR0, в то время как регистр TTBR1 будет по-прежнему содержать адрес таблицы страниц, описывающей отображение виртуальных адресов операционной системы и регистров устройств ввода/вывода.

Размер таблицы страниц, адресуемой через регистр TTBR0, в зависимости от значения регистра TTBCR, может колебаться в пределах от 128 байт до 16 Кбайт (от 32 до 4096 записей), что соответствует размеру виртуального адресного пространства от 32 Мбайт до 4 Гбайт. Если её размер равен 16 Кбайт, то она описывает отображение всех возможных виртуальных адресов в физические. В таком случае регистр TTBR1 для отображения виртуальных адресов не используется. Размер таблицы, адресуемой через регистр TTBR1, всегда равен 16 Кбайт. Физические адреса таблиц страниц, находящиеся в регистрах TTBR0 и TTBR1, должны быть выровнены по границам, равным размерам соответствующих таблиц [9].

Для повышения эффективности работы с памятью блок управления памятью архитектуры ARM использует две оптимизации: одна оптимизирует время отображения виртуального адреса в физический, а вторая — объём используемой памяти для хранения таблицы страниц.

1. Для того, чтобы ускорить процесс отображения виртуального адреса в физический, блок управления памятью использует специальный аппаратный механизм кэширования — TLB (Translation Lookaside Buffer), который поддерживает кэш часто используемых отображений. Обращение к памяти для выборки записей таблицы страниц выполняется лишь тогда, когда соответствующая информация в TLB отсутствует.

2. Многие процессы используют только малую часть доступного им виртуального адресного пространства. Таблицы страниц таких процессов содержат много пустых записей. Поэтому для более эффективного расходования памяти для хранения таблицы страниц, блок управления памятью архитектуры ARM использует двухуровневую иерархию таблиц страниц. Структура таблиц страниц и процесс отображения виртуального адреса в физический подробнее описаны в следующих разделах.

   Механизм страничной адресации памяти даёт целый ряд преимуществ для процессов [10].

2.3.2. Страничная организация виртуальной памяти

Для преобразования виртуальных адресов в физические (номера ячеек физической памяти) блок управления памятью (MMU) в процессорах ARM использует двухуровневую иерархию таблиц страниц. Таблица страниц состоит из упорядоченного массива записей (“дескрипторов”), размер которых равен 4 байт. Размер таблицы зависит от её типа (см. 2.3.3, 2.3.4). Старшие биты записи представляют собой адрес, указывающий на страничный кадр, содержащий следующую в иерархии таблицу страниц, или на страничный кадр, содержащий непосредственно страницу. Младшие биты — это флаги, характеризующие различные состояния страницы, а также определяющие параметры кэширования адресуемой страницы памяти и уровень привилегий, необходимый для обращения к странице. Записи таблиц страниц имеют определённый формат для каждого конкретного случая в зависимости от размера и типа отображаемой страницы. Страничная организация памяти позволяет операционной системе также эффективно управлять правами доступа к памяти. Например, страница может быть доступна только для чтения, для чтения и записи, защищена от исполнения процессором инструкций, записанных в ней, доступна только в привилегированном режиме ядра, причём контроль доступа выполняется на аппаратном уровне блоком MMU.

В версиях, начиная с ARMv6, блок управления памятью поддерживает четыре типа страниц. Страницы больших размеров называются секциями. Размеры поддерживаемых секций и страниц следующие:

- Суперсекция — 16 Мбайт
- Секция — 1 Мбайт
- Большая страница — 64 Кбайт
- Малая страница — 4 Кбайт

Начиная с версии ARMv6 поддержка “крошечных” (tiny) страниц размером 1 Кбайт прекращена. Также в версии ARMv6 для обратной совместимости оставлена поддержка механизма отображения части страницы (подстраницы), но режим использования подстраниц считается устаревшим. Далее в работе этот режим рассматриваться не будет.

Для отображения разных адресов могут использоваться разные размеры страниц. Секции и суперсекции могут быть смешаны с большими и малыми страницами. Используемый метод для отображения конкретного адреса зависит от типа соответствующей записи в таблице страниц. То, что архитектура поддерживает страницы различных размеров не означает, что операционная система обязана использовать все возможные варианты.

Таблица страниц первого уровня описывает преобразование адресов для секций и суперсекций, а также содержит ссылки на таблицы страниц второго уровня, отвечающие за преобразование адресов для больших и малых страниц. Схема страничной организации виртуальной памяти в архитектуре ARM с использованием секций и страниц представлена на Рис. 1. В следующих разделах описывается формат таблиц страниц 1-го и 2-го уровня и механизм отображения виртуального адреса в физический, использующий эти таблицы.

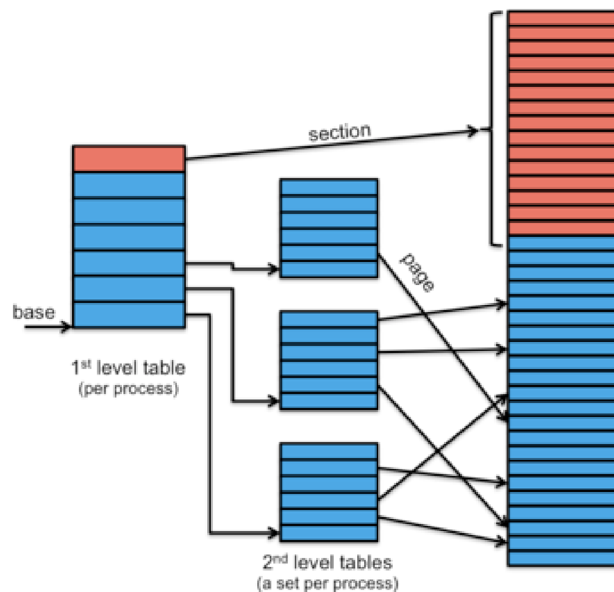

Рис. 1: Страничная организация виртуальной памяти в архитектуре ARM (из [10])

2.3.3. Таблица страниц 1-го уровня

Запись таблицы страниц первого уровня содержит физический адрес секции или суперсекции или ссылку на таблицу страниц второго уровня. Если для отображение виртуального адреса используется секция или суперсекция, то нет необходимости

проходить через два уровня иерархии таблиц страниц. Преимущество отображения памяти достаточно большими блоками, такими как секции и суперсекции, в том, что большие области памяти, например, операционная система, могут быть отображены в виртуальное адресное пространство используя всего одну запись в таблице страниц. Следовательно, для её кэширования требуется всего одна запись в кэше `TLB`, что значительно повышает производительность обращений к адресам, попадающим в отображаемую область памяти.

Если в процессе трансляции отображение запрашиваемого адреса не было найдено в кэше `TLB`, выполняется обход таблиц страниц, начиная с обращения к таблице страниц первого уровня. Как было отмечено ранее (см. 2.3.1), начиная с версии ARMv6, блок управления памятью поддерживает две таблицы страниц первого уровня. Сам процесс трансляции адресов в целом не зависит от того, какая из таблиц используется, поэтому будет описан случай использования таблицы, адресуемой через регистр `TTBR0`, так как другая таблица является лишь частным случаем данной.

Таблица страниц первого уровня содержит от 32 до 4096 записей, в зависимости от значения регистра `TTBCR`, что соответствует размеру виртуального адресного пространства от 32 Мбайт до 4 Гбайт. Для заданного 32-битного виртуального адреса 32-битный физический адрес соответствующей записи таблицы страниц первого уровня формируется из трёх частей (Рис. 2), где X — значение от 0 до 7 включительно, содержащееся в регистре `TTBCR`:

1. Старшие биты 31 – 14-X копируются неизменными из выбранного регистра `TTBR`.

2. Средние биты 13-X – 2 формируются из старших битов (31-X – 20) заданного виртуального адреса и играют роль номера искомой записи в таблице.

3. Два младших бита всегда равны нулю, поскольку адрес записи всегда находится на границе 4 байт.

По полученному физическому адресу находится 4-байтная запись (дескриптор), формат которой приведён на Рис. 3. В случае секции и ссылки на таблицу страниц второго уровня запись описывает отображение одного мегабайта виртуального адресного пространства, в случае суперсекции — 16 Мбайт.

Два младших бита записи определяют тип соответствующей записи. Если их значения равны:

- `0b00` — отображение виртуальных адресов, соответствующих этой записи, в физическую память отсутствует. Попытка обращения по таким адресам вызовет отказ.

