Применение статической бинарной инструментации с целью проведения динамического анализа программ для платформы ARM

М.К.Ермаков <mermakov@ispras.ru>
С.П.Вартанов <svartanov@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. В настоящее время динамический анализ программ активно используется для контроля качества программных продуктов, задач профилирования и поиска уязвимостей. В данной работе рассматриваются особенности одного из методов обработки кода программы в рамках динамического анализа — статической инструментации исполняемого кода, подразумевающей предварительное изменение исполняемых файлов или файлов динамических библиотек. Предлагается метод статической инструментации, позволяющий обрабатывать файлы исполняемого кода в формате ELF для архитектуры ARM.

Предлагаемый метод включает возможности настройки инструментации с помощью пользовательских спецификаций, задающих точки внедрения кода и необходимый внедряемый код. В статье описываются основные шаги метода: обработка пользовательских спецификаций, разбор кода программы и создание блоков инструментационного кода по числу точек инструментации, внедрение блоков инструментационного кода в программу, модификация кода программы с целью осуществления передачи управления на инструментационный код во время выполнения программы.

Модификация файлов исполняемого кода программы проводится в рамках ограничений, накладываемых спецификациями формата ARM ELF — распределение кода, данных и управляющей информации по секциям и сегментам, обрабатываемым динамическим загрузчиком операционной системы, и дополнительными внешними и внутренними зависимостями между секциями, порождаемыми во время генерации исполняемого кода. Непосредственный метод инструментации включает замену блоков инструкций в точках инструментации на инструкции безусловного перехода на соответствующий блок инструментационного кода и перенос заменённых инструкций в блок инструментационного кода для сохранения исходной функциональности программы. Для поддержания корректности работы в блок инструментационного кода также добавляются инструкции сохранения и восстановления состояния и инструкция безусловного перехода для возврата управления после выполнения инструментационного кода. Дополнительно в статье описываются модификации, направленные на добавление информации о внешних символах, используемых в инструментационном коде.

Данные модификации необходимы для поддержания корректности работы динамического загрузчика на этапе разрешения внешних зависимостей.

В статье приведены результаты практических экспериментов по применению разработанной программной системы, реализующей предлагаемый метод. На примере задачи подсчёта базовых блоков разработанная система показала более высокую производительность по сравнению с распространённым средством инструментации Valgrind.

Ключевые слова: статическая бинарная инструментация; динамический анализ; архитектура ARM; формат ELF; Android.

DOI: 10.15514/ISPRAS-2015-27(1)-1

Для цитирования: Ермаков М.К., Вартанов С.П. Применение статической бинарной инструментации с целью проведения динамического анализа программ для платформы ARM. Труды ИСП РАН, том 27, вып. 1, 2015 г., стр. 5-24. DOI: 10.15514/ISPRAS-2015-27(1)-1.

1. Введение

1.1 Автоматический анализ программного обеспечения

В настоящее время возрастание сложности программного обеспечения и повышение степени строгости требований к качеству программного обеспечения приводят ко всё более активному использованию средств автоматического анализа в рамках цикла разработки. Средства статического анализа, обычно интегрированные в среды разработки, позволяют обнаруживать программные дефекты, а также фрагменты, не соответствующие заданным спецификациям и стандартам, непосредственно во время создания программного кода. Средства динамического анализа позволяют проводить оценку различных аспектов программы, таких как производительность, эффективность использования ресурсов и др.; подобные особенности программ являются критичными для конечного пользователя и обычно могут быть рассмотрены только во время реального выполнения программы.

Средства статического анализа обычно предполагают некоторые стандартные подходы к рассмотрению исходного кода (построение таких информационных структур, как синтаксические деревья и базы знаний по модулям и функциям кода) и ориентированы на общие задачи поиска дефектов, обнаружения шаблонов кода, проведение автоматического рефакторинга и т. д. Средства динамического анализа предлагают решения для более широкого спектра задач по исследованию программного обеспечения и отличаются разнообразием применяемых подходов и методов. Данная особенность привела к тому, что было разработано и разрабатывается значительное количество комплексных систем, предоставляющих базовую инфраструктуру для реализации пользовательских инструментов анализа, производящих извлечение релевантной информации во время выполнения программы.

1.2 Инструментация кода

В основе большой группы подобных систем лежит технология инструментации кода — изменение исходного или исполняемого кода с целью добавления функциональности или изменения имеющейся функциональности. Внедрённый таким образом код осуществляет сбор информации и/или непосредственно проводить более сложную обработку для получения результатов анализа. В число распространённых задач, эффективно решаемых данным образом, входит поиск дефектов различного рода, сбор статистики по использованию ресурсов, взаимодействие с внутренним состоянием и данными программы для извлечения другой специфичной информации.

Инструментация исполняемого кода представлена в современных средствах в значительно большей степени, чем инструментация исходного кода; это связано, в первую очередь, с более высоким уровнем организации исходного кода и, соответственно, большей сложности его обработки и модификации.

