Применение программных эмуляторов в задачах анализа бинарного кода ⋆

Довгалюк П.М., Макаров В.А., Падарян В.А., Романеев М.С., Фурсова Н.И.
{Pavel.Dovgaluk, vladimir_makarov, vartan, melon, Natalia.Fursova}@ispras.ru

Аннотация. В статье приводится опыт применения программных эмуляторов в качестве средства динамического анализа бинарного кода: как трассировщика уровня машинных команд и как развитого инструмента интерактивной отладки. Описывается механизм детерминированного воспроизведения, реализованный в эмуляторе QEMU, позволивший обеспечить указанные функциональности.

Ключевые слова: эмулятор, динамический анализ, детерминированное воспроизведение, обратная отладка

1. Введение

Программные эмуляторы успешно применяются в различных областях: как средство консолидации серверов, во время кросс-платформенной разработки, при замещении устаревшей аппаратуры. В последние годы эмуляторы активно начали применять при анализе бинарного кода. Например, изучение вредоносного кода, как правило, ведется в виртуальной машине, а не на реальной аппаратуре. Помимо вопроса защиты окружения от влияния изучаемого кода, решается и обратная задача – разграничивается предмет изучения и инструмент анализа. Факт работы отладчика в общем окружении легко выявляется, после чего изучаемая программа может либо целенаправленно изменить свое поведение, либо попытаться нарушить процесс отладки.

Помимо интерактивной отладки, программные эмуляторы позволяют собирать детальные трассы выполнения уровня машинных команд, которые затем можно анализировать с высокой степенью автоматизации. Разработанная в ИСП РАН среда анализа [1] использует именно такой подход. Возможность выполнения анализа базируется на том, что для целевой процессорной архитектуры существует трассирующий полносистемный эмулятор, т.е. эмулятор, в котором работает операционная система, драйвера устройств, пользовательские приложения. В данной статье приводится опыт работы с такими эмуляторами, а также описываются ключевые

⋆ Работа поддержана грантом РФФИ 12-01-31417

усовершенствования, внесенные в эмулятор с открытым исходом QEMU, ставший основным средством сбора трасс.

2. Опыт сбора трасс в программных эмуляторах

Основной способ добиться понимания, как устроена программа, базируется на анализе потоков управления и данных, включая знания о том, какие значения имеют обрабатываемые данные. Запись одной только последовательности выполнившихся команд не может обеспечить успешность анализа. Для вычисления исполнительных адресов в случае косвенной адресации необходимо знать содержимое регистров общего назначения. Дополнительно необходимы: счетчик команд, слово состояния, значения некоторых системных регистров, например, отвечающих за декодирование и порядок выполнения команд, идентификацию процессов. Перечисленные данные обязательны для восстановления потока управления.

Помимо того, крайне важно контролировать входящие и выходящие потоки данных в виртуальной машине. Если изучается программа, взаимодействующая по сети, то необходимо вести трассу входящих сетевых пакетов. Аналогично, но на другом уровне, требуется вести трассу записей в оперативную память, осуществляемых периферийными DMA-устройствами. В случае внешней памяти (винчестер, USB-накопитель и т.д.) важно уметь отслеживать номера блоков, в которые пишутся данные и из которых идет чтение; знание номеров блоков позволяет более точно восстанавливать поток данных.

Однако не все программные эмуляторы могут быть использованы для исчерпывающего сбора данных. В процессе развития среды анализа бинарного кода произошла смена нескольких полносистемных эмуляторов, использующихся в качестве основного средства сбора трасс. Первый трассировщик был реализован для архитектуры x86 в рамках эмулятора Simics [2] компании Virtutech. Simics поддерживает значительное количество целевых архитектур, куда кроме x86 входят ARM, PowerPC, MIPS, и множество периферийных устройств. Эмулятор обладает развитым и удобным SDK, который позволяет разрабатывать расширения, но при этом сам эмулятор – закрытый коммерческий продукт, полученный для исследований в рамках академической лицензии. В 2010 году компания Virtutech была поглощена Intel и в настоящее время доступность этого эмулятора стала еще более ограниченной. Использование трассировщика в Simics было недолгим из-за того, что в результате исследований в эмуляторе были выявлены неточности. Это привело к тому, что сбор трассы был перенесен в эмулятор SimNow [3] компании AMD. SimNow поддерживает исключительно процессоры производства AMD, но зато обеспечивает большую точность. Однако и он не свободен от ошибок, исправление некоторых сильно затягивалось. Тем не менее, именно этот эмулятор долгое время оставался

основным средством сбора трасс для архитектуры x86 и продолжает использоваться по сей день.

Расширение среды анализа на другие процессорные архитектуры потребовало создания новых трассировщиков. Эмулятор с открытым исходным кодом Dynamips [4] использовался для получения трасс на архитектурах MIPS и PowerPC. Однако его применение ограничено, поскольку в нем эмулируются конкретные модели сетевых маршрутизаторов Cisco. Еще одним негативным фактором стало отсутствие сообщества разработчиков – в период своего существования (2005-2008 гг.) проект фактически развивался единственным человеком. Следует упомянуть, что летом 2013 г. группа из трех заинтересованных разработчиков попыталась возродить проект. Выпускаемый ей эмулятор dynamips-community [5] основан на «официальной» кодовой базе Dynamips, новшества в основном заключаются в исправлении ошибок, улучшении процесса сборки, разработке документации.

Для сбора трасс на архитектуре ARM в первую очередь был разработан трассировщик на основе эмулятора ARMulator [6], который входит в состав ARM Development Studio компании ARM Holdings. Следует различать данный полносистемный эмулятор с одноименным проектом, в котором пытаются разработать эмулятор уровня приложения. К сожалению, скорость работы самого эмулятора оказалась неприемлемо низкой, а накладные расходы, связанные с трассировкой, только ухудшили ситуацию. В качестве альтернативы был выбран эмулятор с открытым исходным кодом QEMU [7], который помимо ARM поддерживает большое количество других процессорных архитектур.

