УДК 519.71

О.Г. Руденко, С.В. Мирошниченко

Харьковский национальный университет радиоэлектроники, Харьков

МЕТОДЫ ДИНАМИЧЕСКОГО ИНСТРУМЕНТИРОВАНИЯ КОДА

В статье рассматриваются такие методы динамического инструментирования, как отладка с единичным шагом и перемещенное выполнение. В результате их анализа был произведен выбор метода для реализации программного средства динамического инструментирования для аппаратной платформы MIPS. Динамическое инструментирование предполагает некоторую модификацию бинарного кода, позволяющую подключить различные функционалы отладки, трассировки, измерения производительности и регистрации интересующих событий.

Ключевые слова: *инструментирование, бинарный код, метод перемещенного выполнения.*

Введение

Инструментирование программ – это техника вставки нового кода в программу, при которой ее оригинальное поведение остается неизменным. Эта фундаментальная техника мониторинга применяется в областях электронной экспертизы, аудита программ, обнаружения вторжений и атак, поведенческого анализа, анализа производительности.

Инструментирование программ – термин, использующийся для обозначения техник, используемых для модификации существующих программ при необходимости сбора во время выполнения дополнительной информации: данные профилирования, трассировка, симуляция кэшей и т.п. Основная техника инструментирования – вставка инструментирующего кода в интересующие точки программы. Имеется также возможность изменять при необходимости семантику программы, например, временно исправлять ошибки, не прибегая к перекомпиляции, либо направить поток выполнения в обход подпрограммы аутентификации.

Задачи, в которых может применяться инструментирование:

- отладка приложений, библиотек и ядра ОС;
- обнаружение порчи и срыва стека [2];
- обнаружение переполнения и порчи кучи;
- обнаружение гонок за данными;
- обнаружение утечек памяти и ресурсов;
- post-mortem анализ – исследование зарегистрированных перед крахом программы событий может использоваться наряду с специальными отладчиками слепков (дампов) памяти;
- анализ производительности [3];
- анализ покрытия кода [4];
- реверсная инженерия.

Обнаружение переполнения и порчи кучи – сложные ошибки, которые могут обнаружиться через длительное время после момента появления и далеко от места появления. Данная проблема может решаться на этапе разработки аллокатором памяти путем пометки границ выделенных областей специальными последовательностями байт – т.н. стражи памяти (memory guards) – и периодический мониторинг их изменений: если в определенный момент значение «стража» изменилось, то это означает нарушение целостности кучи. Однако эта возможность требует несколько большего объема памяти и дополнительных вычислительных расходов на проведение мониторинга, потому такая опция обычно не включена в «финальных» сборках проектов, а порой и не используется вовсе, даже при разработке. Данный тип ошибок наряду со срывом стека являются одними из наиболее часто встречаемых неисправностей программ, и в то же время одними из наиболее трудноуловимых, в особенности в отсутствие принятых мер для облегчения их устранения, наподобие вышеупомянутых стражей памяти. Выходом из ситуации служит инструментирование кода, позволяющее не только облегчить, но и автоматизировать устранение данных проблем.

Обнаружение гонок за данными – проблема, возникающая в многопоточных программах, не может быть отлажена классическими отладчиками, как в силу технических особенностей их работы, так и по причине малой вероятности уловить проблемы синхронизации доступа путем пошагового выполнения, что на практике означает невозможность решения таких проблем вручную.

Обнаружение утечек памяти и ресурсов – возникающих при отсутствии кода, освобождающего выделенную ранее, но уже не использующуюся память. Аналогично можно отслеживать утечки ресурсов любых видов.

Реверсная (обратная) инженерия – в контексте исследования безопасности это детальное изучение работы участка программы. Процесс трудоемкий и кропотливый, и методы инструментирования позволяют множеством способов ускорить этот процесс.

Таким образом, программное обеспечение, реализующее метод инструментирования, предоставляет возможность автоматического поиска неисправ-

ностей, освобождая разработчика, исследователя от необходимости выполнения множества рутинных, но, в то же время, требующих большого внимания действий, значительно упрощая тем самым отладку, повышая ее качество.

1. Методы динамического инструментирования

Динамическое инструментирование предполагает внесение изменений в бинарный программный код – как правило, замену какой-либо инструкции на оператор перехода или вызова исключения. Внедрение кода производится в процессе загрузки либо исполнения исследуемой программы, предварительный анализ путей выполнения не требуется – точки интереса формируются с продвижением исследования. Инструментирование должно быть выполнено заново при каждом перезапуске программы, т.к. модификации кода не сохраняются на диске независимо от того, сколько участков кода и каким образом было инструментировано. Кроме того, произведенные модификации можно динамически откатывать, возвращая участок кода к первоначальному состоянию, в том числе и в плане его производительности.