Методы инструментации классифицируются также по стадии проведения инструментации на статическую и динамическую. Статическая инструментация предполагает предварительную модификацию целевых файлов с целью получения итоговых изменённых вариантов данных файлов. Использование модифицированных файлов вместо исходных приведёт к реальному выполнению внедрённого кода. Динамическая инструментация предполагает перехват управления при выполнении программы и замену блоков кода, подаваемых на процессор, на модифицированные блоки кода с помощью некоторого программного модуля, загружаемого в виртуальную память вместе с целевой программой. Оба подхода предоставляют схожие возможности, однако имеют несколько значительных различий, обуславливающих превосходство одного или другого подхода в условиях конкретной задачи:

- Статическая инструментация производится как некоторый предварительный этап и позволяет использовать модифицированную версию целевой программы, которая может быть использована неограниченное количество раз для получения необходимых результатов на различных наборах входных данных. Накладные расходы при выполнении модифицированной версии целевой программы обычно включают в себя только затраты на непосредственное выполнение внедрённого кода.
- Динамическая инструментация производится непосредственно во время выполнения и включает в себя этап декодирования исходного исполняемого кода и кодирования модифицированного исполняемого кода из некоторого промежуточного представления; эта особенность значительно увеличивает накладные расходы на выполнение целевой программы в режиме инструментации. Несмотря на увеличенные накладные расходы, динамическая инструментация предоставляет более широкие возможности по модификации кода и доступ к текущему состоянию программы уже на этапе инструментации, что значительно

уменьшает сложность структуры дополнительного исполняемого кода для проведения необходимого анализа.

1.3 Статическая инструментация исполняемого кода

Проведение статической инструментации исполняемого кода предусматривает непосредственное изменение одного или нескольких файлов, которые будут использованы вместо исходных. Как правило, форматы, описывающие структуру исполняемых файлов, обладают довольно высокой степенью связности их содержимого (например, смещения относительных переходов между фрагментами кода, смещения констант и специфических конструкций и др.). Поэтому, в отличие от динамической инструментации, при которой инструментационный код не обязан формировать часть целостного образа выполняемой программы в памяти, обеспечение корректности итогового результата является одной из важнейших задач при реализации системы статической инструментации.

Для статической инструментации также более остро стоит проблема разбора исходного исполняемого файла. В то время как при проведении динамической инструментации присутствует значительное количество косвенных признаков, которые можно использовать для идентификации необходимых областей программы, эффективность статической инструментации напрямую зависит от количества информации, которую можно извлечь из целевых файлов.

Несмотря на указанные выше возможные факторы снижения точности статической инструментации, а также более ограниченные возможности инструментационного кода по сравнению с динамической инструментацией, статический подход значительно лучше подходит к решению задач, для которых минимизация накладных расходов является критическим требованием. Методы статической инструментации также являются более подходящим решением при наличии ограничений среды выполнения (если данные ограничения усложняют применение механизмов по перехвату и генерации кода, используемых при динамической инструментации).

1.4 Цели работы

Данная работа представляет описание системы инструментов, позволяющих проводить инструментацию исполняемых файлов и динамических библиотек в формате ELF (Executable and Linkable Format – формат объектных и исполняемых файлов, используемый семейством Unix-подобных операционных систем) для архитектуры ARM. В качестве одной из основных платформ для применения данного инструмента рассматривается система Android. Именно ряд особенностей системы Android, а также относительная ограниченность ресурсов устройств на базе архитектуры ARM являются факторами, определяющими преимущество статической инструментации.

Данная статья имеет следующую структуру:

- Секция 2 описывает общие положения задачи статической инструментации исполняемого кода в формате ARM ELF и особенности целевых файлов, требующие применения специфических механизмов обработки; данная секция также предлагает подробный обзор характеристик системы, разработанной для решения задачи статической инструментации ARM ELF,
- Секция 3 содержит оценку результатов применения полученной реализации для задачи трассировки функций и подсчёта базовых блоков; в качестве целевых объектов выступают динамические библиотеки ARM ELF.
- Секция 4 приводит краткий обзор работ в области инструментации исполняемого кода.
- Секция 5 содержит общую оценку проведённых работ и рассматривает приоритетные направления дальнейшего развития.

2. Статическая инструментация ARM ELF

2.1 Общая структура формата ARM ELF

Исполняемые файлы и динамические библиотеки в формате ARM ELF имеют модульную структуру и разбиваются на заголовок и произвольное количество секций. Секции представляют из себя именованные неразрывные блоки данных, внутренняя организация которых определяется типом секции. Информация о количестве секций, их положении в файле, размерах и управляющих флагах содержится в заголовке файла.

Количество секций и положение секций друг относительно друга не являются фиксированными параметрами, что позволяет свободно добавлять, удалять и перемещать секции внутри одного ARM ELF файла.

Секции, присутствующие в ARM ELF файле делятся на загружаемые и незагружаемые:

- Загружаемые секции объединяются в сегменты и имеют фиксированное смещение в виртуальной памяти от начала образа. Данные смещения назначаются компоновщиком на этапе сборки итогового файла и напрямую связаны с внутренними смещениями, используемыми элементами секции.
- Незагружаемые секции содержат вспомогательную информацию, используемую компоновщиками (только для объектных файлов ELF) и другими инструментами (отладчиками, инструментами анализа и т. д.)

В отличие от положения секции в файле, положение секции в сегменте и её виртуальный адрес не всегда могут быть изменены без внесения дефектов в

итоговой образ. При проведении статической инструментации целевой ARM ELF файл либо полностью преобразовывается с перерасчётом всех смещений, зависимых от положений секций в сегментах, либо производится ограниченная работа с виртуальными адресами, требующая минимальных коррекций для поддержания правильной работы итогового файла.

В рамках представляемой работы применяется второй подход, предполагающий минимизацию изменений, вносимых в имеющиеся секции.