- `0b01` — отображение одного мегабайта виртуального адресного пространства, соответствующего этой записи, описывается посредством "грубой" (coarse) таб-

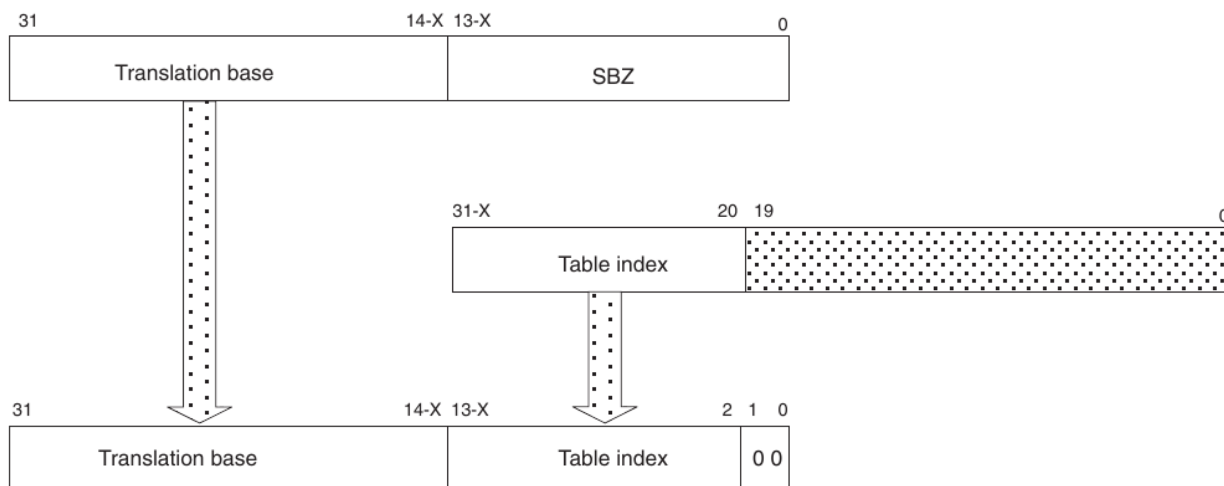

Рис. 2: Формирование адреса записи таблицы страниц первого уровня (из [9])

	31 … 24	23 … 20	19	18	17	16	15	14 … 12	11 10	9	8 … 5	4	3	2	1	0
Fault	IGN														0	0
Coarse page table	Coarse page table base address									IMP	Domain	SBZ			0	1
Section	Section base address		SBZ	0	nG	S	APX	TEX	AP	IMP	Domain	XN	C	B	1	0
Supersection	Supersection base address	Base address [35:32]	SBZ	1	nG	S	APX	TEX	AP	IMP	Base address [39:36]	XN	C	B	1	0
	RESERVED														1	1

Рис. 3: Формат записи таблицы страниц первого уровня (из [9])

лицы страниц (таблица страниц второго уровня). Данная запись содержит физический адрес начала этой таблицы.

- `0b10` — отображаемая область виртуального адресного пространства, соответствующая этой записи, является секцией или суперсекцией. Данная запись содержит физический адрес секции или суперсекции.

- `0b11` — в рассматриваемых версиях архитектуры (ARMv6 и ARMv7) данный тип записи зарезервирован.

Использование суперсекций является необязательным. Тип секции определяется состоянием бита 18 в записи таблицы страниц: 1 соответствует суперсекции, 0 — секции. Схема преобразование виртуального адреса в физический для суперсекций рассмотрена не будет, поскольку не имеет отношения к решению поставленных задач.

Формирование физического адреса ячейки памяти для виртуального адреса, относящегося к секции (её выборка описана выше), происходит следующим образом (Рис. 4):

1. Старшие 12 бит физического адреса формируются из старших битов дескриптора секции (поля `Section base address`). Они определяют физический адрес начала соответствующей секции (области памяти размером 1 Мбайт).

2. Младшие 20 бит физического адреса определяют относительное смещение искомой ячейки памяти внутри секции. Они соответствуют 20 младшим битам заданного виртуального адреса.

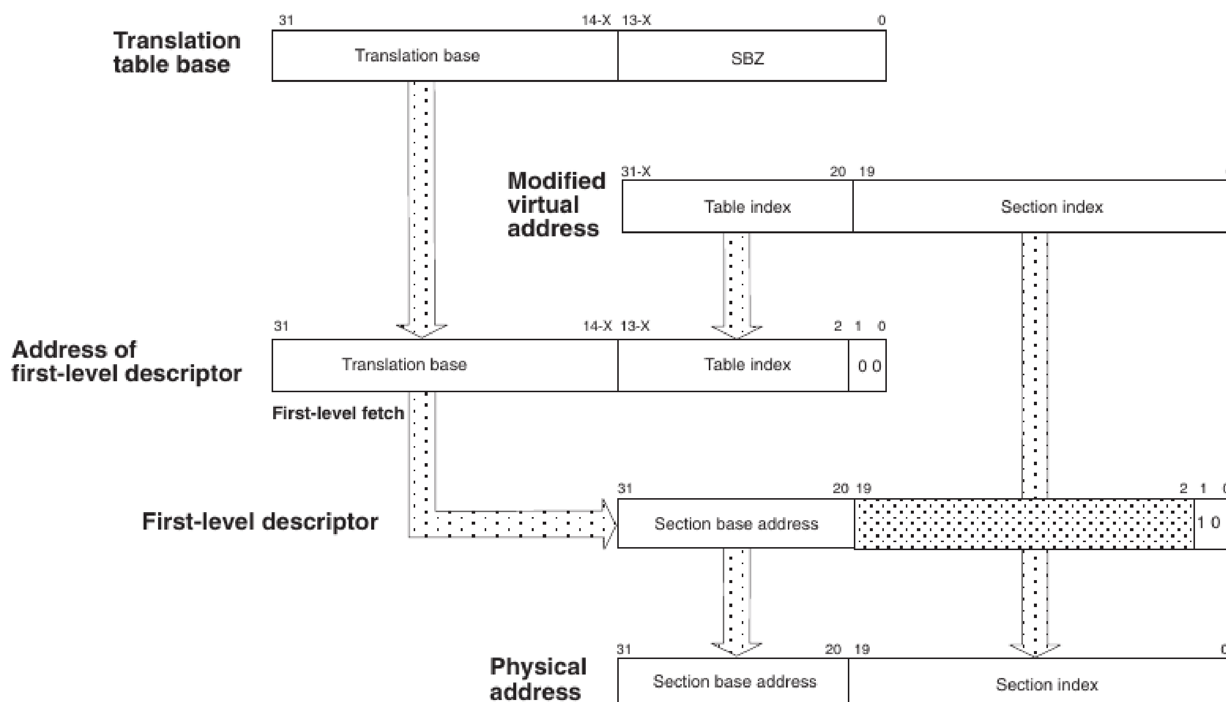

Рис. 4: Отображение виртуального адреса в физический для секций размером 1 Мбайт (из [9])

Если тип записи таблицы страниц первого уровня соответствует дескриптору "грубой" таблицы страниц, то для преобразования заданного виртуального адреса в физический необходимо пройти через второй уровень иерархии таблиц страниц.

2.3.4. Таблица страниц 2-го уровня

Таблица страниц второго уровня, или "грубая" таблица страниц, в отличии от таблицы страниц первого уровня, всегда имеет фиксированный размер и содержит 256 записей размером 4 байт. Суммарный размер таблицы составляет 1 Кбайт и, согласно аппаратным требованиям архитектуры, её адрес должен быть выровнен по границе

1 Кбайт. Таблица страниц второго уровня описывает преобразование виртуальных адресов для больших (64 Кбайт) и малых (4 Кбайт) страниц. Каждая запись таблицы описывает отображение 4 Кбайт виртуального адресного пространства, поэтому записи для больших страниц должны повторяться 16 раз подряд. Ввиду похожести будет рассмотрено отображение только для малых страниц.

Физический адрес необходимой записи таблицы страниц второго уровня для заданного виртуального адреса и полученного на предыдущем шаге дескриптора «грубой» таблицы страниц формируется из трёх частей (Рис. 6):

1. Старшие 22 бита адреса формируются из старших битов дескриптора (поля `Coarse page table base address`). Они определяют физический адрес начала таблицы страниц второго уровня.

2. Средние 8 бит определяют номер искомой записи в таблице, в роли которых выступают биты 19 – 12 заданного виртуального адреса.

3. Два младших бита всегда равны нулю, поскольку адрес записи всегда находится на границе 4 байт.

Формат выбранной 4-байтной записи (дескриптора) изображён на Рис. 5.

<table>
<tr><th></th><th>31 … 16</th><th>15</th><th>14 … 12</th><th>11</th><th>10</th><th>9</th><th>8 7 6</th><th>5 4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>Fault</td><td colspan="10">IGN</td><td>0</td><td>0</td></tr>
<tr><td>Large page</td><td>Large page base address</td><td>XN</td><td>TEX</td><td>nG</td><td>S</td><td>APX</td><td>SBZ</td><td>AP</td><td>C</td><td>B</td><td>0</td><td>1</td></tr>
<tr><td>Extended small page</td><td colspan="3">Extended small page base address</td><td>nG</td><td>S</td><td>APX</td><td>TEX</td><td>AP</td><td>C</td><td>B</td><td>1</td><td>XN</td></tr>
</table>

Рис. 5: Формат записи таблицы страниц второго уровня (из [9])

Как и в случае с таблицей страниц первого уровня, два младших бита записи таблицы страниц второго уровня определяют тип соответствующей записи. Если их значения равны:

- `0b00` — отображение виртуальных адресов, соответствующих этой записи, в физическую память отсутствует. Попытка обращения по таким адресам вызовет отказ.

- `0b01` — отображаемая область виртуального адресного пространства является большой страницей (64 Кбайт). Данная запись содержит физический адрес страничного кадра.

- `0b10/0b11` — отображаемая область виртуального адресного пространства является малой страницей (4 Кбайт). Данная запись содержит физический адрес страничного кадра.

Схема процесса преобразования виртуального адреса в физический для малых страниц приведена на Рис. 6. Формирование физического адреса ячейки памяти происходит следующим образом:

- Старшие 20 бит физического адреса формируются из старших битов дескриптора страницы (поля `Extended small page base address`). Они определяют физический адрес страничного кадра размером 4 Кбайт.

- Младшие 12 бит физического адреса определяют относительное смещение искомой физической ячейки памяти внутри страничного кадра. Они соответствуют младшим битам виртуального адреса.

