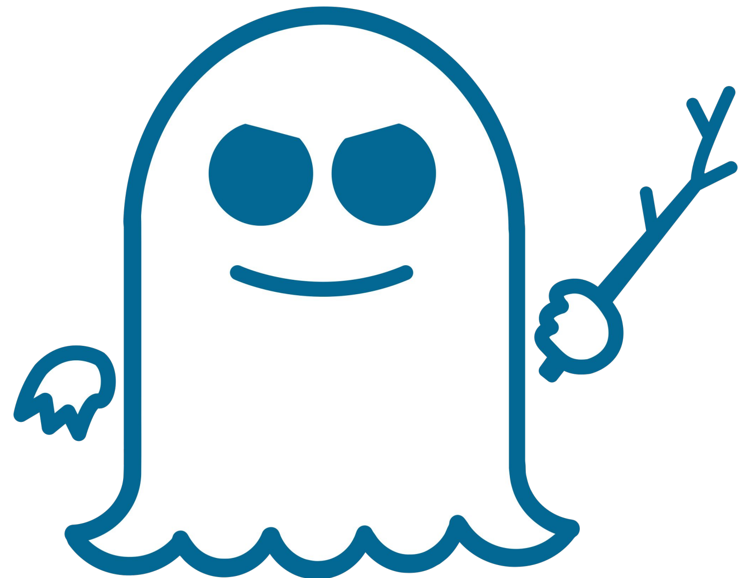
Meltdown & Spectre



Jan Mazur
Instytut Informatyki Uniwersytetu Wrocławskiego
Seminarium: Architektury Systemów Komputerowych

Odpowiedzialni za zamieszanie

Meltdown:

- Jann Horn (Google Project Zero),
- Werner Haas, Thomas Prescher (Cyberus Technology),
- Daniel Gruss, Moritz Lipp, Stefan Mangard, Michael Schwarz(Graz University of Technology)

Spectre:

- Jann Horn (Google Project Zero) and
- Paul Kocher we współpracy z, Daniel Genkin (University of Pennsylvania i University of Maryland), Mike Hamburg (Rambus), Moritz Lipp (Graz University of Technology), oraz Yuval Yarom (University of Adelaide i Data61)

Zainspirowani wpisem na blogu Andersa Fogha (28.07.17), któremu nie udało się jednak stworzyć działającego PoC (*proof of concept*).

Trochę historii

- Błędy były znane co najmniej od początku czerwca (zgłoszone przez Horna do Intela, ARM, AMD itd.).
- Nałożone embargo. Rozpoczęcie pracy nad poprawkami.
- KASLR is Dead: Long Live KASLR. KAISER - poprawa skuteczności KASLR. KAISER -> KPTI - zapobiega atakowi meltdown.
- 20. grudnia artykuł “ The current state of kernel page-table isolation”. KPTI. Seria szybkich poprawek core jądra Linuxa. Merge w 3 miesiące.
- Na listach mailingowych adresy głównych developerów jądra, pracowników Intela, Google, Amazona.
- The mysterious case of the Linux Page Table Isolation patches. Jedne z pierwszych dobrych domysłów.
- Intel 3. stycznia wydaje oświadczenie mówiące, że wszystko jest OK.
- Grupa badaczy z uniwersytetu w Graz wystawia stronę poświęconą atakom oraz Google publikuje post na blogu grupy Project Zero opisujący detale ataków.

Interesujące artykuły

- Triple Meltdown: How So Many Researchers Found a 20-Year-Old Chip Flaw At the Same Time
- How a 22-Year-Old Discovered the Worst Chip Flaws in History
- The Hidden Toll of Fixing Meltdown and Spectre
- Meltdown and Spectre Fixes Arrive—But Don't Solve Everything

Description

CPU hardware implementations are vulnerable to side-channel attacks refe
KAISER and KPTI). These attacks are described in detail by Google Projec
Information Processing and Communications (IAIK) at Graz University of Te

Impact

An attacker able to execute code with user privileges can achieve various ir
kernel memory and bypassing KASLR.

Solution

Replace CPU hardware

The underlying vulnerability is primarily caused by CPU architecture design
requires replacing vulnerable CPU hardware.

Apply updates

Operating system updates mitigate the underlying hardware vulnerability.

Description

CPU hardware implementation[illegible]
KAISER and KPTI). These atta[illegible]
Information Processing and Co[illegible]

```
-       /* Assume for now that ALL x86 CPUs are insecure */
-       setup_force_cpu_bug(X86_BUG_CPU_INSECURE);
+       if (c->x86_vendor != X86_VENDOR_AMD)
+               setup_force_cpu_bug(X86_BUG_CPU_INSECURE);

        fpu__init_system(c);
```

Impact

An attacker able to execute code with user privileges can achieve various ir
kernel memory and bypassing KASLR.

Solution

Replace CPU hardware

The underlying vulnerability is primarily caused by CPU architecture design
requires replacing vulnerable CPU hardware.

Apply updates

Operating system updates mitigate the underlying hardware vulnerability.

Description

CPU hardware implementation

KAISER and KPTI). These atta

Information Processing and Co

```
-        /* Assume for now that ALL x86 CPUs are insecure */
-        setup_force_cpu_bug(X86_BUG_CPU_INSECURE);
+        if (c->x86_vendor != X86_VENDOR_AMD)
+                setup_force_cpu_bug(X86_BUG_CPU_INSECURE);

         fpu__init_system(c);
```

Impact

An attacker able to execute code with user privileges can achieve various ir
kernel memory and bypassing KASLR.

Solution

Replace CPU hardware

The underlying vulnerability is primarily caused by CPU architecture design
requires replacing vulnerable CPU hardware.

Apply updates

Operating system updates mitigate the underlying hardware vulnerability.

Description

CPU hardware implementation[...]
KAISER and KPTI). These atta[...]
Information Processing and Co[...]

```
-        /* Assume for now that ALL x86 CPUs are insecure */
-        setup_force_cpu_bug(X86_BUG_CPU_INSECURE);
+        if (c->x86_vendor != X86_VENDOR_AMD)
+                setup_force_cpu_bug(X86_BUG_CPU_INSECURE);

         fpu__init_system(c);
```

author Tom Lendacky <thomas.lendacky@amd.com> 2017-12-26 23:43:54 -0600

Bryan Lunduke
@BryanLunduke

Obserwuj

This small patch, to the Linux Kernel, is the most epic burn on @Intel by @AMD.

