Динамический анализ исполняемого кода в формате ELF на основе статической бинарной инструментации

Михаил Ермаков
Институт Системного Программирования РАН
Москва, Россия
mermakov@ispras.ru

***Аннотация*—Статья посвящена возможностям применения статической инструментации исполняемых файлов и файлов динамических библиотек в формате ELF для проведения динамического анализа программ. В статье приводится обзор существующих решений в данной и смежных областях и предлагается подход к обработке бинарных файлов ARM ELF. Описываются основные особенности предлагаемого подхода: разметка целевых файлов и генерация инструментационного кода на основе пользовательских спецификаций, использование промежуточного языка описания инструкций ассемблера ARM, компоновка итоговых инструментированных файлов исполняемого кода. Приводятся результаты применения реализации данного подхода для создания модулей трассировки помеченных данных, используемых в рамках инструмента автоматического поиска дефектов Avalanche.**

Ключевые слова—динамический анализ, статическая бинарная инструментация, ELF, платформа ARM

I. Введение

В настоящее время существует большое количество систем и инструментов, производящих автоматическую и полуавтоматическую обработку программного обеспечения для достижения одной или нескольких целей, таких как:

- проведение анализа целевой программы на наличие критических дефектов и несоответствия заявленной функциональности;
- сбор статистических данных о выполнении программы с целью выявления блоков, осуществляющих неэффективное использование системных и временных ресурсов, для проведения последующей оптимизации данных блоков;
- создание блоков защиты от взлома и раскрытия используемых алгоритмов и обратная данной задача — извлечение трасс выполнения, распознавание внутренней организации программы с целью идентификации используемых алгоритмов.

Приведённые примеры взаимодействия с программным обеспечением относятся в первую очередь к аспектам времени выполнения целевой программы. На практике реализация подобной функциональности подразумевает произведение дополнительных действий во время работы целевой программы — либо на уровне внешнего системного инструмента, имеющего возможность отслеживать и влиять на поведение программы, либо на уровне модифицированного исполняемого кода программы или исполняемого кода, внедрённого в программу. Для программного обеспечения, выполняющегося с помощью промежуточных средств (виртуальные машины, интерпретаторы, эмуляторы), добавляется возможность взаимодействия с целевой программой на уровне данных промежуточных средств. Большинство системных инструментов осуществляют внешний контроль за выполнением целевой программы (отслеживание состояния процесса программы, использования выделенных ресурсов, трассировка входных и выходных данных). Данный подход обеспечивает минимизацию влияния на исполнение целевой программы, однако предоставляет ограниченные возможности извлечения внутренних данных программы для проведения анализа. Это в свою очередь повышает сложность самого анализа, если необходимо учитывать конкретные аспекты исполняемого кода, такие как параметры структурных компонент (инструкций, управляющих блоков, функций и т.д.).

Средства инструментации приводят к прямой модификации исполняемого кода целевой программы — дополнительные и изменённые блоки инструкций выполняются непосредственно в контексте её процесса. Инструментация влияет на различные аспекты исследуемой программы — размер областей виртуальной памяти, выделенной под исполняемый код и данные, время работы отдельных функций, взаимодействие с динамической памятью и другими ресурсами. Необходимость гарантировать отсутствие значительного влияния на аспекты, оценка которых является основной целью инструмента анализа, — один из наиболее важных ограничивающих факторов подхода инструментации. Тем не менее, повышенный уровень внедрения в целевую программу позволяет организовывать значительно более точный анализ по сравнению с системными средствами.