Вне зависимости от того, насколько точно перечисленные выше эмуляторы выполняют команды целевых архитектур, все они страдают от одной существенной проблемы – крайне низкой скорости работы во время снятия трассы выполнения. Замедление эмулятора по сравнению с обычной работой составляет 4-5 порядков. При анализе «замкнутых» программ такое замедление негативно сказывается только на требуемом времени. Но если программа состоит из нескольких частей и происходит взаимодействие по сети одной (исследуемой) части программы с другой (неподконтрольной), то замедление при сборе трассы гарантированно повлияет на работу исследуемого кода: как минимум вызовет обрыв сетевых соединений из-за превышения времени ожидания, а в худшем случае поменяет поведение исследуемых алгоритмов. Описанная проблема делает необходимым наличие скоростной трассировки, когда удаленная часть программы не сможет зафиксировать замедления работы других частей, возникающего из-за сбора трассы.

Единственным известным на данный момент способом организации скоростной трассировки без ограничений по длительности сбора трассы является так называемая двухпроходная трассировка на основе детерминированного воспроизведения работы виртуальной машины [8].

Детерминированное воспроизведение – это процесс восстановления хода выполнения программы с использованием заранее записанных входных данных. Происходит два запуска программы: во время первого все входные данные (события) записываются в журнал, во время второго – считываются из журнала. Цель детерминированного воспроизведения – максимально точно повторить ход выполнения программы, которое происходило во время записи событий. В нашем случае в качестве «программы» выступает снимок состояния виртуальной машины, входные данные – внешние, недетерминированные события: асинхронные прерывания, пользовательский ввод, входящие сетевые пакеты.

Полносистемное воспроизведение имеет такие преимущества, как возможности отладки и анализа системных компонентов и многопоточных приложений, а также возможность выполнять анализ любой операционной системы из числа поддерживаемых аппаратной платформой. Когда журнал событий записан, появляется возможность его многократного воспроизведения с целью анализа поведения программы.

Существует ряд подходов к реализации полносистемного воспроизведения. Все они основаны на использовании различных виртуальных машин, и, в зависимости от технологии, используемой при реализации виртуальной машины, могут быть разбиты на три группы: аппаратная виртуализация, «чистая» программная эмуляция, бинарная трансляция.

Первый подход заключается в перехвате внешних событий средствами аппаратной виртуализации; в качестве примеров реализации можно привести системы XenLR [9] и XTRec [10]. Они отличаются невысокими накладными расходами в процессе записи журнала. В качестве недостатка можно отметить, что все эти системы детерминированного воспроизведения ориентированы только на аппаратную платформу x86. Кроме того, XenLR ограничивается одной модельной платформой: операционная система MiniOS собственной разработки и три периферийных устройства (таймер, клавиатура, жесткий диск), что не позволяет применять его для анализа сколь либо содержательных программ.

Второй вариант реализации детерминированного воспроизведения (система ExecRecorder) был реализован в эмуляторе Bochs [11]. Этот эмулятор поддерживает только платформу x86 и работает значительно медленнее, чем основанные на виртуализации и динамической трансляции аналоги.

Третий вариант подхода к реализации детерминированного воспроизведения основан на виртуальной машине с динамической трансляцией бинарного кода. Наиболее интересные результаты были заявлены в публикациях сотрудников VMware [12, 13]: объем журнала – в среднем 4.8 байта на каждую 1000 выполнившихся машинных команд, замедление – около 5% при включении записи журнала. В отладочной версии 6.5 VMware Workstation присутствовал механизм воспроизведения, причем при проигрывании журнала присутствовала возможность получения трассы машинных команд,

дополненных содержимом регистров общего назначения. Проведенные эксперименты показали перспективность подхода, замедление при получении журнала действительно оказалось незначительном, что позволяло анализировать код нового класса программ – работающих с сетью. Однако информации, содержащейся в трассе, было недостаточно. Например, отсутствовали данные о прерываниях и изменениях системных регистров. В отдельных случаях некоторые требуемые данные удавалось восстановить, исходя из выполнявшихся команд и их адресов, но полноценный анализ трассы был невозможен. Более того, в следующем выпуске, Workstation 7.0, трассировка уже не поддерживалась, была предоставлена только возможность обратной отладки. А вскоре поддержка детерминированного воспроизведения была полностью остановлена по причине перехода виртуальной машины Workstation с бинарной трансляции на аппаратную виртуализацию.

Было решено реализовать детерминированное воспроизведение на основе программного эмулятора с открытым исходным кодом. Потенциально пригодными для такой реализации являлись три эмулятора: Bochs, QEMU и VirtualBox [14]. Все перечисленные эмуляторы активно используются в индустрии и обладают сообществами разработчиков. Но VirtualBox и Bochs рассчитаны только на одну целевую архитектуру – x86, а Bochs, как уже упоминалось выше, не использует бинарную трансляцию, что негативно сказывается на скорости работы. Таким образом, эмулятор QEMU представился наиболее подходящей платформой для реализации детерминированного воспроизведения: открытый код, приемлемая скорость работы, поддержка большого количества целевых архитектур (x86, ARM, MIPS, PPC, …), эксплуатация на промышленном уровне и сообщество квалифицированных разработчиков, постоянно улучшающих возможности эмулятора. Более того, производители мобильных устройств нередко используют QEMU как базовую платформу для создания «официальных» эмуляторов в составе распространяемых SDK, в качестве примеров можно привести такие платформы, как Symbian, Android, Maemo и MeeGo. Все это позволяет рассматривать QEMU как единую платформу для получения трасс машинных команд различных целевых архитектур и различных классов исследуемых программ.

Следует упомянуть, что спустя незначительное время с начала исследований вышла публикация коллектива тайваньских исследователей о системе FREE [15], в рамках которой детерминированное воспроизведение также было реализовано на основе QEMU. Исходный код системы был не доступен, но авторы заявляли о быстродействии, превосходящем аналог на основе Bochs и уступающем аппаратной виртуализации. Возможности FREE ограничивались только архитектурой x86 и периферийными устройствами не использующими DMA. Дальнейших публикаций об этой системе не наблюдалось, но результаты, заявленные в [15], подтверждали верность выбора QEMU.

3. Устройство механизма детерминированного воспроизведения

Эмулятор QEMU имеет структуру, показанную на рис. 1.

