DOI: 10.5862/JCSTCS.236.9
УДК 004.05

М.К. Ермаков

ПРОВЕДЕНИЕ ДИНАМИЧЕСКОГО АНАЛИЗА ИСПОЛНЯЕМОГО КОДА ФОРМАТА ARM ELF НА ОСНОВЕ СТАТИЧЕСКОГО БИНАРНОГО ИНСТРУМЕНТИРОВАНИЯ

M.K. Ermakov

DYNAMIC ANALYSIS OF ARM ELF EXECUTABLE CODE USING STATIC BINARY INSTRUMENTATION

Статья посвящена возможностям применения статического инструментирования исполняемых файлов и файлов динамических библиотек в формате ELF для проведения динамического анализа программ. Проведен обзор существующих решений в данной и смежных областях. Предложен подход к обработке бинарных файлов ARM ELF. Описаны основные особенности предлагаемого подхода: разметка целевых файлов и генерация внедряемого кода на основе пользовательских спецификаций, использование промежуточного языка описания инструкций ассемблера ARM, компоновка итоговых инструментированных файлов исполняемого кода. Приведены результаты применения реализации данного подхода для создания модулей трассировки помеченных данных, используемых в рамках инструмента автоматического поиска дефектов Avalanche.
ДИНАМИЧЕСКИЙ АНАЛИЗ; БИНАРНОЕ ИНСТРУМЕНТИРОВАНИЕ; АРХИТЕКТУРА ARM.

Dynamic program analysis methods are widely used in a broad range of activities related to software development; practical implementations of dynamic analysis rely on various code transformation and monitoring techniques. In this paper we focus on one of these techniques, static binary code instrumentation. We provide an overview of the existing tools implementing this technique and show that there are no tools directly applicable to our platform of choice, i.e., ARM/Linux and ELF binary format. We present an approach to perform static binary instrumentation for the platform in question and describe in detail the following points: user-specified instrumentation code and insertion point mapping; intermediate instruction representation used in instrumentation engine; code insertion process; offset correction process. Finally we describe a set of practical experiments of applying static binary instrumentation to Avalanche, a dynamic program analysis tool performing automatic input generation and bug discovery.
DYNAMIC ANALYSIS; BINARY INSTRUMENTATION; ARM ARCHITECTURE.

В настоящее время существует большое количество систем и инструментов, производящих автоматическую и полуавтоматическую обработку программного обеспечения (ПО) с целью анализа, профилирования программ, вскрытия алгоритмов или, наоборот, внедрения блоков препятствования взлому.

Приведенные примеры взаимодействия с ПО 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, внедренного в программу посредством *инструментирования*.

Различают три схемы проведения инструментирования: модификацию исходного кода программы, модификацию исполняемого кода программы во время

выполнения (динамическое инструментирование) и модификацию файлов исполняемого кода перед выполнением (статическое инструментирование).

Модификация исходного кода программы требует наличия исходного кода и позволяет описывать требуемую функциональность высокоуровневыми конструкциями языков программирования. Системы инструментирования исходного кода позволяют делегировать решение практических подзадач изменения программ системам компиляции и сборки, что обеспечивает отсутствие затрат на поддержку различных платформ. В то же время особенности процесса компиляции и оптимизации кода могут негативно влиять на точность анализа, если требуется извлечение низкоуровневой информации о ходе выполнения программы.

Инструментирование исполняемого кода во время выполнения программы предоставляет высокую точность модификации, заниженные затраты на непосредственный разбор кода и возможность обработки кода, который загружается или создается динамически. В то же время разбор кода и инструментирование проводятся на каждом запуске программы. Для методов анализа, требующих проведение множественных запусков, данная особенность негативно влияет на производительность по сравнению с другими режимами инструментирования.

Статическое инструментирование исполняемого кода имеет большую область применения, по сравнению с инструментированием исходного кода, и позволяет легко осуществлять доступ к низкоуровневой информации о выполнении программы. В то же время задача разбора исполняемого кода до запуска программы является достаточно сложной, особенно при работе с обфусцированным или защищенным кодом.

Среди рассмотренного выше применения данных систем можно выделить в высокой степени востребованную задачу поиска дефектов и уязвимостей в программах при наличии только исполняемого кода. Актуальность этой задачи распространяется на мобильные платформы (например, Tizen и Android), программное обеспечение для которых развивается крайне активно. Большинство мобильных устройств работает на базе процессорной архитектуры ARM. Дополнительно, мобильные устройства обычно имеют ограниченные объемы ресурсов. Как было отмечено выше, динамическое инструментирование повышает объем накладных расходов по сравнению со статическими методами.