Стоит отметить, что вышеописанный механизм трансляции виртуального адреса в физический (разбор записей, обход иерархии таблиц, формирование адреса) реализован аппаратно блоком управления памятью процессора (MMU). Также описанный механизм не претендует на свою полноту, а лишь сфокусирован на ключевые моменты организации аппаратного управления памятью, необходимые для решения поставленных задач.

2.4. Особенности реализации виртуальной памяти в ядре ОС Android

В основе операционной системы Android лежит ядро Linux. В ядре Linux реализована поддержка архитектуры ARM, в том числе поддержка виртуальной памяти на этой архитектуре. Каждому процессу в 32-разрядной ОС Android ядро Linux выделяет 4 Гбайт виртуального адресного пространства. Первые 3 Гбайт являются адресным пространством режима пользователя, последний 1 Гбайт — адресным пространством режима ядра. Адресное пространство режима ядра зарезервировано для кода, данных и стека ядра операционной системы. С помощью параметра компиляции ядра `PAGE_OFFSET` граница разделения адресных пространств может быть изменена, но обычно значение `PAGE_OFFSET` равно `0xc0000000`, что соответствует разделению виртуального адресного пространства процесса на 3 и 1 Гбайт для режима пользователя и ядра соответственно.

Ядро Linux разделяет физическую память на две части: нижнюю память (low memory) и верхнюю память (high memory). Нижняя память — это часть физической памяти, для которой существуют логические адреса[3] в пространстве ядра. Во многих

[3]Логические адреса составляют обычное адресное пространство ядра. Эти адреса отображают

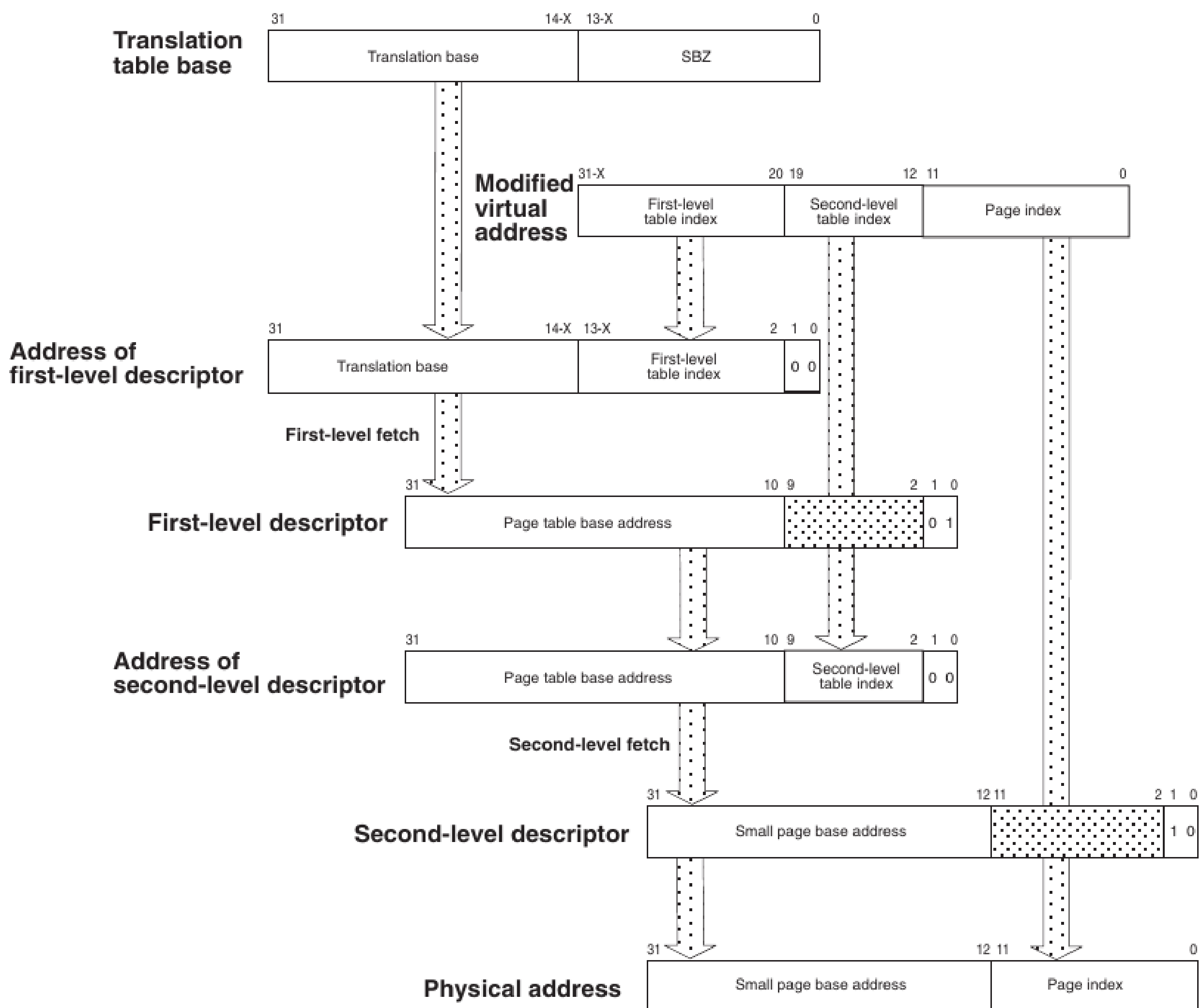

Рис. 6: Отображение виртуального адреса в физический для малых страниц размером 4 Кбайт (из [9])

устройствах с малым объёмом памяти вся память является нижней. Верхняя память — память, логические адреса для которой не существуют, потому как она выходит за пределы адресов, отведённых для виртуальных адресов ядра [11].

Виртуальное пространство режима ядра делится на диапазоны адресов, каждый из которых используется для различных целей. Начало первого диапазона совпадает с началом адресного пространства режима ядра. Этот диапазон прямолинейно отображает физическую память (возможно, всю) так, что адрес PAGE_OFFSET соответствует адресу начала физической памяти. Таким образом, только самая нижняя часть памяти (менее 1 Гбайт) имеет постоянное линейное отображение в пространстве ядра и имеет логические адреса. Для того, чтобы обратиться к странице верхней памяти ядро должно явно задать виртуальное соответствие, чтобы сделать доступной заданную

какую-то часть (возможно, всю) основной памяти и часто рассматриваются, как если бы они были физическими адресами. На архитектуре ARM логические адреса и связанные с ними физические адреса отличаются только на постоянное смещение [11].

страницу в адресном пространстве ядра.

Следующий диапазон адресов как раз используется для отображения в него верхней памяти. Минимальный размер его задаётся параметром сборки ядра `VMALLOC_RESERVE`, значение которого по умолчанию равно 128 Мбайт. Макрос `VMALLOC_END` задаёт виртуальный адрес конца этой области и определяется реализациями различных платформ, обычно в файле `arch/arm/mach-xxx/include/mach/vmalloc.h`. Следовательно, при установленных по умолчанию параметрах сборки ядра, максимальный объём нижней памяти не может превышать 896 Мбайт.

2.4.1. Четырёхуровневая иерархия таблиц страниц

В ядре Linux принята универсальная модель управления страницами, которая подходит как к 32-разрядным, так и к 64-разрядным архитектурам. Для 32-разрядных архитектур (без расширения PAE) достаточно двух уровней управления страницами, а 64-разрядные требуют большего количества уровней (четыре уровня для архитектуры x86_64). Начиная с версии 2.6.11 в ядре Linux используется четырёхуровневая модель, даже если нижележащая архитектура не поддерживает все уровни. Соответствующие четыре типа таблиц страниц называются так: глобальный каталог страниц (PGD), верхний каталог страниц (PUD), средний каталог страниц (PMD), Таблица Страниц (PTE). Как было сказано ранее в разделе 2.3 в 32-разрядной архитектуре ARM используется двухуровневая модель управления виртуальной памятью. Для её реализации ядро Linux фактически исключает из модели поля верхнего и среднего каталогов страниц, считая, что эти поля содержат нулевые биты. Тем не менее ядро сохраняет позиции верхнего и среднего каталогов страниц в цепочке указателей, устанавливая количество записей в них 1 и отображая эти записи в соответствующую запись глобального каталога страниц [8]. Существует путаница в терминологии, поскольку архитектурно-независимый код ядра считает, что внешний уровень таблиц страниц — это PGD, а в коде, реализующем поддержку архитектуры ARM, для обращения к внешнему уровню используются макросы PMD.

В ядре Linux для архитектуры ARM размер таблицы страниц первого уровня, используемой аппаратным блоком управления памятью MMU и адресуемой через регистр `TTBR0`, равен 16 Кбайт. Это означает, что она описывает отображение всего виртуального адресного пространства процесса, как режима пользователя, так и режима ядра. Таблица страниц первого уровня в ядре соответствует глобальному каталогу страниц (PGD).

2.4.2. Существование двух видов таблиц страниц 2-го уровня

Особенным в реализации виртуальной памяти архитектуры ARM в ядре Linux является способ поддержки аппаратных таблиц страниц второго уровня. Блок MMU,

как уже было ранее сказано (см. 2.3.4), для второго уровня использует таблицы страниц размером 1 Кбайт, состоящие из 256 записей размером 4 байт каждая. Однако, код управления памятью в ядре ожидает поддержки некоторых битов в записях таблиц страниц, например, `PRESENT`, `DIRTY` и `YOUNG`, но записи таблиц страниц в ARM их не имеют. Эта информация должна храниться за пределами таблиц, используемых блоком MMU, но всё же в быстро доступном для ядра месте.