Paraphrased in English:
"If the CPU isn't AMD, assume it's not secure."

can achieve various ir

ıre design

ɘrability.

Bryan Lunduke @BryanLunduke · 3.01
I mean, hell. That line of code would make a killer ad campaign. Put it on billboards:

"if (c->x86_vendor != X86_VENDOR_AMD)
  setup_force_cpu_bug(X86_BUG_CPU_INSECURE);"

If I worked in @AMD marketing, that would go up tomorrow. Everywhere.

Description

CPU hardware implementation
KAISER and KPTI). These atta
Information Processing and Co

```
-        /* Assume for now that ALL x86 CPUs are insecure */
-        setup_force_cpu_bug(X86_BUG_CPU_INSECURE);
+        if (c->x86_vendor != X86_VENDOR_AMD)
+                setup_force_cpu_bug(X86_BUG_CPU_INSECURE);

         fpu__init_system(c);
```

author

Bryan Lunduke
@BryanLunduke

Obserwuj

This small patch, to the Linux Kernel, is
the m

can achieve various ir

```
We came up with a list of technically correct acronyms:

  User Address Space Separation, prefix uass_

  Forcefully Unmap Complete Kernel With Interrupt Trampolines, prefix fuckwit_

but we are politically correct people so we settled for

 Kernel Page Table Isolation, prefix kpti_
```

Parap
"If the
secur

Bryan
I mean
billboards:

"if (c->x86_vendor != X86_VENDOR_AMD)
  setup_force_cpu_bug(X86_BUG_CPU_INSECURE);"

If I worked in @AMD marketing, that would go up tomorrow. Everywhere.

erability.

Description

CPU hardware implem
KAISER and KPTI). T
Information Processi

Bryan Lunduke
@BryanLunduke

This small patc
the m

Para
"If t
secu

Bryan
I mean
billboards:

"if (c->x86_v
  setup_forc

If I worked in @AMD marketing, that would go up tomorrow. Everywhere.

```
L x86 CPUs are insecure
G_CPU_INSECURE);
NDOR_AMD)
(X86_BUG_CPU_INSECURE
```

m>   2017-12-26 2 3:54 -0600

ir

```
We came u
d
fu                                          nes, prefix fuckwit_
bu we ar
 Kernel P
```

erability.

Covert channels

Ukryty kanał (*covert channel)* to mechanizm który nie został zaprojektowany z myślą o komunikacji, jednakże może być wykorzystywany do przepływu informacji w sposób który łamie politykę bezpieczeństwa.

"not intended for information transfer at all, such as the service program's effect on system load," ~ Lampson 1973.

Ukrywanie danych i komunikacji w LAN, TCP/IP, OSI, page faulty w dzielonym pliku, komunikacja morsem na egzaminie, selektor w zagadce o miśkach (?) itd.

Side-channels

Boczny kanał (*side channel*) to projektowo niezamierzony kanał, przez który wyciekają informacje nie z powodu teoretycznych słabości a fizycznych cech.

Obie definicje często na siebie nachodzą. Side-channels zazwyczaj używane w odniesieniu do ataków na urządzenia kryptograficzne a covert channels w odniesieniu do większych systemów.

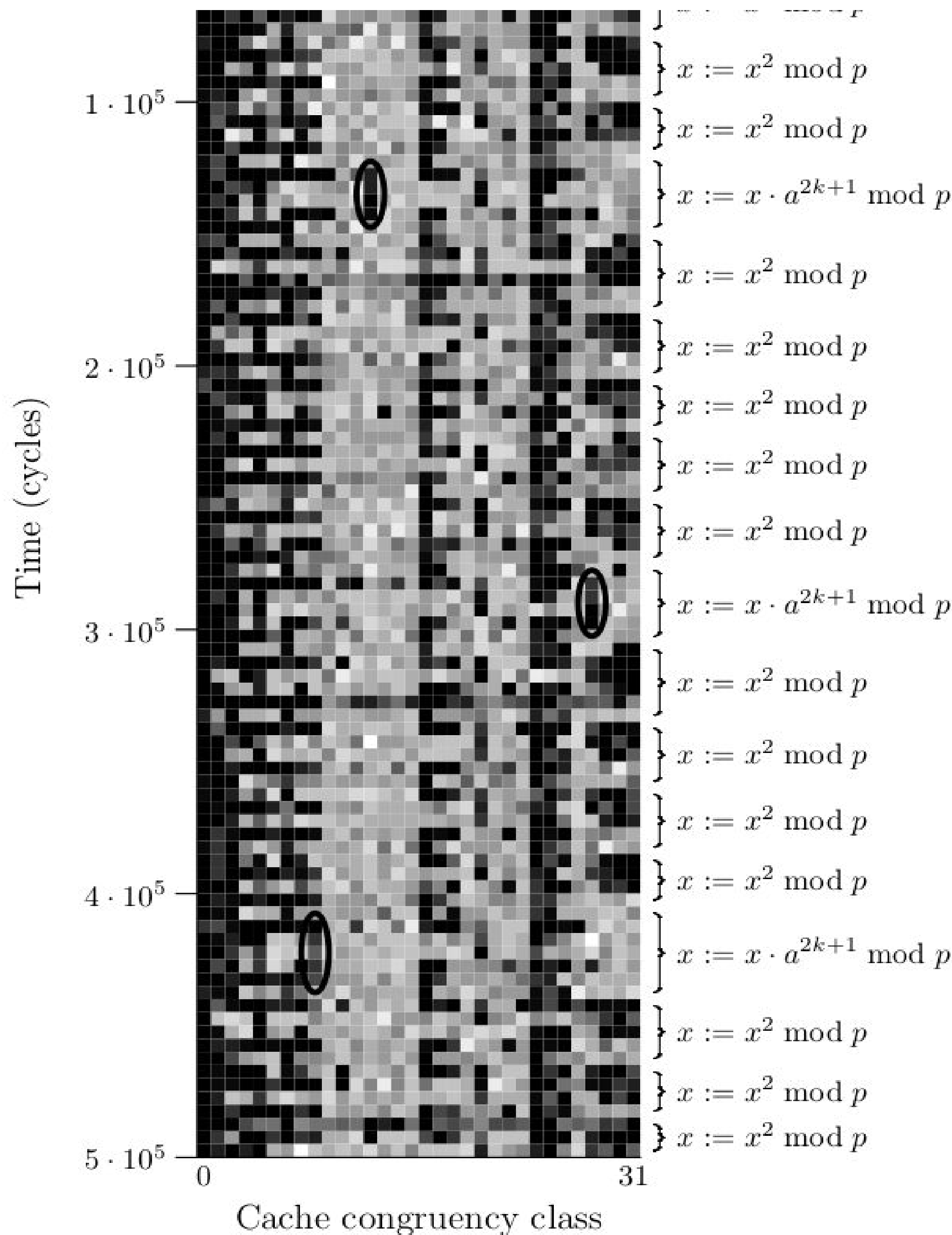
Analiza czasowa, zużycia prądu, akustyczna itd.

