Фаззинг — исследование программ без исходного кода

kAFL, Redqueen, Grimoire.

Какие интересные характеристики можно выделить у фаззера?

- Фаззинг исходного кода/бинарных файлов
- Фаззинг юзерспейса/ядра/…
- Эффективность — скорость работы
- Эффективность — мутаторы
- Эффективность — параллелизация
- …

Фаззинг бинарных файлов с покрытием

Как инструментировать?

Фаззинг бинарных файлов с покрытием

Статическая инструментация

- Разбираем программу на базовые блоки, добавляем инструментацию, собираем обратно
- Сложно реализовать
- Достаточно быстро?
- Пример: Syzygy

Фаззинг бинарных файлов с покрытием

Динамическая инструментация

- JIT перекомпиляция с инструментацией, либо добавление брейкпойнтов с коллбеками
- Сложно реализовать
- Медленнее статической инструментации
- Примеры: DynamoRIO, Intel PIN, QBDI

Фаззинг бинарных файлов с покрытием

Инструментация отладчиком

- Вариант динамической инструментации, вручную ставим нужные брейкпойнты, вручную делаем всё
- Самодельные фаззеры

Фаззинг бинарных файлов с покрытием
Эмуляция

- Динамическая инструментация на основе эмулятора «общего назначения»
- Можно фаззить приложения под любые архитектуры
- Работает, в лучшем случае, медленно
- Примеры: afl-qemu, afl-unicorn

Фаззинг бинарных файлов с покрытием
Аппаратная трассировка

- Снимаем трассы встроенным в процессор функционалом (Intel PT, ARM CoreSight)
- Сложность реализации: 0*
- Скорость работы: практически нативная
- Примеры: PTrix, winafl, kAFL

Работы RUB-SysSec

Ruhr University Bochum — Chair for Systems Security

- (2017) kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels
- (2019) Redqueen: Fuzzing with Input-to-State Correspondence
- (2019) Grimoire: Synthesizing Structure while Fuzzing

kAFL

kAFL

Фаззинг бинарных программ в виртуальной машине QEMU-KVM — можно фаззить как ring3, так и ring0, программы работают с нативной скоростью.

Для снятия покрытия используется Intel PT.

Мутаторы взяты из afl.