2.2 Общая схема проведения инструментации

Отсутствие ограничений на количество, размер и содержимое секций в ARM ELF файле предоставляет возможность организации инструментации следующим образом:

1. Обработка спецификаций типа инструментационных точек и кода обработчиков инструментационных точек, задаваемых пользователем, для разметки целевого файла и генерации непосредственного инструментационного кода.
2. Компиляция инструментационного кода в объектный файл ARM ELF.
3. Добавление секций исполняемого кода и данных из полученного объектного файла в целевой файл ARM ELF в качестве дополнительных секций.
4. Изменение секции исполняемого кода целевого файла для осуществления перехвата управления и передачи его в инструментационный код в точках инструментации.
5. Расширение и модификация специализированных секций целевого файла ARM ELF, используемых динамическим загрузчиком для идентификации внешних зависимостей (только если в инструментационном коде используются внешние зависимости, которые не присутствовали в коде целевого файла).
6. Корректировка смещений и внутренних зависимостей в итоговом файле для поддержания корректности выполнения.

2.3 Подключение инструментационного кода

Первым шагом проведения статической инструментации файла исполняемого кода является добавление непосредственно инструментационного кода; передача управления на инструментационный код и его выполнение будет позволять получать необходимую информацию при работе целевой программы.

Добавление инструментационного кода в целевой ARM ELF файл осуществляется путём добавления двух новый секций в конец файла и размещения непосредственно кода и данных в эти секции. Так как инструментационный код необходимо также использовать во время выполнения, требуется изменить структуру сегментов, описывающих образ, загружаемый в виртуальную память, добавив две секции в конец последнего сегмента.

Передача контроля управления в точках инструментации осуществляется путём замены одной или нескольких инструкций, находящихся в данной точке, на инструкцию безусловного перехода. Обратная передача осуществляется аналогичным образом путём вставки перехода в конец блока инструментационного кода. Схема, представленная на рисунке 1, иллюстрирует структуру исполняемого кода при применении описанного механизма.

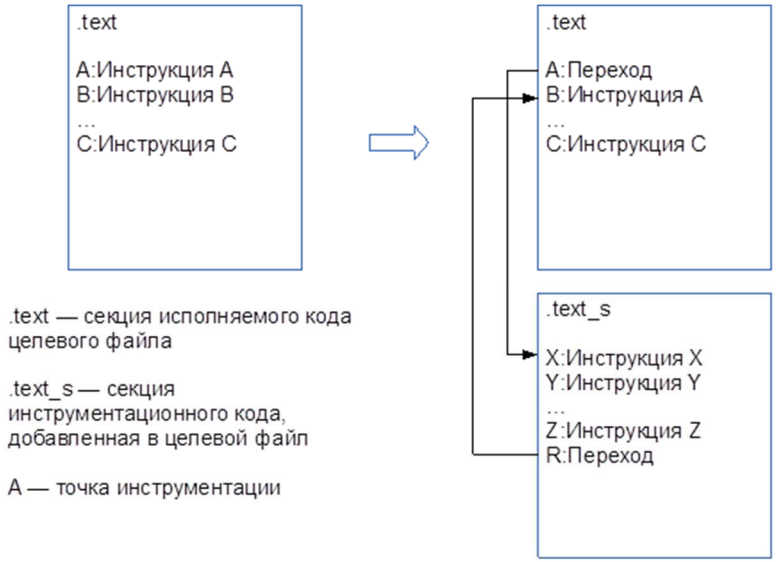

Рисунок 1: Передача управления на код инструментации

Для поддержания корректности выполнения целевой программы и обеспечения полноты инструментации инструментационный код должен удовлетворять следующим требованиям:

1. Инструментационный код должен быть функционально совместим с целевым файлом;
2. Инструментационный код должен сохранять состояние программы во избежание нарушения исходной функциональности;

2.3.1 Функциональная совместимость

В контексте данной задачи под функциональной совместимостью понимается соответствие инструментационного кода по набору инструкций для обеспечения корректного декодирования, осуществляемого процессором. Инструментационный код задаётся пользователем в виде исходного кода и должен быть скомпилирован в объектный файл. На платформе ARM существует несколько наборов инструкций, которые могут быть использованы параллельно; базовыми наборами инструкций являются ARM32 и Thumb-2. Для обеспечения функциональной зависимости производится полный разбор исполняемого кода целевого файла для выявления фрагментов, использующих конкретный набор инструкций. Генерация инструментационного кода производится таким образом, чтобы точке инструментации, находящейся в блоке инструкций ARM32, был сопоставлен блок инструкций, реализующих функциональность, необходимую пользователю, также в формате ARM32.

2.3.2 Сохранение состояния

Модифицированный целевой файл должен полностью сохранять исходную функциональность и выполнять инструментационный код исключительно как некоторый побочный эффект. В контексте схемы инструментации, описанной выше, для обеспечения корректности исходного исполняемого кода необходимо выполнение следующих условий:

1. Сохранение состояния программы после осуществления безусловного перехода на инструментационный код и до его непосредственного выполнения.
2. Восстановление состояния программы после выполнения инструментационного кода и до осуществления обратного безусловного перехода в основной исполняемый код.
3. Выполнение инструкций исполняемого кода целевого файла, заменённых на инструкцию безусловного перехода в инструментационный код.

Механизмы 1 и 2 реализуются с помощью добавления в инструментационный код инструкций сохранения регистров общего пользования и регистра флагов в стек программы и восстановление их из стека. Корректная работа со стеком внутри инструментационного кода гарантируется компилятором (блоки инструментационного кода оформляются в виде отдельных функций, что позволяет пользоваться положениями, зафиксированными в стандартном протоколе вызовов).

Механизм 3 реализуется путём добавления заменённых инструкций непосредственно в блок инструментационного кода после инструкций восстановления состояния, что гарантирует корректность выполнения данных инструкций и полное соответствие исходной функциональности целевого файла. Исключение составляют некоторые классы инструкций, зависимые от конкрет-

ного положения в виртуальной памяти (такие как, например, инструкции перехода, реализующие сдвиги относительно текущего значения счётчика инструкций). Для поддержания корректности в случае, если были заменены именно такие инструкций, необходимо провести их модификацию или заменить функционально аналогичным блоком инструкций.