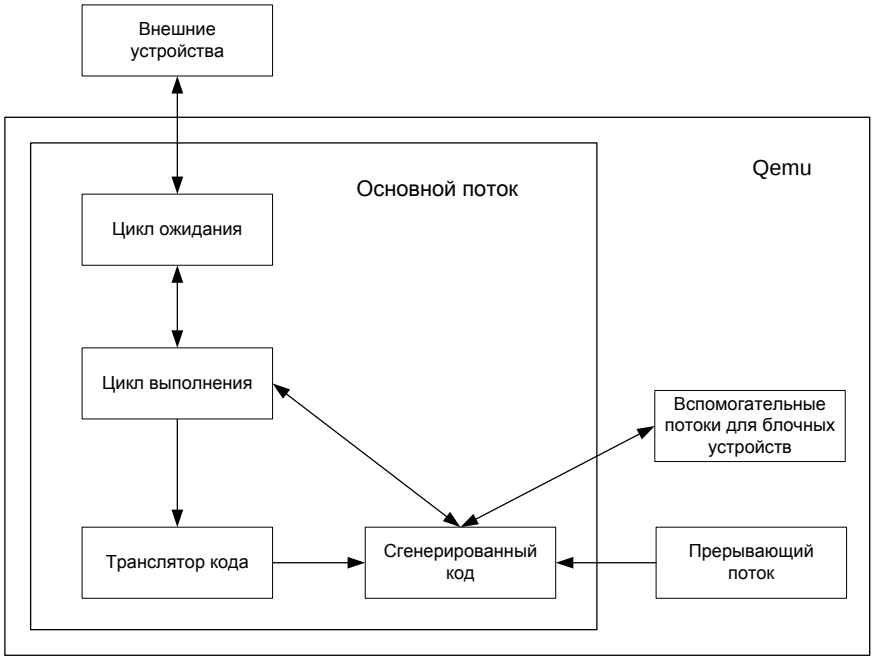

Рис. 1 – Структура эмулятора QEMU.

Цикл выполнения в QEMU производит трансляцию кода симулируемой машины и исполнение сгенерированного кода, обрабатывая небольшую его часть в каждой итерации. Параллельно с этим работает прерывающий поток, останавливающий выполнение кода с определенной периодичностью. При этом цикл выполнения прерывается и происходит возврат в цикл ожидания.

В цикле ожидания эмулятор сначала проверяет состояние внешних устройств и интерфейсов управления эмулятором, а затем возвращает управление в цикл выполнения. Так как прерывающий поток не синхронизирован с основным, детерминированное воспроизведение моментов возникновения прерываний таймера невозможно без переработки этого механизма.

Кроме того, QEMU взаимодействует с внешними устройствами, включая мышь, клавиатуру, сетевую карту, аппаратные таймеры. Через эти внешние устройства в QEMU могут приходить сообщения, которые не являются детерминированными и, следовательно, их обработчики также должны быть доработаны.

Реализация взаимодействия с блочными устройствами, такими как жесткий диск или привод оптических дисков, отличается от взаимодействия с другими внешними устройствами. Блочные устройства, как правило, работают с образами дисков, которые находятся в файлах главной машины. Операции чтения и записи данных из этих файлов выполняются в отдельных потоках для ускорения работы виртуальной машины, что обуславливает необходимость введения дополнительной синхронизации между ними и основным потоком для работы детерминированного воспроизведения.

3.1. Запись журнала событий

Для того чтобы воспроизводить процесс выполнения программ в виртуальной машине, была реализована запись всех недетерминированных событий в специальный журнал.

События, записываемые в журнал, делятся на синхронные и асинхронные. Синхронные события вызываются действиями, выполняемыми основным потоком (выполнение инструкции, итерации цикла ожидания, чтение часов). Асинхронные события приходят в эмулятор извне в произвольный момент времени (прерывание таймерного потока, нажатие клавиши на клавиатуре, движение мыши, приход сетевого пакета).

Выполнение очередной инструкции

Выполнение инструкции является внутренним детерминированным событием. Однако, для того, чтобы корректно воспроизводить моменты возникновения недетерминированных событий (внутренний таймерный поток, события от внешних устройств), необходимо учитывать количество выполненных инструкций. Поэтому был доработан код, отвечающий за трансляцию кода целевой машины для того, чтобы обновлять счетчик выполненных виртуальным процессором инструкций. Данная доработка является платформо-зависимой и выполнялась для всех поддерживаемых аппаратных платформ.

События от часов реального времени

В процессе своей работы эмулятор осуществляет считывание показаний часов реального времени как для своей работы, так и для передачи в виртуальную машину (Рис 2).

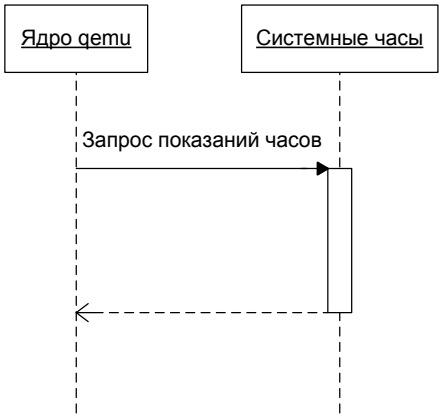

Рис. 2 – Получение показаний часов в исходной версии QEMU.

Чтобы показания часов реального времени, передаваемые внутрь симулируемой системы, не изменились при воспроизведении поведения системы, в модуль, осуществляющий работу с аппаратными часами, были внесены изменения, позволяющие записывать считанные показания часов в журнал (Рис. 3).

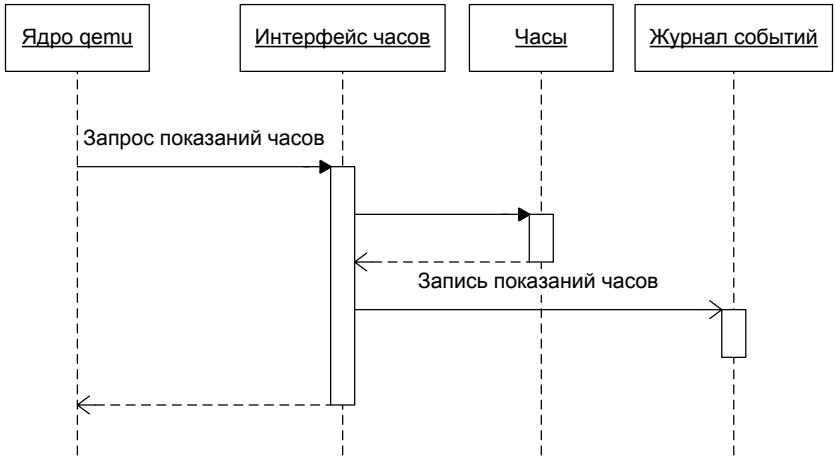

Рис. 3 – Запись показаний часов в журнал.