A. Средства инструментации

Среди средств инструментации можно выделить три основные группы:

1) Статическая инструментация исходного кода

Статическая инструментация исходного кода — автоматическая модификация или расширение исходного кода по заданным спецификациям. Данный подход в первую очередь требует наличия исходного кода и возможности встраиваться в процесс компиляции и сборки программы. Проведение подобной инструментации требует мощного статического анализа для построения модели исходного кода; дополнительно при изменении и добавлении кода необходимо избегать проблем внешних и внутренних зависимостей, конструкций условной компиляции (если таковые поддерживаются) и особенностей компилятора и сборщика. Шаг оптимизации может также нарушить логику инструментационного кода. Тем не менее, инструментация исходного кода позволяет изменять целевые программы для их запуска на разных платформах без необходимости явной поддержки данных платформ (за исключением особенностей компилятора и сборщика).

2) Динамическая инструментация исполняемого кода

Динамическая инструментация исполняемого кода — внедрение в процесс выполнения программы с целью перехвата блоков инструкций и их модификации перед выполнением на процессоре. Модификация блока инструкций требует проведения декодирования инструкций и обратного кодирования во время работы программы под контролем системы инструментации, что вносит дополнительные накладные расходы. Тем не менее, при использовании систем динамической инструментации достаточно лишь контролировать целостность блоков инструкций, а не всего образа исполняемого файла или библиотеки. Размер блоков может быть выбран достаточно малым, чтобы затраты на подобный контроль были минимальными. При этом сохраняется возможность осуществлять доступ к элементам исходного образа исполняемого кода и данных, загруженного в виртуальную память, так как он не изменяется системой инструментации.

3) Статическая инструментация исполняемого кода

Статическая инструментация исполняемого кода — модификация и пересборка исполняемого файла или файла библиотеки для получения целостной версии данного файла с дополнительной функциональностью. В отличие от систем динамической инструментации, статические системы работают не напрямую с блоками инструкций, а с цельными образами, возможно содержащими управляющие структуры, исполняемый код и данные программы. Это приводит к увеличению сложности декодирования блоков целевых файлов и к значительному ограничению возможностей модификации — изменение инструкций внутри блока может приводить к нарушению работы других инструкций блока и корректности информации в управляющих структурах. Обработка защищённых (например, с помощью упаковки) и обфусцированных файлов исполняемого кода с помощью систем статической инструментации крайне затруднена, а программы, осуществляющие генерацию и изменение собственного кода во время выполнения, в общем случае не могут быть обработаны в полном объёме. Тем не менее, этапы декодирования и кодирования производятся один раз, что даёт значительное снижение накладных расходов при множественных запусках целевой программы. Дополнительно, при проведении инструментации существует возможность рассматривать файлы целевой программы для выявления параметров глобальных зависимостей и оптимизации инструментационного кода для связанных блоков инструкций.

B. Структура статьи

В рамках данной статьи будет рассматриваться подход к проведению статической инструментации исполняемого кода и особенности текущей реализации данного подхода для файлов в формате ARM ELF. В секции 2 будут рассмотрены существующие аналоги и другие разработки в данной области. Секция 3 содержит описание этапов инструментации, проводимых в представляемом подходе. В секции 4 приведено краткое описание архитектуры реализации системы инструментации и результаты применения инструментов, разработанных в рамках системы для задачи автоматического поиска дефектов. В секции 5 приводится краткое заключение по представленному материалу и рассматриваются перспективные направления дальнейших исследований.

II. Обзор систем инструментации

Для большинства существующих систем инструментации центральным принципом является предоставление возможностей по проведению настраиваемой обработки целевых программ — блоки исполняемого кода, в которых производится модификация или внедрение дополнительного кода, а также функциональная логика необходимых изменений задаются разработчиком модуля инструментации. Основными факторами, ограничивающими область применимости систем инструментации, являются следующие:

- Поддерживаемые машинные архитектуры и специфические наборы инструкций для данных архитектур;
- Поддерживаемые форматы организации файлов исполняемого кода;
- Предоставляемая разметка точек и блоков инструментации;
- Тип организации инструментационного кода;
- Ограничения, накладываемые на целевые файлы исполняемого кода.

