Внесение неисправностей в программу с использованием детерминированного воспроизведения

П. М. Довгалюк, Ю. В. Маркин
ИСП РАН, Москва
{Pavel.Dovgaluk, ustas}@ispras.ru

Аннотация: В данной работе представлен подход к внесению неисправностей в программу, использующий механизм детерминированного воспроизведения в симуляторе. Внесение неисправностей используется для проверки качества покрытия кода тестами, способности системы реагировать на некорректные данные или механизма обработки исключений. При внесении неисправностей предлагается использовать детерминированное воспроизведение работы программ, что позволит упростить инициализацию системы, ускорить ее тестирование с помощью фаззинга, а также изучить альтернативные пути выполнения программы при ее отладке.

Ключевые слова: детерминированное воспроизведение; виртуальная машина; отладка; внесение неисправностей; фаззинг; симулятор; гейзенбаг; QEMU

1. Введение

Внесение неисправностей (fault injection) – это искажение кода или данных программы с целью проверки качества покрытия кода тестами, способности системы реагировать на некорректные данные или механизма обработки исключений [1].

Внесение неисправностей имеет ряд особенностей, которые затрудняют его применение в определенных случаях. Например, при выполнении фаззинга сетевого протокола может возникнуть потребность в обеспечении одинаковой реакции сервера на каждый инициализирующий пакет, посылаемый тестируемой программой. В некоторых случаях программа сама может быть недетерминированной и из-за этого вести себя по-разному даже на одинаковых наборах входных данных.

В статье предлагается метод, позволяющий решить эти проблемы с помощью детерминированного воспроизведения. Детерминированное воспроизведение программы заключается в записи сценария ее выполнения и последующем его (возможно, многократном) воспроизведении. Детерминированное воспроизведение чаще всего используется для отладки и динамического

анализа программ. Например, при анализе ошибок в многопоточных программах возможность наблюдать один и тот же сбой при каждом запуске программы очень важна. Присутствие множества побочных факторов, влияющих на ход выполнения такой программы, приводит к тому, что повторный запуск на тех же входных данных может задействовать другую траекторию выполнения. Существует специальный термин для обозначения ошибок, для которых «вручную» отследить все условия срабатывания не представляется возможным – гейзенбаг. Без специальных средств инструментальной поддержки процесса отладки, например, детерминированного воспроизведения, гарантировать, что в сложной многопоточной программе при тех же входных данных гейзенбаг проявится в том же месте (и что он вообще проявится) практически невозможно.

Предлагается дополнить метод детерминированного воспроизведения возможностью отклоняться от записанного сценария работы посредством замены части входных данных. Пользователь сможет значительно экономить свое время при анализе и отладке программ с использованием детерминированного воспроизведения. Например, отклонение от записанного хода выполнения программы может понадобиться при изучении покрытия кода тестами, обратной отладке или тестировании методом внесения неисправностей.

2. Подобные работы

Внедрение неисправностей может выполняться на этапе компиляции (мутационное тестирование) или на этапе выполнения программы. Например, XEMU [2] реализует мутационное тестирование в симуляторе при выполнении отдельных пользовательских приложений.

Другой подход к внесению неисправностей – это модификация программы или данных во время выполнения. Существует множество систем, использующих модификацию во время выполнения. Как правило, эти системы выполняют эмуляцию аппаратных неисправностей в микросхемах памяти или процессоре, а также в шинах данных [3], [4], [5], [6]. Другой вид неисправностей – это коммуникационные неисправности – потеря, задержка, искажение или дублирование сообщений [3], [7].

Фаззинг – это генерация заведомо некорректных входных данных для программы с целью вызова ее сбоя [8]. Наиболее популярный способ фаззинга на сегодняшний день – это обращение с системой как с черным ящиком. В этом случае при генерации входных данных не используется никаких знаний о тестируемой системе, кроме, возможно, формата принимаемых ею данных. В работе [9] рассматриваются подходы к фаззингу драйверов USB-устройств. Один из них заключается в помещении между USB-устройством и его драйвером (выполняющимся в виртуальной среде) «прослойки», позволяющей модифицировать или подменять передаваемые пакеты. Другой – в реализации аппаратного устройства, имитирующего требуемый протокол. Главное

достоинство программного решения, использующего виртуальную среду – это полный контроль над выполнением программы, а также возможность сохранять и восстанавливать состояние виртуальной машины. Проблема такого подхода заключается в невозможности эффективного изменения определенного пакета после того, как множество пакетов уже было передано, ведь поведение системы может меняться от запуска к запуску.

Другой подход к фаззингу – это использование символьного выполнения программы [10]. Проводится тестирование «белого ящика» – чтобы найти входные данные, приводящие к сбою программы, производится ее символьное выполнение с последующим решением уравнений, позволяющих восстановить данные, приводящие к повышению степени покрытия кода. Так происходит поиск новых путей исполнения программы. Если в результате на одном из путей программа достигает состояния, которое определяется как сбой, то в ней есть ошибка, проявляющаяся на найденных входных данных. Этот подход доказал свою эффективность на практике, описанной в работе [10], а также широким применением в компании Microsoft при тестировании программ, работающих с файлами сложного формата [11]. В то же время, существующие реализации обладают рядом ограничений. Большинство из них (например, [11], [12]) проводит анализ лишь отдельных программ, откуда следует невозможность анализа драйверов и операционных систем. Существует и реализация полносистемного анализа, описанная в [13]. В ней все необходимые данные постоянно хранятся в памяти, что делает невозможным анализ сложных программных компонентов, либо всего программного стека конкретного протокола. Кроме того, все описанные реализации обладают тем же недостатком, что и подходы к фаззингу черного ящика – невозможность тестирования всех фаз работы драйвера, работающего по определенному протоколу.