При воспроизведении журнала событий вместо считывания показаний аппаратных часов, выполняется чтение их из журнала (Рис. 4). Так как воспроизведение является детерминированным, операция чтения происходит в тот же момент (относительно позиции в журнале), что и при записи.

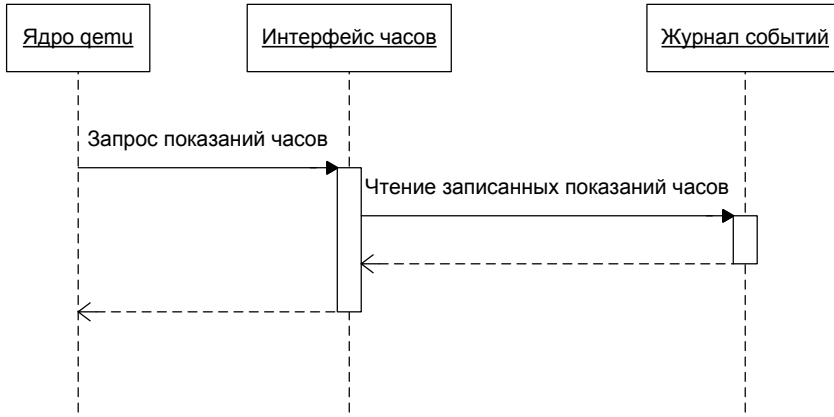

Рис. 4 – Чтение показаний часов из журнала.

Однако, в случае выхода эмулятора в цикл ожидания из цикла выполнения (например, при отладке), могут выполняться операции чтения времени, которых не было в исходном потоке событий. Для того чтобы они завершались успешно, при воспроизведении журнала выполняется кэширование показаний часов. Таким образом, в цикле ожидания будут считываться одни и те же показания, пока выполнение симулируемого кода не продолжится.

Прерывание выполнения симулируемого кода с помощью таймера

Для решения проблемы с синхронизацией потоков внутри QEMU, было решено зафиксировать точки в коде основного потока, в которых он может взаимодействовать с прерывающим потоком. Таким образом, в процессе воспроизведения журнала событий появится возможность точно восстанавливать моменты этих взаимодействий.

Несколько точек возможных прерываний были размещены в циклах ожидания и выполнения, а также был модифицирован код транслятора для добавления таких точек перед выполнением каждой инструкции. Таким образом, становится возможным детерминированное воспроизведение поведения системы даже при пошаговом выполнении кода.

События от периферии

Запись и воспроизведение событий от мыши, клавиатуры и сетевой карты отличается от предыдущих видов событий тем, что они инициируются извне. Поэтому запись информации о них в журнал производится непосредственно в момент возникновения соответствующего события.

Так как основные действия в QEMU, а также запись в журнал, выполняются в одном потоке, сообщения от периферийных устройств будут обрабатываться строго между какими-либо другими событиями, не пересекаясь с ними по времени. Поэтому при воспроизведении журнала (при поиске очередного синхронного события) может встретиться такое асинхронное событие (например, нажатие клавиши), которое может быть тут же обработано.

Работа с физической реализацией блочных устройств выполняется отдельными потоками внутри эмулятора, что позволяет возникнуть состоянию гонок. Например, при одновременной работе с оперативной памятью DMA-контроллера, выполняющего чтение с жесткого диска, и центрального процессора, читающего те же ячейки памяти. Поэтому эмулятор был модифицирован таким образом, чтобы при работе этих вспомогательных потоков выполнение основного потока приостанавливалось.

Экспериментальная оценка характеристик записи журнала

Скорость роста журнала определяет, насколько длительные периоды работы могут быть зафиксированы для последующего воспроизведения. На рис. 5 приводится график скорости роста журнала, т.е. количество байт, записываемых в журнал за 1 секунду. В качестве гостевой системы выступала ОС Windows XP, замер делался на модифицированной версии QEMU 0.13. Никаких внешних воздействий на систему не оказывалось: отсутствуют входящие сетевые пакеты, пользователь не работает с устройствами ввода. Таким образом, график отражает только те данные, которые порождаются работой эмулируемых периферийных устройств. Именно эта составляющая отражает качество механизма воспроизведения – вне зависимости от реализации пользовательский ввод и сеть будут одинаково увеличивать содержимое журнала. После того, как ОС завершила этап загрузки, объем записываемых в журнал данных стабилизировался и, начиная с третьей минуты, составил примерно 32 КБ/с.

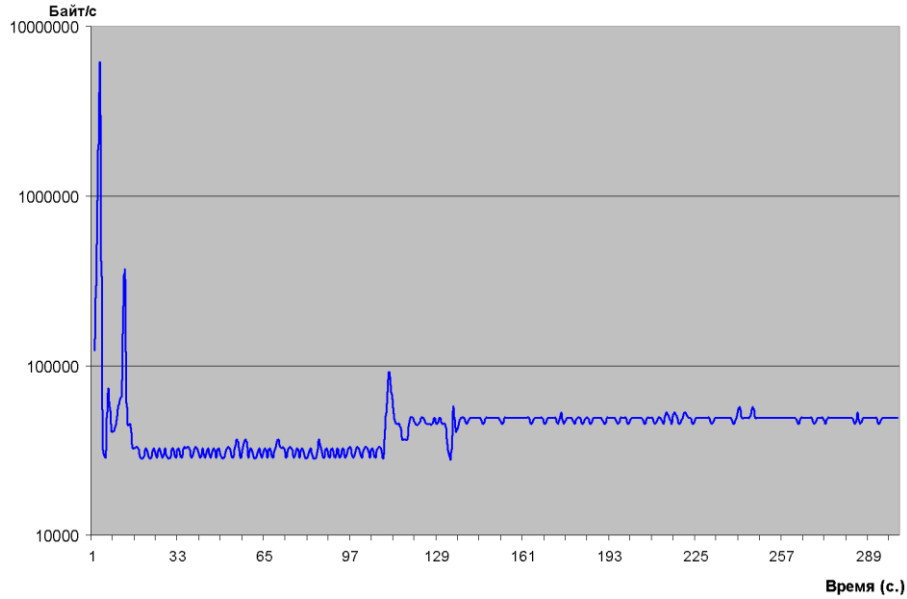

Рис. 5 – Скорость роста журнала.