Można wykraść klucz prywatny nasłuchując dźwięków maszyny podczas deszyfrowania! Używając mikrofonu w telefonie! Oddalonego o 4m!
[ODKRYLI JEDEN PROSTY SPOSÓB]

Paul Kocher - specjalista od kryptograficznych side-channel.

Cache missing for fun and profit ~ Colin Percival

- Liczenie RSA można rozbić na ciąg 2 różnych typów operacji:
  - podniesień do kwadratu $x := x^2 \mod p$
  - mnożeń $x := x \cdot a^{2k+1} \mod p$ dla policzonych **wcześniej** $\{a, a^3, a^5 \dots a^{31}\} \mod p$.
- BN_sqr troszkę szybsze niż BN_mul, ale używa innej tymczasowej przestrzeni do wykonywania algorytmu Karatsuby, więc zostawia inny footprint w cache.
- Daje to nam około 200 z 512 bitów.
- Resztę bitów można uzyskać, odkrywając jakie są adresy mnożników (które są policzone wcześniej) z drugiego typu operacji poprzez ich footprint w cache.
- Średnio dostajemy dodatkowe 110 bitów. (mnożniki w jednej linii cache bądź inne zakłócenia)
- Znamy ~310 bitów z 512 bitowego RSA
- PROFIT



Ślad w cache

Cieniowanie kwadratów odpowiada ilości cykli potrzebnych do dostępu do wszystkich linii w zbiorze cache. Od 120 do 170.

Zaznaczone obszary przenoszą informacje o czynnikach a^{2k+1}.

Covert channel oparty na cache

- Dostępy do pamięci mają wpływ na stan cache.
- Stan cache ma wpływ na czas dostępów do pamięci.
- Mierząc czasy dostępów do pamięci odkrywamy informację o dostępach do pamięci.
    - FLUSH + RELOAD, EVICT + TIME, PRIME + PROBE
- Zazwyczaj używane jako side-channel (nie jest to nowość).
- Istnieją inne covert channels.

Out of Order, spekulacje, przewidywanie skoków

- Instrukcje mogą być wykonywane mniej bądź bardziej równolegle, oraz w innej kolejności niż wynikałoby to z programu.
- Skoki są przewidywane zanim cel skoku jest znany.

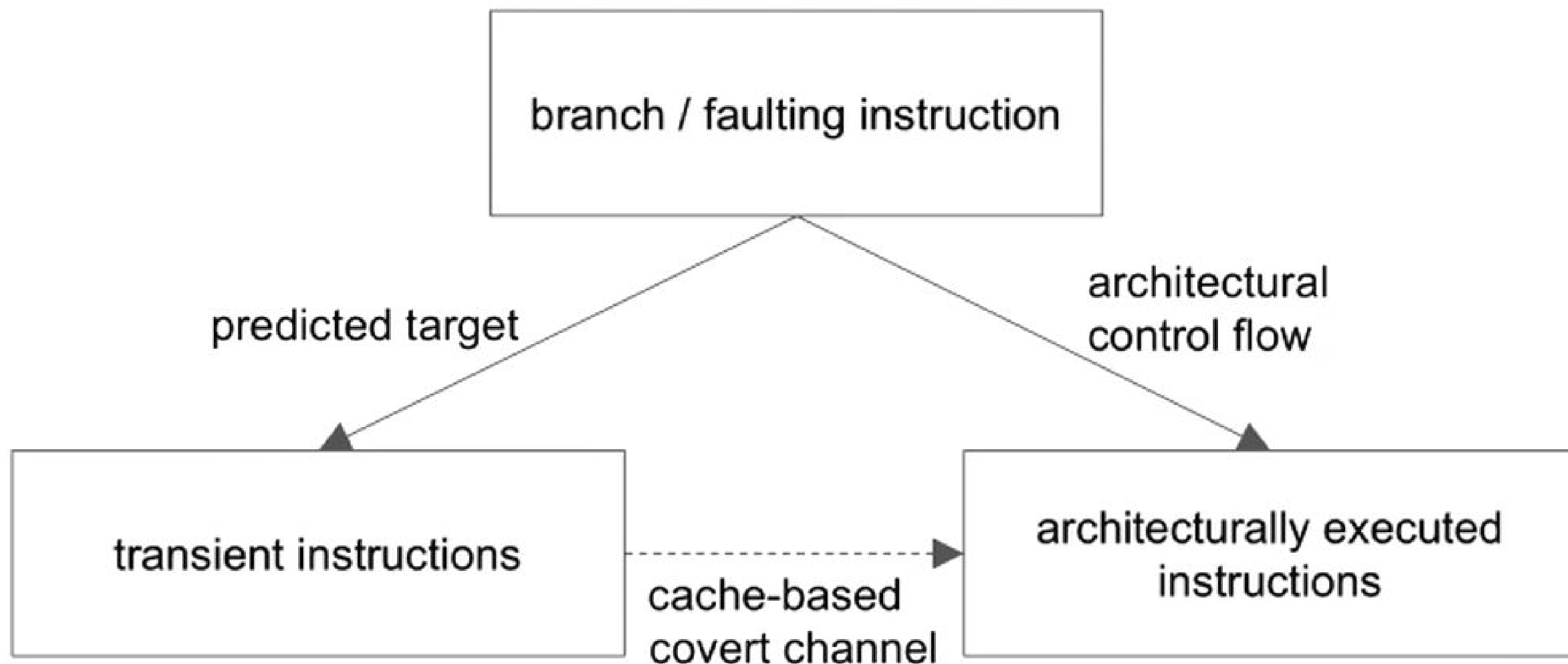
```
1 if (foo_array[index1] ^ foo_array[index2] == 0) {
2       result = bar_array[100];
3 } else {
4       result = bar_array[200];
5 }
```

