Śledzenie ścieżki wykonania procesu (programu)

Robert Święcki - PWNing 2016

Plan prezentacji

- Disclaimer
- Program a proces
- Śledzenie wykonania, czyli właściwie co?
- Motywacje
- Metody programowe
- Metody programowo-sprzętowe
- Metody sprzętowe
- Wydajność

Czym jest śledzenie wykonania kodu?

- *ang.* control flow path tracking (śledzenie ścieżki przebiegu sterowania)
- Czego oczekujemy

Czym jest śledzenie wykonania kodu?

- Wejście - wykonanie procesu (programu)
- Wyjście
  - sekwencja wykonania
  - wykonane bloki
  - zależności czasowe
- W sposób możliwie szybki

Czym jest śledzenie wykonania kodu?

```
void PRINT_B() {
    write(0, "B", 1);
}
void PRINT_A() {
    PRINT_B();
    write(0, "A", 1);
}
void PRINT_NL() {
    write(0, "\n", 1);
}

...
```

```
…

int main(void) {
    for (int i = 0; i < 3; i++) {
        PRINT_A();
    }
    PRINT_B();
    PRINT_NL();
    return 0;
}
```

Czym jest śledzenie wykonania kodu?

```
1     main  inst 1,7
2       PRINT_A   inst 8,11
3         PRINT_B inst 12,18
4                 write inst 19,20
5                 __write_nocancel  inst 21,25
6         PRINT_B inst 26,28
7       PRINT_A   inst 29,33
8         write   inst 34,35
9         __write_nocancel    inst 36,40
10      PRINT_A   inst 41,43
11    main  inst 44,48
12      PRINT_A   inst 49,52
13        PRINT_B inst 53,59
14          write inst 60,61
15          __write_nocancel  inst 62,66
16        PRINT_B inst 67,69
17      PRINT_A   inst 70,74
18        write   inst 75,76
19        __write_nocancel    inst 77,81
20      PRINT_A   inst 82,84
21    main  inst 85,89
22      PRINT_A   inst 90,93
23        PRINT_B inst 94,100
24          write inst 101,102
25          __write_nocancel  inst 103,107
26        PRINT_B inst 108,110
27      PRINT_A   inst 111,115
28        write   inst 116,117
29        __write_nocancel    inst 118,122
30      PRINT_A   inst 123,125
31    main  inst 126,130
32      PRINT_B   inst 131,137
33        write   inst 138,139
34        __write_nocancel    inst 140,144
35      PRINT_B   inst 145,147
36    main  inst 148,149
37      PRINT_NL  inst 150,156
38        write   inst 157,158
39        __write_nocancel    inst 159,163
40      PRINT_NL  inst 164,166
41    main  inst 167,169
```

Motywacje

- Malware research
- Analiza programów binarnych
- Debugowanie kodu
- Profilowanie kodu
- CTF-y
- Fuzzing

Malware research

- Obfuskacja kodu *[demo w/strace]*

```
        .global _start
_start:
        movq $231, %rax
        movq $10, %rdi
        jmp 1f
        .byte 0xC0
1:
        syscall
```

```
00000000004000d4 <_start>:
  4000d4: 48 c7 c0 e7 00 00 00    mov    $0xe7,%rax
  4000db: 48 c7 c7 0a 00 00 00    mov    $0xa,%rdi
  4000e2: eb 01                   jmp    4000e5 <_start+0x11>
  4000e4: c0 0f 05                rorb   $0x5,(%rdi)
```

- Time locks (konieczne b.szybki proces śledzenia)
- Szyfrowanie kodu
- Kod dynamiczny (moduły) i samomodyfikujący się

Programy binarne

- Brak kodu źródłowego (i symboli)
  - Reverse engineering
  - Debugging
  - Bughunting

Debugowanie programów

- Duża złożoność (np. Apache) → wolny single-stepping
- Nietrywialne do analizy skoki warunkowe
- JIT
- Kod samomodyfikujący się
- Języki wysokiego poziomu - bytecode, interpretery

CTF-y

- Zarówno zadania RE jak i PWN
- Pierwszy rzut oka na zadanie - disasembler/dekompilator mogą się mylić
  - **strace ./challenge** *[demo]*