A. Ранние системы инструментации

Ранние системы инструментации создавались для специфических архитектур и технологий и в основном не применимы для использования на данный момент из-за отсутствия поддержки со стороны авторов или устаревания целевых платформ. Тем не менее, данные проекты представляют значительный интерес за счёт развития области и формулирования основных подходов.

Система ATOM [1] была разработана для архитектуры DEC в связке с системой управления и организации исполняемого кода OM [2]. Данная система одной из первых предоставляла пользователю возможности описания логики инструментации, в то время как

непосредственно процесс инструментации выполнялся внутренним модулем ATOM. Дополнительно пользователю необходимо было осуществить разметку файлов целевой программы для обозначения точек вставки дополнительного кода.

Авторы системы EEL [3] сфокусировали внимание на возможности применения структурных элементов исполняемого кода целевой программы (инструкций, блоков инструкций, функций) в качестве точек инструментации. Это позволило значительно упростить процесс разметки целевого кода и спецификации логики модуля инструментации. Дополнительно, подход, используемый в системе EEL, обеспечивал гибкую структуру для поддержки различных архитектур — кодирование и декодирование целевого кода и генерацию инструментационного кода осуществлял специфический модуль, реализующий некоторый общий интерфейс.

Среди систем, разработанных под целевые архитектуры, широко используемые в данный момент, можно выделить BIRD [4] (Windows/x86) и BitRaker Anvil [5] (Linux/ARM).

Подход к инструментации в системе BIRD заключался во вставке инструкций вызова внешних функций в исполняемый код целевой программы и сборка инструментационного кода в виде динамической библиотеки, предоставляющей реализацию данных внешних функций.

Система BitRaker Anvil была основана на использовании кросс-платформенного анализа — в исполняемый код целевой программы для платформы ARM вставлялись инструкции вызова внешней функции. При работе целевой программы в среде эмуляции, функционирующей в рамках устройства x86, вызовы инструментационного кода осуществлялись в процессе среды эмуляции, что значительно повышало скорость проведения анализа.

B. Современные системы инструментации

Более современные системы инструментации активно используются для разработки отдельных модулей или полноценных инструментов анализа. Существует несколько направлений, разработки в которых интегрируются в системы инструментации для повышения их точности, эффективности и гибкости:

- Развитие интерфейса инструментации, доступного пользователю для описания логики необходимого модуля инструментации.
- Поддержка большого количества целевых платформ и разработка специфических оптимизаций для повышения эффективности работы инструментов на данных платформах.
- Повышение точности алгоритмов декодирования фрагментов данных, управляющих структур и кода целевых программ и разработка алгоритмов извлечения дополнительных данных о связях и особенностях объектов целевых программ для использования инструментом анализа.
- Оптимизация генерируемого инструментационного кода.

Среди наиболее значимых систем статической инструментации можно выделить MAQAO [6, 7], PEBIL [8] и Dyninst [9], предоставляющих возможности работы с файлами исполняемого кода для платформ Linux/x86 (PEBIL, Dyninst), Linux/x86-64 (MAQAO, PEBIL, Dyninst) и Linux/ppc, Linux/ppc64 (Dyninst).

Система MAQAO изначально была разработана для разбора исполняемого кода и внедрения модулей регистрации событий для профилирования и оценки эффективности работы компонентов программы. В дальнейшем был разработан язык спецификации точек инструментации MIL [7] и модуль генерации инструментационного кода для поддержки возможности разработки дополнительных инструментов анализа. Важной особенностью системы MAQAO является высокая точность и корректность работы с программами, активно использующими параллельные вычисления — именно для анализа подобных программ было разработано ядро системы.