Тем самым разумно предположить, что для решения задачи анализа программ по исполняемому коду можно эффективно использовать методы статического инструментирования исполняемого кода. К сожалению, современные свободно доступные системы статического инструментирования не предоставляют поддержки архитектуры ARM. Создание системы инструментирования, сравнимой по функциональности с существующими аналогами, является актуальной задачей.

В рамках данной статьи будет рассматриваться подход к проведению статического инструментирования исполняемого кода в формате ARM ELF и особенности программной системы, реализующей данный подход.

Обзор существующих систем инструментирования

Для большинства существующих систем инструментирования 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 инструментирования. Основные факторы, ограничивающие область применимости систем инструментирования, следующие:

- поддерживаемые машинные архитектуры и специфические наборы инструкций для данных архитектур;
- поддерживаемые форматы организации файлов исполняемого кода;
- предоставляемая разметка точек и блоков инструментирования;

• тип организации внедряемого кода;

• ограничения, накладываемые на целевые файлы исполняемого кода.

Ранние системы инструментирования 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 счет 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 инструментирования, в то время как непосредственно процесс инструментирования 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 инструментирования. Это позволило значительно упростить процесс разметки целевого кода и спецификации логики модуля инструментирования. Дополнительно подход, используемый в системе EEL, обеспечивал гибкую структуру для поддержки различных архитектур: кодирование и декодирование целевого кода и генерацию внедряемого 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 инструментированию в системе BIRD заключался во вставке инструкций вызова внешних функций в исполняемый код целевой программы и сборке внедряемого кода в виде динамической библиотеки, предоставляющей реализацию данных внешних функций.

Система BitRaker Anvil 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 внедряемого кода осуществлялись в процессе среды эмуляции, что значительно повышало скорость проведения анализа.

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

• развитие интерфейса инструментирования, доступного пользователю для описания логики необходимого модуля инструментирования;

• 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 внедряемого кода.

Среди наиболее значимых систем статического инструментирования можно выделить MAQAO [6, 7], PEBIL [8] и Dyninst [9], предоставляющие 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 эффектив-

ности работы компонентов программы. В дальнейшем был разработан язык спецификации точек инструментирования MIL [7] и модуль генерации внедряемого кода для поддержки возможности разработки дополнительных инструментов анализа. Важная особенность системы MAQAO – 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 инструментированию и добавляет внедряемый код в виде дополнительной секции. Блоки инструкций в точках инструментирования заменяются на инструкции перехода в добавленные секции. Система PEBIL поддерживает генерацию внедряемого 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 внедряемый код.

Система Dyninst предоставляет широкий интерфейс описания внедряемого кода и спецификаций точек инструментирования. Внутренние модули обеспечивают проведение необходимых модификаций, причем возможно проведение как статического инструментирования на основе полномасштабного разбора файлов целевой программы, так и динамического инструментирования на основе стандартного подхода перехвата и модификации блоков инструкций.

Инструментирование файлов ARM ELF

Как было сказано ранее, ни одна из представленных выше современных систем инструментирования не предоставляет поддержку обработки исполняемого кода в инструкциях архитектуры ARM. Система BitRaker Anvil не предоставляет поддержку современных версий набора инструкций архитектуры. Данное исследование направлено на разработку подхода инструментирования и создание инструментального средства, предоставляющего возможности работы с исполняемым кодом в инструкциях ARM. В качестве первичных задач для применения системы инструментирования рассматриваются задачи, связанные с автоматическим поиском дефектов и уязвимостей в программах.

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

фокус на инструментирование структурных единиц исполняемого кода (инструкций) для сбора подробной трассы, включающей типы и аргументы совершаемых операций;

поддержка создания внедряемого 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.

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

структурные точки, задающие все инструкции определённой функциональной группы (арифметические и логические инструкции, инструкции доступа к памяти, вызова процедур и т. д.);

модульные точки, задающие позицию в некотором логическом блоке (входные и выходные инструкции функции, входные инструкции базовых блоков, блоки условного выполнения).