Инструментационный код конструируется таким образом, чтобы иметь набор зарезервированных фрагментов, заполненных инструкциями, не имеющими эффекта. Именно на данные зарезервированные места осуществляется добавление дополнительных инструкций (сохранение состояния, заменённые инструкции целевого кода). Подробную схема трансформации инструментационного кода представлена на рисунке 2.

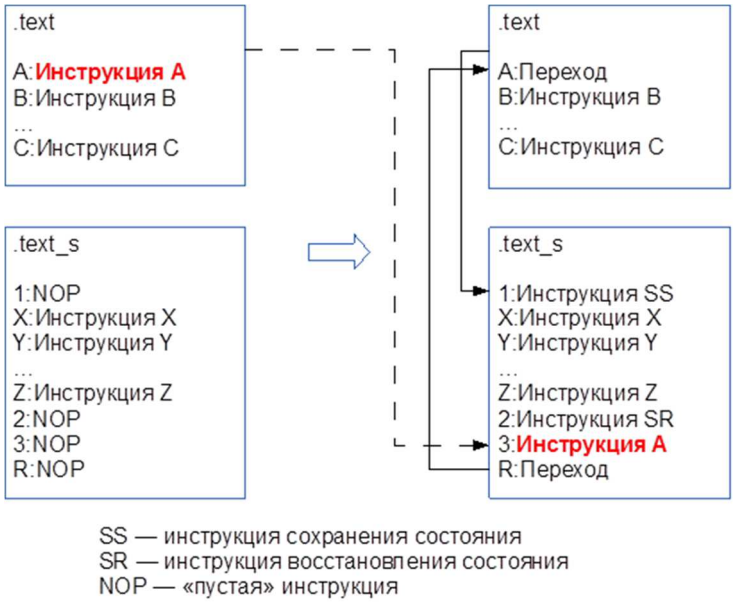

Рисунок 2: Схема структуры инструментационного кода

2.4. Подключение внешних зависимостей инструментационного кода

Одним из этапов работы динамического загрузчика операционной системы Linux (и систем, базирующихся на ней) при работе с файлами в формате ELF является разбор внешних зависимостей, заявленных загружаемым файлом и поиск необходимых ресурсов для удовлетворения данных зависимостей. Для задания внешних зависимостей и последующей работы с ними в файле формата ELF используется целый набор специфических секций, включающих в себя следующие:

1. .dynsym и .dynstr — секции, содержащие статическую информацию о внешних зависимостях: имена символов динамических библиотек и функций, контрольные флаги символов функций. Имена символов непосредственно используются динамическим загрузчиком для осуществления поиска по реестру библиотек, доступных в системе.
2. .got и .plt — секции, загружаемые в образ исполняемого файла в виртуальной памяти и используемые исполняемым кодом напрямую для осуществления перехода на инструкции в динамических библиотеках, реализующих внешние зависимости.
3. .rel.plt – секция, содержащая указания для динамического загрузчика о соответствии статической информации внешних зависимостей явным относительным смещениям в секции .plt.
4. .dynamic — секция, содержащая сокращённую информацию о наиболее важных параметрах образа целевого файла, загружаемого в виртуальную память. В данную информацию входит список внешних зависимостей по динамическим библиотекам.

Информация, заключённая в данных секциях, а также данные и код, находящиеся в секциях .got и .plt обеспечивают корректную загрузку и использование внешних библиотек во время выполнения целевого файла. Только зависимости, отражённые в данных секциях, будут обрабатываться корректно — использование в коде зависимостей, не известных динамическому загрузчику по указанным секциям, приводит к невозможности выполнения файла.

В случае, если инструментационный код имеет внешние зависимости, не отражённые в целевом файле, необходимо осуществить добавление информации о данных зависимостях в рассмотренные выше секции целевого файла. Данная задача обычно выполняется компоновщиком при создании исполняемых файлов и динамических библиотек из объектных файлов. Так как компоновщик не поддерживает работу уже с исполняемым ELF файлом, его применение невозможно.

Непосредственные модификации включают в себя следующие этапы:

1. Расширение секций .dynstr, .dynsym, .got, .plt, rel.plt и .dynamic на размер соответствующих блоков по числу внешних зависимостей инструментационного кода, отсутствующих в исходном целевом файле.
2. Конструирование блоков данных секций для работы с внешними зависимостями инструментационного кода и вставка в «пустые» фрагменты секций, появившиеся после их расширения.
3. Реорганизация таблицы сегментов и обновление данных в секции .dynamic.

В то время как непосредственное расширение и модификация секций являются исключительно техническими задачами, проведение реорганизации секций требует учёта многих особенностей отдельных секций. Как было отмечено ранее, загружаемые секции объединяются в сегменты и обычно располагаются

в данных сегментах как общий единый блок (т. е. в виртуальной памяти соседние секции не имеют «пустого» промежутка между собой). Расширение загружаемых секций (все указанные выше секции являются загружаемыми) приводит к образованию наложений в таблице сегментов. Данные наложения приводят к порче данных при загрузке образа в виртуальную память и должны быть разрешены путём сдвига секций и изменения порядка следования в образе.

Некоторые загружаемые секции (такие как секции исполняемого кода и динамических релокаций) содержат данные и управляющие конструкции, зависимые от относительного положения в образе, загружаемом в виртуальную память. Поэтому при проведении реорганизации структуры сегментов данные особенности необходимо учитывать. Для минимизации необходимых последующих корректировок больший приоритет имеют операции перемещения независимых секций. В случае, если это невозможно, производятся разбор затронутых секций и автоматическая модификация необходимых элементов.