Błędne przewidywanie (*misprediction*)

- wyjątki i źle przewidziane skoki powodują unieważnienie **chwilowych** (*ang. transient*) instrukcji.
- stare wartości rejestrów są zachowywane, mogą być odtworzone (przemianowanie rejestrów)
- zapisy do pamięci są buforowane, mogą być odrzucone
- **zmiany w cache nie są cofane -** stan cache nie jest bezpośrednio widoczny dla programisty; jest to stan na poziomie mikroarchitektury

Ukryty kanał poprzez błędne przewidywanie

- chwilowe instrukcje pozwalają na przesyłanie danych przez ukryty kanał bazowany na cache



3 warianty 2 ataków

Spectre		Meltdown
• CVE-2017-5753 • "Variant 1" • "Bounds Check Bypass" • Primarily affects interpreters/JITs	• CVE-2017-5715 • "Variant 2" • "Branch Target Injection" • Primarily affects kernels/hypervisors	• CVE-2017-5754 • "Variant 3" • "Rogue Data Cache Load" • Affects kernels (and architecturally equivalent software)

Wariant 1 - bounds check bypass

```
if (x < array1_size)
    y = array2[array1[x] * 256];
```

- Wykonanie powyższego kodu bez spekulacji jest bezpieczne - nigdy nie przeczytamy z array1 poza zakresem.
- Aby wykorzystać spekulatywne wykonanie, atakujący musi zapewnić odpowiednie warunki:
  - odpowiednio wytrenować predyktor skoków
  - sprawić, żeby array1_size i tablica array2 nie były w cache
- Atakujący podaje x spoza zakresu tak aby array1[x] było sekretną wartością
  - Taki dostęp zmienia stan cache w sposób zależny od array1[x].
  - Po sprawdzeniu wartości array1_size instrukcje są unieważniane i stan jest przywracany, ale nie stan na poziomie mikroarchitektury!
- Atakujący wykrywa zmianę w cache (FLUSH + RELOAD, EVICT + RELOAD, PRIME + PROBE)
  - Dostęp do array2[k*256] będzie szybki gdy k=array1[x] ponieważ wartość ta znajduje się w cache.

- Jeśli atakujący ma dostęp do array2 to mierzy czas dostępów do array[i*256] dla każdego i w przedziale [0...255]
- Jeśli nie ma dostępu to może wykorzystać technikę PRIME + PROBE
  - atakujący wypełnia cały cache swoimi danymi. Uruchamia kod ofiary. Sprawdza czasy dostępów do cache (jedna linia najprawdopodobniej została wymieniona).
- Atakujący może również chwilę po złośliwie podanym x wywołać kod ofiary z x’ w zakresie i zmierzyć czas. Jeśli array1[x’] = k to wartość w array2 była w cache i dostęp będzie szybszy niż gdy array1[x’] != k
- Istnieją również inne metody na wywnioskowanie k.

`arr1->length`, `arr2->data[0x200]` i `arr2->data[0x300]` nie są w cache, pozostałe dane są, predyktor skoków przewiduje warunek jako spełniony, procesor wykonuje następujące instrukcje zanim `arr1->length` zostanie przeczytane

- odczyt `arr1->data[untrusted_offset_from_caller]`
- zaczynając odczyt `arr2->data`, ładując odpowiadającą linię cache

```
struct array *arr1 = ...; /* small array */
struct array *arr2 = ...; /* array of size 0x400 */
/* >0x400 (OUT OF BOUNDS!) */
unsigned long untrusted_offset_from_caller = ...;
if (untrusted_offset_from_caller < arr1->length) {
 unsigned char value = arr1->data[untrusted_offset_from_caller];
 unsigned long index2 = ((value&1)*0x100)+0x200;
 if (index2 < arr2->length) {
   unsigned char value2 = arr2->data[index2];
 }
}
```

Mierząc czas potrzebny na załadowanie arr2->data[0x200] i arr2->data[0x300], atakujący może wywnioskować czy wartość index2 podczas spekulatywnego wykonania była 0x200 czy 0x300 - co daje informację o tym czy arr1->data[untrusted_offset_from_caller]&1 wynosi 0 lub 1.

eBPF JIT/interpreter

Może by tak ułatwić sobie życie i wykorzystać samodzielnie napisany kod, który będzie działał w ograniczonej dla niego przestrzeni (najlepiej w przestrzeni jądra)?

W oczywisty sposób podatne są więc JIT!

Jądro Linuxa wspiera eBPF. Nieuprzywilejowany kod użytkownika może wysyłać kod bajtowy do jądra, który jest weryfikowany przez jądro oraz:

- albo interpretowany przez interpreter kodu bajtowego w jądrze
- lub kompilowany do natywnego kodu maszynowego i uruchamiany w kontekście jądra używając silnika JIT.

Skutek:

Można czytać pamięć jądra mając uprawnienia zwykłego użytkownika!

Czy można wykonać atak w JS? Można!

JavaScript działa w piaskownicy (*sandbox)*

- Nie może czytać dowolnej pamięci
- Brak wskaźników, dostępy do tablic są sprawdzane pod względem zakresów

Oszukamy trochę silnik JS żeby wygenerował nam podatny kod maszynowy.

w zakresie przy trenowaniu predyktora, poza zakresem podczas ataku

sprawdzenie zakresu, żeby JIT tego nie robił. Podczas ataku nie będzie w cache.

trik optymalizacyjny (wynik jest int)

```
1 if (index < simpleByteArray.length) {
2   index = simpleByteArray[index | 0];
3   index = (((index * TABLE1_STRIDE)|0) & (TABLE1_BYTES-1))|0;
4   localJunk ^= probeTable[index|0]|0;
5 }
```

żeby JIT nie wyoptymalizował dostępu do probeTable

4096

żeby JIT nie dodał sprawdzeń zakresów, dodajemy je sami

W JS nie ma clflush więc cache opróżniamy czytając adresy odległe o 4096 w jakiejś dużej tablicy. Pozwoliło to atakującym na usunięcie z cache adresów o tych samych bitach 11-6.

Sekretny bajt odczytujemy wykonując dostępy do probeTable[n*4096] dla n w przedziale [0..255].