Несмотря на то, что шаг трассировки зачастую привязывается к единицам модульности, например, функциям, двоичное инструментирование как процесс внесения изменений в исполняемый код проводится по отношению к одной инструкции, а не к участку кода, состоящему из непрерывной последовательности инструкций. Это делается с целью избежать комплексного и сложного анализа, связанного с потребностью выявления и модификации команд с относительной адресацией, указывающих на середину инструментируемой последовательности инструкций. Отсюда становится очевидным, что способов выполнения трассируемой инструкции два: либо она может находиться по своему оригинальному адресу, либо же она перемещается в отдельное хранилище и выполняется там.

1.1. Метод отладки с единичным шагом (single-step)

Классический алгоритм установки и обработки точек останова, который традиционно используется в отладчиках [1], основан на механизме исключений. В контексте инструментирования интерес представляет механизм программных точек останова, которых, в отличие от аппаратных (имеется в виду заполнение специальных регистров некоторых процессоров, вызывающих исключения при доступе к указанным ячейкам памяти), может быть неограниченное количество. Пункты, относящиеся к логическому связыванию физического размещения точки останова в исполняемом коде с соответствующим оператором исходного текста, не рассматриваются. Следует лишь отметить относительную сложность этой задачи, например, при отладке шаблонов C++, которые приводят к необходимости решать проблемы размерности «один-ко-многим».

Прерывания точек останова обычно реализованы в виде специальной машинной инструкции генерации исключения точки останова, вызывающей соответствующий обработчик исключения в ядре ОС. Этому обработчику в свою очередь предоставляется контекст процесса, представленный заполненной структурой в оперативной памяти, из которого можно извлечь такую информацию как адрес инструкции останова, идентификаторы используемых ресурсов, значения регистров процессора на момент останова.

В архитектурах процессоров с переменной длиной инструкций для точки останова обычно используется инструкция наименьшей длины. Отладчик имеет возможность чтения и записи бинарного кода отлаживаемой программы через специальный интерфейс ОС: подлежащая инструментированию инструкция запоминается, а на ее место записывается инструкция точки останова. Далее показаны шаги записи инструкции генерации отладочного исключения (с кодом операции 0xCC) для ассемблерного кода архитектуры x86.

```
ba 07 00 00 00    mov $0x7,%edx
b9 b4 90 04 08    mov $0x80490b4,%ecx
Шаг 1 – Участок оригинального кода

cc 07 00 00 00    breakpoint; original instruction
b9 b4 90 04 08    mov $0x80490b4,%ecx
Шаг 2 – Установка точки останова
```

Когда счетчик команд исследуемого процесса доходит до этой инструкции и выполняет ее – процессор передает управление модулю ОС, отвечающему за обработку прерываний и исключений сообщит отладчику о прерывании. В операционной системе Linux обработчики устанавливаются функциями set_intr_gate() и set_system_gate_ist(), вызываемыми процедурой инициализации trap_init() и заменяемыми затем отладчиком при запуске. Таким образом, управление, в конце концов, получает отладчик.

По выполнении необходимых разработчику либо исследователю действий (инспектирование переменных и состояния процессора, анализ текущего пути выполнения и др.) возникает необходимость выбора дальнейших действий – либо управление сразу передается отлаживаемой программе, либо процессор перед этим переводится в режим single-step (режим единичного шага). В этом режиме процессор генерирует дополнительное исключение отладки автоматически при выполнении каждой инструкции, что и обозначается единичным шагом. Однако такой режим поддерживается не всеми платформами, например, большинством RISC-архитектур, потому выходом из ситуации является установка точки останова на следующей за текущей инструкции исследу-

емой программы, что требует несколько большего количества выполняемых операций.

Нельзя упускать из виду также тот факт, что при выходе из обработчика исключения и возврата управления отлаживаемой программе счетчик команд по умолчанию установлен в общем случае на середину той инструкции, в начало которой была записана инструкция генерации отладочного исключения. В архитектурах с фиксированной длиной команд это будет следующая инструкция. Потому отладчик записывает исходную команду по ее оригинальному местоположению, и должен сдвинуть счетчик в направлении, обратном выполнению, на длину инструкции исключения.