Решение данной проблемы заключается в следующем: ядро считает, что глобальный каталог страниц состоит из 2048 записей 8 байт каждая, тогда как соответствующая таблица страниц первого уровня в ARM имеет 4096 записей 4 байт каждая. Другими словами, одна запись PGD содержит две ссылки на аппаратные таблицы страниц второго уровня. Также ядро считает, что каждая Таблица Страниц состоит из 512 записей — двух подряд идущих аппаратных таблиц страниц второго уровня, используемых блоком MMU и состоящих из 256 записей каждая. Перед этими таблицами в Таблице Страниц ядра располагается еще две таблицы по 256 записей каждая — специальные программные записи, используемые только ядром Linux для хранения в них недостающих битов для нижележащих записей двух смежных таблиц. В результате Таблица Страниц состоит из 1024 записей — двух программных таблиц и двух аппаратных таблиц, суммарным размером 4 Кбайт. Описанная структура таблиц страниц изображена на Рис. 7.

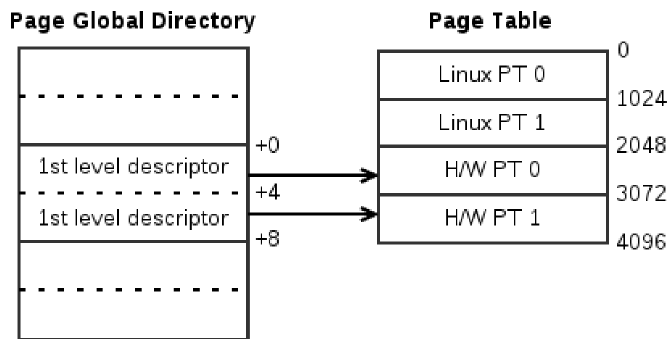

Рис. 7: Структура таблиц страниц архитектуры ARM в ядре Linux

Для любой записи из аппаратной таблицы относительное смещение соответствующей ей программной записи в PTE составляет 2048 байт. MMU никогда не использует дополнительные программные таблицы, созданные ядром. Когда необходимо создать новую аппаратную таблицу страниц второго уровня, ядро создает PTE, в которой инициализирует сразу две смежные аппаратные таблицы страниц и две смежные программные таблицы, и создаёт соответствующую аппаратным таблицам запись в PGD.

В версиях ядра Linux до 2.6.38 программные таблицы в PTE располагались после аппаратных таблиц. Изменение порядка, начиная с версии 2.6.38, обусловлено упрощением арифметики, связанной с вычислением относительных смещений записей.

3. Архитектура решения

В рамках данной дипломной работы исследования касаются операционной системы Android, работающей на архитектуре ARM. Для разработки и тестирования алгоритмов рассматриваются и используются только наиболее распространённые на данный момент версии архитектуры ARM — ARMv6 и ARMv7. Предполагается, что имеется файл образа оперативной памяти, снятый с целевого мобильного устройства с помощью метода LiME (см. 2.1.1).

Поставленные в данной работе задачи разработки общего подхода к восстановлению адресного пространства процессов из образа памяти ОС Android решаются в несколько этапов:

1. С помощью алгоритма, описанного в разделе 3.1, в образе памяти находятся все работающие процессы, а именно для каждого процесса находится табличная структура глобальный каталог страниц (Page Global Directory). Данный алгоритм использует особенности программно-аппаратного взаимодействия архитектуры ARM и не зависит от работающей на анализируемой системе версии ядра Linux. Фактически, для каждого активного процесса в системе находится физический адрес глобального каталога страниц, хранящийся в регистре `TTBR0` процессора.

2. После нахождения каталога страниц выполняется восстановление виртуального адресного пространства соответствующего ему процесса с помощью алгоритма, описанного в разделе 3.2. Для этого выполняется обход в глубину его записей, тем самым получается перебор виртуальных адресов процесса в порядке возрастания). Во время обхода из образа памяти последовательно извлекаются страницы, относящиеся к процессу, и записываются в отдельный выходной файл.

3. С помощью разработанного и описанных в разделе 3.3 методов находятся структуры ядра ОС Android, описывающие процессы в системе. С помощью них восстанавливается имя процесса, адресное пространство которого было получено на предыдущем этапе. Также с помощью этого метода из найденных процессов выявляются те, которые могли быть скрыты руткитами[4].

3.1. Нахождение процессов

Каждый пользовательский процесс в операционной системе должен работать в своём адресном пространстве, определённом набором соответствующих структур управ-

[4]rootkit — программа, основной задачей которой является скрытие следов присутствия в системе себя или другой программы.

ления страницами памяти. Следовательно, одним из методов нахождения всех процессов в образе памяти является нахождение в образе вышеупомянутых структур. Анализируя образ памяти, следует понимать семантику необработанных данных находящихся в образе. В первую очередь она зависит от работающей системы (аппаратное обеспечения, операционная система).

3.1.1. Аппаратный и программный подходы

Можно выделить два подхода к поиску структур управления страницами памяти (таблиц страниц): используя особенности реализации ядра операционной системы и используя особенности реализации аппаратного обеспечения.

Первый подход опирается на работу ядра операционной системы. Так как ядро обслуживает все структуры управления страницами памяти, можно найти их расположение проанализировав структуры данных ядра. Сложность применения данного подхода к анализу памяти рассматриваемых мобильных устройств обусловлена существованием большого количества различных версий и сборок ядра Linux, которое используется в семействе ОС Android. В разных версиях и сборках ядра структуры, которые описывают процесс и соответствующие ему таблицы страниц, могут существенно различаться: иметь разные размеры, разные смещения полей, некоторые поля могут отсутствовать. Сложно придумать надёжные методы поиска необходимых структур в образе опять, применимые для всех версий ядра ОС Android. Еще один недостаток данного подхода заключается в возможности получения недостоверной информации в случае скомпрометированного ядра и его структур данных.

Второй подход является более приемлемым. Он опирается на определённые особенности программно-аппаратного взаимодействия, которым должно удовлетворять даже вредоносное программное обеспечение. В работе [12] описан алгоритм, применяющий данный подход для устройств с архитектурой x86. Этот алгоритм находит структуры управления страницами памяти, которые используются аппаратным обеспечением для управления виртуальной памятью процессов ОС. Предложенный алгоритм оказался достаточно эффективным и устойчивым к изменениям версий ядра ОС. Его идея лежит в основе разработанного в рамках этой дипломной работы алгоритма нахождения процессов.

Таким образом в данной работе применяется второй подход: для нахождения структур управления страницами памяти в образе используются особенности реализации аппаратного обеспечения мобильных устройств, в частности, архитектуры ARM.

3.1.2. Алгоритм нахождения потенциальных PGD

Глобальный каталог страниц (PGD) создаётся ядром, но, т.к. используется аппаратным обеспечением, его формат должен удовлетворять строгим спецификациям

архитектуры ARM, описанные в разделе 2.3. Алгоритм для нахождения потенциальных PGD в образе памяти в первую очередь фокусируется на аппаратных флагах, выставленных в записях PGD, соответствующих отображению адресов пространства ядра. Первая часть адресного пространства ядра используется для отображения нижней памяти (логических адресов) “один к одному” (см. 2.4). В ядре Linux для отображения нижней памяти используются секции размером 1 Мбайт. Соответствующие 32-битные дескрипторы этих секций должны иметь следующий вид (Рис. 8):

- Два младших бита (тип записи) равны `0b10`, т.к. отображаемая область является секцией (см. 2.3.3).
- Бит 2 (`B`) равен 1, указывает, что записываемые в отображаемую память данные могут накапливаться в буферах процессора и физически записываться позже.
- Бит 3 (`C`) равен 1, указывает, что данные, считываемые из отображаемой памяти, могут помещаться в кэш и позже извлекаться по возможности оттуда. Установленные одновременно флаги B и C задают использование внешнего и внутреннего кэша с отложенной записью (write-back).
- Бит 15 (`APX`) равен 0, биты 11–10 (`AP`) равны `0b01`, разрешают доступ к отображаемой памяти для чтения и записи только из привилегированного режима ядра и запрещают доступ из режима пользователя.
- Бит 18 равен 0, т.к. отображаемая область не является суперсекцией.

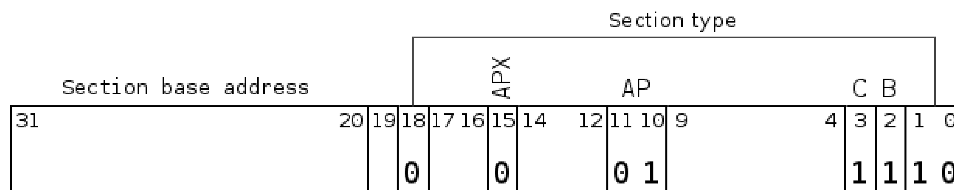

Рис. 8: Дескриптор секции глобального каталога страниц (отображение нижней памяти в пространстве ядра Linux)

Поскольку нижняя память отображается в виртуальное адресное пространство режима ядра каждого процесса, то каждый действительный каталог страниц в памяти, вне зависимости от версии используемого ядра и платформы устройства, должен иметь определённое количество записей такого типа, расположенных в его последней четверти. Последняя четверть — это записи, относящиеся к отображению адресного пространства ядра.