Система PEBIL производит разбор исполняемых файлов целевой программы, модификацию секций кода и управляющих структур с целью подготовки к инструментации и добавляет инструментационный код в виде дополнительной секции. Блоки инструкций в точках инструментации заменяются на инструкции перехода в добавленные секции. Система PEBIL поддерживает генерацию инструментационного кода в виде внешней динамической библиотеки или коротких блоков инструкций, не требующих подключения дополнительных библиотек. В системе используется двухпроходный декодер инструкций и модуль оптимизации, осуществляющий минимизацию количества инструкций, не относящихся к логике инструмента анализа, но необходимых для корректной передачи управления на инструментационный код.

Система Dyninst предоставляет широкий интерфейс описания инструментационного кода и спецификаций точек инструментации. Внутренние модули обеспечивают произведение необходимых модификаций, причём возможно проведение как и статической инструментации на основе полномасштабного разбора файлов целевой программы, так и динамической инструментации на основе стандартного подхода перехвата и модификации блоков инструкций.

III. Инструментация ARM ELF

Разработка подхода, описанного в рамках данной статьи, была обоснована отсутствием систем статической инструментации, предоставляющих поддержку архитектуры ARM на базе операционной системы Linux (перспективные целевые платформы — Android и Tizen). Востребованность подхода статической инструментации продиктована возможностями разработки модулей динамического анализа, осуществляющегося на множестве входных данных и путей выполнения для обеспечения достаточного покрытия целевой программы, — при проведении подобного анализа накладные расходы

статической инструментации меньше, чем накладные расходы динамической инструментации [8].

Среди основных принципов разрабатываемого подхода, определяемых целевыми задачами динамического анализа, можно выделить следующие:

- Фокус на инструментации структурных единиц исполняемого кода (инструкций) для сбора подробной трассы, включающей типы и аргументы совершаемых операций.
- Поддержка создания инструментационного кода в виде исходного кода на языке C/C++ с возможностью использования прямых ассемблерных вставок.
- Поддержка возможности задания областей исполняемого кода целевой программы на основе фильтрации по функциям и модулям.

A. Этапы инструментации

1) Разбор исполняемого кода и генерация инструментационного кода

Первым этапом процесса инструментации является обработка пользовательских спецификаций и проход декодера файлов исполняемого кода целевой программы. Язык задания спецификаций позволяет описывать целевые типы точек инструментации, набор фильтров для каждой типа и непосредственно инструментационный код, выполнение которого необходимо в точках инструментации. В числе поддерживаемых типов точек инструментации выступают следующие:

- Структурные точки инструментации, задающие все инструкции определённой функциональной группы (арифметические и логические инструкции, инструкции доступа к памяти, вызова процедур и т.д.).
- Модульные точки инструментации, задающие позицию в некотором логическом блоке (входные и выходные инструкции функции, входные инструкции базовых блоков, блоки условного выполнения).

Задаваемые фильтры обеспечивают возможность осуществлять инструментацию точек определённого типа внутри отдельных функций или групп функций. Язык описания фильтров поддерживает групповые символы для возможности использования префиксов (например, для работы с функциями классов C++).

Задание инструментационного кода осуществляется путём написания коротких блоков исходного кода на языке C/C++, которые автоматически компилируются в исполняемый код, который впоследствии внедряется в файлы целевой программы. Существует возможность использования ассемблерных вставок, глобальных переменных (с помощью задания заголовка, общего для всех блоков) и внешних зависимостей (для которых необходимо задать имя подключаемой библиотеки).

Блоки инструментационного кода привязаны к инструкциям, находящимся в точках инструментации. При написании инструментационного кода существует возможность использования специального интерфейса, предоставляющие операции над инструкциями двух типов:

- Операции доступа к статическим элементами инструкций, таким как номера регистров операндов, значения константных операндов, флаги, специфичные для отдельных групп инструкций.
- Операции доступа к динамическим элементами инструкций, таким как значения регистров и содержимого ячеек памяти, значения регистра флагов для работы с блоками условного выполнения и т.д.