Задаваемые фильтры обеспечивают возможность осуществлять инструментирование для точек определенного 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 внедряемого кода осуществляется путем 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 внедряемого кода привязаны к инструкциям, находящимся в точках инструментирования. При написании внедряемого кода существует возможность использования специального интерфейса, предоставляющего операции над инструкциями двух типов:

- операции доступа к статическим элементам инструкций, таким как номера регистров операндов, значения константных операндов, флаги, специфичные для отдельных групп инструкций;
- операции доступа к динамическим элементам 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 внедряемом коде. Работа блока оптимизации при компиляции внедряемого кода позволяет понизить накладные расходы по осуществлению доступа к статическим элементам. Для реализации операций второго типа производится генерация блока инструкций, выполнение которого приведет к вычислению необходимых значений при выполнении внедряемого кода.

Подготовка файлов целевой программы к инструментированию. Вторым этапом инструментирования 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 внедряемого кода и данных;
- перемещение и расширение ряда секций, содержащих управляющую информацию и исполняемый код, для добавления внешних зависимостей, указанных во внедряемом 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.

Инструментирование подготовленных образов. Непосредственное инструментирование заключается в замене одной инструкции или блока инструкций в указанных точках на инструкцию перехода в секцию внедряемого кода, добавленную на предыдущем этапе. Последующие действия связаны с обеспечением корректности модифицированного кода путем вставки дополнительных инструкций сохранения состояния программы и восстановления заменённых инструкций.

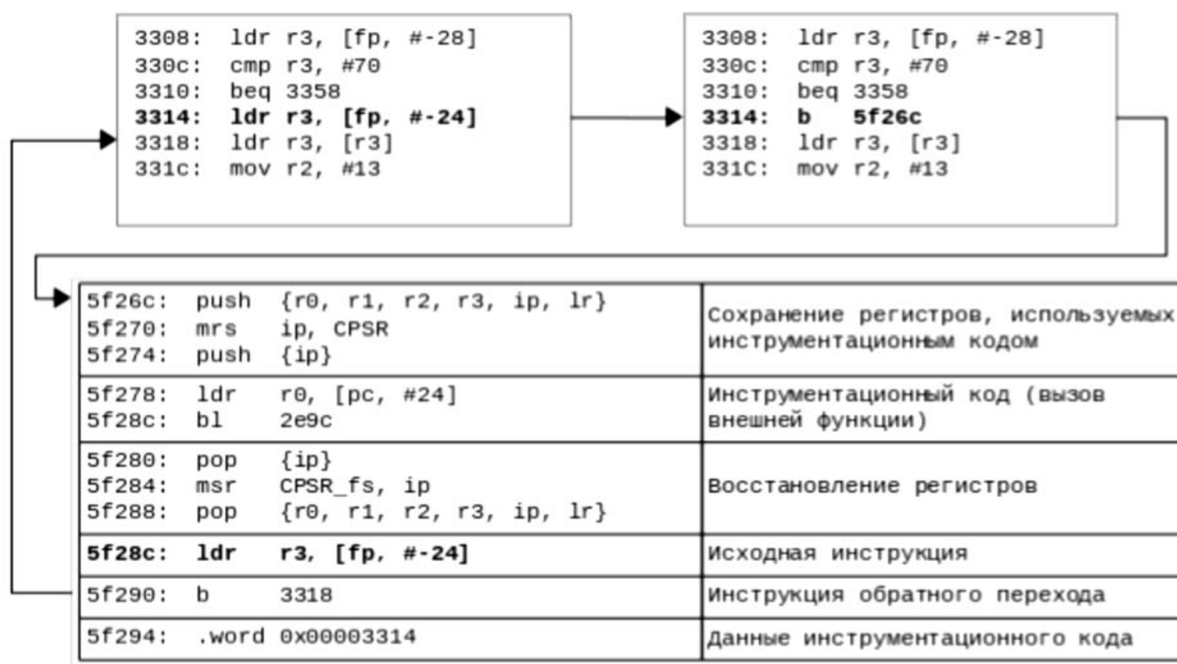
Сгенерированный внедряемый код состоит из набора обособленных сегментов, каждый из которых соответствует точке инструментирования. При этом сегменты имеют блочную структуру, разделённую на четыре области: начальный «пустой» блок, состоящий из инструкций, не имеющих никаких побочных эффектов, непосредственно блок кода, реализующий необходимую для анализа функциональность, дополнительный «пустой» блок и блок данных. Начальный и дополнительный «пустые» блоки имеют размер, определяемый особенностями инструкций в точке инструментирования, и используются для вставки служебных инструкций.