У системы ExecRecorder (основанной на Bochs) этот показатель составляет примерно 1.5 МБ/с. Сопоставление с системой Retrace затруднительно, т.к. авторами заявлялась характеристика в виде числа записываемых байт на 1000 выполненных команд, но грубый пересчет, исходя из оценки скорости работы виртуального процессора в 500 МГц, дает примерно 2 МБ/с, что также

ощутимо больше, нежели полученные в нашей работе результаты. Без учета входящего сетевого трафика такой характер роста журнала (32 КБ/с) позволяет использовать обычные жесткие диски для записи часов и даже суток непрерывной работы виртуальной машины.

В другой характеристике, замедлении при снятии журнала, имеет место некоторый проигрыш: двукратное замедление (+100%) в сравнении +5% у системы Retrace. Тем не менее, достигнутый уровень замедления уже не позволяет выявить факт снятия журнала неконтролируемой частью исследуемой программы, работающей на другом компьютере. Замедление такого порядка может быть обусловлено различными другими причинами, например: плохим сетевым каналом, недостаточной производительностью или высокой загруженностью хостовой системы.

3.2. Детерминированное воспроизведение журнала событий

Технология детерминированного воспроизведения журнала событий заключается в выполнении кода эмулятора и считывании необходимых для воспроизведения программы данных из журнала:

- Считывание показаний часов из журнала, когда их запрашивает какой-либо код. Если соответствующих показаний нет – возвращается кэшированное значение.
- Передача сообщений от клавиатуры, мыши, звуковой и сетевой карт в тот момент, когда они считываются из журнала в соответствующие обработчики.
- Обновление внутреннего счетчика команд, когда происходит выполнение очередной инструкции.
- Выполнение заданий на чтение/запись данных жесткого диска, когда соответствующие события поступают из журнала.

Чтобы воспроизведение журнала было детерминированным, оно должно начинаться с того же самого состояния симулируемой системы, что и запись. Состояние системы включает в себя состояния всех симулируемых устройств, включая образы используемых дисков. Остальные устройства инициализируются при старте эмулятора и, поэтому, их состояния не отличаются при записи и воспроизведении.

3.3. Контроль корректности воспроизведения журнала

Для того чтобы контролировать, является ли процесс воспроизведения детерминированным, при считывании событий из файла журнала проверяется следующее ограничение: в момент, когда ожидается наступление события «выполнение инструкции» не должно возникать ни события от часов, ни события «цикл ожидания».

Если это ограничение нарушается, это означает, что ход эмуляции был нарушен. В этом случае выдается сообщение об ошибке и процесс воспроизведения останавливается. Это позволяет контролировать отсутствие

ошибок в механизме воспроизведения и выполнять отладку при их возникновении.

4. Сбор трасс для последующего автоматизированного анализа

Одной из целей доработок, вносимых в QEMU, является динамический анализ программ с помощью снятой с них трассы выполнения. Трасса представляет собой последовательность из выполняемых инструкций, а также состояний процессора (включающих значения регистров), соответствующих этим инструкциям. Каждая группа «выполняемая инструкция – значений регистров» называется шагом трассы.

Реализация механизма снятия трасс выполнения состоит из нескольких частей:

1. Платформенно-независимая часть, сохраняющая сформированные шаги трассы в файл. При этом используется сжатие с помощью библиотеки LZO, так как сохраняемый объем данных достаточно велик.

2. Изменения в платформенно-независимом модуле пользовательского монитора, включающие поддержку команд начала и завершения процесса снятия трассы.

3. Изменения в платформенно-зависимом модуле трансляции выполняющегося в виртуальной машине бинарного кода. Транслятор вставляет перед каждой выполняющейся командой дополнительный код, заполняющий шаг трассы и передающий его модулю сохранения для записи в файл.

В настоящее возможность снятия трассы реализована для платформ x86 и ARM. Трудозатраты на реализацию этого механизма для других платформ достаточно малы, т.к. требуется внести изменения только в механизм трансляции, добавив код, который заполняет шаг трассы и вызывает механизм его сохранения.

Снятие трассы предполагает запись выполненных инструкций и значений регистров, а учесть данные, которые пишутся в память или считываются из нее без участия процессора, не представляется возможным. В QEMU был реализован механизм сбора логов обращений к жесткому диску. Лог содержит данные, которые были прочитаны с жесткого диска или записаны на него, а также информацию о номере сектора, количестве секторов, имени устройства и, если QEMU работает в режиме детерминированного воспроизведения, то номер текущего шага воспроизведения (в качестве номера шага выступает число выполненных процессором инструкций, начиная с начала процесса воспроизведения). Следует учитывать, что работа с блочными устройствами ведется в двух режимах – PIO (Programmed Input/Output) и DMA (Direct

Memory Access); в первом случае при считывании или записи данных участвует эмулируемый центральный процессор, а во втором нет. QEMU работает как с PIO, так и с DMA, для DMA транзакций в лог попадает так же адрес памяти, куда контроллер запишет или прочитает блок данных.

5. Расширение возможностей отладки

Детерминированное воспроизведение можно использовать не только для непрерывного многократного выполнения сценария, но и для перехода в произвольные места записанного сценария для просмотра состояния виртуальной машины – ее регистров, памяти и т.п. Также возможность быстрого перехода в нужное состояние дает дополнительные возможности при отладке программы, выполняющейся в виртуальной машине.

5.1. Сохранение состояний виртуальной машины

QEMU изначально включал в себя механизм сохранения состояний виртуальной машины. Сохраненные состояния можно в последующем загружать и, таким образом, неоднократно начинать выполнение с сохраненной позиции. Чтобы иметь возможность неоднократно переходить в нужное состояние, достаточно сохранить стартовое состояние виртуальной машины и, в дальнейшем, загружая его, воспроизводить журнал событий до нужной точки.

Также в механизм записи журнала была внесена возможность автоматического сохранения состояний виртуальной машины с заданной периодичностью. Это позволяет пользователю быстрее переходить к нужной точке журнала, загружая ближайшее состояние.

Для того чтобы пользователь мог перемещаться между точками журнала в режиме его воспроизведения, в подсистему пользовательского монитора была внесена поддержка дополнительной команды, выполняющей данную операцию.