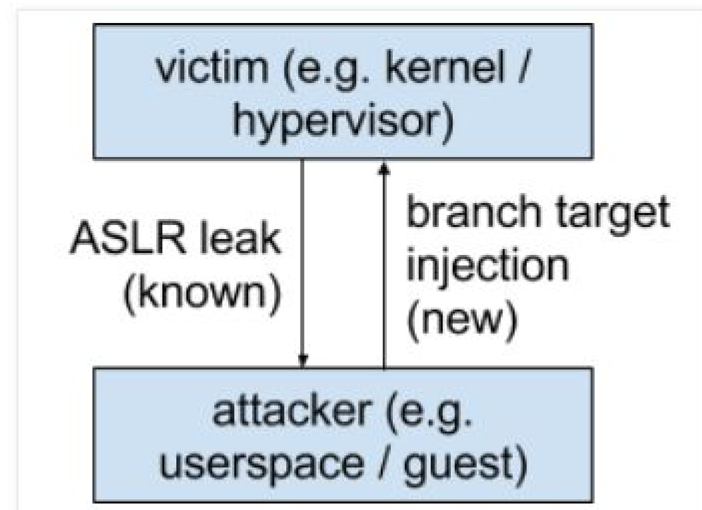
Niektóre silniki JS specjalnie zaburzają dokładność timera, ale HTML5 Web Worker pozwala na stworzenie osobnego wątku który dekrementuje zmienną w dzielonej pamięci, co pozwala osiągnąć wystarczającą dokładność pomiarów.

```
1 cmpl r15,[rbp-0xe0]          ; Compare index (r15) against simpleByteArray.length
2 jnc 0x24dd099bb870           ; If index >= length, branch to instruction after movq below
3 REX.W leaq rsi,[r12+rdx*1]   ; Set rsi=r12+rdx=addr of first byte in simpleByteArray
4 movzxbl rsi,[rsi+r15*1]      ; Read byte from address rsi+r15 (= base address+index)
5 shll rsi, 12                 ; Multiply rsi by 4096 by shifting left 12 bits}\%\
6 andl rsi,0x1ffffff           ; AND reassures JIT that next operation is in-bounds
7 movzxbl rsi,[rsi+r8*1]       ; Read from probeTable
8 xorl rsi,rdi                 ; XOR the read result onto localJunk
9 REX.W movq rdi,rsi           ; Copy localJunk into rdi
```

Listing 3: Disassembly of Speculative Execution in JavaScript Example (Listing 2).

Wariant 2 - Branch Target Injection

Wykonanie instrukcji skoku z adresem w rejestrze może być opóźnione ze względu na pobranie adresu skoku z pamięci.

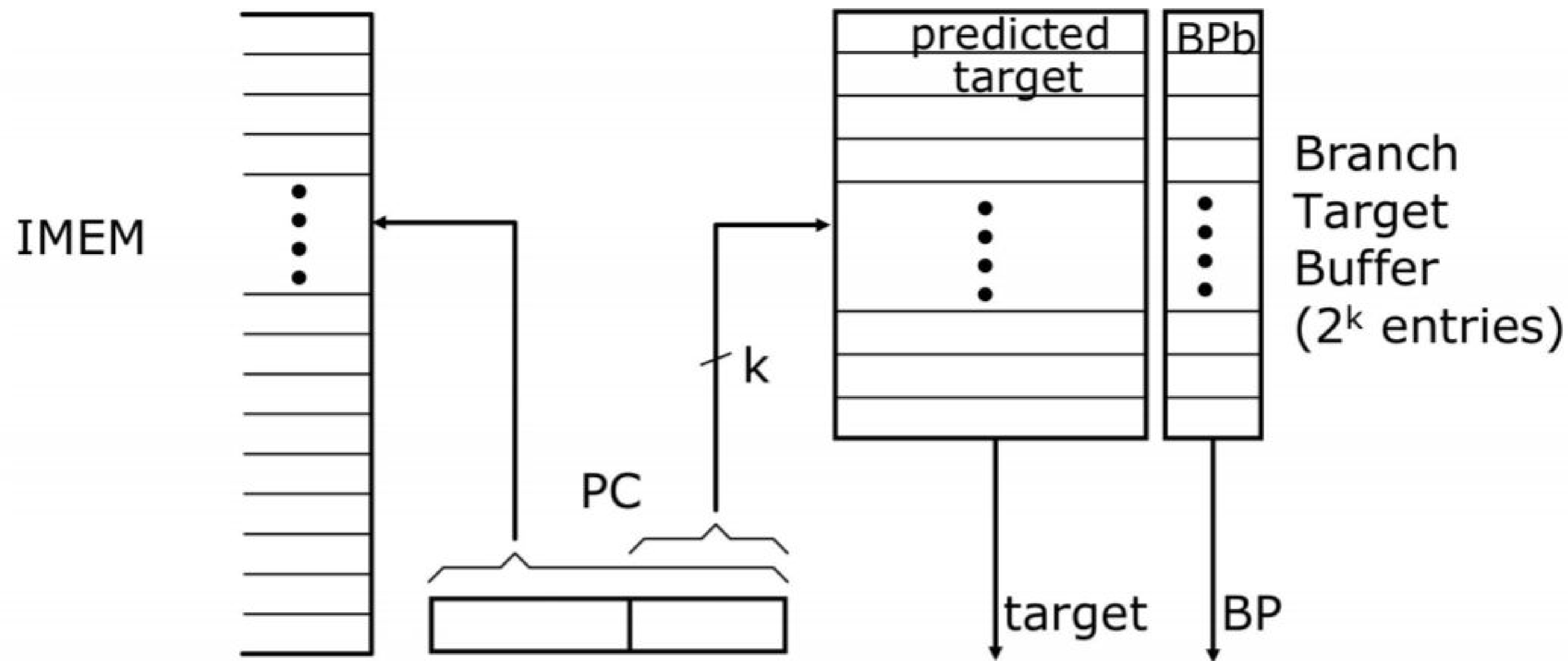


W wyniku tego procesor wykona instrukcje spekulatywnie w oparciu o stan predyktora skoków.

Możliwe jest więc odpowiednie wytrenowanie predyktora skoków aby procesor wykonał tymczasowe instrukcje pod podanym przez atakującego adresem. Jeśli te instrukcje mogą zmienić stan cache to atak jest możliwy do przeprowadzenia.

Jump over ASLR: Attacking branch predictors to bypass ASLR

Branch Target Buffer



Na różnych procesorach predyktory skoków mogą wyglądać różnie. Nie wiadomo dokładnie jak, ale można eksperymentalnie sporo wywnioskować (po ilu bitach indeksują, ile ostatnich skoków przechowują itd.). Aliasing.