Стандартные спецификации таблиц сегментов и секций, используемые в базовых компиляторных системах, организованы таким образом, что при практическом проведении инструментации, реорганизация секций приводит к минимальным изменениям.

2.5. Поддержание корректности ARM ELF

Этап проведения постобработки файла ARM ELF после проведения инструментации включает в себя задачи, относящиеся к корректировке смещений, используемых некоторыми инструкциями исполняемого кода. Следующие три группы инструкций рассматриваются в качестве целевых:

- Инструкции, осуществляющие обработку данных инструментационного кода;
- Инструкции перехода из секции кода A в секцию кода B, если одна из этих секций была модифицирована во время инструментации;
- Инструкции, которые были заменены на переходы в инструментационный ход и перенесены в блоки инструментационного кода.

Так как заданный пользователем исходный инструментационный код непосредственно компилируется в объектный файл, но добавляется в целевой ELF файл напрямую в виде двух секций (секции кода и секции данных), то стандартные модификации, проводимые компоновщиком для корректировки смещений из секции кода в секцию данных не производятся. Информация о позициях, в которых необходимо проставить смещение между виртуальными адресами данных секций, находится в объектном файле в секции .rel.text. В рамках проведения инструментации производится автоматический перенос информации из секции .rel.text (с учётом конкретных значений виртуальных адресов уже в целевом ELF файле) для модификации смещений в секции исполняемого кода инструментации.

Инструкции условных и безусловных переходов из секции A в секцию B используют относительные смещения, учитывающие разницу виртуальных адресов данных секций. Перенос одной из секций приводит к нарушению корректности инструкций перехода и требует автоматической коррекции.

При проведении инструментации производится перенос инструкций в точках инструментации в блоки инструментационного кода. В наборе инструкций ARM32 и Thumb-2 присутствуют типы инструкций, напрямую использующие значение регистра PC — счётчика инструкций. За счёт данного свойства результат выполнения данных инструкций зависит от их положения в образе целевого файла, загружаемого в виртуальную память; перенос инструкций из одной секции в другую может нарушить корректность их выполнения. Для устранения подобного эффекта необходимо проведение дополнительных корректировок данных инструкций для учёта разности виртуальных адресов положения в исходной секции исполняемого кода и положения в секции инструментационного кода.

Сложность модификаций, необходимых для инструкций, зависит от конкретного типа инструкций. Инструкции безусловного перехода требует простого изменения относительного смещения, закодированного в инструкцию (в некоторых случаях требуется использование альтернативных кодировок, поддерживающих более широкие диапазоны значений). Некоторые типы инструкций, такие как, например, инструкций загрузки из памяти (сохранения в память) по смещению от значения регистра PC, требует замены на блок арифметических инструкций, проводящий конструирование итогового эффективного адреса загрузки (сохранения).

3. Практические результаты

Для проведения эффективности реализации средства инструментации был проведён ряд тестов с использованием набора базовых библиотек платформы Android и набора библиотек для обработки файлов мультимедиа форматов. Тестовые запуски проводились на устройстве Pandaboard (Dual-core ARM Cortex A-9 MPCore, 1 GHz) с установленной версией системы Android 4.0.4.

При проведении первого этапа тестирования был рассмотрен ряд библиотек для работы с файлами специализированных форматов с целью извлечения необходимой пользователю информации:

- libjpeg — библиотека для работы с изображениями формата JPEG;
- libmpeg2 — библиотека для работы с видео в формате MPEG-2;
- libxml2 — библиотека для работы с файлами в формате XML;
- swftools — набор средств для работы с файлами Flash.

В рамках задачи оценки эффективности средства инструментации была проведена обработка библиотек с целью добавления кода, осуществляющего подсчёт количества базовых блоков инструкций, выполненных во время работы приложения. Практическая реализация необходимой функциональности зани-

мает 14 инструкций из набора ARM32 (учитывая инструкции, используемые базовыми механизмами инструментации) при замене одной инструкции из набора ARM32 в каждой из точек инструментации.

Для сравнения с имеющимися аналогами, предоставляющими поддержку архитектуры ARM, аналогичная по функциональности инструментация была проведена для рассматриваемых библиотек с помощью средства Valgrind.

Средние результаты запусков и информация о количестве точек инструментации по отношению к общему числу инструкций исполняемого кода в целевом файле и числу отдельных функций в составе исполняемого кода представлена в таблице 1.

Таблица 1: Инструментация с целью проведения подсчёта базовых блоков

Целевая программа	Базовое время работы, с	Время работы, с (статическая инструментация)	Точки инструментации	Время работы, с (Valgrind)
cjpeg (libjpeg)	3.965	9.944 (+150.8 %)	5649/78080/483	14.33 (+261 %)
djpeg (libjpeg)	9.135	22.168 (+142.7 %)	5649/78080/483	33.60 (+268 %)
mpeg2dec (libmpeg2)	10.469	15.493 (+48 %)	3849/44729/133	37.21 (+255 %)
xmllint (libxml2)	5.022	14.563 (+190 %)	61558/380491/3065	24.64 (+391 %)
png2swf (swftools)	39.382	47.32 (+19 %)	3157/23252/432	132.2 (+231 %)

Для оценки эффективности применения статической бинарной инструментации для проведения анализа библиотек платформы Android была проведена модификация набора базовых библиотек Android, использующихся для отображения графического интерфейса системы. В качестве целевой инструментации рассматривалось добавление функциональности по трассировке точек входа и выхода из функций, описанных в библиотеках. Замеры временных задержек проводились на одном из возможных сценариев работы с системой Android — запуске и инициализации графической оболочки.