Реализация операций первого типа настроена таким образом, чтобы приводить к генерации явных константных значений в исходном инструментационном коде. Работа блока оптимизации при компиляции инструментационного кода позволяет понизить накладные расходы по осуществлению доступа к статическим элементам. Для реализации операций второго типа производится генерация блока инструкций, выполнение которого приведёт к вычислению необходимых значений при выполнении инструментационного кода.

2) Подготовка файлов целевой программы к инструментации

Вторым этапом инструментации является изменение структуры файлов целевой программы в рамках формата ARM ELF для получения корректных образов. В список операций, проводимых на данном этапе, входят следующие:

- Копирование исходной структуры файлов и добавление дополнительных секций инструментационного кода и данных.
- Перемещение и расширение ряда секций, содержащих управляющую информацию и исполняемый код, для добавления внешних зависимостей, указанных в инструментационном коде.
- Добавление исполняемого кода и заполнение управляющих структур в расширенных областях секций согласно стандарту ARM ELF и протоколам генерации кода, используемых компиляторами и сборщиками, для корректной работы загрузчика при выполнении инструментированных исполняемых файлов и библиотек.

3) Инструментация подготовленных образов

Непосредственная инструментация заключается в замене одной инструкции или блока инструкций в указанных точках на инструкцию перехода в секцию инструментационного кода, добавленную на предыдущем этапе. Последующие действия связаны с обеспечением корректности модифицированного кода путём вставки дополнительных инструкций сохранения состояния программы и восстановления заменённых инструкций.

Сгенерированный инструментационный код состоит из набора обособленных сегментов, каждый из которых

соответствует точке инструментации. При этом сегменты имеют блочную структуру, разделённую на 4 области: начальный «пустой» блок, состоящий из инструкций, не имеющих никаких побочных эффектов, непосредственно блок кода, реализующий необходимую для анализа функциональность, дополнительный «пустой» блок и блок данных, используемый инструментационным кодом. Начальный и дополнительный «пустые» блоки имеют размер, определяемый особенностями инструкций в точке инструментации, и используются для вставки служебных инструкций.

В процессе инструментации в первый «пустой» блок осуществляется вставка инструкций сохранения регистров, изменяемых целевым блоком, во второй «пустой» блок осуществляется вставка инструкций восстановления регистров и вставка инструкций кода, которые были заменены в точке инструментации на инструкцию перехода. Последней во второй «пустой» блок вставляется инструкция возвратного перехода в основную секцию кода на позицию после точки инструментации. При наличии двух смежных точек инструментации с целью снижения накладных расходов вместо возвратного перехода может осуществляется переход на следующий блок инструментационного кода. Пример структуры кода представлен на рис. 1, показывающем три блока инструкций: исходный блок инструкций, содержащий точку инструментации, блок инструкций после вставки перехода на инструментационный код и соответствующий блок инструментационного кода.

Во время переноса исходных инструкций во второй «пустой» блок может возникнуть необходимость модифицировать параметры инструкций или заменить инструкцию на блок, производящий идентичный эффект. В первую очередь подобные сложности вызывают инструкции, которые напрямую или косвенно используют значение счётчика команд: инструкции перехода по относительному смещению, инструкции доступа к памяти по адресу, вычисляющемуся по смещению от текущего и т.д.

4) Коррекция смещений в инструментационном коде

Последним этапом инструментации является восстановление смещений, используемых инструкциями доступа к глобальными переменным и внешним зависимостям в инструментационном коде. Стандартный процесс сборки исполняемого кода из нескольких объектных файлов включает в себя выставление корректных смещений в сегментах исполняемого кода по общей секции работы с глобальными данными и зависимостями. В процессе инструментации сгенерированный инструментационный код подключается к существующему исполняемому файлу или библиотеки сторонними от системы сборки средствами (использование стандартных сборщиков напрямую невозможно из-за отсутствия дополнительных таблиц в исходных файлах целевых файлов) и, соответственно, требует проведения подобного шага.