Использование детерминированного воспроизведения для решения задач, подобных внесению неисправностей, уже становилось предметом внимания исследователей и разработчиков ПО. В отладчике EPDB [14], реализующем метод обратной отладки, существует возможность изменения входных данных, чтобы отклониться от записанного сценария. Основное ограничение этого отладчика в том, что он существует лишь для программ на языке Python. К тому же, в настоящее время проект не развивается.

Другим примером, где имеется подобная возможность, является система динамического анализа Crosscut [15]. В ней детерминированное воспроизведение используется для сбора трасс программ, выполняющихся внутри виртуальной машины. Чтобы получать данные из гостевой операционной системы и запущенных в ней процессов, Crosscut в определенные моменты времени выполняет фрагменты кода, отличающиеся от тех, что выполнялись при записи сценария работы. Например, для трассировки программы на языке Perl в интерпретатор встраивается код, активируемый на определенных шагах воспроизведения. При этом перед

выполнением кода трассировки необходимо сохранять состояние процессора и оперативной памяти, чтобы затем использовать их для продолжения воспроизведения работы виртуальной машины.

Гораздо более близкий аналог предлагаемого метода реализуется в симуляторе Simics [16], [17]. Этот симулятор позволяет выполнять обратную отладку сложных (в том числе и состоящих из нескольких компьютеров) систем. Однако симулятор является закрытым, и никаких деталей о принципах и методах, лежащих в его основе, не публиковалось.

Отличие разработанного метода от отладчиков уровня приложений (таких как EPDB) заключается в том, что он работает на уровне всей системы и поэтому может применяться для изменения поведения на более низком уровне (например, при работе в ядре операционной системы). В реализациях подобных Crosscut возможно выполнение ограниченного фрагмента кода в нужной точке воспроизведения, но невозможно создание ответвлений с продолжением работы программы от произвольной точки некоторого сценария ее работы. Кроме того, предлагаемый метод позволяет выполнять полноценный фаззинг протоколов с тестированием полного стека реализации протокола, а не отдельных приложений в рамках операционной системы. Симулятор Simics ориентирован лишь на обратную отладку и не содержит в себе механизмов для фаззинга выполняющихся в нем программ.

Описанные выше программные системы, реализующие внесение неисправностей (и, в частности, фаззинг) обладают следующими недостатками. Многократные запуски одного сценария с частично изменяющимися входными данными могут иметь фазы работы, которые не отличаются от запуска к запуску. Это приводит к выполнению лишней работы при повторении этих фаз. Кроме того, программа может по-разному себя повести при разных запусках независимо от подаваемых входных данных, что изменит ход выполнения, и при тестировании могут проявиться совсем разные дефекты.

Вместо предлагаемого детерминированного воспроизведения для перехода к определенной фазе выполнения можно использовать предварительно сохраненные снимки системы. Но снимки занимают значительное место на диске (ведь нужно как минимум делать копию оперативной памяти системы), а на их создание затрачивается больше времени, чем на запись журнала недетерминированных событий. Поэтому сохранение снимков может повлиять на работу системы, замедляя ход ее выполнения, что сделает анализ приложений реального времени невозможным.

Для проверки результатов внесения неисправностей можно использовать сравнение трасс выполнения программ. Предлагаемый метод позволяет уменьшить число и объем используемых трасс. В качестве эталона можно использовать трассу оригинального сценария выполнения, а дополнительные трассы получать, начиная воспроизведение с моментов внесения изменений. Кроме того, использование детерминированного воспроизведения

гарантирует, что отличия в трассах выполнения будут обусловлены только внесенными неисправностями, а не какими-либо другими внешними факторами.

3. Описание метода

Детерминированное воспроизведение работы системы означает, что во время записи сценария работы с ней создается журнал недетерминированных событий.

Но полностью детерминированное (без возможности внести изменения) воспроизведение не всегда удобно. Например, при тестировании или отладке программ бывает необходимо изменить часть данных, поступающих в систему. В этом случае пользователю придется записать новый сценарий работы с программой. При этом в ряде случаев он может столкнуться с затруднениями:

1. Сценарий достаточно длительный, а изменения нужно внести ближе к его концу.
2. На работу программы влияют источники недетерминизма (многопоточность, неинициализированная память, внешняя среда), из-за чего сложно воспроизвести ту же самую ошибку, которая была записана в оригинальном сценарии.
3. Подготовка к запуску сценария требует сложной настройки окружения.

Поэтому предлагается не записывать новый сценарий, а создавать ответвление от текущего посредством изменения некоторых данных, поступающих из журнала в определенный момент воспроизведения. На рис. 1 изображен пример. Во время воспроизведения сценария программы в точке B была произведена замена записанных входных данных на другие. В результате, выполнение пошло по другому пути, и программа перешла в состояние C' вместо C, которое было бы достигнуто при нормальном воспроизведении.

