Санкт-Петербургский Государственный Университет
Математико-Механический факультет
кафедра системного программирования

Применение эмуляции для обратной инженерии

Курсовая работа студента 445 группы Вадима Еварда
Научный руководитель Максим Викторович Баклановский

2013 год

Введение

Обратной инженерией какой-либо системы называют исследование принципов её работы по её структуре, функциональным возможностям и наблюдаемому поведению. К традиционным задачам обратной инженерии относят:

- Реализацию взаимодействия системы с внешними объектами, не предусмотренного при создании системы;
- Создание, актуализация и исправление документации на существующую систему;
- Исправление тривиальных ошибок;
- Анализ безопасности системы;
- Промышленный шпионаж, в том числе восстановление алгоритмов, составляющих основную ценность системы;
- Преодоление методов защиты интеллектуальной собственности, в том числе снятие лицензионных ограничений и создание нелицензионных копий.

Выделим два основных подхода к анализу программных систем:

- Статический анализ представляет собой исследование системы без её фактического запуска, то есть исследование её хранимого образа, например, разбор структуры исполняемого файла и декомпиляция секций кода. К достоинствам данного подхода относят автоматизируемость типичных задач, а также отсутствие необходимости в работоспособности системы и её окружения. Недостатком статического анализа является то, что эффективно его можно реализовать только для достаточно узкого класса программ, например, обычно не поддерживается декомпиляция самомодифицирующегося кода.
- Динамический анализ — исследование работы системы при запуске в контролируемом окружении. При динамическом анализе могут быть выявлены сложные и динамические связи (например, выбор системой той или иной стратегии в зависимости от внешних обстоятельств), а также появляется возможность слежения за состоянием системы и его изменение. Однако изменения, вносимые в среду исполнения для достижения нужного уровня контроля, могут существенно исказить картину работы системы (например, для кода, чувствительного к времени своего исполнения).

Динамический анализ может проводиться как в «родной» среде, для которой проектировалась и создавалась исследуемая система, так и с применением

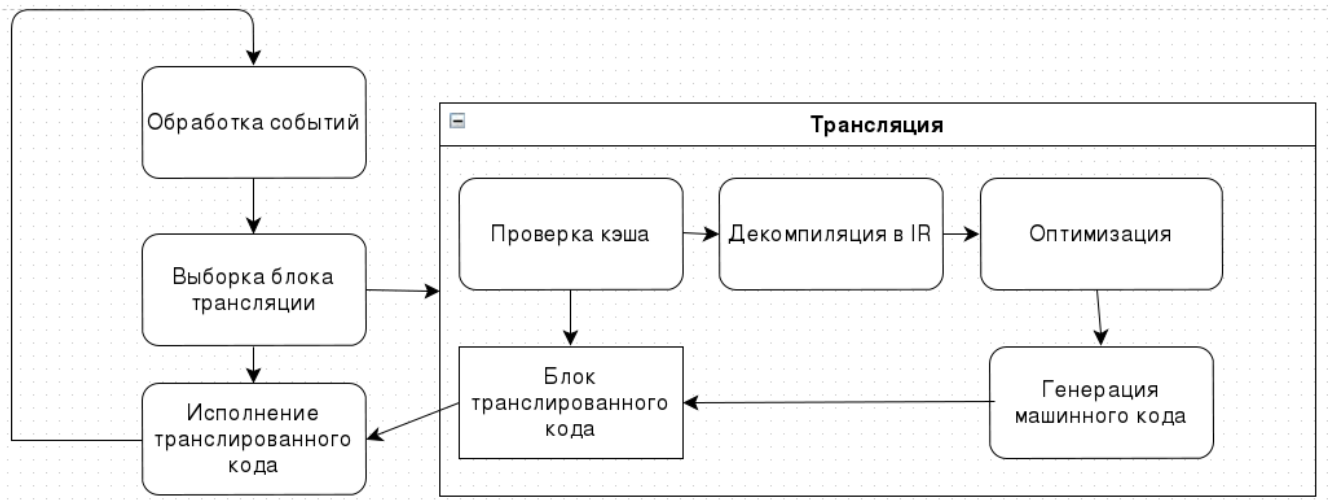
эмуляции. Эмуляцией называется воспроизведение программными или аппаратными средствами либо их комбинацией работы других программ и устройств.

В настоящей работе исследовались возможности динамического анализа, предоставляемые кроссплатформенным эмулятором QEMU, поддерживающим эмуляцию распространённых компьютерных архитектур (PC с процессорами x86/x86_64 и типичной периферией, ARM- и MIPS-платы и другие устройства).