- Analiza przebiegu dla zadanego inputu
- Fuzzowanie (np. CGC DARPA)
  - głównie AFL+QEmu-User lub modyfikacja dynamiczna kodu

Fuzzing

- Nie - jak się program dokładnie wykonuje (sekwencje)
- Lecz - co program właściwie wykonuje (coverage)
- Sekwencja wykonania jest o wiele mniej istotna od prostego określenia wykonanych bloków kodu → code coverage
  - sekwencje → tuples i liczniki
- Code coverage → maksymalizacja attack surface
  - Galeria błędów na stronie AFL i Honggfuzz
  - Np. błąd Critical w OpenSSL **CVE-2016-6309** (zgłoszony przeze mnie) został znaleziony przy użyciu instrumentacji SanCov i za pomocą fuzzera Honggfuzz

Fuzzing

[demo]

```
int check0() {
}
int check1() {
}
int check2() {
}

int main(void) {
    char buf[4] = {};
    read(0, buf, 4);
...
```

```
...
if (buf[0] == 'A') {
        check0();
        if (buf[1] == 'B') {
            check1();
            if (buf[2] == 'C') {
                check2();
                if (buf[3] == 'D') {
                    int *i = 0x41414141;
                    *i = 0x12345678;
                }
            }
        }
    }
    return 0;
}
```

Metody programowe

- Brak specjalizowanego wsparcia w sprzęcie
- Większośc współczesnych CPU ogólnego przeznaczenia z MMU je wspiera

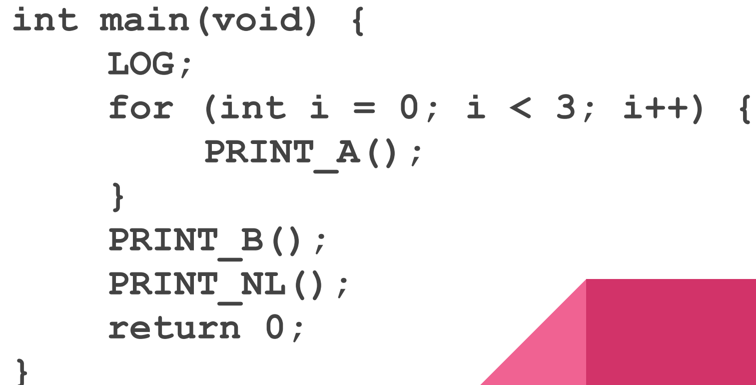
Metody programowe - printf-based debugging

```
#define LOG printf("\nFunkcja: %s,
linia: %d\n", __func__, __LINE__)

void PRINT_B() {
     LOG;
     write(0, "B", 1);
}

void PRINT_A() {
     LOG;
     PRINT_B();
     write(0, "A", 1);
}
```

```
void PRINT_NL() {
     LOG;
     write(0, "\n", 1);
}
```



```
int main(void) {
     LOG;
     for (int i = 0; i < 3; i++) {
          PRINT_A();
     }
     PRINT_B();
     PRINT_NL();
     return 0;
}
```

Metody programowe - printf-based debugging

```
Funkcja: main, linia: 20
Funkcja: PRINT_A, linia: 9
Funkcja: PRINT_B, linia: 4
BA
Funkcja: PRINT_A, linia: 9
Funkcja: PRINT_B, linia: 4
BA
Funkcja: PRINT_A, linia: 9
Funkcja: PRINT_B, linia: 4
BA
Funkcja: PRINT_B, linia: 4
B
Funkcja: PRINT_NL, linia: 15
```

Metody programowe - instrument. (kompilacja)

- GCC/Clang
- Funkcje - wejście i wyjście

```
void __cyg_profile_func_enter(void *func, void *caller) {
     printf("Wywołano %p z funkcji %p\n", func, caller);

}
void __cyg_profile_func_exit(void* func, void *caller) {
     printf("Powrót z funkcji %p do funkcji %p\n", func, caller);
}
$ gcc -finstrument-functions -ggdb -c prog.c -o prog-instr.o
$ gcc -ggdb prog-instr.o instrgcc.c -o prog-instr
$ ./prog-instr | grep Wyw | cut -f2 -d" " | \
 addr2line -sfe ./prog-instr
```