Анализируя работу алгоритма, можно заметить, что в ситуации, когда требуется постоянная точка останова (после отработки исходной инструкции на ее место должна быть возвращена команда отладочного исключения), возникает окно во время замен команд. Наличие окна позволяет судить о ненадежности механизма регистрации трассы, что делает невозможным отладку многопоточных приложений, участков общих для нескольких потоков кода.

1.2. Метод перемещенного выполнения

Можно выделить предусловия, при которых использование данного метода будет наиболее целесообразным: недопустимость недетерминированной длительности остановки исследуемого процесса и недопустимость пропуска регистраций трассы из-за возникающего окна во время замены инструкций.

Следствием из этих требований является невозможность использования каких-либо блокирующих объектов синхронизации в алгоритмах инструментирования. Это означает, что внесение изменений в бинарный исполняемый код, находящий в памяти, можно производить лишь при включении и выключении инструментирования данного конкретного участка кода. Такие ограничения ведут к тому, что трассируемая инструкция не может выполняться по своему первоначальному адресу. Вместо этого ее можно выполнить по другому адресу, расположенному в пространстве исследуемого потока.

Исходя из вышеперечисленного, единственным путем развития является копирование оригинальной трассируемой инструкции в специально выделенную для нее область памяти в пространстве исследуемого процесса, а затем ее выполнение в контексте этого процесса.

Трасса представляется специальным объектом, являющегося аналогом точки останова и выполняющего ту же функцию; в контексте динамического инструментирования такие объекты называют датчиками либо сенсорами (англ. probe), причем второе название точнее отражает сущность этого объекта.

Информация о сенсорах хранится в глобальной системной хеш-таблице (для быстрого поиска в дальнейшем), ключами которой служат модуль (местоположение в системе) и адрес инструкции, а значением – идентификатор нового сенсора либо указатель на область памяти, его содержащую.

Алгоритм работы метода состоит из нескольких шагов:

– шаг 1: выбор трассируемой инструкции;
– шаг 2: выделение памяти для нового сенсора;
– шаг 3: создание записи в глобальной системной хеш-таблице;
– шаг 4: копирование исходной инструкции в глобальное хранилище;
– шаг 5: запись инструкции, активирующей сенсор;
– шаг 6: сброс кэшей.

Как правило, активирующая сенсор инструкция является командой генерации отладочного исключения. Записывается она по адресу оригинальной инструкции, после чего на некоторых архитектурах требуется сброс кешей инструкций [7].

При срабатывании сенсора в адресном пространстве системного процесса, в котором это произошло, создается локальное хранилище (если оно еще не было создано) путем выделения памяти. В это хранилище копируется исходная инструкция и необходимое обрамление (код, возвращающий управление трассировщик, как правило, с помощью все той же команды отладочного исключения), затем на нее передается управление. Таким образом, одному потоку достаточно лишь одного хранилища размером в несколько инструкций.

Рассмотрим работу метода на примере ассемблерного кода функции memcpy() для процессоров SPRARC. Допустим, необходимо инструментировать первую инструкцию приведенной последовательности:

```
add %o1, 2, %o1
srl %o3, 8, %o4
st  %o4, [%o0]
st  %o3, [%o0 + 4]
```

Далее схематично показан этап сохранения инструкции в глобальной системной хеш-таблице в адресном пространстве ядра операционной системы:

```
add %o1, 2, %o1 -->   | 0x104c | add %o1, 2, %o1 |
srl %o3, 8, %o4
st  %o4, [%o0]
st  %o3, [%o0 + 4]
```

Замена трассируемой инструкции командой генерации отладочного исключения (называемой в данном контексте ловушкой) показана в следующем листинге:

```
ta  0x38     !trap       | 0x104c | add %o1, 2, %o1 |
srl %o3, 8, %o4
st  %o4, [%o0]
st  %o3, [%o0 + 4]
```

Расположение инструкций после проведения инструментирования продемонстрировано в следующем участке кода:

```
add %o1, 2, %o1 <--  | 0x104c | add %o1, 2, %o1 |
<возврат выполнения>
---------
ta  0x38
srl %o3, 8, %o4
st  %o4, [%o0]
st  %o3, [%o0 + 4]
```

1.3. Анализ метода перемещенного выполнения

Одним из достоинств метода является отсутствие использования в алгоритмах, непосредственно связанных с заменой инструкций, перехватом управления и регистрацией трасс блокирующих системных объектов синхронизации, ограничиваясь атомарными переменными и атомарными процессорными инструкциями сравнения и записи при совпадении, что означает невозможность возникновения взаимных блокировок, гонок за данными и недетерминированных задержек. Это позволяет судить об абсолютной надежности метода, позволяющей инструментировать любые участки кода: как пользовательские приложения и системные библиотеки, так и компоненты и модули ядра операционной системы, а также драйвера виртуальных и физических устройств.