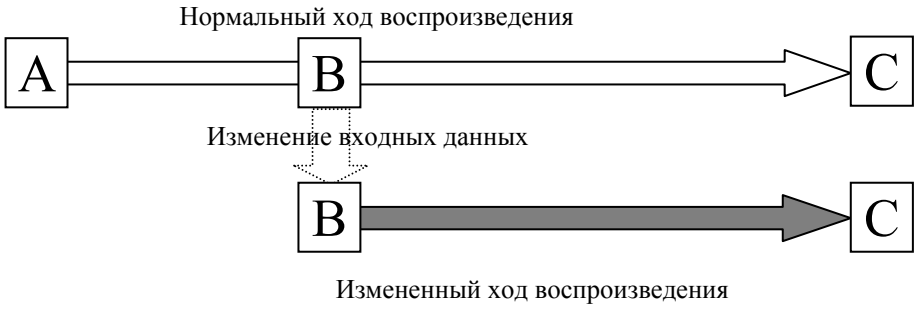

Рис. 1. Изменение состояния программы во время ее воспроизведения.

В рамках работы было реализовано несколько способов перехода к выполнению программы из режима ее воспроизведения. В зависимости от того, с какой целью восстанавливается выполнение программы, варианты могут быть следующие:

1. Изменение переменных, регистров или ячеек памяти с помощью отладчика.
2. Изменение получаемого сетевого пакета, записанного в журнал недетерминированных событий.
3. Включение с помощью специальной команды в произвольный момент воспроизведения журнала. Этот способ может быть использован для того, чтобы изменить входные данные, подаваемые в программу через ее пользовательский интерфейс.

В режиме воспроизведения работы программы весь ввод данных в систему симулируется – соответствующие события загружаются из журнала и передаются виртуальным устройствам ввода (мышь, клавиатура, сетевая карта, микрофон). После переключения на режим выполнения, виртуальные мышь, клавиатура и микрофон снова становятся доступны для передачи команд от пользователя.

Во время воспроизведения работы системы весь ввод в нее поступает из заранее записанного журнала событий. В тот момент, когда воспроизведение прерывается и начинается нормальное выполнение, необходимо прекратить чтение событий из журнала и начать взаимодействие с пользователем.

Основная проблема при переключении на источники входных данных заключается в том, что источники реального времени также считаются внешними устройствами и их показания записываются в журнал событий. Поэтому при переходе в режим выполнения программы из режима воспроизведения, показания счетчиков времени должны соответствовать новому состоянию системы.

Реализация детерминированного воспроизведения, используемая в данной работе, основана на симуляторе QEMU [18] и подробно описана в работе [19]. Для реализаций платформ i386 и ARM, которые поддерживают детерминированное воспроизведение, симулируются следующие аппаратные счетчики времени:

- TSC (Time Stamp Counter) – счетчик тактов с последнего сброса процессора. Возвращается как результат работы инструкции RDTSC процессора i386.
- i8254 – микросхема программируемого таймера, используемая начиная с самых первых моделей IBM PC.
- MC146818 – микросхема часов реального времени, впервые появившаяся в IBM PC/AT.

- HPET (High Precision Event Timer) – таймер событий высокой точности, призванный заменить i8254 и MC146818 в задачах замера небольших промежутков времени.
- ACPI PM timer (Advanced Configuration and Power Interface Power Management timer) – таймер в составе интерфейса управления конфигурацией и питанием.
- PL031 – часы реального времени ARM PrimeCell.
- DS1338 – микросхема часов реального времени с интерфейсом I2C.

Программа не должна «заметить» переключение симулятора из режима воспроизведения в режим выполнения по необычной работе одного из таймеров – в противном случае работа программы может по тем или иным причинам измениться по сравнению с ожиданиями пользователя. Это изменение может быть случайно или намеренно заложено программистом. Например, в [20] описывается ряд антиотладочных приемов, основанных на использовании инструкции RDTSC.

В QEMU все виртуальные счетчики времени реализованы через использование нескольких источников реального времени хостовой машины. Они же используются для измерения периодов, не относящихся к часам, например, промежутков времени между обновлениями экрана:

- Функция QueryPerformanceCounter (Windows) или clock_gettime (Linux)
- Функция gettimeofday
- Функция gmtime_r
- Функция time
- Счетчик тактов хостового процессора

Все эти источники являются внешними для виртуальной машины, и поэтому обращения к ним записываются в журнал событий. Во время воспроизведения нужные показания источников реального времени извлекаются из журнала. Таким образом, в случае переключения из режима воспроизведения в режим выполнения программы значения таймеров моментально изменятся на текущие для хостовой машины значения, поскольку записанные показания таймеров отличаются от текущих.