Metody programowe - instrument. (kompilacja)

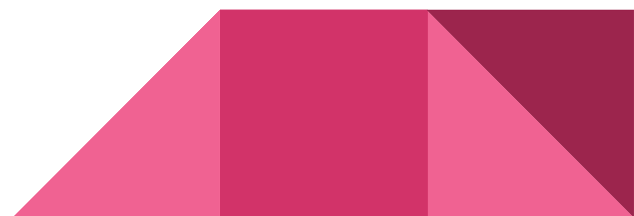
```
./prog-instr | grep Wywołano | \
 cut -f2 -d" " | \
 addr2line -sfe ./prog-instr
```

```
BABABAB
main
prog.c:14
PRINT_A
prog.c:5
PRINT_B
prog.c:1
PRINT_A
prog.c:5
PRINT_B

...
```

```
...

prog.c:1
PRINT_A
prog.c:5
PRINT_B
prog.c:1
PRINT_B
prog.c:1
PRINT_NL
prog.c:10
```



Metody programowe - instrument. (kompilacja)

```
void __sanitizer_cov_trace_pc(void)
{
    printf("Wywołanie z adresu: %p\n", __builtin_return_address(0));
}

void __sanitizer_cov_trace_pc_indir(void *callee)
{
    printf("Skok z adresu %p do adresu %p\n", callee,
    __builtin_return_address(0));
}
```

- Bloki, nie funkcje

Metody programowe - instrument. (kompilacja)

```
$ ./prog-instrclang
Wywołanie bloku: 0x4005ed
Wywołanie bloku: 0x40060e
Wywołanie bloku: 0x4005fb
Wywołanie bloku: 0x400594
Wywołanie bloku: 0x40056f
BA
Wywołanie bloku: 0x40060e
Wywołanie bloku: 0x4005fb
Wywołanie bloku: 0x400594
Wywołanie bloku: 0x40056f
BA
Wywołanie bloku: 0x40060e
Wywołanie bloku: 0x4005fb
```

```
Wywołanie bloku: 0x400594
Wywołanie bloku: 0x40056f
BA
Wywołanie bloku: 0x40060e
Wywołanie bloku: 0x400619
Wywołanie bloku: 0x40056f
B
Wywołanie bloku: 0x4005c3

Wywołanie bloku: 0x400637
```

- addr2line

Metody programowe - kod binarny

- Instrumentacja programu (pliku na dysku)
- Instrumentacja procesu
  - gdb-style - int3 (break)
  - Linux → ptrace()
  - Windows → WinAFL
- Pakiety
  - DynamoRIO
  - DynInst
  - Intel PIN
- In-flight instrumentation
- Intel PIN dla zadań DARPA CGC (emulacja syscalli)

Metody programowe - kod binarny

- Przykład dla Intel PIN

```
export LD_BIND_NOW=1
../../../pin -t ./obj-intel64/calltrace.so -- ./prog
```

```
main
PRINT_A
PRINT_B
__write
__write
PRINT_A
PRINT_B
__write
__write
...
```

```
...
PRINT_A
PRINT_B
__write
__write
PRINT_B
__write
PRINT_NL
__write
exit
```

Metody programowe - śledzenie syscalli

- Linux
  - szeroki zakres użycia interfejsu ptrace()
- Relatywnie wysokopoziomowe śledzenie procesu → interakcja procesu z zasobami nielokalnymi, ujednolicone adresy wywołań (np. poprzez glibc)

```
$ strace -i ./prog
...
[000000000040f690] write(0, "B", 1)      = 1
[000000000040f690] write(0, "A", 1)      = 1
[000000000040f690] write(0, "B", 1)      = 1
[000000000040f690] write(0, "A", 1)      = 1
[000000000040f690] write(0, "B", 1)      = 1
[000000000040f690] write(0, "A", 1)      = 1
[000000000040f690] write(0, "B", 1)      = 1
[000000000040f690] write(0, "\n", 1)     = 1
[000000000040f608] exit_group(0)         = ?
```