В то время как непосредственно затраты на вставку инструментационного кода и выполнение инструкций базовых механизмов инструментации вносят минимальные задержки, отдельные аспекты организации инструментационного кода приводят к появлению более значительных накладных расходов. К основным аспектам, приводящим к замедлению работы, относятся следующие:

- запись данных в файл на устройстве;
- вычисление текущего абсолютного времени события входа или выхода из функции;
- использование средств синхронизации из стандартной библиотеки pthread для защиты доступа к внутренним буферам и ресурсам.

Таблица 2 приводит результаты измерений времени загрузки среды Android (от момента запуска устройства до внутреннего индикатора о завершении необходимой инициализации, после которой пользователь может выполнять действия с устройством). В таблице представлены следующие значения — время загрузки системы без инструментации каких-либо библиотек, время загрузки системы с использованием библиотеки, инструментированной «пустым» кодом, время загрузки системы с использованием библиотеки, инструментированной трассировочным кодом, и количество отслеживаемых событий входа/выхода из функций, зафиксированных за время выполнения.

Таблица 2: Инструментация библиотек Android

Библиотека	Загрузка, с	Загрузка, с (базовая инструментация)	Загрузка, с (целевая инструментация)	Вызовы инстр. кода
libsurfaceflinger.so	42.431	43.204 (+1.8 %)	51.706 (+21.8 %)	452880
libui.so	42.431	43.667 (+2.9 %)	58.964 (+38.9 %)	928812
libgui.so	42.431	43.364 (+2.1 %)	50.043 (+17.9 %)	406780
libEGL.so	42.431	43.105 (+1.5 %)	45.373 (+6.9 %)	129132
libGLES_android.so	42.431	43.226 (+1.8 %)	50.381 (+18.7 %)	500677

Полученные результаты укладываются в рамки ожидаемых результатов замедления при использовании инструментированных библиотек, активно применяемых несколькими ключевыми процессами системы Android. Замедление, вносимое инструментационным кодом, равномерно распределено по коду целевых библиотек и не влияет на корректность результатов профилирования библиотек по производительности отдельных функций.

4. Обзор существующих решений

Проекты систем инструментации, позволяющие пользователю создавать средства для проведения специализированного анализа, начали появляться уже в конце XX века. Некоторые из них были разработаны для определённых платформ, что позволяло использовать специфические особенности данных платформ для повышения гибкости и эффективности инструментов; современные системы, наиболее широко применяемые при разработке программ, покрывают целый ряд архитектур и основаны на относительно общих принципах работы с целевым кодом. Среди инструментов статической инструментации

следует отметить такие проекты, как ATOM[1], EEL[2], Etch[3], BIRD[4], PEBIL[5] и Dyninst[6]. Для проведения анализа на основе динамической инструментации широко применяются инструменты Pin [7], Valgrind [8] , DynamoRIO [9] и Dyninst.

Система ATOM (Analysis Tools with OM) была разработана для процессорной архитектуры DEC Alpha и предоставляла возможности для анализа целевых программ, доступных в виде объектных модулей. Система включала в себя интерфейс генерации кода, обращения к данным и работы с потоком управления целевой программы. Данный интерфейс позволял создать непосредственно код инструмента анализа, который впоследствии будет взаимодействовать с целевой программой, а также разметить объектные модули целевой программы для обозначения точек инструментации, т.е. точек в которых происходит выполнение инструментационного кода. ATOM осуществляет вставку инструкций инструментационного кода в указанные точки и, используя систему работы с объектными модулями OM, создаёт итоговый исполняемый файл. При запуске полученного файла исходная целевая программа работает с инструментом, заданным пользователем, параллельно в рамках одного процесса на устройстве. На основе системы ATOM были разработаны инструменты профилирования кэша и динамической памяти, счётчики инструкций и вызовов функций, а также система повышения эффективности предсказания переходов.

В рамках проекта EEL была разработана система, предоставляющая возможность внесения новой функциональности и изменения имеющейся функциональности программного кода, на основе работы со структурными компонентами — функциями, базовыми блоками и отдельными инструкциями, и комплексными производными сущностями, такими как граф потока управления. Данные компоненты в рамках инструмента обобщались под общим термином «абстракция». Абстракции позволяли производить архитектурно-независимую инструментацию — инструмент анализа, разработанный пользователем, также был выражен на языке абстракций и операций с ними; кодирование и декодирование в исполняемый код конкретной архитектуры выполнялось автоматически на основе описания языка соответствия. На основе системы EEL был реализован ряд инструментов профилирования и трассировки в рамках процессорной архитектуры SPARC.

Инструменты Etch и BIRD были разработаны для проведения инструментации программ на платформе Windows/x86.

Система Etch предоставляет модуль обхода программного кода с целью выявления иерархической структуры (модуль, функция, блок инструкций, инструкция) и разметки точек инструментации. Код инструмента анализа оформляется в виде обработчиков, применяющихся к отдельным узлам выявленной структуры. В результате выполнения инструментации целевой исполняемый файл полностью трансформируется для включения дополнительной функциональности. Помимо стандартных применений методов инструментации (профилирование и трассировка), Etch предлагает возможности модификации исходного исполняемого кода с целью оптимизаций (например, путём перестановки инструкций для повышения эффективности конвейера).