Глобальный каталог станиц в памяти занимает 16 Кбайт и его адрес выровнен по границе 16 Кбайт. Он содержит 4096 (0 – 4095) 4-байтных аппаратных записей

(см. 2.4.2). Таким образом, в образе памяти с шагом 16 Кбайт перебираются блоки данных размером 16 Кбайт. Каждый такой выбранный блок — это потенциальный каталог страниц, для которого необходимо выполнить проверку валидности. Для него считается число записей, находящихся в последней четверти блока (записи с номерами 3072 – 4095) и имеющих вид дескрипторов секций, описанный выше. Эти записи соответствуют отображению нижней памяти (возможно, всей физической памяти) в пространстве ядра. Поскольку каждая такая запись определяет отображение одного мегабайта нижней памяти, то полученное число сравнивается с пороговым значением, равным немного меньше min(объём физической памяти Мбайт, 896). Значения 896 (Мбайт) соответствует максимально возможному размеру нижней памяти в ядре Linux по умолчанию (см. 2.4) и является настраиваемым параметром алгоритма. Если полученное значение меньше порогового, то выбранный блок данных исключается из рассмотрения и осуществляется переход к следующему блоку размером 16 Кбайт.

Если для потенциального каталога страниц выполняется вышеописанное пороговое условие, осуществляется переход к следующему этапу проверки корректности всех его записей, отвечающих за отображение как адресов пространства пользователя, так и пространства ядра (номера 0 – 4095). Если два младших бита записи равны `0b11`, то данный блок сразу же исключается из рассмотрения, т.к. запись такого типа является зарезервированной в рассматриваемых версиях архитектуры ARM (см. 2.3.3). Ядро не может использовать их для отображения виртуальной памяти процесса, следовательно, потенциальный каталог страниц не должен содержать записей такого вида.

Чтобы усилить метод нахождения действительных каталогов страниц, можно дополнительно провести еще один этап проверки всех его записей, чтобы быть уверенным, что найден именно каталог страниц. В случае ссылки на таблицу страниц второго уровня (младшие биты записи равны `0b01`) в записи присутствует поле `Coarse page table base address`, содержащее физический адрес начала соответствующей таблицы страниц второго уровня (Рис. 3). В случае дескриптора секции (младшие биты записи равны `0b10`) поле `Section base address` записи содержит физический адрес начала секции, соответствующей этой записи (Рис. 3). В каждом из случаев выполняется проверка, что физический адрес попадает в допустимый диапазон (не превосходит объём оперативной памяти). Тестирование алгоритма на образах памяти различных устройств показало хороший результат его работы и без этого дополнительного этапа проверки корректности потенциального каталога страниц.

Если рассматриваемый блок (потенциальный каталог страниц) удовлетворяет всем описанным выше условиям, то он помечается как действительный каталог страниц. Этот каталог страниц идентифицирует очередной процесс в образе памяти.

Такой метод нахождения процессов не зависит от версии ОС Android, ядра Linux и модели мобильного устройства. Дополнительным преимуществом этого подхода является способность находить недавно завершённые процессы, а также скрытые вре-

доносными программами (руткитами), т.к. даже скрытый процесс в системе должен иметь свой каталог страниц, удовлетворяющий описанным выше условиям.

3.2. Восстановление адресного пространства процесса

Существует одна особенность в организации физической памяти устройств с архитектурой ARM, отличная от других архитектур, например, Intel. Далеко не во всех ARM устройствах оперативная память начинается с нулевого физического адреса. Для таких устройств в ядре Linux задаётся специальный параметр сборки `PHYS_OFFSET`, который равняется физическому адресу начала блока оперативной памяти. Все функции в ядре, выполняющие преобразования логических адресов в физические, делают поправку на это значение. Таким образом, утверждение о том, что смещению X в оперативной памяти соответствует смещение X в файле образа памяти в общем случае неверно. Для того, чтобы корректно находить в файле образа памяти смещения, соответствующие физическим адресам, необходимо знать значение `PHYS_OFFSET`.

В случае, когда образ памяти был получен в формате LiME с помощью метода, описанного в разделе 2.1.1, необходимая информация о смещении оперативной памяти уже находится в файле образа. В случае же "сырого" образа, значение смещения может быть получено из первой записи глобального каталога страниц, отвечающей за отображение пространства ядра (запись с номером 3072). Эта запись отображает первый мегабайт нижней (физической) памяти (см. 2.4) и содержит базовый физический адрес начала блока оперативной памяти.

Каждый найденный описанным в предыдущем разделе алгоритмом каталог страниц (PGD) определяет отдельный процесс в системе (его адресное пространство). После того, как выбран нужный каталог страниц, можно приступить к восстановлению адресного пространства соответствующего ему процесса. Для этого выполняется обход в глубину связанной с ним иерархии таблиц страниц следующим образом:

1. Перебираются по порядку аппаратные PDE (Page Directory Entry) записи выбранного каталога страниц. В зависимости от того, какой тип имеет запись, производятся соответствующие действия. Тип записи определяется значением двух младших битов, возможны три варианта (см. 2.3.3):

   - Два младших бита записи равны `0b00` — виртуальные адреса, соответствующие этой записи, не имеют отображения в физическую память. Такая запись пропускается.
   - Два младших бита записи равны `0b01` — запись ссылается на аппаратную таблицу страниц второго уровня, которая является составной частью соответствующей ей Таблицы Страниц. Подробнее об особенностях реализации таблиц страниц в ядре Linux было описано в разделе 2.4. В данном случае

из записи извлекается поле `Coarse page table base address` (биты 31–10) — базовый физический адрес аппаратной таблицы страниц второго уровня, на которую ссылается эта запись (Рис. 3). Далее по базовому адресу вычисляется её расположение в образе памяти (смещение от начала файла). Поскольку эта таблица страниц имеет размер 1 Кбайт, её смещение в файле образа равняется `Coarse page table base address` · 1024 байт. После того, как в образе памяти найдена таблица страниц второго уровня, осуществляется переход к пункту 2.

- Два младших бита записи равны `0b10` — запись описывает отображение области памяти (секции) размером 1 Мбайт. Из записи извлекается базовый физический адрес соответствующей ей секции в оперативной памяти — поле `Section base address` (биты 31–20) Рис. 3. Так как секция в данном случае имеет размер 1 Мбайт, смещение этой секции в файле образа равняется `Section base address` · 2^{20} байт. В ядре Linux суперсекции используются только для отображения 36-битных адресов физической памяти и, следовательно, записи, имеющие формат суперсекций, не могут присутствовать в глобальном каталоге страниц, если объём физической памяти устройства не превосходит 4 Гбайт.

2. Перебираются PTE (Page Table Entry) записи полученной на шаге 1 таблицы страниц второго уровня. Ядро Linux для отображение виртуальной памяти не использует большие страницы размером 64 Кбайт, поэтому записи такого типа не могут находиться в таблице страниц второго уровня. Тип записи, как и в случае с каталогом страниц, определяется значением двух младших битов, возможны два варианта (см. 2.3.4):

   - Два младших бита записи равны `0b10` или `0b11` — запись указывает на страницу размером 4 Кбайт (малая страница), находящуюся в оперативной памяти, а, следовательно, данная страница присутствует и в снятом образе памяти. Базовый физический адрес этой страницы равняется полю `Extended small page base address` (биты 31–12) Рис. 5, а смещение в файле образа — `Extended small page base address` · 4096 байт.

   - Два младших бита записи равны `0b00` — страница не принадлежит адресному пространству процесса, либо соответствующий страничный кадр еще не был назначен процессу (выделение страниц по требованию). Для получения информации об этой странице необходимо изучать структуру ядра `vm_area_struct`, относящуюся к данному процессу. Записи такого типа также могут определять страницы, выгруженные в область подкачки, — страницы, которые не помещаются в оперативной памяти. Однако в стандартной

конфигурации ОС Android механизм подкачки страниц не используется, поэтому такие записи можно не рассматривать, что значительно упрощает процесс восстановления адресного пространства.

Для каждой найденной страницы во время обхода иерархии таблиц страниц известны номера записей, по которым совершался переход в каталоге страниц и таблице страниц второго уровня, следовательно, можно вычислить соответствующий ей виртуальный адрес. Пусть

$$pde_index = \text{“номер записи PDE (начиная с 0) в каталоге страниц”},$$
$$pte_index = \text{“номер записи PTE (начиная с 0) в таблице страниц второго уровня”}.$$

Тогда виртуальный адрес этой страницы равняется

$$pde_index \ll 20 \mid pte_index \ll 12,$$

где знаком “<<” обозначена операция побитового сдвига влево, а знаком “|” — операция побитового ИЛИ. Процесс трансляции виртуального адреса в физический с использованием малых страниц описан в разделе 2.3.4.

Аналогично для каждой найденной секции известен номер соответствующий ей записи в каталоге страниц. Пусть

$$pde_sect_index = \text{“номер записи PDE (начиная с 0) в каталоге страниц”},$$

тогда виртуальный адрес этой секции равняется

$$pde_sect_index \ll 20.$$

Подробнее процесс трансляции виртуального адреса в физический для секций описан в разделе 2.3.3.