Predyktory skoków zazwyczaj nie przetrzymują żadnych informacji ani nie wykonują sprawdzeń w celach zapewnienia bezpieczeństwa.

Rozważmy przypadek gdy atakujący kontroluje rejestry R1 i R2 podczas wykonania instrukcji skoku z adresem w rejestrze. Istnieje sporo funkcji które operują na zewnętrznych danych i zachowują rejestry (czasem ich nawet nie używają, zapisują je na stosie). Atakujący może je wykorzystać.

Jeśli CPU pozwala wykonywać tylko instrukcje wykonywalne przez ofiarę (np. nadzorca nie może wykonywać kodu gościa), to atakujący musi przekierować przepływ kontroli przez istniejące już kawałki kodu nazywane *gadgets* (gadżety). Jeśli ciąg gadżetów pozwala modyfikować stan cache to atak jest możliwy do wykonania.

Przykładowy gadget może wykonywać operację na R1 (add, xor, sub) i zapisywać w R2 a potem wykonywać dostęp do adresu w R2.

Ataki oparte o ciąg gadżetów znane są jako Return Oriented Programming, jednak tutaj gadgety nie zwracają danych explicite, a poprzez cache covert channel.

W oknie spekulacyjnym można wykonać ~200 instrukcji włącznie ze skokami.

Wyskakiwanie z maszyn wirtualnych

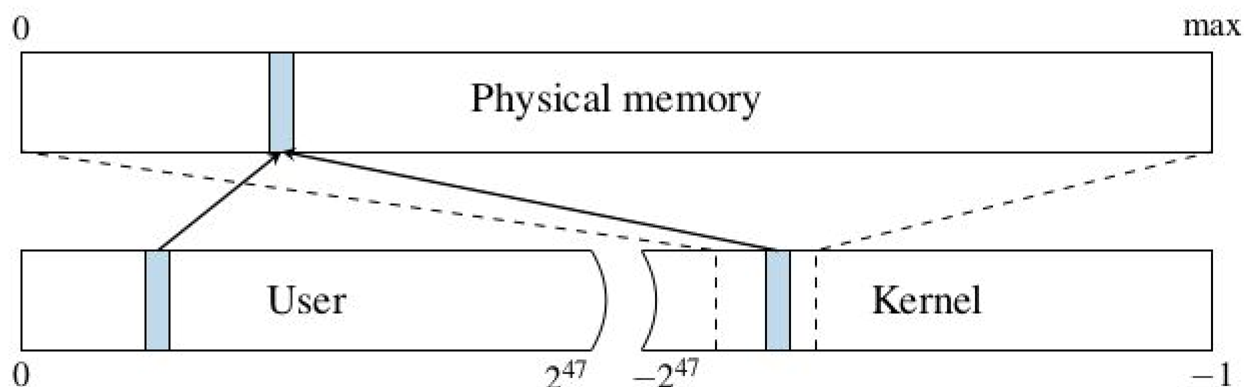
- łamiemy ASLR nadzorcy wykorzystując predyktor skoków
- powodujemy złe przewidzenie pierwszego skoku z adresem w rejestrze po zmianie kontekstu na kontekst nadzorcy
- nie chcemy adresu skoku w cache ~200 cykli na pobranie adresu
- stan rejestrów gościa jest widoczny w kontekście nadzorcy
- gość i nadzorca nie dzielą przestrzeni adresowej ale nadzorca ma zmapowane strony gościa
- można wykorzystać interpreter eBPF; wywołanie przez odpowiedni gadget

```
ffffffff81514edd: mov    rsi,r9
ffffffff81514ee0: call   QWORD PTR [r8+0xb0]
```

```
static unsigned int __bpf_prog_run(void *ctx, const struct bpf_insn *insn)
```

Wariant 3 - Meltdown

- CPU przechowuje w rejestrze sprzętowym adres wierzchołka wielopoziomowej tablicy stron.
- Przy zmianie kontekstu rejestr ten jest podmieniany.
- Każda przestrzeń adresowa ma zmapowane strony jądra które są dostępne jedynie w trybie uprzywilejowanym CPU
- Wymuszane jest to przez bity uprawnień w tabeli stron
- pamięć fizyczna jest bezpośrednio zmapowana w jądrze pod danym adresem wirtualnym
- KASLR przy każdym uruchomieniu ładuje jądro pod innymi adresami ale KASLR można łatwo obejść.



µOperacje

W nowoczesnych procesorach instrukcje są rozbijane na wiele instrukcji zwanych µoperacjami, które przypominają instrukcje RISC.

Instrukcja jest zatwierdzana bądź unieważniana w momencie gdy wszystkie jej µoperacje zostały wykonane (precyzyjność wyjątków co do instrukcji a nie co do µoperacji). Wtedy również podnoszony jest ewentualny wyjątek.

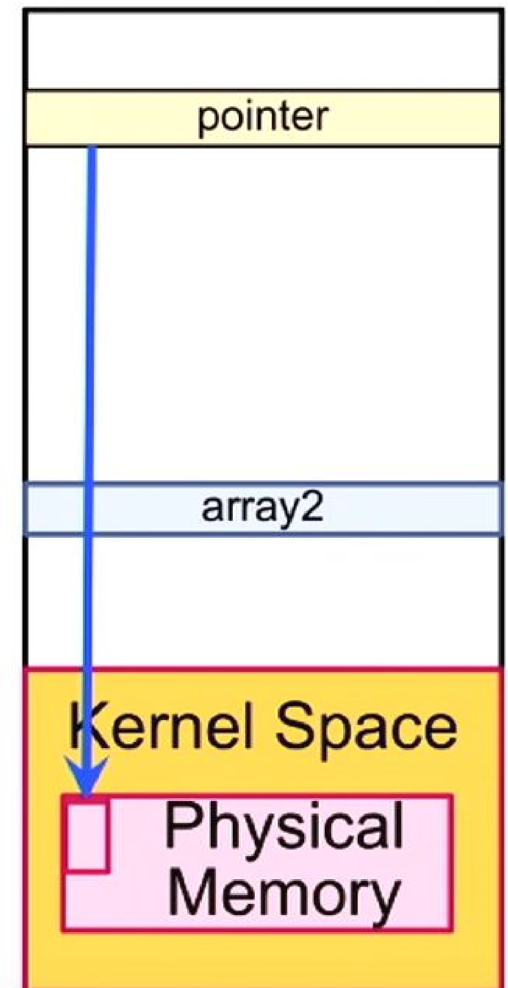
µoperacje instrukcji podnoszącej wyjątek mogą działać w przeplocie z µoperacjami innych instrukcji. Co jeśli inne µoperacje były zależne od µoperacji instrukcji podnoszącej wyjątek, w szczególności od µoperacji czytającej niedozwoloną pamięć?