5.2. Детерминированная и обратная отладка

Детерминированная отладка – это способ поиска ошибок в недетерминированных приложениях, при котором недетерминированность устраняется с помощью записи сценария работы системы (или программы) в журнал. Разработанный метод воспроизведения работы программ позволяет выполнять детерминированную отладку недетерминированных приложений следующим образом:

1. Тестировщик записывает сценарий, при выполнении которого проявляется дефект в тестируемой программе, в журнал, а затем передает этот журнал вместе с образами дисков системы разработчику. Здесь сценарий выступает не только в роли исходных

данных для отладки, но и в роли описания способа воспроизведения дефекта.

2. Разработчик может неоднократно проигрывать полученный сценарий в эмуляторе, анализируя причины появления дефекта. Разработчику не нужно настраивать сложное окружение системы так же, как это делал тестировщик, так как все особенности взаимодействия с этим окружением уже записаны в журнал.

Таким образом, отладка недетерминированных приложений с применением разработанного метода становится детерминированной, что позволяет сократить время, затрачиваемое разработчиками на локализацию дефектов в программе, а тестировщиками – на описание процесса их воспроизведения.

Обратная отладка – возможность перехода отлаживаемой программы к ранее пройденным состояниям. В нашем случае вся виртуальная машина (т.е. гостевая операционная система и выполняющиеся в ней программы) рассматривается как отлаживаемая программа. Обратная отладка позволяет пользователю переходить от момента в программе, когда обнаруживается исключительная ситуация (например, обращение к памяти через испорченный указатель) к моменту времени, когда формируются данные, вызвавшие эту ошибку (запись некорректного значения указателя).

5.3. Известные инструменты обратной отладки

Встроенные возможности по обратной отладке в отладчике gdb (для x86), а также в Trace32 (для ARM) реализованы с помощью записи состояний процессора и значений ячеек памяти [16, 17]. Таким образом, можно просматривать ранее пройденные состояния и наблюдать значения переменных. Но такой подход имеет и ограничения. Записывается только определенное число состояний процессора (откат возможен на ограниченное число шагов). Если не использовать специальные аппаратные средства для записи данных процессора, возникает значительное замедление, способное повлиять на ход работы отлаживаемой программы. Кроме того, т.к. записываются только состояния процессора и ячеек памяти, при возврате назад невозможно наблюдать полное состояние системы (например, вывод программы на экран).

Подобный подход также используется в реализации отладчиков для различных программных платформ, например, Java, C# и т.п. При этом в отлаживаемую программу добавляется специальный код, записывающий интересующие пользователя данные, что может повлиять на поведение отлаживаемой программы [18, 19].

Кроме методов, записывающих состояния процессора, существуют также методы обратной отладки, использующие детерминированное воспроизведение программ [20, 21]. Детерминированное воспроизведение в самом простом случае предполагает сохранение начального состояния программы и результатов системных вызовов в процессе ее выполнения. Так

появляется возможность перехода к заданному шагу программы с помощью загрузки ее начального состояния и выполнения с подменой системных вызовов. В случае большого количества системных вызовов, использования разделяемой памяти или многопоточных приложений, использования слишком многих системных библиотек, применение данного подхода становится слишком трудоемким.

Если же детерминированное воспроизведение необходимо для системы целиком, отладка требует выполнения дополнительного прохода с записью журнала недетерминированных событий. В журнал должны попадать как начальное состояние системы, так и все внешние события (пользовательский ввод или работа с сетью), позволяющие восстановить заданное состояние системы в дальнейшем. После записи сценария выполнения программы в журнал событий, становится возможным воспроизводить его и уже выполнять непосредственно отладку.

Преимущества подходов, основанных на детерминированном воспроизведении, в том, что в отлаживаемую программу не вносится никаких изменений, а также становится возможной многократная отладка одного и того же сценария выполнения программы, в котором может проявляться ошибка. Кроме того, процесс отладки становится детерминированным, что особенно важно для ошибок, проявляющихся от случая к случаю.

Однако большинство из современных решений были разработаны для процессоров с архитектурой x86 и не поддерживают широко распространенную в настоящее время архитектуру ARM. Описываемый в данной статье метод лишен этого недостатка.

5.4. Реализация обратной отладки

Эмулятор QEMU включает в себя механизм, позволяющий с помощью отладчика gdb подключаться к виртуальной машине и управлять процессом выполнения ее кода: останавливать и возобновлять процесс выполнения, задавать точки останова и контрольные точки данных, считывать и записывать значения регистров и памяти, а также выполнять пошаговую отладку.

Начиная с 7-й версии, отладчик gdb поддерживает команды обратной отладки, такие как шаг назад и поиск первой сработавшей контрольной точки в обратном направлении. Для того чтобы интерпретировать эти команды, был доработан модуль взаимодействия QEMU с gdb. Кроме этого, были внесены доработки в основной модуль детерминированного воспроизведения для реализации сценариев обратной отладки.

Чтобы выполнить действие «шаг назад», QEMU загружает ближайшее из сохраненных состояний виртуальной машины, предшествующих этому шагу, и продолжает выполнение, пока нужный шаг не будет достигнут.

Команда «выполнение в обратном направлении до первой точки останова» (reverse-continue) выполняется в несколько этапов (Рис. 6). Сначала

загружается состояние виртуальной машины, предшествующее текущему положению, и выполняется поиск точки останова, которая срабатывает последней перед достижением текущего шага. Если такой точки останова не найдено, данный этап повторяется для предыдущего состояния. После нахождения сработавшей точки останова происходит переход к ней с помощью загрузки ближайшего состояния и воспроизведения журнала событий до этой точки.