Перед выполнением обхода таблиц страниц для исследуемого процесса создаётся отдельный выходной файл образа памяти процесса, в который последовательно копируются найденные описанным алгоритмом страницы и секции. Этот файл содержит упорядоченную последовательность областей памяти, относящихся к виртуальному адресному пространству процесса. Эти области идут в файле в порядке возрастания их виртуальных адресов, но диапазоны адресов любых двух соседних областей не обязаны быть смежными. Для того, чтобы иметь возможность по виртуальному адресу определять соответствующее ему смещение в файле, необходимо поддерживать дополнительную индексную структуру. Для каждой записанной в выходной файл области в ней хранится соотношение “диапазон виртуальных адресов — смещение”. В простейшем случае такой структурой может служить массив пар (диапазон, смещение).

Таким образом, с помощью описанного в этом разделе алгоритма для выбранного процесса восстанавливается его виртуальное адресное пространство, как режима пользователя, так и режима ядра.

3.3. Нахождение структур ядра, описывающих процесс

Чтобы восстановить имена процессов, адресные пространства которых были получены описанным в предыдущем разделе способом, необходимо обратиться к связанным с процессами структурам ядра ОС Android.

Прежде чем описывать, как ядро отслеживает процессы в системе, стоит отметить важность определённых структур, реализующих двунаправленные списки. В ядре Linux определена структура `list_head`, два поля которой, `next` и `prev`, содержат указатели на следующий и предыдущий элементы двунаправленного списка общего назначения. При этом важно отметить, что указатели содержат адреса других структур `list_head`, а не адреса структур, включающих в себя структуру `list_head` (Рис. 9).

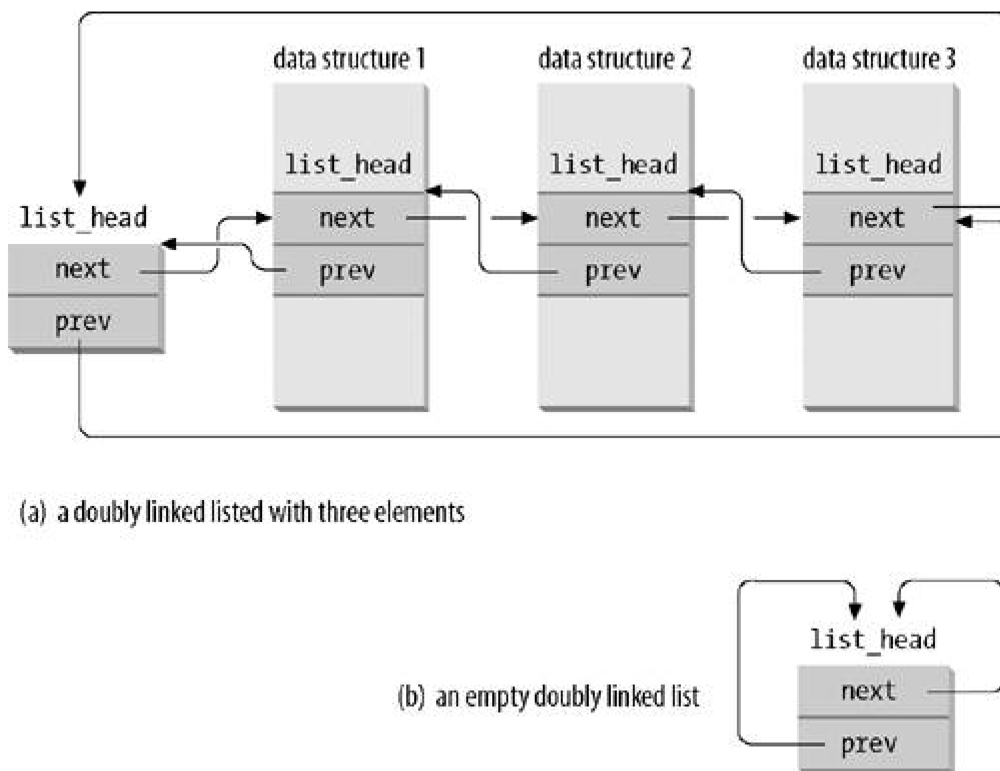

Рис. 9: Двунаправленные списки, построенные из структур list_head (из [8])

В ядре Linux основной структурой, ассоциированной с каждым процессов, является `task_struct`. Данная структура (дескриптор процесса) описывает процесс и содержит указатели на другие, связанные с процессом, структуры. Каждая структура `task_struct` включает в себя поле список процессов `tasks` типа `list_head`. Поля `tasks.next` и `tasks.prev` указывают соответственно на следующий и предыдущий

процессы в связном списке по отношению к текущему. Голова списка процессов представляет собой дескриптор `init_task` типа `task_struct`. Это дескриптор так называемого нулевого (`swapper`) процесса, всегда присутствующего в системе.

Имя процесса определяется в структуре `task_struct` полем `comm` типа `char[16]`. Дескриптор каждого пользовательский процесса в системе содержит указатель `mm` на структуру `mm_struct`. Данная структура содержит всю информацию об адресном пространстве, относящемся к процессу. В рамках поставленной в этой работе задачи интерес представляет поле `pgd` структуры `mm_struct`. Для каждого процесса оно содержит адрес соответствующего ему глобального каталога страниц (Page Global Directory). Описанная связь структур изображена на Рис. 10.

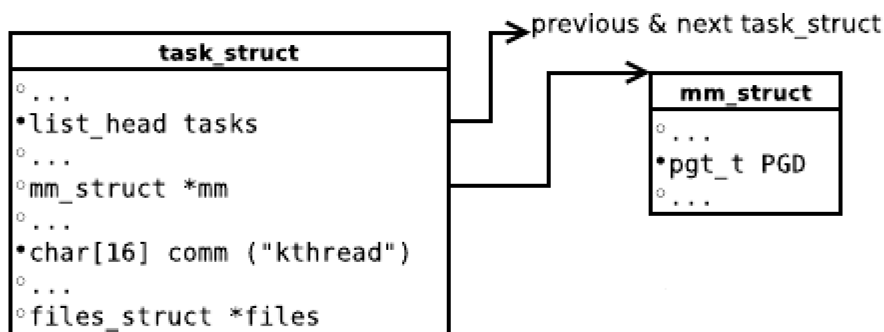

Рис. 10: Дескриптор процесса структура task_struct (из [13])

Сложность поиска структуры `task_struct` в образе памяти обусловлена тем, что для общего случая сложно подобрать устойчивые шаблоны (сигнатуры) для её идентификации. Порядок полей и их смещения внутри структур сильно зависят от версии и конфигурации ядра Linux, а также от выбранных опций компиляции. Далее описаны возможные подходы для нахождения структур `task_struct` и вычисления смещений полей `tasks`, `mm`, `comm` и `pgd`.

3.3.1. Нахождение процесса swapper

Для получения адреса дескриптора `init_task` процесса `swapper` будет использоваться таблица символов ядра. При подключении загружаемого модуля ядра в ОС Android все ссылки на глобальные символы ядра (переменные и функции) в объектном коде модуля должны быть заменены на соответствующие адреса. Для этого ядро использует специальную таблицу символов. Эта таблица хранится в секциях `__ksymtab` и `__ksymtab_strings` сегмента кода ядра, а следовательно, в первых мегабайтах физической памяти, куда загружается ядро. В таблице символов ядра присутствует символ с именем `init_task`, адрес которого равняется адресу структуры `task_struct` процесса `swapper`.

Секция `__ksymtab` состоит из подряд идущих записей размером 8 байт. Каждая запись состоит из двух адресов: адрес структуры, на которую ссылается символ, (стар-

шие 4 байт) и адрес строковой константы — имя символа (младшие 4 байт). Ниже секции __ksymtab в памяти располагаются строковые константы символов ядра — секция __ksymtab_strings. Эта секция состоит из большого количества подряд записанных строк, оканчивающихся нулевым символом, многие из которых постоянны и всегда присутствуют во всех версиях и сборках ядер Linux, что позволяет использовать сигнатурный метод для нахождения этой секции в образе памяти. После её нахождения из секции извлекаются все строковые константы с их адресами (смещениями от начала файла). Далее в файле образа памяти выше секции __ksymtab_strings перебирается смещение секции __ksymtab. Её размер выбирается равным "количество строковых констант в секции __ksymtab_strings" · 8 байт. Для того чтобы определить, что выбранная область в файле является корректной секцией __ksymtab, необходимо проверить, что она состоит из 8-байтных записей, младшие 4 байт которых являются адресами извлечённых строковых констант из найденной секции __ksymtab_strings.

Таким образом, чтобы получить адрес символа init_task, необходимо найти строковую константу "init_task\0" в секции __ksymtab_strings (её адрес будет равняться смещению от начала файла), после чего по адресу строковой константы найти соответствующую запись в секции __ksymtab и извлечь из неё адрес символа (старшие 4 байт).

Получив адрес структуры task_struct процесса swapper, можно вычислить смещение поля comm внутри структуры, так как известно, что оно равняется "swapper\0".

3.3.2. Нахождение смещения поля tasks структуры task_struct

Для получения смещения поля tasks структуры task_struct может быть применено два разных метода. Первый — метод списков, который подробно описывается в работе [13]. Данный метод заключается в переборе смещения для головы списка (структуры list_head) и проверки двусвязности и замкнутости полученного списка. Смещение для поля tasks перебирается в диапозоне от 0 до смещения поля comm. Так как структура task_struct содержит различные связные списки, такие как список дочерних процессов или процессов "братьев" (имеющих того же родителя), список максимальной длины определяет смещение поля tasks.