Архитектура эмулятора QEMU

QEMU поддерживает различные варианты исполнения «гостевых» приложений:

- «Пользовательский» режим. Эмулируется только процессор, обращение к API операционной системы транслируется настоящей системе.
- Задействование средств аппаратной виртуализации (KVM, гипервизор XEN). Применяется в ситуации, когда архитектура эмулируемой системы совпадает с архитектурой хостовой. Обеспечивает наибольшую производительность.
- Программная эмуляция системы (системный режим). В этом режиме эмулируется исполнение инструкций процессора и обращение к периферии. Он обеспечивает наиболее полный контроль над исполнением гостевой программы, и поэтому был использован для решения задач динамического анализа.



Большая часть логики работы QEMU включена в «основной цикл». Он состоит из четырёх этапов:

1. Сбор и обработка новых событий: нажатия клавиш, пришедший сетевой пакет и т. п. Информация об этих событиях сохраняется в состоянии соответсвующих эмулируемых устройствах гостевой системы.

2. Выбирается участок машинных инструкций гостевой системы, который может быть исполнен «за раз»: в нём отсутствуют команды, модифицирующие поток управления (т.о., это линейный участок) и существенным образом меняющие режим работы эмулируемого процессора (например, переход в другой режим, настройка сегментных регистров). В этот момент можно задействовать встроенную в QEMU возможность отладочной печати дизассемблерного листинга выбранного блока трансляции.
3. Осуществляется трансляция произвольного машинного кода гостевой системы в безопасный машинный код хостовой.
    1. Машинный код гостевой системы декомпилируется в более простые инструкции RISC-подобного машинного языка IR (Intermediate Representation). При этом обращения к памяти заменяются на обращение к так называемому SoftMMU – механизму, гарантирующему, что транслированный код может обращаться только к памяти, выделенной для хранения физической памяти гостевой системы. Состояние гостевого процессора (содержимое его регистров, режимы работы, флаги) взаимно однозначно отображается на регистры IR-процессора и дополнительные области памяти.
    2. Осуществляется оптимизация сгенерированного IR-кода: удаляется «мёртвый» код и избыточные инструкции.
    3. IR-код транслируется в машинный код архитектуры хоста: состояние IR-процессора однозначно отображается на регистры хостового процессора и дополнительные области памяти.
    4. Генерируется эпилог, возвращающий управление из сгенерированного кода обратно в основной цикл QEMU.
4. Управление передаётся на сгенерированный машинный код. В этот момент доступна отладочная печать дизассемблерного листинга сгенерированного кода. По окончании исполнения сгенерированного блока управление вернётся в начало основного цикла QEMU.

В качестве механизма, предоставляющего возможности динамического анализа на базе QEMU, была использована печать отладочных трасс машинного кода гостевой системы.

Методика

Для генерации отлачдочных листингов QEMU необходимо запускать с ключом -d in_asm. Пример такого листинга, выводимый при старте эмулируемой

PC-системы: процессор находится в реальном режиме, управление передаётся на BIOS по адресу FFFF:FFF0.

```
----------------
IN:
0xffffffff0:  ljmp   $0xf000,$0xe05b

----------------
IN:
0x000fe05b:  cmpl   $0x0,%cs:-0x2f2c
0x000fe062:  jne    0xfc792

----------------
IN:
0x000fe066:  xor    %ax,%ax
0x000fe068:  mov    %ax,%ss

----------------
IN:
0x000fe06a:  mov    $0x7000,%esp

----------------
IN:
0x000fe070:  mov    $0xf4f83,%edx
0x000fe076:  jmp    0xfc641

----------------
IN:
0x000fc641:  mov    %eax,%ecx
```