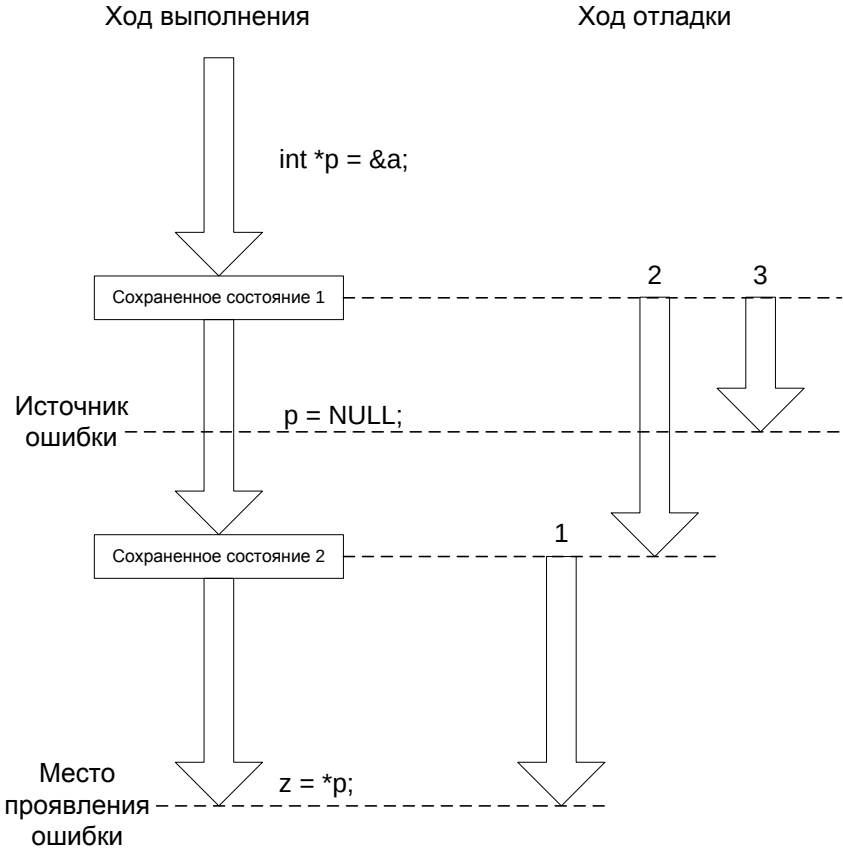

Рис. 6 – Ход выполнения обратной отладки при поиске ближайшей точки останова

Таким образом, использование сохраненных в процессе записи сценария состояний виртуальной машины позволяет находить нужную точку останова за время в лучшем случае равное удвоенному периоду времени между записанными состояниями. В худшем случае это время будет равняться удвоенному времени выполнения сценария целиком. На практике реально

сохранять состояние виртуальной машины каждые 4-5 секунд, что означает, что команда reverse-continue чаще всего будет выполняться за 8-10 секунд.

6. Заключение

В настоящий момент механизм детерминированного воспроизведения работает в модифицированной версии QEMU 1.5 и поддерживает платформы x86 и ARM. Благодаря динамической трансляции эмулятор поддерживает множество других аппаратных платформ [7], для которых также может быть реализовано детерминированное воспроизведение.

Уже поддерживаемый механизмом детерминированного воспроизведения список устройств включает все виртуальные устройства (то есть не взаимодействующие с чем-либо, кроме других виртуальных устройств), а также такие устройства, как аудиоадаптер, сетевая карта, последовательный порт, мышь, клавиатура, внешние USB-устройства.

Открытость исходного кода эмулятора позволяет реализовывать поддержку дополнительных периферийных устройств, не представленных в исходной версии эмулятора. В том случае, если периферийное устройство является виртуальным, для поддержки детерминированного воспроизведения не требуется выполнять никаких дополнительных действий.

Кроме того, в эмулятор можно добавить специфическое устройство, общающееся с внешним миром (например, сетевую карту). В этом случае требуется расширение механизма детерминированного воспроизведения для того, чтобы получаемые извне данные записывались в журнал. Затем, при воспроизведении этого журнала, они должны считываться и подаваться на вход устройства, заменяя реальный обмен данными с внешним миром.

Список литературы

[1] В.А. Падарян, А.И. Гетьман, М.А. Соловьев. Программная среда для динамического анализа бинарного кода. // Труды Института Системного Программирования, том: 16, 2009. стр. 51-72.

[2] Full System Simulation. http://www.windriver.com/products/simics/ дата обращения 2 декабря 2013

[3] SimNow™ Simulator http://developer.amd.com/tools-and-sdks/cpu-development/simnow-simulator/ дата обращения 2 декабря 2013

[4] Cisco 7200 Simulator http://www.ipflow.utc.fr/blog/ дата обращения 2 декабря 2013

[5] GNS3 / dynamips https://github.com/GNS3/dynamips дата обращения 2 декабря 2013

[6] ARM Software development tools http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0058d/Chdcdbib.html дата обращения 2 декабря 2013

[7] QEMU – Open Source Processor Emulator. http://wiki.qemu.org/Main_Page дата обращения 2 декабря 2013

[8] Dunlap, George W. and King, Samuel T. and Cinar, Sukru and Basrai, Murtaza A. and Chen, Peter M. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. // ACM SIGOPS Operating Systems Review - OSDI '02: Proceedings of the 5th

symposium on Operating systems design and implementation, vol. 36, 2002, pp. 211-224.
[9] Haikun Liu, Hai Jin, Xiaofei Liao, Zhengqiu Pan. XenLR: Xen-based Logging for Deterministic Replay. // In proc. of Japan-China Joint Workshop on Frontier of Computer Science and Technology, 2008. pp. 149-154.
[10] Amit Vasudevan, Ning Qu, Adrian Perrig. XTRec: Secure Real-time Execution Trace Recording on Commodity Platforms. // In Proceedings of the 44th Hawaii International Conference on System Sciences (HICSS'11), 2011. pp. 1-10.
[11] Daniela A. S. de Oliveira, Jedidiah R. Crandall, Gary Wassermann, S. Felix Wu, Zhendong Su, and Frederic T.Chong. ExecRecorder: VM-based full-system replay for attack analysis and system recovery. // Proc. of the 1st workshop on Architectural and system support for improving software dependability (ASID '06), 2006. pp. 66-71
[12] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. // In Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation, MoBS, San Diego, CA, June, volume 3, pages 4--2, 2007
[13] Jim Chow, Tal Garfinkel, Peter M. Chen. Decoupling dynamic program analysis from execution in virtual environments. // Proceedings of the 2008 Annual USENIX Technical Conference, June 2008. pp. 1-14
[14] Oracle VM VirtualBox https://www.virtualbox.org/ дата обращения 2 декабря 2013
[15] Chia-Wei Hsu, Shiuhpyng Shieh. FREE: A Fine-grain Replaying Executions by Using Emulation. // The 20th Cryptology and Information Security Conference (CISC 2010), Taiwan, 2010.
[16] GDB and Reverse Debugging. http://sourceware.org/gdb/news/reversible.html, дата обращения 2 декабря 2013
[17] Microprocessor Development Tools. http://www.lauterbach.com/frames.html?home.html, дата обращения 2 декабря 2013
[18] Omniscient Debugging. http://www.lambdacs.com/debugger/ODBDescription.html, дата обращения 2 декабря 2013
[19] How Does VS2010 Historical Debugging Work? http://www.wintellect.com/CS/blogs/jrobbins/archive/2009/06/16/how-does-vs2010-historical-debugging-work.aspx, дата обращения 2 декабря 2013
[20] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging Operating Systems with Time-Traveling Virtual Machines. ATEC '05 Proceedings of the annual conference on USENIX Annual Technical Conference, Berkeley, CA, USA, 2005, pp. 1-15
[21] Toshihiko Koju, Shingo Takada, and Norihisa Doi. An efficient and generic reversible debugger using the virtual machine based approach. VEE '05 Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments, New York, NY, USA, 2005, pp. 79-88