Система BIRD осуществляет инструментацию, минимизируя количество изменений, которые необходимо внести в целевой исполняемый код. Инструментационный код оформляется в виде динамической библиотеки, вызовы к которой вставляются в обрабатываемый файл путём замены инструкций (в тех местах, в которых замены возможны). Для разбора исполняемого кода используется статический декодер, производящий разметку точек инструментации. BIRD также предоставляет модуль динамического анализа для покрытия тех фрагментов кода, которые не удалось разметить статически. На основе системы BIRD был разработан инструмент, осуществляющий внедрение модуля защиты от несанкционированного изменения кода во время выполнения.

Инструмент PEBIL является одним из наиболее близких аналогов системы, разрабатываемой в рамках данной работы. PEBIL нацелен на работу с исполняемыми файлами и динамическими библиотеками в формате ELF для архитектур x86 и x86_64. Инструмент PEBIL производит расширение секций ELF в целевом файле, тем самым создавая пространство для внесения дополнительного кода. В коде целевой программы инструкции в точках инструментации заменяются на инструкции перехода на дополнительный код, откуда в свою очередь производится вызов методов динамической библиотеки, включающей непосредственную реализацию кода инструментации (однако присутствуют и ограниченные возможности по использованию прямых ассемблерных вставок). Инструмент PEBIL применяется в рамках нескольких систем оценки производительности работы программы (подсчёт статистики вызовов функций и блоков инструкций определённого типа, предсказание переходов).

Система Dyninst предоставляет возможности как статической, так и динамической инструментации исполняемого кода. Реализация подхода статической инструментации аналогична инструменту PEBIL — внедрение дополнительных фрагментов кода, замена инструкций на инструкции перехода и перенос заменённых инструкций для выполнения после инструментации. В отличие от PEBIL Dyninst максимизирует размер заменяемого блока инструкции с целью минимизации секций кода, которые необходимо расширить.

Системы Dyninst, Valgrind, Pin и DynamoRIO предоставляют широкие возможности по проведению динамической инструментации бинарного кода. Специализированные инструменты анализа разрабатываются на основе интерфейсов, предоставляемых данными системами для обработки кода в некотором внутреннем представлении. При выполнении программы блоки инструкций переводятся в данной представление ядром системы и передаются инструменту анализа, который осуществляет трансформацию кода для получения нужной функциональности. Изменённый блок инструкций декодируется обратно в машинное представление и передаётся на выполнение процессору.

5. Заключение

Реализованный в рамках проекта комплекс инструментов предоставляет пользователю возможность провести статическую бинарную инструментацию с указанием типа точек перехвата управления; дополнительная функциональность, которая будет выполняться во время выполнения также задаётся пользователем в форме исходного кода на языке C. Полнота реализации инструмента сравнима с рассмотренными аналогами, проводящими статическую инструментацию исполняемого кода. Инструмент предоставляет поддержку файлов исполняемого кода в формате ELF для платформы ARM; поддержка данной архитектуры отсутствует у прямых рассмотренных аналогов и представлена только среди инструментов динамической инструментации.

Разработанная система была применена в рамках задачи анализа производительности набора нативных библиотек платформы Android путём добавления трассирующего инструментационного кода в точки входа и выхода из внутренних функций библиотек. Дополнительно система инструментации была использована при анализе реализации подсистемы Android Binder, осуществляющей межпроцессное взаимодействие, путём трассировки внутренних функций библиотеки реализации Binder и извлечения параметров вызовов функций во время выполнения.

5.1 Направления дальнейших исследований

В настоящее время проводятся дополнительные исследования по расширению гибкости системы инструментации и оценка эффективности применения системы для реализации произвольных инструментов динамического анализа. В качестве прямых задач по развитию функциональности системы рассматриваются следующие:

- Расширение набора поддерживаемых типов точек инструментации (дополнительные типы инструкций, поддержка замены блоков).
- Расширение встроенных механизмов работы с внутренним состоянием программы и обработки инструкций (поддержка пользовательского интерфейса, доступного при разработке кода инструментации для таких целей как доступ к параметрам функций, извлечение явных значений операндов инструкций и т. д.).
- Оптимизация базовых механизмов обработки конкретных инструкций и блоков исполняемого кода, используемых в специализированных секциях (например, секции .plt) для упрощения добавления поддержки наборов инструкций отличных от ARM архитектур.

Практические результаты применения методов статической инструментации по сравнению с методами динамической инструментации, представленные в работах [5], [10] и в данной статье, показывает превосходство методов первой группы по производительности и величине накладных расходов. При этом современные средства динамической инструментации предоставляют более

широкие возможности по разработке инструментационного кода за счёт простоты встраивания дополнительной функциональности уже во время исполнения и отсутствия необходимости проводить комплексную корректировку целевого файла. На основе данных утверждений можно предположить, что поддержка дополнительных механизмов, которые позволят достигнуть уровня гибкости динамической инструментации при проведении статической инструментации, является в значительной степени важной задачей. Указанные выше направления развития инструмента посвящены решению данной задачи.

Список литературы

[1]. Srivastava Amitabh, Eustace Alan. ATOM: A System for Building Customized Program Analysis Tools, WRL Research Report 94/2, Western Research Laboratory, Palo Alto, CA, USA (http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-94-2.pdf)

[2]. Larus James R., Scnharr Eric. EEL: Machine-Independent Executable Editing. PLDI '95 Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation, 1995, pp. 291-300.

[3]. Romer Ted, Voelker Geoff, Lee Dennis, Wolman Alec, Wong Wayne, Levy Hank, Bershad Brian, Chen Brad. Instrumentation and Optimization of Win32/Intel Executables Using Etch. Proceedings of the USENIX Windows NT Workshop, 1997.