В процессе инструментирования в пер-

вый «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 инструментирования на инструкцию перехода. Последней во второй «пустой» блок вставляется инструкция возвратного перехода в основную секцию кода на позицию после точки инструментирования. При наличии двух смежных точек инструментирования с целью снижения накладных расходов вместо возвратного перехода может осуществляться переход на следующий блок внедряемого кода. Пример структуры кода представлен на рисунке, показывающем три блока инструкций: исходный блок инструкций, содержащий точку инструментирования, блок инструкций после вставки перехода на внедряемый код и соответствующий блок внедряемого 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 счетчика команд: инструкции перехода по относительному смещению, инструкции доступа к памяти по адресу, вычисляющемуся по смещению от текущего, и т. д.

Коррекция смещений во внедряемом коде. Последним этапом инструментирования является восстановление смещений, используемых инструкциями доступа к глобальными переменным и внешним зависимостям во внедряемом 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 инструментирования сгенерированный внедряемый код подключается к существующему исполняемому файлу или библиотеке 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.



Пример инструментирования одной точки

Особенности практической реализации подхода

Система инструментирования интегрирована с набором инструментов binutils [10]. Базовые модули binutils включают в себя библиотеку для разбора и управления файлами в формате ELF, причем 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 инструментирования и автоматической генерации исходного кода инструментирования. Инструмент objdump 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 внедряемого 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 разработан инструмент addsymbol. Наконец, модуль расширения для библиотеки libopcodes, используемый для разметки точек инструментирования, интегрирован с новым инструментом rewriter, проводящим непосредственное инструментирование.

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 внедряемом коде, однако прямой доступ к значениям параметров значительно упрощает структуру внедряемого кода и снижает накладные расходы при проведении анализа за счет единовременного извлечения статических данных на этапе инструментирования.

Практическое применение системы инструментирования. В рамках практических экспериментов по использованию системы статического инструментирования разработаны модули, позволяющие проводить динамический анализ, нацеленный на автоматический поиск дефектов. Данные модули разрабатывались для существующей системы анализа Avalanche [11], стандартно использующей динамическое инструментирование 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, причем на каждой итерации целевая программа выполняется под контролем двух модулей, извлекающих дополнительные данные, – tracegrind (модуль инструментирования с целью получения полной трассы инструкций, оперирующих помеченными данными) и covgrind (модуль инструментирования 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 специальный режим, при использовании которого на целевом устройстве осуществляется только запуск программы, а полная обработка трасс и работа решателя булевых формул выполняется на хост-устройстве.

Реализованные на основе статического инструментирования версии модулей covgrind и tracegrind использовались в рамках указанного режима работы и позволили значительно ускорить анализ набора целевых программ. Различия в обработке инструкций системами статического и динамического инструментирования привели к малому изменению порядка обхода ветвей, однако повышение скорости обхода ветвей было зафиксировано на всех проектах. Обнаруженные дефекты соответствуют дефектам, найденным системой Avalanche ранее на рассмотренных проектах для архитектуры x86_64. В таблице приведен краткий обзор результатов работы системы с использованием двух типов инструментирования.

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 четырех целевых программ из пяти при применении статического инструментирования было достигнуто пятикратное увеличение количества итераций анализа. Дефекты, известные по предыдущим результатам работы Avalanche, были получены 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.

Рассмотренный подход позволяет создавать точные и эффективные инструменты анализа, осуществляющие трассировку дан-

Результаты работы инструмента Avalanche

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

ных 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 основная задача реализации подхода заключается в работе с платформой ARM, однако интеграция с системой binutils обеспечивает возможность поддержки других систем (в рамках формата ELF) без необходимости модификации базовых модулей системы инструментирования.

В настоящее время рассматриваются следующие направления будущей деятельности:

Расширение множества поддерживаемых типов точек инструментирования и интерфейса, предоставляемого пользователю для описания внедряемого кода.

Исследования возможностей оптимизации инструментирования за счет уменьшения количества инструкций, обеспечивающих корректность перехвата управления, и реорганизации переходов между исполняемым кодом целевой программы и блоками внедряемого 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 внедряемого 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 [13]).

СПИСОК ЛИТЕРАТУРЫ

1. **Srivastava A., Eustace A.** ATOM: A System for Building Customized Program Analysis Tools // WRL Research Report 94/2. Western Research Laboratory, 1994.

2. **Srivastava A., Wall D.** A Practical System for Intermodule Code Optimization at Link-Time // J. of Programming Language. 1993. Vol. 1(l). Pp. 1–18.

3. **Larus J., Scnharr E.** EEL: Machine-Independent Executable Editing // Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation. 1995. Pp. 291–300.