При этом первые два блока трансляции будут преобразованы в следующий вид (запуск с ключом -d out_asm):

```
OUT: [size=42]
0x41577000:  mov    $0xf000,%ebp
0x41577005:  mov    %ebp,0x50(%r14)
0x41577009:  mov    $0xf0000,%ebp
0x4157700e:  mov    %ebp,0x54(%r14)
0x41577012:  mov    $0xe05b,%ebp
0x41577017:  mov    %ebp,0x20(%r14)
0x4157701b:  xor    %eax,%eax
0x4157701d:  mov    $0x7f6ac1e60576,%r10
0x41577027:  jmpq   *%r10

OUT: [size=156]
0x41577030:  mov    0x54(%r14),%ebp
0x41577034:  add    $0xd0c4,%ebp
0x4157703a:  mov    %ebp,%esi
0x4157703c:  mov    %ebp,%edi
0x4157703e:  shr    $0x7,%esi
0x41577041:  and    $0xfffff003,%edi
0x41577047:  and    $0x1fe0,%esi
0x4157704d:  lea    0x378(%r14,%rsi,1),%rsi
0x41577055:  cmp    (%rsi),%edi
0x41577057:  mov    %ebp,%edi
0x41577059:  jne    0x41577063
0x4157705b:  add    0x10(%rsi),%rdi
0x4157705f:  mov    (%rdi),%ebp
0x41577061:  jmp    0x41577074
```

```
0x41577063:  xor    %esi,%esi
0x41577065:  mov    $0x7f6ac11c2a60,%r10
0x4157706f:  callq  *%r10
0x41577072:  mov    %eax,%ebp
0x41577074:  mov    $0x10,%ebx
0x41577079:  mov    %ebx,0x30(%r14)
0x4157707d:  xor    %ebx,%ebx
0x4157707f:  mov    %ebx,0x28(%r14)
0x41577083:  mov    %ebp,0x2c(%r14)
0x41577087:  test   %ebp,%ebp
0x41577089:  jne    0x415770b4
0x4157708f:  jmpq   0x41577094
0x41577094:  mov    $0xe066,%ebp
0x41577099:  mov    %ebp,0x20(%r14)
0x4157709d:  mov    $0x7f6ab39e5080,%rax
0x415770a7:  mov    $0x7f6ac1e60576,%r10
0x415770b1:  jmpq   *%r10
0x415770b4:  mov    $0xc681,%ebp
0x415770b9:  mov    %ebp,0x20(%r14)
0x415770bd:  xor    %eax,%eax
0x415770bf:  mov    $0x7f6ac1e60576,%r10
0x415770c9:  jmpq   *%r10
```

Полученные трассы позволяют собирать различные статистические сведения об исполняющемся в эмуляторе гостевом коде.

Результаты

Для эталонной трассы, полученной запуском тестового образа Linux, собранного для x86, выявлены следующие закономерности:

- Средняя длина блока трансляции около 5.11 машинных инструкций, не изменяется от запуска к запуску;
- Средняя длина линейного участка (последовательности команд, из которых изменяет поток управления только последняя) составляет 5.14 инструкций;
- 40% исполлняемых инструкций являются инструкциями копирования данных (общая мнемоника **mov** для перемещения данных между регистрами и/или паятью), около 8% - операции с аппаратным стеком (**push**, **pop** и аналогичные инструции), 9% - операции сравнения **cmp** и **test**, столько же — различные варианты условных переходов, 2% - безусловный переход **jmp**, около 30% - совокупность логических и арифметических инструкций. Инструкции вызова подпрограммы **call** (4%) встречаются в 2 раза чаще инструкций выхода (**ret**, 2%);
- Линейный участок завершается примерно в 50% случаев инструкцией условного перехода **je**, 45% - инструкцией **call**.

При оценке этих результатов необходимо учесть, что они являются средним между количеством количеством инструкций, исполняемых виртуальным процессором (в этом случае количество инструкций **call** и **ret** должно примерно совпадать) и количеством инструкций, содержащихся в запускаемых исполняемых файлах гостевой ОС (транслированные блоки сохраняются в кэше, поэтому при повторном вызове подпрограммы она не будет транслирована заново; на эту картину накладывается то обстоятельство, что при переполнении кэша он полностью сбрасывается).

Выводы и перспективы

Эмулятор QEMU спроектирован для максимально производительной и переносимой эмуляции аппаратных систем. Эти требования привели к решениям, существенно снижающим предоставляемые возможности динамического анализа: промежуточное представление, его оптимизация, кэш транслированных блоков.

Несмотря на упомянутые ограничения, были собраны базовые статистики исполняемого кода: средние длины блоков трансляции и линейных участков, процентное соотношение исполняемых машинных инструкций.

Основной режим работы QEMU не предоставляет возможностей пошагового исполнения кода и сбора нетривиальных статистик. Возможным продолжением данной работы является построение инструмента для сбора таких статистик на основе связки QEMU и GDB или другого эмулятора (Bochs).

Литература

1. Документация проекта QEMU, http://wiki.qemu.org/.
2. Обзор архитектуры QEMU, http://blog.vmsplice.net/2011/03/qemu-internals-overall-architecture-and.html.
3. Pandora's Bochs: Automatic Unpacking of Malware, *http://archive.hack.lu/2009/PandorasBochs.pdf*.