IV. Особенности практической реализации подхода

Система инструментации интегрирована с набором инструментов binutils[10]. Базовые модули binutils включают в себя библиотеку для разбора и управления файлами в формате ELF, причём текущая версия комплекса уже включает в себя средства удаления, добавления и перемещения секций, представления управляющих структур в необходимом формате. Наконец, в составе комплекса binutils существует библиотека libopcodes, предоставляющая гибкий интерфейс для реализации модулей декодирования набора инструкций определённой архитектуры.

Инструменты readelf и objdump в составе комплекса binutils используются для извлечения структуры целевых файлов ARM ELF, разметки точек инструментации и автоматической генерации исходного инструментационного кода. Инструмент objdump

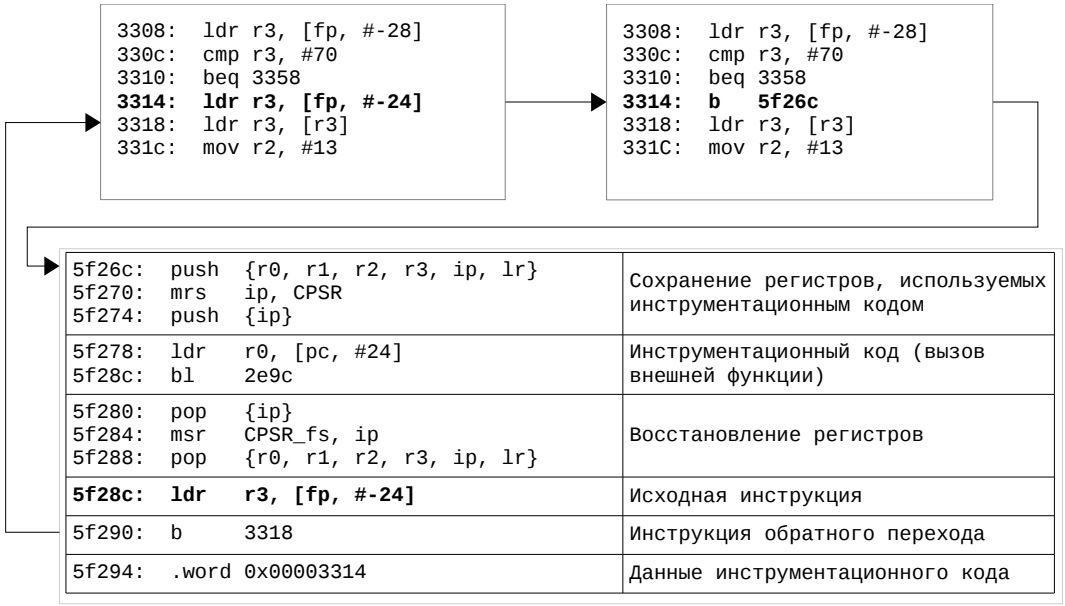

Рис. 1: Структура кода при инструментации

реализует линейное дизассемблирование; данный подход в отличие от метода рекурсивного разбора накладывает дополнительные ограничения на входные файлы (например, наличие таблицы символов) для обеспечения точности результата. Трансформация инструментационного кода в объектные файлы осуществляется с помощью компиляторной системы gcc, конкретная версия которой может быть предоставлена пользователем. Перераспределение исходных секций и добавление новых секций осуществляется с помощью инструмента objcopy. Для коррекции смещений и сегментов, осуществляющих работу с внешними зависимостями, на основе внутренней библиотеки работы с форматом ELF binutils был разработан инструмент addsymbol. Наконец, модуль расширения для библиотеки libopcodes, используемый для разметки точек инструментации, интегрирован с новым инструментов rewriter, проводящим непосредственно инструментацию.