4. **Nanda S., Li W., Lam L., Chiueh T.** BIRD: Binary Interpretation using Runtime Disassembly // Proc. of the Internat. Symp. on Code Generation and Optimization. 2006.

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

6. **Djoudi L., Barthou D., Carribault P., Lemuet C., Acquaviva J., Jalby W.** MAQAO: Modular Assembler Quality Analyzer and Optimizer for Itanium 2 // Workshop on Explicitly Parallel Instruction Computing Techniques. 2005.

7. **Charif-Rubial A., Barthou D., Valensi C., Shende S., Malony A. Jalby W.** MIL: A Language to Build Program Analysis Tools through Static Binary Instrumentation // Proc. of the 20th Annual Internat. Conf. on High Performance Computing. 2013.

8. **Laurenzano M., Tikir M., Carrington L., Snavely A.** PEBIL: Efficient Static Binary Instrumentation for Linux. // Proc. of the Internat. Symp. on Performance Analysis of Systems & Software. 2010. Pp. 175–183.

9. **Miller B., Bernat A.** Anywhere, Any Time Binary Instrumentation // ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. 2011. Pp. 9–16.

10. Страница проекта GNU binutils [Электронный ресурс] // URL: 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. 2010. № 4. С. 51–67.

12. **Nethercote N., Seward J.** Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation // In Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation. San Diego, California, USA, 2007. Pp. 89–100.

13. **Dullien T., Porst S.** REIL: A platform-independent intermediate representation of disassembled code for static code analysis // In CanSecWest. 2009.

REFERENCES

1. **Srivastava A., Eustace A.** ATOM: A System for Building Customized Program Analysis Tools. *WRL Research Report 94/2*, Western Research Laboratory, 1994.

2. **Srivastava A., Wall D.** A Practical System for Intermodule Code Optimization at Link-Time. *Journal of Programming Language*, 1993, Vol. 1(l), Pp. 1–18.

3. **Larus J., Scnharr E.** EEL: Machine-Independent Executable Editing. *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation,* 1995, Pp. 291–300.

4. **Nanda S., Li W., Lam L., Chiueh T.** BIRD: Binary Interpretation using Runtime Disassembly. *Proceedings of the International Symposium on Code Generation and Optimization*, 2006.

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

6. **Djoudi L., Barthou D., Carribault P., Lemuet C., Acquaviva J., Jalby W.** MAQAO: Modular Assembler Quality Analyzer and Optimizer for Itanium 2. *Workshop on Explicitly Parallel Instruction Computing Techniques*, 2005.

7. **Charif-Rubial A., Barthou D., Valensi C., Shende S., Malony A. Jalby W.** MIL: A Language to Build Program Analysis Tools through Static Binary Instrumentation. *Proceedings of the 20th Annual International Conference on High Performance Computing*, 2013.

8. **Laurenzano M., Tikir M., Carrington L., Snavely A.** PEBIL: Efficient Static Binary Instrumentation for Linux. *Proceedings of the International Symposium on Performance Analysis of Systems & Software,* 2010, Pp. 175–183.

9. **Miller B., Bernat A.** Anywhere, Any Time Binary Instrumentation. *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2011, Pp. 9–16.

10. *GNU binutis*. Available: http://www.gnu.org/software/binutils/.

11. **Isayev I.K., Sidorov D.V.** Primeneniye dinamicheskogo analiza dlya generatsii vkhodnykh dannykh, demonstriruyushchikh kriticheskiye oshibki i uyazvimosti v programmakh [The Use of Dynamic Analysis for Generation of Input Data that Demonstrates Critical Bugs and Vulnerabilities in Programs]. *Programmirovaniye* [*Programming*], 2010, No. 4, Pp. 51–67. (rus)

12. **Nethercote N., Seward J.** Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, California, USA, 2007, Pp. 89–100.

13. **Dullien T., Porst S.** REIL: A platform-independent intermediate representation of disassembled code for static code analysis. *CanSecWest*, 2009.

ЕРМАКОВ Михаил Кириллович – *младший научный сотрудник Института системного программирования РАН.*

109004, Россия, Москва, ул. Александра Солженицына, д. 25.

E-mail: mermakov@ispras.ru

ERMAKOV Mikhail K. *Institute for System Programming of the Russian Academy of Sciences.*

109004, Alexander Solzhenitsyn Str. 25, Moscow, Russia.

E-mail: mermakov@ispras.ru

© Санкт-Петербургский политехнический университет Петра Великого, 2016