Второй метод для определения смещения поля tasks использует уже найденный процесс swapper и известное смещение поля comm структуры task_struct. В ОС Android следующим за процессом swapper запускается процесс init, который является предком всех пользовательских процессов. Этот процесс продолжает работать до выключения системы. Вслед за процессом init запускается поток ядра kthreadd, выполняющий функцию диспетчера потоков ядра. Потоки ядра, как и обычные пользовательские процессы, также описываются в ядре структурой task_struct. В связный список дескрипторы процессов добавляются в том же порядке, в котором запуска-

ются соответствующие им процессы, т.е. первые три элемента этого списка должны быть дескрипторами процессов с именами “swapper”, “init”, и “kthreadd” соответственно. Таким образом, для проверки корректности перебираемого смещения поля `tasks`, в определяемом им связном списке необходимо проверить совпадение имён первых трёх процессов с описанными значениями.

3.3.3. Нахождение смещения поля mm структуры task_struct

Поле `mm` структуры `task_struct` указывает на дескриптор памяти, принадлежащий данному процессу, а поле `active_mm` того же типа, что и `mm`, расположенное сразу же за ним, — на дескриптор памяти, используемый процессом, когда он выполняется. В случае обычных процессов эти два поля содержат один и тот же указатель. Однако потоки ядра не имеют своего дескриптора памяти, и их значение поля `mm` всегда равняется `NULL`. Когда поток ядра выбирается для выполнения, его значение поля `active_mm` устанавливается равным значению поля `active_mm` процесса, выполнявшегося перед этим [8].

Зная адрес дескриптора процесса `swapper` и смещение поля `tasks`, можно получить адрес дескриптора процесса `init`. Поскольку процесс `swapper` является потоком ядра, его поле mm должно содержать `NULL`. Процесс `init` является обычным процессом, поэтому его поля `mm` и `active_mm` должны быть равны и должны содержать корректный виртуальный адрес из адресного пространства ядра. Для этого проверяется, что он больше порогового значения `0xc0000000` и, что с помощью найденных таблиц страниц управления памятью (см. 3.1.2) можно произвести его отображение в физический адрес.

Стоит также отметить, что во всех версиях ядра Linux до v3.8, вне зависимости от выбранной конфигурации, выполняется следующее: по нулевому смещению структуры `mm_struct` находится указатель на структуру `vm_area_struct`, а по нулевому смещению структуры `vm_area_struct` — указатель обратно на ту же структуру `mm_struct`. Это же условие упоминается в работе [13], а также используется в инструменте Volatilitux [7] (см. 2.2.2) для нахождения смещения поля `mm`. Начиная с версии ядра v3.8 (дата выхода: 18.02.2013), смещение указателя в структуре `vm_area_struct`, ведущего обратно на структуру `mm_struct`, изменилось и стало равным 32 байт. Разумеется, во время написания работы [13] и создания инструмента Volatilitux еще не существовало ядра версии v3.8, поэтому данное условие распространялось на все версии ядер.

В процессе анализа исходных кодов структуры `task_struct`, были получены следующие возможные смещения поля `mm` относительно поля `tasks`:

- 24 байт в версиях ядер v2.6.11 – v2.6.26.8
- 8 байт в версиях v2.6.27 – v2.6.29.6

- 28 байт в версиях v2.6.30 – v2.6.37.6
- 8 или 28 байт в версиях v2.6.38 – v3.9.4

Перебор перечисленных смещений с проверкой всех описанных выше условий определяет смещение поля mm в структуре `task_struct`.

3.3.4. Нахождение смещения поля pgd структуры mm_struct

Осталось определить смещение поля `pgd` в структуре `mm_struct`. Имея в распоряжении дескриптор процесса `swapper` и известное смещение поля `tasks` в структуре `task_struct`, можно получить список дескрипторов всех процессов в системе. Для каждого перебираемого смещения поля `pgd` в структуре `mm_struct`, можно из данного списка построить новый. Каждый элемент `task` исходного списка заменить на значение `task->mm->pgd` или исключить элемент из списка, если тот не содержит адрес указателя на структуру `mm_struct` в поле `mm` (например, потоки ядра). Для корректно определённого смещения поля `pgd` новый список должен состоять по большей части из адресов, полученных алгоритмом нахождения потенциальных PGD (см. 3.1.2).

Таким образом, используя найденные смещения в структурах, файлу образа восстановленного адресного пространства процесса (см. 3.2) можно поставить в соответствие имя процесса, однако некоторые файлы, возможно, не получат имени. Эти файлы могут являться образами памяти недавно завершённых процессов, а также процессов, скрытых руткитами, так как их дескрипторы не содержатся в списке дескрипторов активных процессов в структурах ядра ОС Android. Для более точного определения и анализа скрытых процессов необходимо проводить дальнейшие исследования. Подобные исследования по обнаружению скрытых процессов в операционных системах на архитектуре x86 проведены в работе [12].

3.4. Особенности реализации в Volatility Framework

Разработанные в работе алгоритмы и методы были реализованы в виде системы плагинов на языке Python в проекте с открытым исходным кодом Volatility Framework (далее Volatility) (см. 2.2.1). На данный момент в Volatility для проведения анализа образов памяти ОС с ядром Linux необходимо для каждой конкретной версии и сборки ядра создавать специальный профиль: zip файл, содержащий информацию о структурах ядра и смещениях полей в них, а также символьную таблицу адресов переменных и функций, используемых ядром. Эту информацию Volatility использует для нахождения и разбора в образе памяти важных для криминалистического анализа данных.

Вместо этого был разработан единых динамический программный профиль, который использует описанные в работе алгоритмы и методы для автоматического опре-

деления адресов и смещений необходимых символов и структур и может применяться для любых ОС с ядром Linux на платформе ARM (архитектуры ARMv6 и ARMv7), в том числе ОС Android.

Далее приведены краткие описания разработанных команд (плагинов) для этого профиля. Также показаны примеры вывода команд для образа памяти мобильного телефона HTC Explorer, снятого с помощью модуля LiME (см. 2.1.1). Запуск команд в Volatility для снятого файла образа памяти с использованием разработанного динамического профиля в общем виде выглядит следующим образом:

```
$ ./vol.py -f ram_image_file --profile=LinuxAutoARM command_name
[command options]
```

Листинг 1: Формат запуска команд в Volatility с использованием динамического профиля для ОС с ядром Linux на платформе ARM

3.4.1. Вывод адреса символа ядра (linux_auto_ksymbol)

Команда `linux_auto_ksymbol` выводит адрес запрашиваемого символа ядра. Реализация этой команды для получения адреса использует метод, описанный в разделе 3.3.1. Имя символа передается с помощью параметра командной строки `-K` (`--ksymbol`). Пример запуска команды для получения адреса символа ядра `init_task`:

```
$ ./vol.py [...] linux_auto_ksymbol -K init_task

Volatile Systems Volatility Framework 2.3_alpha
Kernel symbol: init_task @ 0xc05d3678
```

Листинг 2: Вывод адреса символа ядра init_task

3.4.2. Вывод списка потенциальных PGD (linux_auto_dtblist)

Команда `linux_auto_dtblist` позволяет вывести физические адреса всех потенциальных PGD (Page Global Directory) в образе памяти, найденных с помощью алгоритма, описанного в разделе 3.1.2. В проекте Volatility принято другое обозначение для адреса глобального каталога страниц PGD — DTB (Directory Table Base). Пример запуска команды:

```
$ ./vol.py [...] linux_auto_dtblist

Volatile Systems Volatility Framework 2.3_alpha
DTB
```

```
----------
0x12c04000
0x12c28000
0x13844000
0x13898000
0x138dc000
[...]
0x2bdf4000
0x2be00000
0x2c1cc000
0x2c1d8000
0x2c248000
```

Листинг 3: Вывод списка адресов потенциальных PGD

3.4.3. Вывод списка процессов (linux_auto_pslist)

Команда `linux_auto_pslist` предназначена для вывода списка найденных процессов в образе памяти. Сначала выводятся процессы из связного списка, полученного с помощью метода, описанного в разделе 3.3. Вывод состоит из трёх полей: виртуального адреса дескриптора процесса (Offset), имени процесса (Name) и физического адреса глобального каталога страниц соответствующего процесса (DTB). Некоторые процессы, например, потоки ядра, не имеют адреса DTB.