A. Внутреннее представление инструкций

При работе инструмента objdump происходит разбор инструкций файлов исполняемого кода целевой программы и перевод инструкций во внутреннее представление, описывающее их основные элементы и параметры. Среди наиболее значимых элементов можно выделить следующие:

- Тип аргументов инструкции (регистры, константные значения).
- Значения аргументов инструкции (для констант).
- Код операции.
- Дополнительные параметры, специфичные для групп операций (режимы индексирования, триггеры обновления регистра флагов).

Основная функциональная мощность по обработке данных параметров лежит на инструментационном коде, однако прямой доступ к значениям параметров значительно упрощает структуру инструментационного кода и снижает накладные расходы при проведении анализа за счёт единовременного извлечения статических данных на этапе инструментации.

B. Практическое применение системы инструментации

В рамках практических экспериментов по использованию системы статической инструментации были разработаны модули инструментации для проведения динамического анализа, нацеленного на автоматический поиск дефектов. Данные модули разрабатывались для существующей системы анализа Avalanche [11], стандартно использующей динамическую инструментацию на основе комплекса Valgrind [12] для сбора необходимых данных.

Система Avalanche осуществляет обход возможных путей выполнения целевой программы на основе отслеживания потока помеченных данных, составления систем булевых формул для описания путей выполнения и автоматической генерации новых входных данных с помощью решателя булевых формул. Система осуществляет итеративный анализ, причём на каждой итерации целевая программа выполняется под контролем двух модулей, извлекающих дополнительные данные, — tracegrind (модуль инструментации с целью получения полной трассы инструкций, оперирующих помеченными данными) и covgrind (модуль инструментации с целью вычисления метрики покрытия на основе количества выполненных базовых блоков). Для работы с удалёнными и внешними устройствами, обладающими ограниченными вычислительными мощностями, в системе Avalanche предусмотрен специальным режим, при использовании которого на целевом устройстве осуществляется только запуск программы, а полная обработка трасс и работа решателя булевых формул выполняется на хост-устройстве.

Реализации модулей covgrind и tracegrind на основе статической инструментации были использованы в рамках указанного режима работы и позволили значительно ускорить анализ набора целевых программ. Различия в обработке инструкций систем статической и динамической инструментации привели к малому изменению порядка обхода ветвей, однако повышение скорости обхода ветвей были зафиксированы на всех проектах. Обнаруженные дефекты соответствуют дефектам, найденным системой Avalanche ранее на рассмотренных проектах для архитектуры x86_64. В таблице 1 приведён краткий обзор результатов работы системы с использованием двух типов инструментации.

ТАБЛИЦА I. РЕЗУЛЬТАТЫ РАБОТЫ СИСТЕМЫ AVALANCHE

Программа	**Итерации анализа**		**Количество дефектов**		**Время обнаружения дефекта, с**	
	Стат.	***Дин.***	***Стат.***	***Дин.***	***Стат.***	***Дин.***
swfdump	575	105	4	1	4	31
mpeg2dec	302	55	1	1	3	25
cjpeg	1331	236	1	1	20	600
qtdump	1027	189	1	1	33	1269
mpeg3dump	125	54	2	1	40	112

Время анализа для всех запусков было ограничено двумя часами (стандартная практика для инструмента Avalanche), в качестве начальных входных данных использовались файлы, соответствующие предыдущим результатам применения Avalanche. В список проанализированных программ входят:

- утилита swfdump (swftools-0.9.0) для извлечения информации о содержимом файлов формата SWF;
- утилита mpeg2dec (libmpeg2-0.5.1) для декодирования файлов формата MPEG-2;
- утилита cjpeg (libjpeg-7) для кодирования файлов в формат JPG;
- утилита qtdump (libquicktime-1.1.3) для декодирования файлов нескольких форматов (например, AVI)
- утилита mpeg3dump (libmpeg3-1.8) для декодирования файлов в формате MPEG-3.