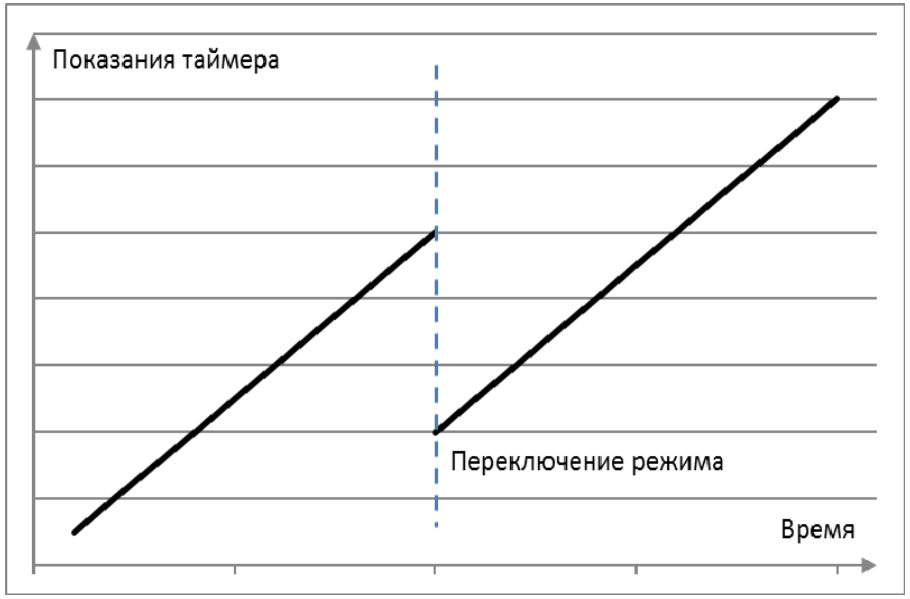

Рис. 2. Изменение показаний таймера при переключении из режима воспроизведения в режим выполнения.

Чтобы такого «скачка времени» не происходило, необходимо изменить способ вычисления значений таймеров в режиме выполнения. При передаче показаний таймера хостовой машины виртуальным устройствам нужно вычитать из них смещение – разницу между показаниями таймера хоста и показаниями таймера, считанными из журнала событий в тот момент, когда происходит переключение режима.

С подключением реальных устройств ввода к клавиатуре, мыши или микрофону таких проблем не возникает, так как они получают извне только требуемые изменения в своем состоянии. Поэтому скачков в передаваемых в гостевую систему значениях не происходит.

Совершенно иначе обстоит дело с сетевым адаптером. Здесь получаемые системой пакеты формируются на удаленной машине. А ее состояние невозможно восстановить при переходе программы в режим выполнения. Поэтому при подмене сетевого пакета дальнейшее взаимодействие программы с внешними сервисами уже не имитируется. Тем не менее, можно проверить непосредственную реакцию программы на заменяемый пакет с учетом всего предыдущего взаимодействия (т.е. осуществлять фаззинг сетевого протокола). Но ту часть взаимодействия с сервером, которая должна следовать за подмененным пакетом, проверить этим способом не удастся.

4. Применение метода

В качестве примера продемонстрируем возможности предлагаемого метода для фаззинга сетевого протокола. Для этого было разработано приложение, подключающееся к запущенному симулятору и выполняющее несколько итераций фаззинга заданного сетевого пакета. Каждая итерация состоит из трех этапов:

- переход на начальный шаг исследуемого сценария;
- изменение содержимого входящего сетевого пакета;
- продолжение выполнения программы до момента наступления некоторого события.

Отличительная особенность фаззера – восстановление контекста обработки сетевых пакетов на каждой новой итерации (для stateful-протоколов). Это происходит благодаря детерминированному воспроизведению при переходе на начальный шаг исследования – в каждой итерации будет получено одно и то же состояние системы после выполнения этого перехода.

4.1. Входные параметры

Чтобы начать фаззинг сетевого приложения, необходимо задать следующие параметры:

- Список динамических библиотек, производящих разбор (построение) сетевых пакетов. Для того чтобы модифицировать сетевые пакеты, необходимо осуществлять разбор этих пакетов. Структура полезной нагрузки сетевого пакета определяется приложением, в работе которого данный сетевой пакет используется. Для каждого нового формата создается динамическая библиотека, в которую помещается код разборщика (построителя) сетевых пакетов в соответствии с заданным форматом.
- Идентификатор сетевого пакета, содержимое которого в рамках данного исследования будет изменяться. Для этого пакета необходимо указать динамическую библиотеку, при помощи которой будет производиться разбор (построение), а также файл с набором параметров, необходимых для разбора (например, ключи шифрования)
- Список целевых адресов. Если значение счетчика инструкций процессора совпадет хотя бы с одним значением из этого списка, задача фаззинга будет считаться выполненной. По целевому адресу не обязательно должен располагаться код анализируемого сетевого приложения – там, к примеру, может находиться код динамической библиотеки, которую анализируемое приложение использует в своей работе.

- Начальный и конечный шаги исследования. Шаги журнала событий, которые соответствуют получению выбранных для модификации сетевых пакетов, должны лежать в диапазоне начального и конечного шагов исследования. Попадание управления на какой-либо адрес из списка целевых адресов должно произойти в диапазоне начального и конечного шагов исследования. Таким образом, в качестве начального шага исследования всегда подходит шаг журнала событий, соответствующий получению сетевого пакета, содержимое которого будет изменено. В качестве конечного шага исследования можно выбирать достаточно большие значения. В то же время, чем меньше разность конечного и начального шагов, тем меньше времени потребуется для проведения одной итерации исследования.
- Модифицирующий файл. Необходимо указать файл, содержимое которого будет использовано для модификации сетевого пакета.