Application of software emulators for the binary code analysis*

Dovgalyuk P.M., Makarov V.A., Padaryan V.A., M.S. Romaneev, Fursova N.I.
{Pavel.Dovgaluk, vladimir_makarov, vartan, melon, Natalia.Fursova}@ispras.ru
ISP RAS, Moscow, Russia

Annotation. The paper describes the experience of using software emulators as a means of dynamic analysis of binary code tools. Emulator is considered as tracer of machine commands layer and as interactive debugging tool. It describes a mechanism of deterministic replay implemented in emulator QEMU.

Deterministic replay is a process of recovery of program execution using a pre-recorded input. To replay process of program execution in virtual machine recording of all nondeterministic events to journal was implemented. Such events are indications of real time clock, messages from keyboard, mouse, sound and network cards. Currently the deterministic replay mechanism works in a modified version of QEMU 1.5 and supports x86 and ARM platforms.

To solve the problems of binary code analysis tracing is used, but it slows down the system, so it is easier to do with deterministic replay. Trace is a sequence of executed instructions and processor state (including register values). Each group "executed instruction - values of registers" is called a trace step. Currently tracing is implemented for x86 and ARM platforms. Trace does not contain information about the read and written memory, so logging of hard disk drive accesses was implemented.

Deterministic debugging is a way to find errors in nondeterministic applications, in which nondeterminism is eliminated by writing the scenario of system work. By means of deterministic replay nondeterministic debugging becomes deterministic strongly reducing the time spent on the localization of defects in the program and their description.

Reverse debugging is the possibility of studying the past states of the program. In our case the entire virtual machine is considered the program being debugged.

Emulator QEMU includes mechanism to let GNU debugger connect to virtual machine and manage the process of execution. GNU debugger supports reverse debugging commands, such as reverse-step and reverse-continue.

* The paper is supported by RFBR grant 12-01-31417

Keywords: emulator, dynamic analysis, deterministic replay, reverse debugging

References

[1]. Padaryan V.A., Get'man A. I., Solov'ev M. A. Programmnaya sreda dlya dinamicheskogo analiza binarnogo koda [Software environment for dynamic analysis of binary code]. Trudy ISP RAN [The Proceedings of ISP RAS], 2009, vol. 16, pp. 51-72 (in Russian).
[2]. Full System Simulation. http://www.windriver.com/products/simics/
[3]. SimNow™ Simulator. http://developer.amd.com/tools-and-sdks/cpudevelopment/simnow-simulator/
[4]. Cisco 7200 Simulator. http://www.ipflow.utc.fr/blog/
[5]. GNS3 / dynamips. https://github.com/GNS3/dynamips
[6]. ARM Software development tools. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0058d/Chdcdbib.html
[7]. QEMU – Open Source Processor Emulator. http://wiki.qemu.org/Main_Page
[8]. Dunlap, George W. and King, Samuel T. and Cinar, Sukru and Basrai, Murtaza A. and Chen, Peter M. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. ACM SIGOPS Operating Systems Review - OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation, vol. 36, 2002, pp. 211-224.
[9]. Haikun Liu, Hai Jin, Xiaofei Liao, Zhengqiu Pan. XenLR: Xen-based Logging for Deterministic Replay. In proc. of Japan-China Joint Workshop on Frontier of Computer Science and Technology, 2008. pp. 149-154.
[10]. Amit Vasudevan, Ning Qu, Adrian Perrig. XTRec: Secure Real-time Execution Trace Recording on Commodity Platforms. In Proceedings of the 44th Hawaii International Conference on System Sciences (HICSS'11), 2011. pp. 1-10.
[11]. Daniela A. S. de Oliveira, Jedidiah R. Crandall, Gary Wassermann, S. Felix Wu, Zhendong Su, and Frederic T.Chong. ExecRecorder: VM-based full-system replay for attack analysis and system recovery. Proc. of the 1st workshop on Architectural and system support for improving software dependability (ASID '06), 2006. pp. 66-71
[12]. M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, and B. Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. In Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation, MoBS, San Diego, CA, June, volume 3, pages 4--2, 2007
[13]. Jim Chow, Tal Garfinkel, Peter M. Chen. Decoupling dynamic program analysis from execution in virtual environments. Proceedings ofthe 2008 Annual USENIX Technical Conference, June 2008. pp. 1-14
[14]. Oracle VM VirtualBox . https://www.virtualbox.org/
[15]. Chia-Wei Hsu, Shiuhpyng Shieh. FREE: A Fine-grain Replaying Executions by Using Emulation. The 20th Cryptology and Information Security Conference (CISC 2010), Taiwan, 2010.
[16]. GDB and Reverse Debugging. http://sourceware.org/gdb/news/reversible.html
[17]. Microprocessor Development Tools. http://www.lauterbach.com/frames.html?home.html
[18]. Omniscient Debugging. http://www.lambdacs.com/debugger/ODBDescription.html

[19]. How Does VS2010 Historical Debugging Work? http://www.wintellect.com/CS/blogs/jrobbins/archive/2009/06/16/how-does-vs2010-historical-debugging-work.aspx
[20]. Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging Operating Systems with Time-Traveling Virtual Machines. ATEC '05 Proceedings of the annual conference on USENIX Annual Technical Conference, Berkeley, CA, USA, 2005, pp. 1-15
[21]. Toshihiko Koju, Shingo Takada, and Norihisa Doi. An efficient and generic reversible debugger using the virtual machine based approach. VEE '05 Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments, New York, NY, USA, 2005, pp. 79-88