Takie zależne µoperacje mogą należeć do instrukcji która w kodzie programu występuje później. Jednak µoperacje opuszczają bufor przestawiania tak szybko jak mają gotowe operandy. Istnieje zatem możliwość, że µoperacje instrukcji następujących po instrukcji podnoszącej wyjątek zdążą się wykonać (np. zmodyfikować cache), zanim poprzedzające je instrukcje zdążą być unieważnione!

Po wykonaniu dostępu mającego pobrać żądany sekretny bajt będziemy wykonywać 256 dostępów w takich odległościach aby każdy należał do innej linii cache.

```
i = *pointer;
y = i * 256;
z = array2[y];
```

Cache



Dostęp do pamięci jądra generuje błąd strony.

Stan cache się zmienia.

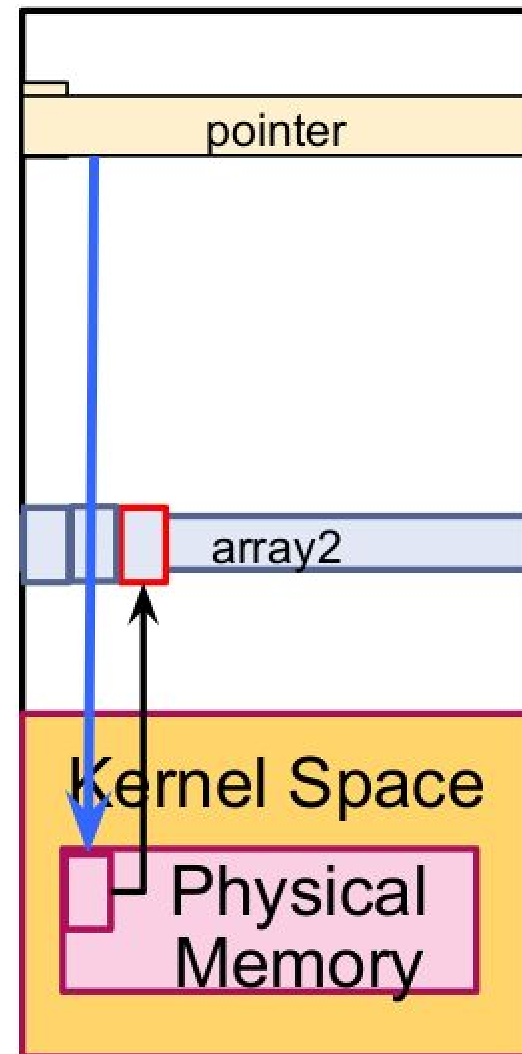
data == 3

architectural
attack code

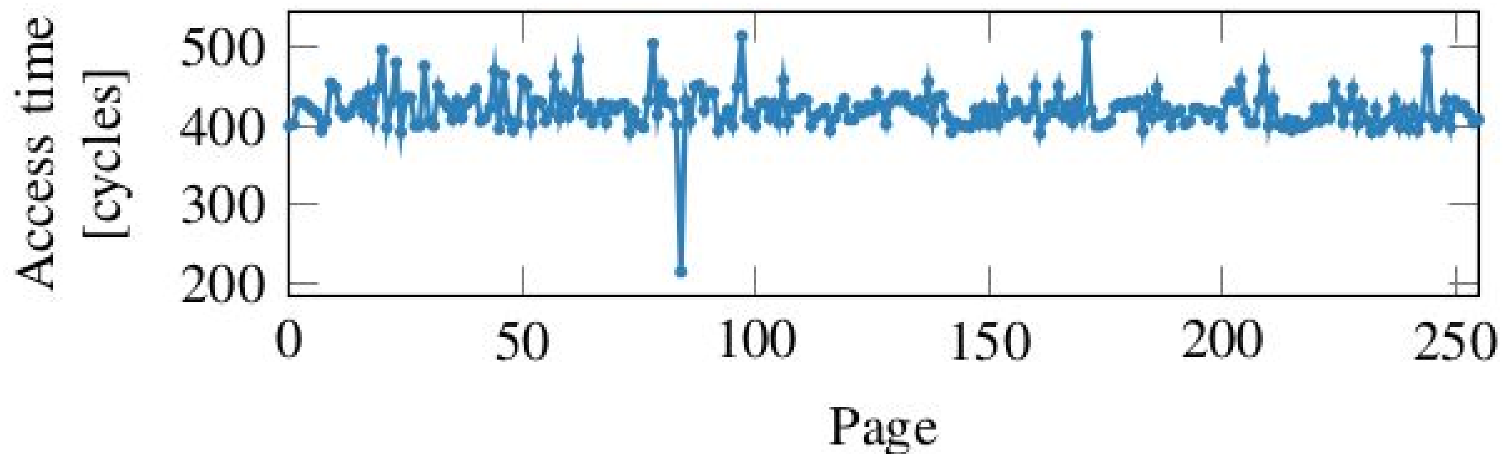
```
i = *pointer;
y = i * 256;
z = array2[y];
```

Cache

FLUSH + RELOAD do odzyskania informacji.



Wyjątek można złapać (C++), obsłużyć bądź zignorować sygnał (C) , użyć mechanizmu transakcyjnej pamięci (TSX), wykonywać dostęp w procesie dziecku, który może się wywalić, wykonać dereferencje wskaźnika z jądra jako tymczasową instrukcję w źle przewidzianym skoku itd.



- Wyścig między sprawdzeniem uprawnień a dostępem do pamięci.
- Wyjątki są obsługiwane podczas unieważniania / zatwierdzania instrukcji w buforze przestawiania.

- Sprawdzanie uprawnień bazuje na PTE
- Sprawdzenia uprawnień mogą być wykonywane asynchronicznie
- Zależne instrukcje mogą się wykonać na poziomie mikroarchitektury, zanim zostaną unieważnione w buforze przestawiania!
- Łatwy atak: wyciek keszowanych danych z jądra
- TU Graz: Dane niebędące w cache również mogą być wykradzione.