Для 4 целевых программ из 5 при применении статического инструментации было достигнуто 5-кратное увеличение количества итераций анализа (количество итераций пропорционально количеству обойдённых ветвей и проверенных наборов формул). Известные по предыдущим результатам работы Avalanche дефекты были достигнуты ранее и для двух проектов было найдено больше различных дефектов. Для программы mpeg3dump было зафиксировано менее значительное ускорение, что связано с большим количеством ветвей выполнения, приводящих к зацикливанию программы. Зацикливание приводит к задержке по ожиданию срабатывания таймера и к снижению полезного времени анализа.

V. Заключение

Рассмотренный подход позволяет создавать точные и эффективные инструменты анализа, осуществляющие трассировку данных и инструкций. Наибольший выигрыш по производительности по сравнению с существующими динамическими аналогами достигается при выполнении многочисленных запусков модифицированных целевых программ (с целью проверки различных путей выполнения и сценариев работы). На данный момент основной фокус реализации подхода стоит на работе с платформой ARM, однако интеграция с системой binutils обеспечивает возможность поддержки других систем (в рамках формата ELF) без необходимости модификации базовых модулей системы инструментации.

В настоящее время рассматриваются следующие направления будущей деятельности:

- Расширение множества поддерживаемых типов точек инструментации и интерфейса, предоставляемого пользователю для описания инструментационного кода.
- Исследования возможностей оптимизации инструментации за счёт уменьшения количества инструкций, обеспечивающих корректность перехвата управления, и реорганизации переходов между исполняемым кодом целевой программы и блоками инструментационного кода.
- Повышение точности работы декодера исполняемого кода при работе с файлами ELF, не содержащими дополнительные секции управляющей информации.
- Изменение структуры внутреннего представления инструкций для повышения гибкости интерфейса, предоставляемого для написания инструментационного кода, и адаптации к поддержке различных наборов инструкций. Одним из возможных решений в данном направлении является полномасштабная трансформация блоков инструкций в операции промежуточного языка (например, REIL [14]).

Ссылки

[1] Amitabh Srivastava, Alan Eustace. ATOM: A System for Building Customized Program Analysis Tools, WRL Research Report 94/2, Western Research Laboratory, Palo Alto, CA, USA.

[2] Amitabh Srivastava and David W. Wall. A Pratical System for Intermodule Code Optimization at Link-Time. Journal of Programming Language, l(l), pp 1-18, March 1993.

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

[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 .

[5] B. Calder, T. Austin, D. Yang, T. Sherwood, S. Sair, D. Newquist, and T. Cusac. BitRaker Anvil: Binary Instrumentation for Rapid Creation of Simulation and Workload Analysis Tools. In Global Signal Processing (GSPx) Conference, 2004 .

[6] Lamia Djoudi, Denis Barthou, Patrick Carribault, Christophe Lemuet, Jean-Thomas Acquaviva, and William Jalby. MAQAO: Modular Assembler Quality Analyzer and Optimizer for Itanium 2. In Workshop on Explicitly Parallel Instruction Computing Techniques, Santa Jose, California, March 2005.

[7] Andres S. Charif-Rubial, Denis Barthou, Cedric Valensi, Sameer S Shende, Allen D. Malony and William Jalby. MIL: A language to build program analysis tools through static binary instrumentation. In 20th Annual International Conference on High Performance Computing, HiPC'13, Hyderabad, India, December 2013.

[8] Michael A. Laurenzano, Mustafa M. Tikir, Laura Carrington, Allan Snavely. PEBIL: Efficient static binary instrumentation for Linux. 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS), 2010 .

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

[10] GNU binutils, http://www.gnu.org/software/binutils/. Дата обращения: 10.08.2015.

[11] Исаев И.К., Сидоров Д.В. Применение динамического анализа для генерации входных данных, демонстрирующих критические ошибки и уязвимости в программах. Программирование, №4, 2010.

[12] Nicholas Nethercote and Julian Seward. 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 .

[13] Thomas Dullien and Sebastian Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. In CanSecWest, 2009.