Кроме того, на протяжении работы не возникает окон, создаваемых при частой перезаписи инструкций. Тем самым обеспечивается безопасность отладки и исследования многопоточных программ.

Среди прочего следует отметить возможность снятия и установки сенсора на лету, облегчая работу с программами без исходных кодов. Снятие сенсора означает возврат участка кода к его первоначальному состоянию, что означает полное отсутствие накладных расходов в этом случае.

Метод не лишен существенных недостатков, одним из которых является уже упомянутые проблемы с производительностью, общие для всех методов динамического инструментирования. Кроме того, инструментированы таким образом могут быть не все инструкции, ведь выполнение по смещенному адресу команд, содержащих относительную адресацию операндов, ведет к тому, что результирующие адреса будут отличаться от требуемых. Потому требуется либо ограничивать доступных для инструментирования команд, либо решать проблему интенсивным путем.

Интрузивность метода затрудняет исследование вредоносных программ, проверяющих целостность собственного кода [13].

2. Реализация метода перемещенного выполнения

Рассмотрены этапы решения задачи реализации средства динамического инструментирования с использованием метода перемещенного выполнения для устройств архитектуры MIPS32.

В качестве программной основы был выбран DTrace – данный фреймворк позволяет трассировать как ядро, так и пользовательское пространство в реальном времени [6]. DTrace может использоваться для наблюдения за количеством потребляемой памяти, процессорным временем, файловыми системами и сетевыми ресурсами, используемыми активными процессами, на работающей системе. Также можно получить более детальную информацию, например, список аргументов, с которыми вызывается каждая функция, или список процессов, использующих определённый файл.

DTrace спроектирован в виде системного комплекса, состоящего из двух крупных частей: динамически загружаемого модуля ядра и программы адресного пространства пользователя.

Модуль, выполняющийся в адресном пространстве ядра, содержит в себе функционал всех провайдеров, предоставляющих функционал инструментирования различных участков кода различными способами. Работа на уровне ядра операционной системы обусловлена необходимостью: во-первых, для возможности инструментирования кода самого ядра, его модулей и драйверов устройств, во-вторых, для возможности отслеживания системных вызовов и событий планировщиков, в-третьих, для обеспечения доступа к структурам, содержащим контексты процессов. Кроме того, требуется проводить перехват некоторых прерываний, что не всегда невозможно из пользовательского пространства.

Часть DTrace, выполняющаяся в адресном пространстве пользователя, нагружена функционалом интерпретации программ инструментирования, написанных на языке D, а также предоставляет интерфейс взаимодействия с разработчиком/исследователем для выполнения таких действий, как получение списка всех сенсоров, добавление, активация и деактивация сенсоров, перенаправление результатов инструментирования (журналов трасс) в указанный файл.

MIPS32 – архитектура RISC с фиксированной длиной машинных инструкций 32 бита, 32 32-битными регистрами общего назначения, регистрами специального назначения (счетчик команд, TLB и др.), доступными через сопроцессор 0. Гарвардская архитектура процессорных кэшей: раздельные кэши для инструкций и данных. Основная сфера использования процессоров архитектуры MIPS это системы на кристалле. Форматы машинных инструкций представлены тремя видами [5].

Основным функционалом DTrace является трассировка функций. Самым эффективным методом для этого является вставка кода в начало функции (entry point) и в точки возврата из нее (return points) [8]. Для этого можно воспользоваться тем фактом, что исходя из общепринятой функциональ-

ной направленности регистров, функции и процедуры языков C, C++ и многих других, транслируются кросс-компиляторами в такие структуры исполняемого кода, которые могут быть непосредственно сопоставлены с упомянутыми точками входа и возврата. Такие структуры называются, соответственно, прологом и эпилогом функции.

В процессе реализации были решены такие возникшие проблемы, как эмуляция инструкций с относительной адресацией, эмуляция слота задержки, эмуляция устаревших инструкций, перехват прерываний, перехват межпроцессорных прерываний.

Для оценки влияния производительности программы, инструментируемой методом перемещенного выполнения, необходимо учитывать параметры, от которых она зависит. Одно лишь количество инструментированных инструкций не может обеспечить информацией для предварительной оценки вносимых задержек, так как не до каждой такой инструкции может дойти управление. Потому длительность вносимых задержек зависит от количества передач управления трассировщику, которое зависит от того, как часто поток управления будет достигать инструментированных участков кода.