kAFL — архитектура

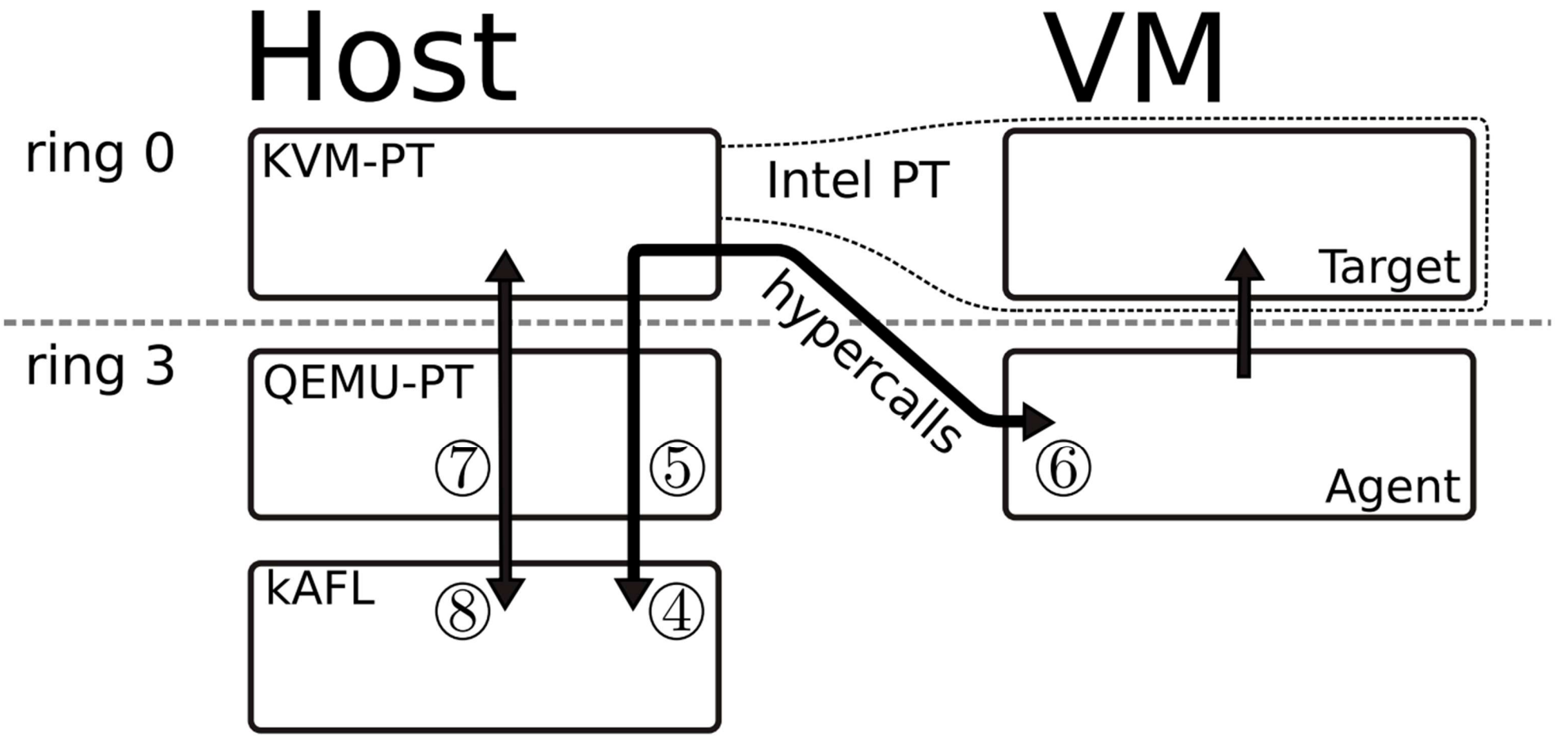
Фаззер в юзерспейсе хоста

ВМ, запущенная в “kAFL-QEMU”

“kAFL-QEMU” использует “kAFL-KVM”

Агент внутри ВМ, принимает тесты от фаззера и каким-то образом использует их для тестирования