После этого выводятся оставшиеся процессы (скрытые или завершённые), для которых были найдены адреса глобальных каталогов страниц с помощью команды `linux_auto_dtblist`, но не были найдены соответствующие им дескрипторы в структурах ядра. Для таких процессов известны только их DTB адреса. Пример запуска команды:

```
$ ./vol.py [...] linux_auto_pslist

Volatile Systems Volatility Framework 2.3_alpha
Offset     Name                 DTB
---------- -------------------- ----------
0xd9c28c60 init                 0x2c2bc000
0xd9c28040 kthreadd             ----------
0xd9c2ec80 ksoftirqd/0          ----------
0xd9c34080 khelper              ----------
[...]
0xd62e43e0 ndroid.systemui      0x28ee8000
0xd62e5000 .android.htcime      0x28f64000
0xd63b50e0 m.android.phone      0x28fb8000
0xd0c22500 om.htc.launcher      0x2384c000
0xd0e2e780 e.process.gapps      0x23a34000
```

```
0xd0ea6080 com.htc.bgp          0x23aac000
0xc0d0a740 loop0                ----------
0xd914d1e0 kdmflush             ----------
0xcc257080 kcryptd_io           ----------
0xc34224e0 kcryptd              ----------
0xc0c0e0a0 htcime:provider      0x1dc64000
0xc0c882c0 LocationService      0x16174000
0xd255a6c0 d.process.acore      0x138dc000
0xc54023e0 tc.android.mail      0x1eca4000
0xc12d53c0 oid.htccontacts      0x13ef8000
0xc56405e0 com.android.mms      0x1ecfc000
0xc37c02e0 htc.taskmanager      0x28c40000
0xcb265380 Smack Packet Ha      0x23a34000
---------- -------------------- 0x12c04000
---------- -------------------- 0x12c28000
---------- -------------------- 0x13844000
[...]
---------- -------------------- 0x2c1d8000
---------- -------------------- 0x2c248000
---------- -------------------- 0x2c28c000
```

Листинг 4: Вывод списка процессов (в том числе скрытых или завершённых)

3.4.4. Восстановление памяти процесса (linux_auto_dump_map)

Команда `linux_auto_dump_map` выполняет восстановление (дефрагментацию) виртуального адресного пространства выбранного процесса согласно алгоритму, описанному в разделе 3.2. Для выбора процесса необходимо указать его адрес DTB с помощью параметра командной строки `--proc-dtb`. Если этот параметр не указан, выполняется восстановление адресного пространства всех процессов. Список возможных адресов DTB может быть получен из вывода команд `linux_auto_dtblist` или `linux_auto_pslist`. С помощью необязательных параметров `-s` (`--va-start`) и `-e` (`--va-end`) можно задать начальный и конечный виртуальный адрес диапазона адресов для восстановления. Параметр `-D` (`--dump-dir`) задаёт директорию, в которую записывается выходной файл образа восстановленной памяти процесса и связанный с ним индексный файл (см. 3.2). Ниже приведён пример запуска команды для восстановления адресного пространства режима пользователя всех процессов и приведён краткий листинг выходной директории.

```
$ ./vol.py [...] linux_auto_dump_map -D ./dump_dir -e 0xc0000000
[...]

$ ls -1 ./dump_dir
DTB_0x12c28000.bin
```

```
DTB_0x12c28000.bin.index
DTB_0x13844000.bin
DTB_0x13844000.bin.index
[...]
m.android.phone_0x28fb8000.bin
m.android.phone_0x28fb8000.bin.index
mediaserver_0x2bc40000.bin
mediaserver_0x2bc40000.bin.index
ndroid.systemui_0x28ee8000.bin
ndroid.systemui_0x28ee8000.bin.index
netd_0x2bc34000.bin
netd_0x2bc34000.bin.index
[...]
```

Листинг 5: Восстановление адресного пространства режима пользователя всех процессов

4. Тестирование реализации

Для проведения тестирования программной реализации изложенных в работе алгоритмов использовались образы памяти, снятые с различных мобильных устройств под управлением ОС Android, в том числе с программного эмулятора, поставляемого в наборе инструментальных средств разработки Android SDK [14]. Для доказательства общности подхода к решению поставленных задач дополнительно в процесс тестирования был вовлечён образ памяти, снятый с мини-компьютера Raspberry Pi [15]. Это устройство на базе процессора ARM под управлением ОС Raspbian (основана на Debian GNU/Linux). Характеристики используемых для тестирования устройств приведены в Таблице 1. Образы памяти снимались с устройств в формате LiME с помощью загружаемого модуля ядра (см. 2.1.1).

Model	**ROM**	**Kernel version**	**Architecture**
HTC Explorer	Stock	2.6.38.6-g565872f	ARMv7
Sony Xperia J	Stock	3.0.8-perf	ARMv7
Android emulator	Stock goldfish 4.2.2	2.6.29	ARMv7
Android emulator	Stock goldfish 4.2.2	3.4	ARMv7
Raspberry Pi	Stock Raspbian “wheezy”	3.6.11	ARMv6

Таблица 1: Используемые для тестирования устройства

Для всех тестируемых устройств программной реализации удалось определить правильные смещения полей описанных в работе структур ядра Linux. Сравнение выводов реализованных команд `linux_auto_dtblist` и `linux_auto_pslist` показало, что разработанный алгоритм поиска потенциальных PGD способен найти как минимум все пользовательские процессы, для которых существуют соответствующие им дескрипторы в структурах ядра. В большинстве случаев он также способен найти и такие процессы, от которых в структурах ядра уже не осталось следов.

Для тестирования корректности восстановления адресного пространства процесса была написана вспомогательная программа, которая запускалась на целевом устройстве перед снятием памяти. Механизм её работы достаточно прост: она выделяет в своём виртуальном адресном пространстве несколько достаточно больших участков памяти, выровненных по границе 4 Кбайт. После выделения очередного участка памяти, она заполняет его определённым образом: первые 4 Кбайт — байтами со значением `0x00`, следующие 4 Кбайт — байтами со значением `0x01` и так далее. После заполнения очередной области размером 4 Кбайт байтами со значением `0xFF`, процедура повторяется снова, начиная со значения `0x00`. Таким образом, каждая страница, относящаяся к такому участку памяти, будет заполнена байтами с одинаковыми значениями. После выделения и заполнения всех участков памяти запущенный экземпляр вспомогательной программы переходит в режим ожидания пользовательского ввода и остаётся доступным в списке активных процессов ОС.

Далее производилось снятие образа памяти и восстановление адресного пространства процесса вспомогательной программы. Для всех устройств, приведённых в Таблице 1, разработанной программной реализации удалось корректно восстановить все выделенные специальным образом участки памяти.

Заключение

Результаты

В рамках данной дипломной работы её автором были достигнуты следующие результаты:

- Проведено исследование предметной области и описаны основные преимущества использования анализа оперативной памяти в ходе проведения криминалистической экспертизы мобильного устройства.

- Проведено исследование механизмов работы виртуальной памяти в ОС Android.

- Разработан общий, независимый от версии ОС Android и модели мобильного устройства, алгоритм нахождения всех запущенных (в том числе скрытых) процессов в образе памяти.

- Разработан алгоритм восстановления виртуального адресного пространства найденного процесса.

- Разработан общий метод нахождения структур ядра ОС Android, описывающих найденный процесс, для восстановления информации о нём, в частности, его имени. Разработанный метод применим к любой версии ядра ОС Android.

- Реализованы разработанные алгоритмы в проекте с открытым исходным кодом Volatility Framework.

- Проведено успешное тестирование и проверка корректности реализации.

Таким образом, поставленные перед автором этой работы задачи были успешно решены.

Дальнейшее развитие

Восстановленное разработанными методами адресное пространство процесса может быть полезным для дальнейшего анализа и восстановления из него структур виртуальной машины Dalvik, а также такой информации, как изображения, текстовые документы, исполняемые файлы, зашифрованные или запакованные вредоносные программы. Исследования в этом направлении представлены в работе [16].

Список литературы

[1] E. Casey, “Digital Evidence and Computer Crime, Second Edition”. Elsevier. 2004

[2] IDC, “Android and iOS Combine for 91.1% of the Worldwide Smartphone OS Market in 4Q12 and 87.6% for the Year, According to IDC”. URL: http://www.idc.com/getdoc.jsp?containerId=prUS23946013 (дата обращения: 30.05.2013)

[3] Google™ Official Blog, “The first Android-powered phone”. URL: http://googleblog.blogspot.ru/2008/09/first-android-powered-phone.html (дата обращения: 30.05.2013)

[4] J. Sylve, A. Case, L. Marziale, G. G. Richard, “Acquisition and analysis of volatile memory from android devices”. Digital Investigation, vol. 8, no. 3–4, pp. 175–184, Feb. 2012.

[5] Digital Forensics Solutions LCC, “LiME — Linux Memory Extractor v1.1”. 2012. URL: http://lime-forensics.googlecode.com/files/LiME_Documentation_1.1.pdf (дата обращения: 30.05.2013)

[6] Volatile Systems LLC, “The Volatility Framework: Volatile memory artifact extraction utility framework”. URL: https://www.volatilesystems.com/default/volatility (дата обращения: 30.05.2013)

[7] E. Girault, “Volatilitux: Memory forensics framework to help analyzing Linux physical memory dumps”. URL: https://code.google.com/p/volatilitux/ (дата обращения: 30.05.2013)

[8] DP Bovet, M. Cesati, “Understanding the Linux Kernel, Third Edition”. Sebastopol, O’Reilly Media, Inc. 2005.

[9] ARM Limited, “ARM Architecture Reference Manual”. 2005.

[10] P. Krzyzanowski, “Memory Management: Paging”. URL: http://www.cs.rutgers.edu/~pxk/416/notes/09a-paging.html (дата обращения: 30.05.2013)

[11] J. Corbet, G. Kroah-Hartman, A. Rubini, “Linux Device Drivers, Third Edition”. O’Reilly Media, Inc. 2005

[12] K. Saur, J. B. Grizzard. “Locating x86 paging structures in memory images”. Digital Investigation, vol. 7, no. 1–2, pp. 28–37, Oct. 2010.

[13] I. Kollar, “Forensic RAM dump image analyser”. Charles University in Prague. 2010.

[14] “Android Developer Tools”. URL: http://developer.android.com/tools/index.html (дата обращения: 30.05.2013)

[15] “Raspberry Pi”. URL: `http://www.raspberrypi.org/` (дата обращения: 30.05.2013)

[16] H. Macht, “Live Memory Forensics on Android with Volatility”. Friedrich-Alexander University Erlangen-Nuremberg. January, 2013.