4.2. Критерий остановки

Выполнение программы, использующей измененный сетевой пакет, прекращается, если наступает одно из событий:

- достижение конечного шага исследования;
- аварийное завершение работы эмулятора.

Каждая итерация характеризуется отдельным файлом, в который записываются инструкции, исполняемые симулятором во время этой итерации. В случае наступления одного из перечисленных выше событий, сохранение исполняемого кода прекращается и начинается его анализ. Список целевых адресов сопоставляется с адресами инструкций файла данной итерации. Если найдено хотя бы одно совпадение, анализатор принимает решение о прекращении исследования. В противном случае начинается новая итерация. В случае аварийного завершения работы симулятора происходит его перезапуск.

4.3. Тестирование

Анализ корректности работы фаззера проводится на контрольном примере. Используется предварительно записанный сценарий выполнения программы «FuzzTest.exe», а также файл «Original.pcap», в котором сохранен трафик виртуального сетевого интерфейса симулятора. Программа «FuzzTest.exe» отправляет сетевой пакет, содержащий запрос ip-адресов серверов с доменным именем «lenta.ru». После получения ответа программа сравнивает значение последнего из пришедших ip-адресов с ip-адресом «0.0.0.0». В случае совпадения происходит ошибка – разыменование нулевого указателя.

Перед запуском фаззера необходимо найти адрес инструкции, выполнение которой приводит к ошибке. Это можно сделать с помощью дизассемблера IDA Pro. Интересующая нас инструкция находится по адресу «0x40135E». Для

работы с пакетами протокола DNS используется модуль «Nslookup.dll». В окне добавления сетевого пакета вводим идентификатор пакета (349). Никаких дополнительных параметров в данном случае не требуется.

После нажатия на кнопку «Start Fuzzing» появится окно, отображающее промежуточное представление выбранного сетевого пакета (рис. 3). Необходимо выбрать *модифицирующий* файл. Каждый модифицирующий файл представляет собой последовательность блоков. Блок включает в себя идентификатор вершины промежуточного представления, а также набор новых значений для модификаций:

```
[id_1]
id_1_new_value_1
id_1_new_value_2
...
id_1_new_value_n_1

[id_2]
id_2_new_value_1
id_2_new_value_2
...
id_2_new_value_n_2
...
[id_m]
id_m_new_value_1
id_m_new_value_2
...
id_m_new_value_n_m
```

В качестве модифицирующего использовался файл следующего содержания:

```
[24]
01020304
19203940
00000000
```

На рис. 3 тег с идентификатором «24» соответствует ip-адресу, который будет изменяться. В данном случае будет выполнено три итерации, то есть ip-адрес будет изменен три раза, так как именно столько различных IP-адресов записано в модифицирующем файле. По окончании работы фаззера в консоль выводится сообщение «Fuzzing completed». В папке «FuzzerLog» появилась папка с результатами фаззинга, содержащая четыре файла: «asm_log_1.txt», «asm_log_2.txt», «asm_log_3.txt», «fuzz_log.txt». В первых трех файлах находится ассемблерный код, исполняемый виртуальной машиной. В последнем файле для каждой проведенной итерации сохранен список соответствующих изменений сетевого пакета. В поле «Result» последней итерации записано значение «success». Это значит, что на последней итерации симулятор выполнил интересующую нас инструкцию по адресу «0x40135E».