Metody programowe - śledzenie wyw. bibliot.

```
$ ltrace -w2 ./prog-dyn

__libc_start_main([ "./prog-dyn" ])

write(0, "B", 1) = 1
 > prog-dyn(PRINT_B+0x1e) [400544]
 > prog-dyn(PRINT_A+0xd) [400553]
 > prog-dyn(main+0x1a) [4005aa]

write(0, "A", 1) = 1
 > prog-dyn(PRINT_A+0x28) [40056e]
 > prog-dyn(main+0x1a) [4005aa]
 > libc.so.6
(__libc_start_main+0xf0)
[7fe10783d730]

write(0, "B", 1) = 1
 > prog-dyn(PRINT_B+0x1e) [400544]
 > prog-dyn(PRINT_A+0xd) [400553]
 > prog-dyn(main+0x1a) [4005aa]

...
```

```
...

write(0, "\n", 1) = 1
 > prog-dyn(PRINT_NL+0x1e) [40058e]
 > prog-dyn(main+0x38) [4005c8]
 > libc.so.6
(__libc_start_main+0xf0)
[7fe10783d730]

+++ exited (status 0) +++
```

Metody programowe - śledzenie wyw. bibliot.

- Wady
  - Aplikacje dynamiczne (konieczne zmiany w GOT/PLT)
  - Tylko wywołania biblioteczne i systemowe

Metody programowe - emulacja instrukcji

- Emulatory dla programów - np. qemu-user
- Konieczność translacji/emulacji każdej instrukcji → dostęp do informacji o wykonaniu/adresie każdej emulowanej instrukcji CPU
- Użycie mechanizmów JIT skutkuje relatywnie szybkim procesem wykonania
- Zmodyfikowany qemu-user-i386 - używany w DARPA CGC przez większośc zespołów do określania code coverage wraz z feedback-guided fuzzerami
- Może śledzić wywołania systemowe

Metody programowe - emulacja instrukcji

```
$ qemu-x86_64 -strace -d exec -B 0x40000000 ./prog 2>&1 >/dev/null | grep -E \
  "main|PRINT"
```

```
Trace 0x55bbb1c78620 [0000400998] main
Trace 0x55bbb1c786b0 [00004009b7] main
Trace 0x55bbb1c78750 [00004009a9] main
Trace 0x55bbb1c787b0 [000040094e] PRINT_A
Trace 0x55bbb1c78820 [000040092e] PRINT_B
Trace 0x55bbb1c78a60 [000040094b] PRINT_B
Trace 0x55bbb1c78ac0 [000040095c] PRINT_A
Trace 0x55bbb1c78b30 [0000400975] PRINT_A
Trace 0x55bbb1c78b90 [00004009b3] main
Trace 0x55bbb1c78750 [00004009a9] main
Trace 0x55bbb1c78a60 [000040094b] PRINT_B
Trace 0x55bbb1c78ac0 [000040095c] PRINT_A
Trace 0x55bbb1c78b30 [0000400975] PRINT_A
Trace 0x55bbb1c78b90 [00004009b3] main
...
```

```
...
Trace 0x55bbb1c78a60 [000040094b] PRINT_B
Trace 0x55bbb1c78ac0 [000040095c] PRINT_A
Trace 0x55bbb1c78b30 [0000400975] PRINT_A
Trace 0x55bbb1c78b90 [00004009b3] main
Trace 0x55bbb1c78c50 [00004009bd] main
Trace 0x55bbb1c78820 [000040092e] PRINT_B
Trace 0x55bbb1c78a60 [000040094b] PRINT_B
Trace 0x55bbb1c78cb0 [00004009c7] main
Trace 0x55bbb1c78d10 [0000400978] PRINT_NL
Trace 0x55bbb1c78da0 [0000400995] PRINT_NL
Trace 0x55bbb1c78e00 [00004009d1] main
```

Metody programowe - emulacja instrukcji

- Qira
  - Autorstwo: George *geohot* Hotz - 2015/2016
  - *Timeless Debugger*
  - Oparty o zmodyfikowany qemu-user
  - Działa w przeglądarce
  - Stworzony z myślą o CTF-ach, ale nie tylko
  - “Nagrywa przebieg wykonania” - podobnie do “record btrace” w gdb