Этот факт позволяет спроектировать синтетический тест, показывающий зависимость скорости выполнения инструментированной программы от количества вызовов трассировщика. В качестве аппаратной платформы для замеров производительности была использована целевая система на кристалле Broadcom BCM7405, имеющая два процессора с частотой 400 МГц. Средняя стоимость одного вызова трассировщика, выраженная задержкой выполнения исследуемой программы, остается неизменной при любой сложности вычислительной задачи и насыщенности инструментируемой программы дополнительным кодом. Этот факт подтверждает детерминированность вносимых задержек. Следовательно, это является необходимой и достаточной метрикой оценки производительности средства динамического инструментирования.

Подобный эксперимент был проведен на процессоре архитектуры x86 Core 2 Quad Q8400 с частотой 2.6 ГГц. Задержка, вносимая трассировкой одной функции, оставляет 0,02 мс, что позволяет судить о крайней эффективности реализации данного метода для архитектуры MIPS32.

Выводы

В процессе исследований, результаты которых приведены статье, были изучены методы динамического инструментирования. Был выбран метод перемещенного выполнения, на основе которого реализовано расширение программного средства DTrace для возможности инструментирования программного кода для процессоров архитектуры MIPS. Проведено сравнение быстродействия и внесения накладных расходов с реализацией метода для процессоров архитектуры x86. Дальнейшие исследования направлены на инструментирование и трассировку самомодифицирующегося кода.

Список литературы

1. Rosenberg J.B. How Debuggers work: algorithms, data structure, and architecture [Text] / J.B. Rosenberg. – Wiley, 1996. – 263 p.

2. Dynamic Code Instrumentation to Detect and Recover from Return Address Corruption [Text] / S. Gupta, P. Pratap, H. Saran, S. Arun-Kumar // International workshop on Dynamic systems analysis. – 2006. – №5. – P. 65.

3. Metz E. Efficient Instrumentation for Performance Profiling [Text] / E. Metz, R. Lencevicius // ICSE Workshop on Dynamic Analysis. – 2003. – №3. – P. 10.

4. Lencevicius R. Tracing Execution of Software for Design Coverage [Text] / R. Lencevicius, E. Metz, A. Ran // IEEE international conference on Automated software engineering. – 2001. – №16. – P. 328.

5. Sweetman D. See MIPS Run Linux [Text] / D. Sweetman. – Morgan Kaufmann, 2006. – 512 p.

6. Cantrill B.M. Dynamic Instrumentation of Production Systems [Text] / B.M. Cantrill, M.W. Shapiro, A.H. Leventhal // USENIX Annual Technical Conference. – 2004. – №10. – P. 15.

7. Bernat A.R. Efficient, Sensitivity Resistant Binary Instrumentation [Text] / A.R. Bernat, K. Roundy, B.P. Miller // International Symposium on Software Testing and Analysis. – 2011. – №11. – P. 89.

8. HMTT: A Hybrid Hardware/Software Tracing System for Bridging Memory Trace's Semantic Gap [Text] / Yungang Bao, Jinyong Zhang, Yan Zhu и др. // ACM SIGMETRICS – 2008. – №36. – P. 229.

Поступила в редколлегию 10.07.2012

Рецензент: д-р техн. наук, проф. С.Г. Удовенко, Харьковский национальный университет радиоэлектроники, Харьков.

МЕТОДИ ДИНАМІЧНОГО ІНСТРУМЕНТУВАННЯ КОДУ

О.Г. Руденко, С.В. Мірошниченко

У статті проводиться аналіз різних методів інструментування, в яких в код програми додаються невеликі ділянки коду, що вносять додатковий функціонал, як правило, для налагодження, дослідження або аналізу продуктивності. Метою даної роботи є дослідження методу переміщеного виконання. Основним завданням є побудова засоби динамічного інструментування апаратної платформи MIPS32. В результаті виконання роботи була реалізована підтримка даної архітектури на засіб інструментування DTrace.

Ключові слова: *інструментування, бінарний код, метод переміщеного виконання.*

METHODS FOR DYNAMIC INSTRUMENTATION OF BINARY CODE

O. G. Rudenko, S.V. Miroshnychenko

This paper describes dynamic instrumentation – method to instrument binaries by adding small code snippets to existing code fragments. Two main techniques of dynamic instrumentation are described: traditional single-step method and method of displaced execution. To prove effectiveness and safety of displaced execution technique it was implemented as additional module for Dtrace facility to support MIPS32 architecture.

Keywords: *instrumentation, binary code, displaced execution.*