Цель — драйвер, библиотека, что угодно

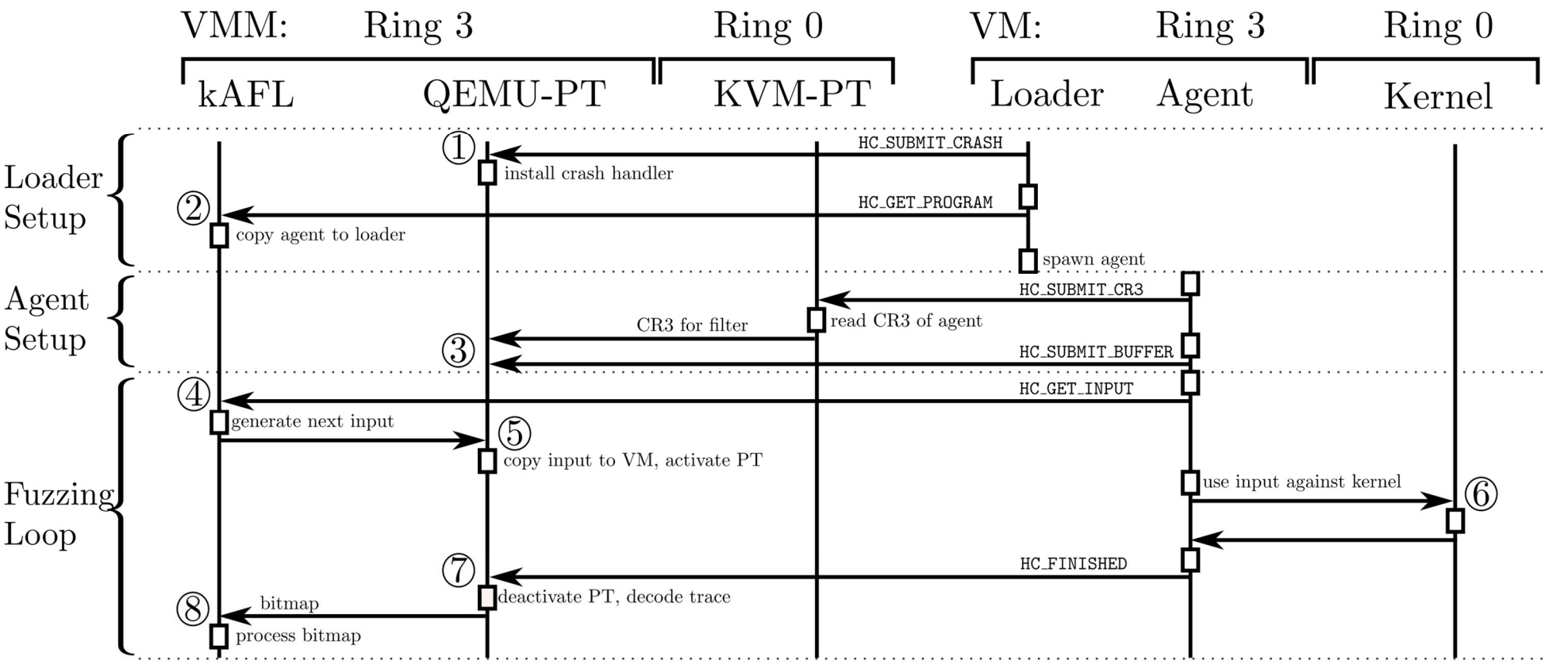

Host
VM
ring 0
ring 3
KVM-PT
Intel PT
Target
hypercalls
QEMU-PT
⑦
⑤
⑥
Agent
kAFL
⑧
④

kAFL — подготовка к фаззингу

1. Создание виртуальной машины с *загрузчиком* и исследуемой программой
2. Запуск ВМ в kAFL-QEMU
3. Запуск загрузчика внутри ВМ — загрузчик делает гипервызов, по которому kAFL-QEMU создаёт снапшот виртуальной машины

kAFL — фаззинг

1. Снапшот запускается в kAFL-QEMU
2. Загрузчик передаёт адреса функций `panic`/`KeBugCheck`/…, принимает и запускает *агент*
3. Агент передаёт CR3 и адрес буфера для тестов
4. Агент в цикле принимает и запускает тесты


VMM:
Ring 3
Ring 0
VM:
Ring 3
Ring 0
kAFL
QEMU-PT
KVM-PT
Loader
Agent
Kernel
Loader Setup
Agent Setup
Fuzzing Loop
①
HC_SUBMIT_CRASH
install crash handler
②
HC_GET_PROGRAM
copy agent to loader
spawn agent
HC_SUBMIT_CR3
read CR3 of agent
CR3 for filter
③
HC_SUBMIT_BUFFER
④
HC_GET_INPUT
generate next input
⑤
copy input to VM, activate PT
use input against kernel
⑥
⑦
HC_FINISHED
deactivate PT, decode trace
⑧
bitmap
process bitmap

kAFL

Плюсы: можно фаззить бинарные программы, можно фаззить драйвера/BIOS/…, фаззинг происходит в виртуалке — никаких последствий для хоста, можно гибко настраивать окружение

Минусы: фаззинг происходит в виртуалке — занимает много места

Redqueen

Ситуация №1

```
if (test[4] == 'X') {...}
```

afl пройдёт эту проверку за ~256 попыток.

Символьное исполнение пройдёт проверку за 1 попытку.

Ситуация №2

```
if (*(int*)&test[4] == 0xCAFEF00D) {...}
```

afl пройдёт эту проверку за ~4’300’000’000 попыток.

Символьное исполнение пройдёт проверку за 1 попытку.

Ситуация №3

```
if (hash(test+4, test+20) == 179) {...}
```

afl будет генерировать удачные тесты с шансом 1/256.

Символьное исполнение может застрять на этой проверке.

Redqueen

Мутаторы из afl это хорошо, но хочется что-то более умное.

Символьное исполнение это хорошо, но хочется что-то более быстрое.

Redqueen

1. Ставим брейкпойнт на проверке
2. Случайно меняем данные в тесте (красим тест) так, чтобы исполнение дошло до проверки
3. Проверка имеет вид A ?= C, где C это константа. Если мы знаем, на каких позициях в тесте есть A, то мы можем попробовать сгенерировать тест, проходящий эту проверку.

```
if (*(int*)&test[4] == 0xCAFEF00D) {...}

// test = “\0\0\0\0\0\0\0\0”
if (0 == 0xCAFEF00D) {...}  // false

// Redqueen: test = “\x00\xD3!K;\xD9)\x80”
if (0x8029D93B == 0xCAFEF00D) {...}

// Result: test = “\x00\xD3!K\x0D\xF0\xFE\xCA”
if (0xCAFEF00D == 0xCAFEF00D) {...}  // true
```

Бонус: обход хешей

Если таким образом мы можем пройти проверку, но значение меняется при изменении теста, то это может быть хеш.