Metody programowo-sprzętowe - sampling

- Mechanizm oparty of timery
- Okresowy zapis parametrów procesu (np. rejestrów CPU)
- Niezbyt dobry mechanizm do określania ścieżki przebiegu (mała precyzja)
- Częściej używany do profilowania programów

```
$ perf record --per-thread -e instructions:u -- ./prog
$ perf script
 prog 31059        1 instructions:u: 400812 _start+0x2
 prog 31059        1 instructions:u: 400c60 __libc_start_main+0x0
 prog 31059        6 instructions:u: 400c72 __libc_start_main+0x12
 prog 31059       60 instructions:u: 400dc6 __libc_start_main+0x166
 prog 31059      322 instructions:u: 400d2e __libc_start_main+0xce
 prog 31059     3160 instructions:u: 410960 _dl_aux_init+0x300
 prog 31059     6531 instructions:u: 410ac8 _dl_non_dynamic_init+0x128
```

Metody programowo-sprzętowe - rejestry DR

- Debug Register
- Specjalizowane rejestry w CPU (w x86 - DR0-DR3 + DR6/7)
- Generowanie wyjątku w przypadku spełnienia warunku
  - Adres PC osiąga zadaną wartość
  - Odczyt z zadanego adresu
  - Zapis do zadanego adresu
  - Dostęp do szyny IO po zadanym adresem
- Wyjątek obsługiwany przez ten sam lub inny proces
- Może posłużyć do śledzenia przepływu sterowania
  - niezbyt szybkie (liczne context switche)
  - mała liczba rejestrów → konieczne częste zmiany ich zawartości
- Implementacja **hardware breakpoints** w gdb

Metody programowo-sprzętowe - single-stepping

- Jedna z najwcześniejszych metod stosowania w debuggingu
  - wolna, lecz pozwala na określenie ścieżki przebiegu sterowania
  - zazwyczaj bit w rejestrze kontrolnym CPU → **TF** w RFLAGS/EFLAGS dla x86

- Typowy schemat:
  - Ustaw bit TF
  - Wznów wykonanie procesu
  - Wyjątek jest generowany przez CPU przy następnej instrukcji maszynowej
  - Bit TF jest czyszczony przez CPU
  - Proces nadzorcy obsługuje wyjątek
    - OS może przekazać sterowanie do innego procesu
  - Ustaw bit TF…...

Metody programowo-sprzętowe - single-stepping

- Przykładowe makro dla gdb

```
define tfloop
    while(1)
        stepi
    end
end
start
tfloop
```

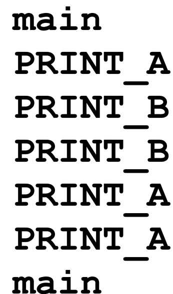
Metody programowo-sprzętowe - BTF

- Branch Trap Flag
  - Wspierane zarówno przez CPU Intel jak i AMD
  - Dla Intel-a - od procesorów Intel Core
  - Jak **single-stepping**, lecz dla instrukcji skoków (jmp, call, ret, itp.)
  - W Linuksie, **ptrace(PTRACE_SINGLEBLOCK)**
  - Własne patche dla gdb
- Context-switche sprawiają, że jest to jedna z najwolniejszych dostępnych metod śledzenia ścieżki przebiegu procesu
  - Użyteczna, lecz bardziej dla gdb-style debuggingu

Metody programowo-sprzętowe - BTF

```
$ ./step ./prog 2>&1 | addr2line -fe
./prog | grep -E "^main|^PRINT_" | wc
-l
115

main
PRINT_B
PRINT_B
PRINT_B
PRINT_B
PRINT_B
PRINT_B
PRINT_B
PRINT_B
PRINT_B
PRINT_B
main
```