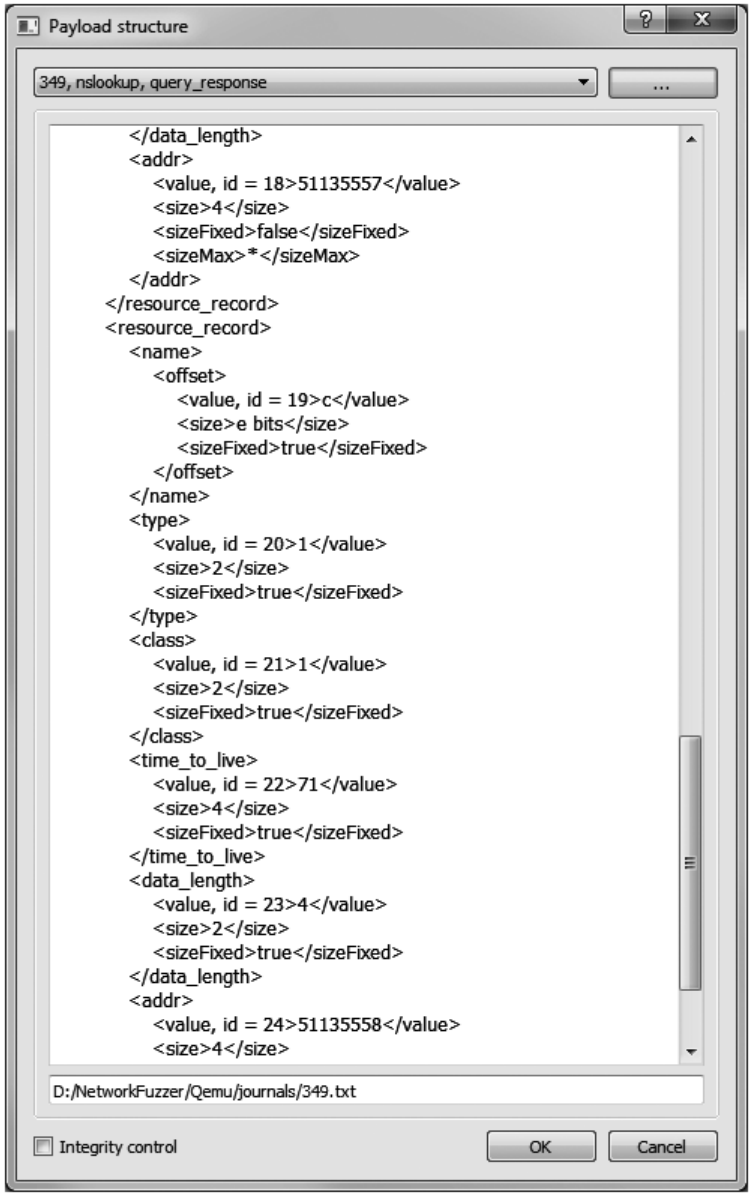

Рис. 3. Окно промежуточного представления.

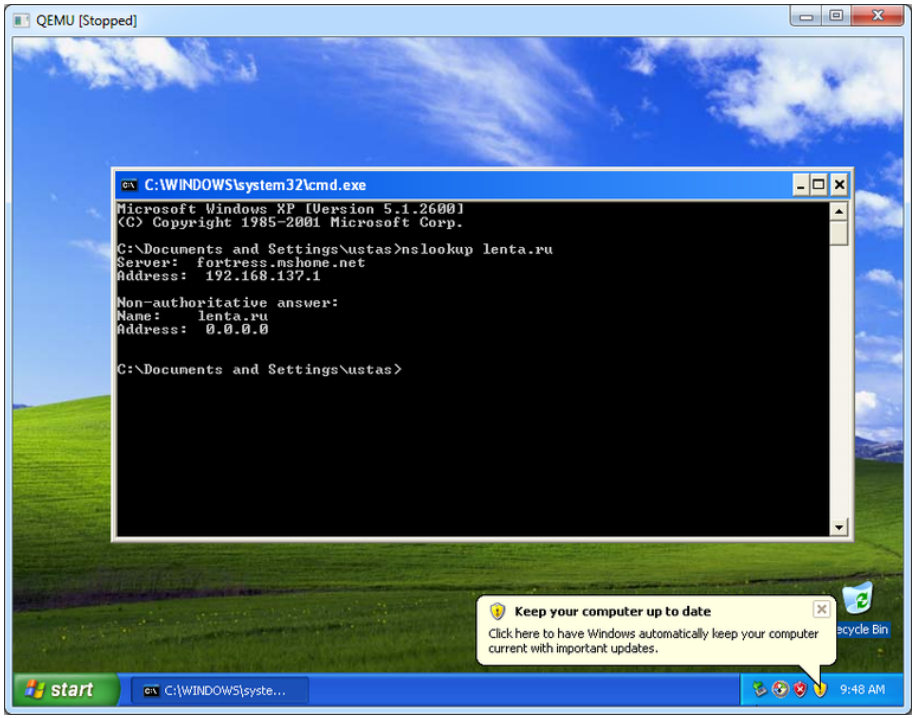

Рис. 4. Результат последней итерации фаззинга.

5. Заключение

В рамках работы был разработан метод, позволяющий переходить от воспроизведения программ к их выполнению. Благодаря этому методу расширяются возможности по внесению неисправностей, а также отладке приложений. Например, при фаззинге сетевых протоколов каждый сгенерированный сетевой пакет не требует нового сеанса взаимодействия с удаленным компьютером в отличие от обычных методов тестирования. Таким образом, разработанный метод способен сократить время, затрачиваемое на тестирование как клиентских, так и серверных приложений.

Предполагается дальнейшее развитие метода по ряду направлений. Во-первых, при отладке может оказаться полезной возможность построения дерева путей выполнения программы, по которому можно передвигаться вверх и вниз (назад и вперед во времени), переключаясь между ветвями и изучая состояние программы в зависимости от входных данных и взаимодействия с пользователем.

Второе направление заключается в разработке средств символьного выполнения. Само по себе символьное выполнение работает медленно, и это может повлиять на realtime-приложения. Использование детерминированного воспроизведения может позволить анализировать такие приложения. Похожий подход реализован в системе динамического анализа SAGE [11], которая работает на уровне отдельных приложений.

Еще одно направление исследований – это дополнение разработанного метода для проведения более полного фаззинга сетевых протоколов. Текущая реализация не позволяет продолжить сетевое взаимодействие после изменения определенного пакета. Решением этой проблемы могут стать синхронные запись и воспроизведение работы сразу нескольких виртуальных машин, например как в симуляторе Simics [16].