Poprawki wariantu 1

Poprawki wymagają przekompilowania istniejącego kodu, żeby nie generować podatnych wzorców. Przekompilowania najbardziej wymagają systemy operacyjne oraz aplikacje wykonujące kod z niepewnego źródła.

LLVM + Clang , GCC wprowadzają funkcję __builtin_load_no_speculate()

ARM również wypuszcza podobną poprawkę jako C header.

Intel zaleca w używać instrukcji lfence, co powinno zapobiec spekulacji za tą instrukcją. Skoro wrażliwe są określone schematy kodu, to można je wykryć podczas statycznej analizy i powstawiać instrukcje lfence.

Poprawki wariantu 2

Intel wraz z mikrokodem wprowadza mechanizmy:

- Indirect Branch Restricted Speculation (IBRS): Ogranicza spekulacje skoków z adresem w rejestrze.
- Single Thread Indirect Branch Predictors (STIBP): Izoluje predyktor skoków pomiędzy wątkami (Hyperthread).
- Indirect Branch Predictor Barrier (IBPB): Zapewnia, że wcześniejszy kod nie wpływa na predykcje późniejszych skoku z adresem danym w rejestrze.

Nie chwali się jak działają (jeśli działają).

Google proponuje retpoline. Imlementacja w LLVM oraz GCC.

Retpolines

```
class Base {
 public:
  virtual void Foo() = 0;
};

class Derived : public Base {
 public:
  void Foo() override { … }
};

Base* obj = new Derived;
obj->Foo();
```

Skoki warunkowe z adresem w rejestrze występują w wielu przypadkach gdy adres skoku należy wyliczyć bądź wybrać podczas pracy programu (kod polimorficzny, tablice skoków, powroty z funkcji).

Nie możemy łatwo programowo zatrzymać spekulacji, ale co gdybyśmy moglą ją kontrolować w pewien sposób?

Adres powrotu z funkcji (return) może być keszowany.
Return Stack Buffer (RSB) - Intel, Return Address Stack (RAS) - AMD , Return Stack - ARM.

Ta wartość może być inna niż adres odłożony na stosie przez call.

Jump z adresem w rejestrze

```
jmp *%r11  |   call set_up_target;    (1)
           |  capture_spec:           (4)
           |    pause;
           |    jmp capture_spec;
           |  set_up_target:
           |    mov %r11, (%rsp);     (2)
           |    ret;                  (3)
```

1. call do set_up_target; adres znany podczas kompilacji więc brak spekulacji. Generuje to inne adresy w RSB i na stosie przy powrocie do capture_spec.
2. Zmieniamy adres odłożony na stosie przez (1), na %r11. RSB ciągle pokazuje na capture_spec.
3. Wracamy z call
   - Wykonanie spekulatywne używa RSB wygenerowanego przez (1), i kręci się w pętli (4). Ta ścieżka jest wykonywana tylko spekulatywnie.
   - Our return is ultimately retired, the on-stack value is used to locate the actual new instruction pointer and the benign results of any speculative execution in the loop at (4) are discarded

Call z adresem w rejestrze

```
call *%r11    jmp set_up_return;
             inner_indirect_branch:
               call set_up_target; }
             capture_spec:         }
               pause;              }
               jmp capture_spec;   } Indirect branch
             set_up_target:        } sequence.
               mov %r11, (%rsp);   }
               ret;                }
             set_up_return:
               call inner_indirect_branch; (1)
```

Używamy dwóch call. W przeciwieństwie do skoku, call musi ostatecznie wrócić do następnej instrukcji.

Zewnętrzny call ustawia adres powrotu, a wewnętrzny call pełni rolę mechanizmu bezpieczeństwa. RSB i adres na stosie ustawiony przez (1) i są identyczne. Powoduje to, że dodatkowy call jest jedynym narzutem, który jest wymagany aby powrócić poprawnie do wykonania kodu.

KPTI - poprawka wariantu 3

KPTI powoduje, że każdy proces ma 2 tabele stron. Normalną, ze zmapowanym jądrem i drugą z minimalną ilością stron jądra. Druga tabela stron jest używana gdy proces wykonuje się w przestrzeni użytkownika.

Każda zmiana sprzętowego rejestru wskazującego na wierzchołek tablicy stron wymaga wyczyszczenia TLB -> syscall czyści TLB!

Można wykorzystać PCID process context identifier. Wpisy TLB tagowane PCID danej przestrzeni adresowej -> możliwe dwa wpisy do tego samego adresu -> nie trzeba czyścić całego TLB.

Gdy proces wykonuje się w przestrzeni jądra to przestrzeń użytkownika jest niewykonywalna (SMEP).

Inne procesory

AMD niepodatne na wariant 3, podatne na 1 i 2 który jest ciężko wykonać z powodu różnic w mikroarchitekturze między Intelem.

ARM wypuścił tabelkę podatności. Na meltdown podatny tylko 1 chip.

Procesory Qualcomm podatne - brak szczegółów.

IBM nie potwierdza ani nie zaprzecza tego, że procesory z rodziny POWER są podatne, ale mówią o “potential impact” i wypuszczają poprawki.

MIPS nie powinien być podatny na meltdown.

Raspberry Pi używają takich procesorów ARM które nie są podatne na żaden wariant.

Urządzenia Apple, CISCO, chmury Google, Amazon i Microsoft, oraz wieeele innych.

Czy wydane poprawki naprawdę coś poprawiają?

Wydane poprawki to jedynie doraźne “naprawienie” błędów. Błędy te powinny być naprawione na poziomie mikroarchitektury.

Modyfikacja mikroarchitektury może zająć trochę czasu...

- rollback cache?
- ASIDy ?
- mechanizm bezpieczeństwa w BTB?
- Mniejsze współdzielenie komponentów sprzętowych?

Nie warto rezygnować ze spekulacji. Byłby to powrót do epoki procesora łupanego.

Źródła

1. arXiv:1801.01203 [cs.CR]
2. arXiv:1801.01207 [cs.CR]
3. github.com/IAIK/meltdown
4. CERT
5. youtu.be/6O8LTwVfTVs Spectre and Meltdown: Data leaks during speculative execution | J. Horn (Google Project Zero)
6. googleprojectzero.blogspot.com
7. Intel Analysis of Speculative Side Channel
8. POCs
9. http://lkml.iu.edu/hypermail/linux/kernel/1801.2/04628.html
10. Negative Result: Reading Kernel Memory From User Mode