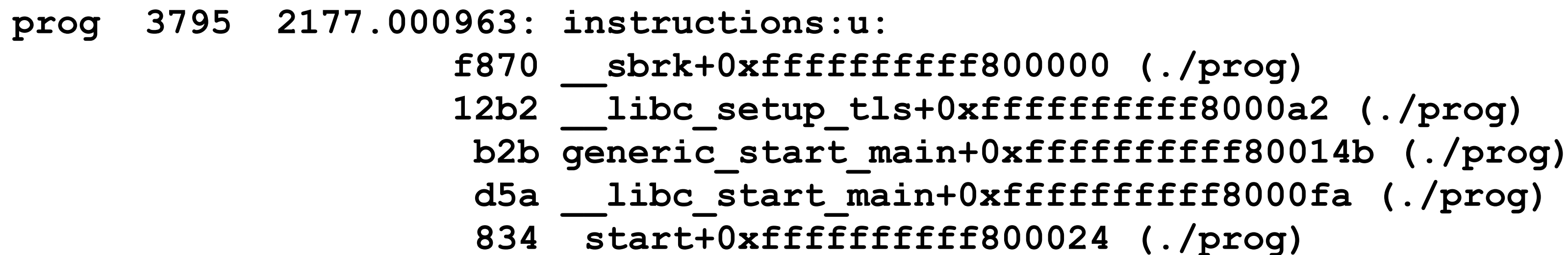
```
$ ./block ./prog 2>&1 | addr2line -fe
./prog | grep -E "^main|^PRINT_" | wc
-l
29

main
PRINT_A
PRINT_B
PRINT_B
PRINT_A
PRINT_A
main
```

Metody programowo-sprzętowe - LBR

- Intel Last Branch Record
  - Od 4 do 16 specjalnych rejestrów MSR
  - Rejestracja ostatnich punktów skoku (jmp, call, ret itp.)
- Przejściowe wsparcie (patche) dla interfejsu ptrace
- W linuksie wsparcie poprzez perf
- Szybsze od DR, Single-, Block- steppingu, konieczne context-switche
  - Podstawa dla implementacji innych mechanizmów (np. BTS)

Metody programowo-sprzętowe - LBR



Metody programowo-sprzętowe - BTS

- Branch Trace Store
- Nowsze procesory Intel x86
- Rozwinięcie mechanizmu LBR
- Wykorzystanie mechanizmu DS - Debug Store
  - Bufory cykliczne dla LBR
  - Brak context switchy
- Wykorzystuje MMU (wolne)
- Linux
  - Pierwotnie w ramach regularnego mechanizmu perf
  - Od wersji kernela 4.2 - mechanizm AUXTRACE

Metody programowo-sprzętowe - BTS

< 4.2

```
. ... raw event: size 24 bytes
.  0000:  09 00 00 00 01 00 18 00 29 8e 1a b8 ff ff ff ff  ........).......
.  0010:  34 7c 00 00 34 7c 00 00                          4|..4|..
0x2e90 [0x18]: PERF_RECORD_SAMPLE(IP, 0x1): 0xffffffffb81a8e29 period: 2 addr: 0
 ... thread: prog:31796
 ...... dso: [kernel.kallsyms]
            prog 31796 branch-instructions:  ffffffffb81a8e29 perf_event_task_tick+0x259
([kernel.kallsyms])
0x2ea8 [0x18]: event: 9
```

>= 4.2

```
.  00005be0:  eb f0 40 00 00 00 00 00 01 f1 40 00 00 00 00 00 00 00 00 00 00 00 00 00
40f0eb -> 40f101 miss
.  00005bf8:  30 f1 40 00 00 00 00 00 80 f1 40 00 00 00 00 00 00 00 00 00 00 00 00 00
40f130 -> 40f180 pred
.  00005c10:  87 f1 40 00 00 00 00 00 10 f1 40 00 00 00 00 00 00 00 00 00 00 00 00 00
40f187 -> 40f110 miss
.  00005c28:  50 f1 40 00 00 00 00 00 30 f1 40 00 00 00 00 00 00 00 00 00 00 00 00 00
40f150 -> 40f130 miss
```

Metody programowo-sprzętowe - BTS

[demo]

Metody programowo-sprzętowe - Intel PT

- Intel Processor Trace - od procesorów Intel x86 serii Broadwell
- >90 stron w Manualu x86 (Vol. 3C)
- Per-Core
- Omija MMU
- Mikrokod
  - kompaktowy zapis
  - kompresja adresów
  - wiele rodzajów pakietów: skoki, timing, adresy, wyjątki, stronnicowanie
- Śledzenie różnych poziomów uprzywilejowania
  - SMM
  - Ring 0-3
  - VM