Аналогично фаззингу сетевых протоколов, можно использовать детерминированное воспроизведение и при фаззинге протоколов взаимодействия USB-устройств с виртуальной машиной. USB-устройство является внешним по отношению к виртуальной машине, поэтому при фаззинге протокола возникают те же сложности, что и при работе с удаленной системой по сети. Например, в работе [9] описаны подходы к фаззингу протоколов работы с USB-устройствами. Авторам удалось обеспечить устойчивое воспроизведение найденных сбоев лишь при подмене инициализирующего пакета, а в остальных случаях недетерминированность обмена с устройством не давала возможности воспроизводить сбои.

Список литературы

[1] Bieman J. M., Dreilinger D., Lin L. Using Fault Injection to Test Software Recovery Code // Final report, Colorado advanced software institute, 1995, 48 pages.

[2] Becker M., Baldin D., Kuznik C., Joy M. M., Xie T., Mueller W. XEMU: an efficient QEMU based binary mutation testing framework for embedded software. // Proceedings of the Tenth ACM International Conference on Embedded Software, ACM New York, NY, USA 2012, pp. 33-42.

[3] Han S., Shin K. G., Rosenberg H. A. DOCTOR: An Integrated Software Fault Injection Environment for Distributed Real-time Systems. // In Proc. 2nd Annual IEEE Int. Computer Performance and Dependability Symp. (IPDS'95). Erlangen, Germany, 1995. P. 204-213.

[4] Kanawati J., Abraham J. FERRARI: A Tool for the Validation of System Dependability Properties. // In Proc. 22nd IEEE Int. Symp. on Fault Tolerant Computing (FTCS-22). Boston, Massachusetts, USA, 1992. P. 336-344.

[5] Segall Z., Vrsalovic D., Siewiorek D., Yaskin D., Kownacki J., Barton R., Dancey A., Robinson T. FIAT – Fault Injection Based Automated Testing Environment. // In Proc. 18th IEEE Int. Symp. on Fault Tolerant Computing (FTCS-18). Tokio, Japan, 1988. P. 102-107.

[6] Li Y., Xu P., Wan H. A Fault Injection System Based on QEMU Simulator and Designed for BIT Software Testing // Applied Mechanics and Materials, vol. 347-350, 2013. pp. 580-587.

[7] Dawson S., Jahanian F., Mitton T. ORCHESTRA: a probing and fault injection environment for testing protocol implementations. In proceeding of: Computer Performance and Dependability Symposium, 1996, page 56.
[8] B.P. Miller, L. Fredriksen, and B. So, An Empirical Study of the Reliability of UNIX Utilities, Communications of the ACM 33, 12 (December 1990), pages 32-44
[9] Jodeit M., Johns M. USB Device Drivers: A Stepping Stone into Your Kernel. // Proceedings of the 2010 European Conference on Computer Network Defense, IEEE Computer Society, Washington, DC, USA, 2010. pp. 46-52
[10] Cha S. K., Avgerinos T., Rebert A., Brumley D. Unleashing mayhem on binary code. // SP '12 Proceedings of the 2012 IEEE Symposium on Security and Privacy. IEEE Computer Society Washington, DC, USA 2012, pp. 380-394.
[11] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. Queue 10, 1, Pages 20 (January 2012), 8 pages.
[12] The KLEE Symbolic Virtual Machine. http://klee.github.io/klee, 21.04.2014
[13] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: a platform for in-vivo multi-path analysis of software systems. SIGARCH Comput. Archit. News 39, 1 (March 2011), 265-278
[14] EPDB – a reversible debugger for Python. https://code.google.com/p/epdb, 21.04.2014
[15] Chow J., Lucchetti D., Garfinkel T., Lefebvre G., Gardner R., Mason J., Small S., Chen P. M. Multi-stage replay with Crosscut // VEE '10 Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments. – New York, NY, USA : ACM, 2010. – pp. 13-24.
[16] Jakob Engblom . Back to Reverse Execution. http://blogs.windriver.com/tools/2013/06/back-to-reverse-execution.html, 21.04.2014
[17] Engblom, J. A review of reverse debugging. System, Software, SoC and Silicon Debug Conference (S4D), 2012, pages 1-6.
[18] Bellard F. QEMU, a fast and portable dynamic translator. // In USENIX 2005 Annual Technical Conf. pages 41–46, Apr. 2005.
[19] Довгалюк П. Детерминированное воспроизведение процесса выполнения программ в виртуальной машине / Труды Института системного программирования РАН. Т. 21 / под ред. В.П. Иванникова. М.: ИСП РАН. 2011. С. 123-132.
[20] Bania P. Playing with RDTSC. http://www.piotrbania.com/all/articles/playing_with_rdtsc.txt, 21.04.2014

Using Deterministic Replay for Software Fault Injection

P. M. Dovgalyuk, Y. V. Markin
ISP RAS, Moscow, Russia
{Pavel.Dovgaluk, ustas}@ispras.ru

Abstract: This paper presents method of improving software fault injection by using deterministic replay. Fault injection and fuzzing are the methods of testing used for checking code coverage quality, improving error handling, and robustness testing. Fuzzing can hardly be applied for stateful communication protocols because of program state could change when restarting an application. The main idea of our method is to inject faults while replaying program deterministically. Deterministic replay requires program execution recording for latter replaying. Recorded log includes user input, incoming network packets, USB input, and hardware timers. During replay we read these events from the log and put them back into the simulator instead of reading inputs or receiving packets from the network. After injecting the fault in replay mode the program execution is different. It means that we should stop the replaying and start normal program execution from that program state. During the execution we simulate all hardware timers to make this mode switching imperceptible to the program. With the help of deterministic replay we can accelerate system initialization, eliminate non-deterministic data sources effect, and simplify environment setup, because the whole program execution before injecting fault is recorded. On the basis of the method the network fuzzer was built. The fuzzer modifies selected network packet saved during session recording and sends it back into the simulator. This phase is repeated from the same program state until the bug in the program was found.