Можно попробовать запатчить проверку, пофаззить и проверить, что мы всё ещё умеем честно проходить проверку без её удаления.

Redqueen — оценка

Детерминированное время работы.

Если новый тест возможно найти из старого с помощью Redqueen, это произойдёт быстро и точно.

Есть смысл запускать в первую очередь, перед более простыми мутаторами из afl.

Redqueen реализован в kAFL (2020) и afl++.

Grimoire

Ситуация

Фаззим интерпретатор языка, тест “`print(1)`”.

Простые мутации постоянно нарушают структуру теста. Как автоматически вывести элементы структуры?

Grimoire

Синтаксически корректный текст “`print(1)`” породил новое покрытие — вызов функции.

Будем пытаться выкидывать из теста символы, пока получаем покрытие, отличающееся от базового.

Grimoire

“`rint(1)`” — некорректный код, «пустое» покрытие

“1” — корректный код, выводим из него токен «1»

“`print()`” — корректный код, выводим из него шаблон “`print(□)`”

Grimoire

Шаблоны:

1. `if(x){□}else{□}`
2. `print(□)`
3. `throw □`

Комбинация: `if(x){throw 1}else{print(y)}`

Grimoire — оценка

Работает лучше базовых мутаций afl, но хуже мутаторов на основе грамматик.

По оценке авторов, лучше всего проявляет себя в связке с грамматическим мутатором, позволяет найти дополнительные пути.

kAFL (2020)

kAFL (2020)

Форк изначального kAFL, в который влиты Redqueen и Grimoire.

В данный момент не развивается, как и всё предыдущее, но иногда обновляется под новые версии QEMU и Linux.

Intel PT

Intel PT

«Интерфейс» в процессорах Intel для получения трассы исполнения и некоторых других данных.

Трасса записывается в сжатом виде, для её использования нужен нетривиальный декодировщик.

Intel PT — сжатие

- Записывается только нижняя изменённая часть IP, остальное надо брать из предыдущего значения
- IP после RET не передаётся, если адрес возврата на стеке не изменился
- Условные прыжки со статически известными адресами переходов сжимаются в 1 бит: прыгнули/не прыгнули

Intel PT — фильтрация трассы

1. По уровню исполнения, CPL 0/3
2. По контексту процесса — значение CR3
3. По диапазонам IP, до 4 диапазонов

Intel PT — фильтрация трассы

Table 35-2. IP Filtering Packet Example

Code Flow	Packets
Bar: jmp RangeBase // jump into filter range RangeBase: jcc Foo // not taken add eax, 1 Foo: jmp RangeLimit+1 // jump out of filter range RangeLimit: nop jcc Bar	TIP.PGE(RangeBase) TNT(0) TIP.PGD(RangeLimit+1)

Intel PT — проблемы

- Для декодирования нужен исполняемый код программы
- Прерывания влияют на трассу
- Множество деталей взаимодействия с другими компонентами процессора: VMX, SGX, TSX, SMM, …
- Десятки багов (в микроархитектуре Skylake), которые невозможно исправить программно

Intel PT…

014 TIP.PGD may not have target IP payload
023 VM entry that clears TraceEn may generate a FUP
030 FUP may be dropped after OVF
036 OVF packet mey be lost if immediately preceding a TraceStop
039 Writing a non-canonical value to an LBR MSR does not signal a #GP when Intel PT is enabled
041 Buffer overflow may result in incorrect packets
050 CYCTresh value of 13 is not supported
067 ToPA PMI does not freeze performance monitoring counters
108 CYC packets can be dropped when immediately preceding PSB
109 VM-entry indication depends on the incorrect VMCS control field
114 Intel PT may drop all packets after an internal buffer overflow
115 ToPA tables read from non-cacheable memory during an Intel TSX transaction may lead to processor hang
116 Performing an XACQUIRE to a ToPA table may lead to processor hang
119 CYC packet can be dropped when immediately preceding PSB
125 Intel PT trace may drop second byte of CYC packet
127 PSB+ packets may be omitted on a C6 transition
128 PacketEn change on C-state wake may not generate a TIP packet

fin

Ссылки

kAFL (2017): https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo

kAFL (2017): https://github.com/RUB-SysSec/kAFL

Redqueen: https://github.com/RUB-SysSec/redqueen

Grimoire: https://github.com/RUB-SysSec/grimoire

kAFL (2020): https://github.com/IntelLabs/kAFL

Intel PT: Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B, 3C & 3D), Chapter 35 "Intel Processor Trace"