Metody programowo-sprzętowe - Intel PT

Table 36-22. FUP Packet Definition

Name	Flow Update (FUP) Packet		
Packet Format	*(see packet format table below)*		
Dependencies	TriggerEn & ContextEn. (Typically depends on BranchEn and FilterEn as well, see Section 36.2.4 for details.)	Generation Scenario	Asynchronous Events (interrupts, exceptions, INIT, SIPI, SMI, VM exit[1], #MC), XBEGIN, XEND, XABORT, XACQUIRE, XRELEASE, (EENTRY, EEXIT, ERESUME, EEE, AEX,)[2], INT 0, INT 3, INT n, a WRMSR that disables packet generation.
Description	Provides the source address for asynchronous events, and some other instructions. Is never sent alone, always sent with an associated TIP or MODE packet, and potentially others.		
Application	FUP packets provide the IP to which they bind. However, they are never standalone, but are coupled with other packets. In TSX cases, the FUP is immediately preceded by a MODE.TSX, which binds to the same IP. A TIP will follow only in the case of TSX aborts, see Section 36.4.2.8 for details. Otherwise, FUPs are part of compound packet events (see Section 36.4.1). In these compound cases, the FUP provides the source IP for an instruction or event, while a following TIP (or TIP.PGD) uop will provide any destination IP. Other packets may be included in the compound event between the FUP and TIP.		

Packet Format:

	7	6	5	4	3	2	1	0
0	IPBytes			1	1	1	0	1
1	IP[7:0]							
2	IP[15:8]							
3	IP[23:16]							
4	IP[31:24]							
5	IP[39:32]							
6	IP[47:40]							
7	IP[55:48]							
8	IP[63:56]							

Metody programowo-sprzętowe - Intel PT

- Tylko tyle informacji, ile potrzebne do odtworzenia ścieżki sterowania
  - Pakiet TNT (Taken-Not-Taken Branch)
  - Potrzebny program binarny, aby odtworzyć ścieżkę
- Fuzzing nie wymaga programu binarnego, kosztem precyzji
  - Implementacja w honggfuzz (--linux_perf_ipt_block)
- Zapis → szybki
- Dekodowanie → wolne
- Implementacje
  - Dekodowanie - biblioteka libipt.so (od Intel-a)
  - Linux: Kernel >= 4.2 - wymaga AUXTRACE
  - Linux: Niestabilna implementacja, nawet w 4.8
  - Linux: Simple-PT od Andi Kleen
  - Windows: https://github.com/talos-vulndev/TalosIntel[illegible]ver