Keywords: deterministic replay; virtual machine; debugging; fault injection; fuzzing; simulator; heisenbug; QEMU

References

[1] Bieman J. M., Dreilinger D., Lin L. Using Fault Injection to Test Software Recovery Code. Final report, Colorado advanced software institute, 1995, 48 pages.
[2] Becker M., Baldin D., Kuznik C., Joy M. M., Xie T., Mueller W. XEMU: an efficient QEMU based binary mutation testing framework for embedded software. Proceedings of the Tenth ACM International Conference on Embedded Software, ACM New York, NY, USA 2012, pp. 33-42. doi: 10.1145/2380356.2380368
[3] Han S., Shin K. G., Rosenberg H. A. DOCTOR: An Integrated Software Fault Injection Environment for Distributed Real-time Systems. In Proc. 2nd Annual IEEE Int. Computer Performance and Dependability Symp. (IPDS'95). Erlangen, Germany, 1995. P. 204-213. doi: 10.1109/IPDS.1995.395831
[4] Kanawati J., Abraham J. FERRARI: A Tool for the Validation of System Dependability Properties. In Proc. 22nd IEEE Int. Symp. on Fault Tolerant Computing (FTCS-22). Boston, Massachusetts, USA, 1992. P. 336-344. doi: 10.1109/FTCS.1992.243567
[5] Segall Z., Vrsalovic D., Siewiorek D., Yaskin D., Kownacki J., Barton R., Dancey A., Robinson T. FIAT – Fault Injection Based Automated Testing Environment. In Proc. 18th IEEE Int. Symp. on Fault Tolerant Computing (FTCS-18). Tokio, Japan, 1988. P. 102-107. doi: 10.1109/FTCS.1988.5306

[6] Li Y., Xu P., Wan H. A Fault Injection System Based on QEMU Simulator and Designed for BIT Software Testing. Applied Mechanics and Materials, vol. 347-350, 2013. pp. 580-587.
[7] Dawson S., Jahanian F., Mitton T. ORCHESTRA: a probing and fault injection environment for testing protocol implementations. In proceeding of: Computer Performance and Dependability Symposium, 1996, page 56. doi: 10.1109/IPDS.1996.540200
[8] B.P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. Communications of the ACM 33, 12 (December 1990), pages 32-44. doi: 10.1145/96267.96279
[9] Jodeit M., Johns M. USB Device Drivers: A Stepping Stone into Your Kernel. Proceedings of the 2010 European Conference on Computer Network Defense, IEEE Computer Society, Washington, DC, USA, 2010. pp. 46-52. doi: 10.1109/EC2ND.2010.16
[10] Cha S. K., Avgerinos T., Rebert A., Brumley D. Unleashing mayhem on binary code. SP '12 Proceedings of the 2012 IEEE Symposium on Security and Privacy. IEEE Computer Society Washington, DC, USA 2012, pp. 380-394. doi: 10.1109/SP.2012.31
[11] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. Queue 10, 1, Pages 20 (January 2012), 8 pages. doi: 10.1145/2090147.2094081
[12] The KLEE Symbolic Virtual Machine. http://klee.github.io/klee
[13] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: a platform for in-vivo multi-path analysis of software systems. SIGARCH Comput. Archit. News 39, 1 (March 2011), 265-278. doi: 10.1145/1961295.1950396
[14] EPDB – a reversible debugger for Python. https://code.google.com/p/epdb
[15] Chow J., Lucchetti D., Garfinkel T., Lefebvre G., Gardner R., Mason J., Small S., Chen P. M. Multi-stage replay with Crosscut. VEE '10 Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments. – New York, NY, USA : ACM, 2010. – pp. 13-24. doi: 10.1145/1735997.1736002
[16] Jakob Engblom . Back to Reverse Execution. http://blogs.windriver.com/tools/2013/06/back-to-reverse-execution.html
[17] Engblom, J. A review of reverse debugging. System, Software, SoC and Silicon Debug Conference (S4D), 2012, pages 1-6.
[18] Bellard F. QEMU, a fast and portable dynamic translator. In USENIX 2005 Annual Technical Conf. pages 41–46, Apr. 2005.
[19] Dovgalyuk P. Determinirovannoe vosproizvedenie protsessa vypolneniya programm v virtual'noj mashine [Deterministic replay of software in virtual machine]. Trudy Instituta sistemnogo programmirovaniya RAN [The Proceedings of ISP RAS], T. 21, pod red. V.P. Ivannikova. M.: ISP RAN. 2011. S. 123-132. (in Russian)
[20] Bania P. Playing with RDTSC. http://www.piotrbania.com/all/articles/playing_with_rdtsc.txt