[4]. Susanta Nanda, Wei Li, Lap-Chung Lam, Tzi-cker Chiueh. BIRD: Binary Interpretation using Runtime Disassembly. International Symposium on Code Generation and Optimization, 2006. doi:10.1109/CGO.2006.6

[5]. Laurenzano Michael A., Tikir Mustafa M., Carrington Laura, Snavely Allan. PEBIL: Efficient static binary instrumentation for Linux. 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS), 2010, pp. 175-183. doi:10.1109/ISPASS.2010.5452024

[6]. Miller Barton P. and Bernat Andrew R., Anywhere, Any Time Binary Instrumentation, ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE), Szeged, Hungary, 2011, pp. 9-16. doi: 10.1145/2024569.2024572

[7]. Luk Chi-Keung, Cohn Robert, Muth Robert, Patil Harish, Klauser Artur, Lowney Geoff, Wallace Steven, Reddi Vijay Janapa, Hazelwood Kim. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, 2005, pp.190-200. doi:10.1145/1065010.1065034

[8]. Nethercote Nicholas and Seward Julian. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), San Diego, California, USA, 2007, pp. 89-100. doi:10.1145/1250734.1250746

[9]. Bruening Derek L. Efficient, Transparent, and Comprehensive Runtime Code Manipulation. Doctor of Philosophy Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

[10]. Hazelwood Kim, Klauser Artur. A Dynamic Binary Instrumentation Engine for the ARM Architecture. Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems (CASES'06). New York, NY, USA, 2006, pp. 261-270

Dynamic Analysis of ARM ELF Shared Libraries Using Static Binary Instrumentation

M.K. Ermakov <mermakov@ispras.ru>
S.P. Vartanov <svartanov@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, Russia, 109004.

Abstract. Dynamic program analysis is a prominent approach towards software quality control allowing to perform automatic profiling, defect detection and other activities during software development. In this paper we focus on static binary code instrumentation – a technique to automatically modify program executable code in order to extract data necessary for dynamic analysis. We discuss the key features of this technique within context of dynamic analysis and propose a method to perform static binary code instrumentation for ELF executable and shared library files specifically targeting the ARM architecture.

We describe the main steps of the proposed method including the following: instrumentation specification and target code parsing, executable instrumentation code generation and finally target executable code file modification in order to insert instrumentation code and ensure that control transfer from original code to instrumentation code and vice versa will happen at runtime.

Executable code file modification is performed within bounds of ARM ELF specifications and is designed to minimize the changes introduced in actual executable code blocks. Instrumentation code is appended to target files as a set of separate sections; we implement control transfer to instrumentation code through unconditional jump instructions which replace small blocks of original instructions at instrumentation points. In order to preserve the original functionality we wrap instrumentation code blocks with instructions that save and restore program state; additionally, instructions replaced at instrumentation points are transferred to the instrumentation code blocks. We also describe a set of modifications performed in order to introduce instrumentation code external dependencies to the target executable files.

The proposed method was implemented in an instrumentation framework. We provide a brief overview of practical experiments using basic block counting and function entry/exit tracing as base instrumentation applications. The results show better performance in comparison to popular dynamic instrumentation framework Valgrind and low overhead for system-wide tracking of native Android libraries.

Key words: static binary instrumentation; dynamic analysis; ARM architecture; ELF format; Android.

DOI: 10.15514/ISPRAS-2015-27(1)-1

For citation: Ermakov M.K., Vartanov S.P. Dynamic Analysis of ARM ELF Shared Libraries Using Static Binary Instrumentation. Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 1, 2015, pp. 5-24 (in Russian). DOI: 10.15514/ISPRAS-2015-27(1)-1

References

[11]. Srivastava Amitabh, Eustace Alan. ATOM: A System for Building Customized Program Analysis Tools, WRL Research Report 94/2, Western Research Laboratory, Palo Alto, CA, USA (http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-94-2.pdf)

[12]. Larus James R., Scnharr Eric. EEL: Machine-Independent Executable Editing. PLDI '95 Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation, 1995, pp. 291-300.

[13]. Romer Ted, Voelker Geoff, Lee Dennis, Wolman Alec, Wong Wayne, Levy Hank, Bershad Brian, Chen Brad. Instrumentation and Optimization of Win32/Intel Executables Using Etch. Proceedings of the USENIX Windows NT Workshop, 1997.

[14]. Susanta Nanda, Wei Li, Lap-Chung Lam, Tzi-cker Chiueh. BIRD: Binary Interpretation using Runtime Disassembly. International Symposium on Code Generation and Optimization, 2006. doi:10.1109/CGO.2006.6

[15]. Laurenzano Michael A., Tikir Mustafa M., Carrington Laura, Snavely Allan. PEBIL: Efficient static binary instrumentation for Linux. 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS), 2010, pp. 175-183. doi:10.1109/ISPASS.2010.5452024

[16]. Miller Barton P. and Bernat Andrew R., Anywhere, Any Time Binary Instrumentation, ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE), Szeged, Hungary, 2011, pp. 9-16. doi: 10.1145/2024569.2024572

[17]. Luk Chi-Keung, Cohn Robert, Muth Robert, Patil Harish, Klauser Artur, Lowney Geoff, Wallace Steven, Reddi Vijay Janapa, Hazelwood Kim. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, 2005, pp.190-200. doi:10.1145/1065010.1065034

[18]. Nethercote Nicholas and Seward Julian. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), San Diego, California, USA, 2007, pp. 89-100. doi:10.1145/1250734.1250746

[19]. Bruening Derek L. Efficient, Transparent, and Comprehensive Runtime Code Manipulation. Doctor of Philosophy Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

[20]. Hazelwood Kim, Klauser Artur. A Dynamic Binary Instrumentation Engine for the ARM Architecture. Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems (CASES'06). New York, NY, USA, 2006, pp. 261-270