Metody programowo-sprzętowe - Intel PT

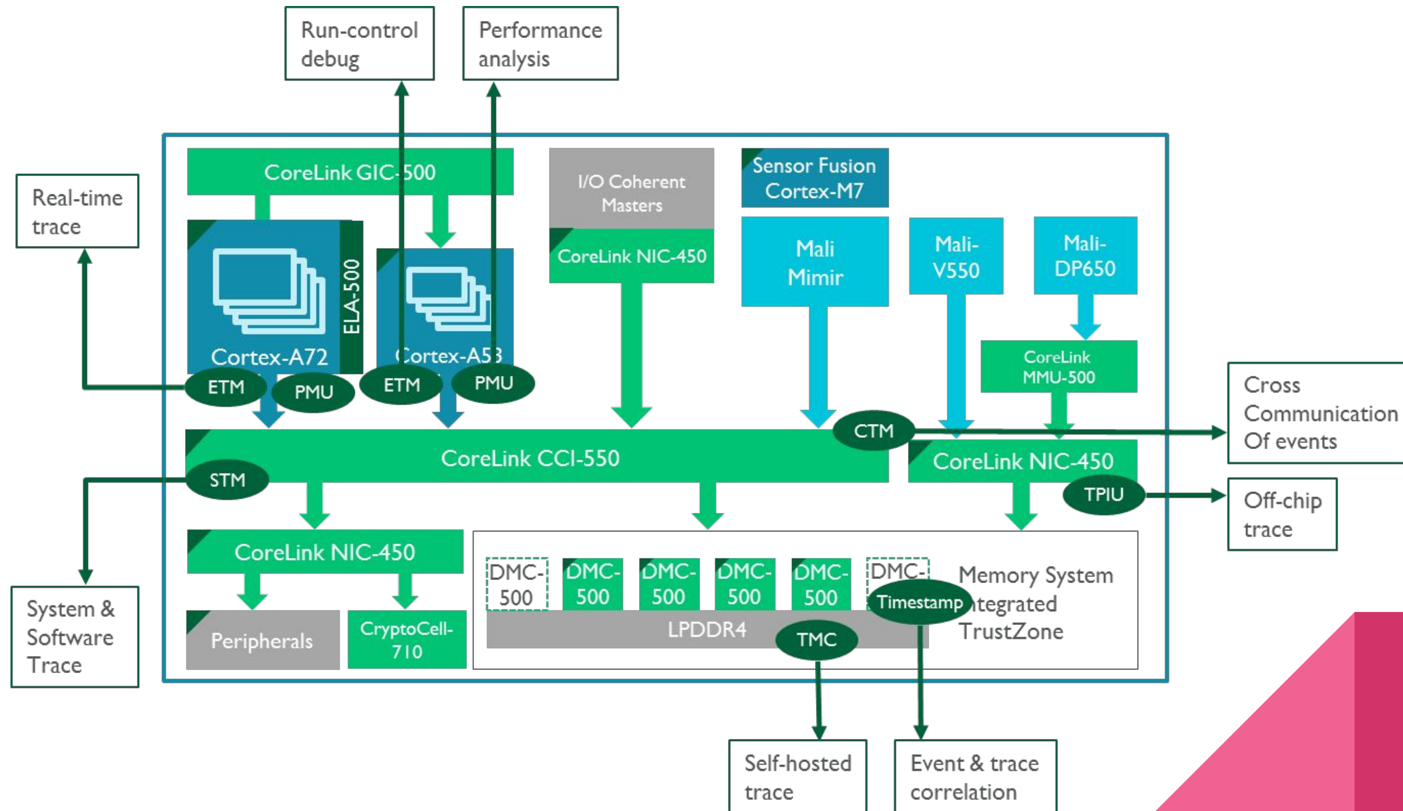
```
. ... Intel Processor Trace data: size 5824 bytes
.  00000000:  02 82 02 82 02 82 02    // PSB
.  00000010:  00 00 00 00 00 00          PAD
.  00000016:  19 74 6c 6b f2 38 00 00    TSC 0x38f26b6c74
.  0000001e:  00 00 00 00 00 00 00 00    PAD
.  00000026:  02 73 cc c1 00 60 00 00    TMA CTC 0xc1cc FC
0x60
.  0000002e:  00 00                      PAD
.  00000030:  02 03 2a 00                CBR 0x2a
.  00000034:  02 23                      PSBEND
.  00000036:  59 3a                      MTC 0x3a
.  00000038:  59 3b                      MTC 0x3b
.  000000d5:  99 01                      MODE.Exec 64
.  000000d7:  71 c0 fc 38 0b 8d 7f 00    TIP.PGE
0x7f8d0b38fcc0
.  000000df:  00                         PAD
.  000000e0:  59 89 00 00 00 00 00       MTC 0x89
.  000000e7:  7d c0 fc 38 0b 8d 7f 00    FUP 0x7f8d0b38fcc0
.  000000ef:  00                         PAD
.  000000f0:  01                         TIP.PGD no ip
```

Metody programowo-sprzętowe - ARM CoreSight

"CoreSight provides components for the implementation solutions for debug access, instruction tracing, cross-triggering and time-stamping"

- On-chip PMU dla ARM
- Natywne narzędzia dostępne od ARM i od 3rd party vendors
- Linux - istnieje podstawowe wsparcie w drivers/hwtracing/coresight
  - Wydaje się nie udostępniać funkcjonalności śledzenia procesu *by default*
  - Wymagany patche od OpenCSD by Linaro - https://github.com/Linaro/OpenCSD
  - Patche te wspierają platformy ARMv7 Vertex oraz ARMv8 Juno, wydają się nie działać np. z Raspberry Pi3
- Potencjalnie dobra technologia dla platformy Android

Metody programowo-sprzętowe - ARM CoreSight



Metody sprzętowe

- Wymagany dostęp DMA
  - Karty PCI (PCMCIA, CardBus)
  - Interfejs FireWire - firescope - http://gate.crashing.org/~benh/firescope.tar.gz
  - Thunderbolt

Wydajność

Q&A