# Meltdown

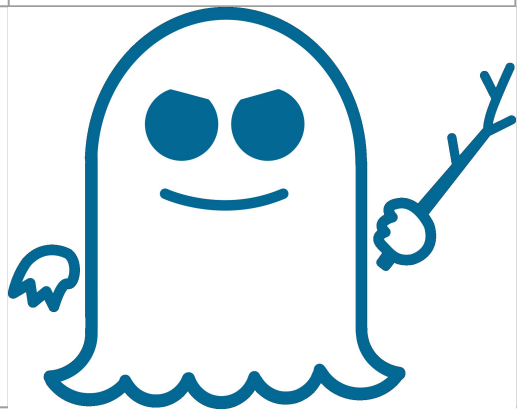
Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg

# Wstęp

Meltdown i Spectre to rodzina zbliżonych do siebie ataków wykorzystujących mechanizmy spekulatywnego wykonania kodu obecne w nowoczesnych procesorach. Dzięki tym mechanizmom możliwy jest wyciek wrażliwych danych z pamięci RAM komputera.

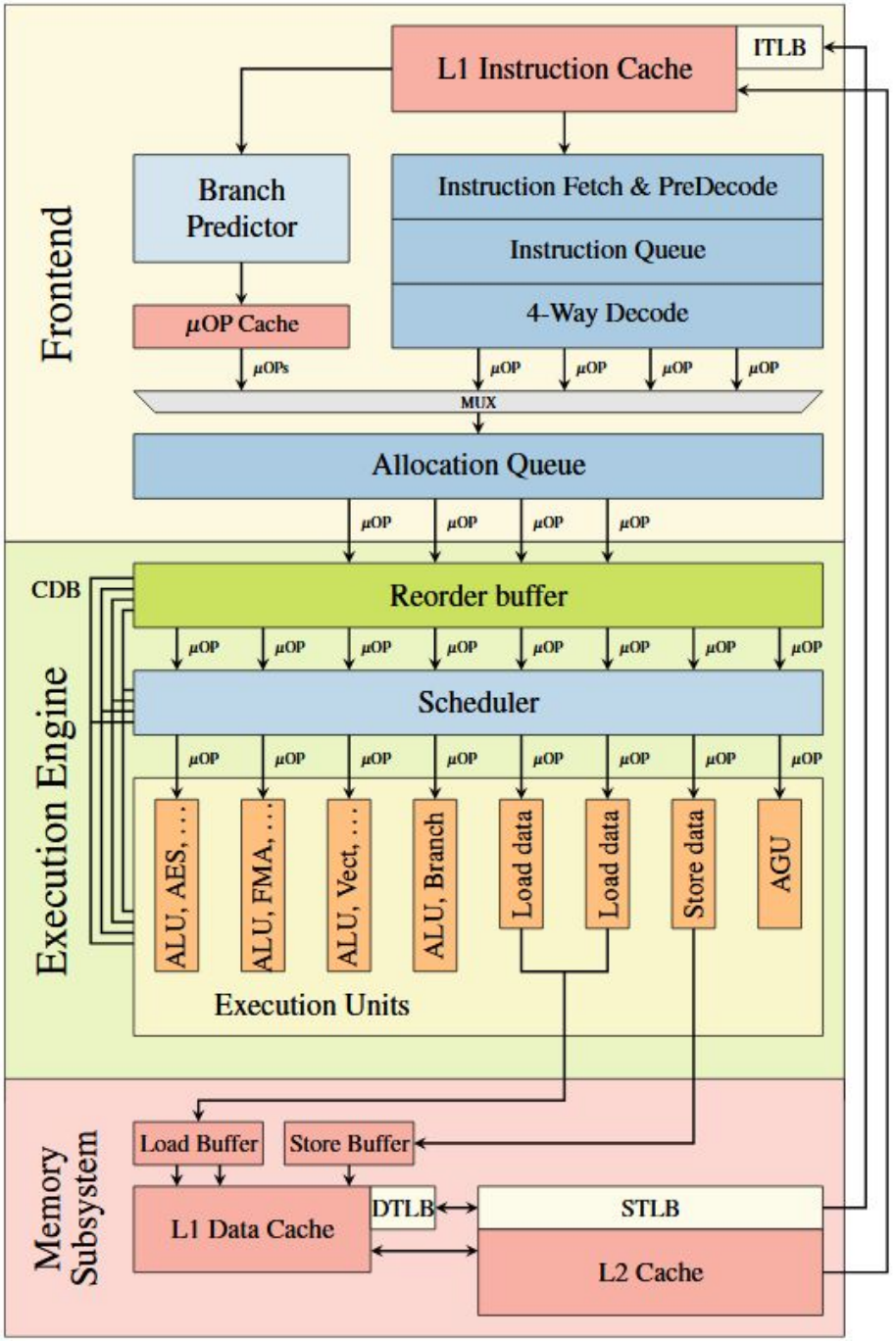
Prace opisujące te ataki zostały opublikowane na przełomie 2017 i 2018 roku. Składy zespołów pracujących nad tymi pracami w dużej mierze pokrywają się.

# Porównanie Meltdown i Spectre

| | Meltdown | Spectre |
|---|---|---|
| Podatne procesory | Intel, od 2010 roku | Intel, AMD, ARM |
| Szybkość wycieku | 503 KB/s | 10 KB/s |
| Błąd przy odczytywaniu danych | 0,02% - 0,03% | NA |
| Obrazek |  |  |

# Jak działa procesor

- Instrukcje są dekodowane
- Poszczególne instrukcje są zamieniane na **mikrooperacje**
- Mikrooperacje są wstawiane do kolejki
- Kolejność mikrooperacji jest **optymalizowana** z zachowaniem zależności
- Scheduler wrzuca instrukcje do poszczególnych jednostek wykonawczych
- Kiedy **wszystkie** mikroinstrukcje składające się na polecenie asemblera się wykonają, ich efekt jest aplikowany do stanu procesora


Frontend
L1 Instruction Cache
ITLB
Branch Predictor
Instruction Fetch & PreDecode
Instruction Queue
4-Way Decode
µOP Cache
µOPs
µOP
µOP
µOP
µOP
MUX
Allocation Queue
µOP
µOP
µOP
µOP
Execution Engine
CDB
Reorder buffer
Scheduler
ALU, AES, ...
ALU, FMA, ...
ALU, Vect, ...
ALU, Branch
Load data
Load data
Store data
AGU
Execution Units
Memory Subsystem
Load Buffer
Store Buffer
L1 Data Cache
DTLB
STLB
L2 Cache

# Przykład

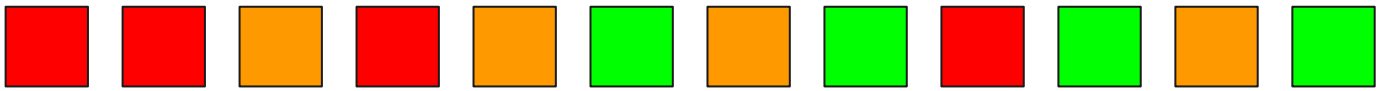
Ciąg instrukcji

MOV RCX, [adres]

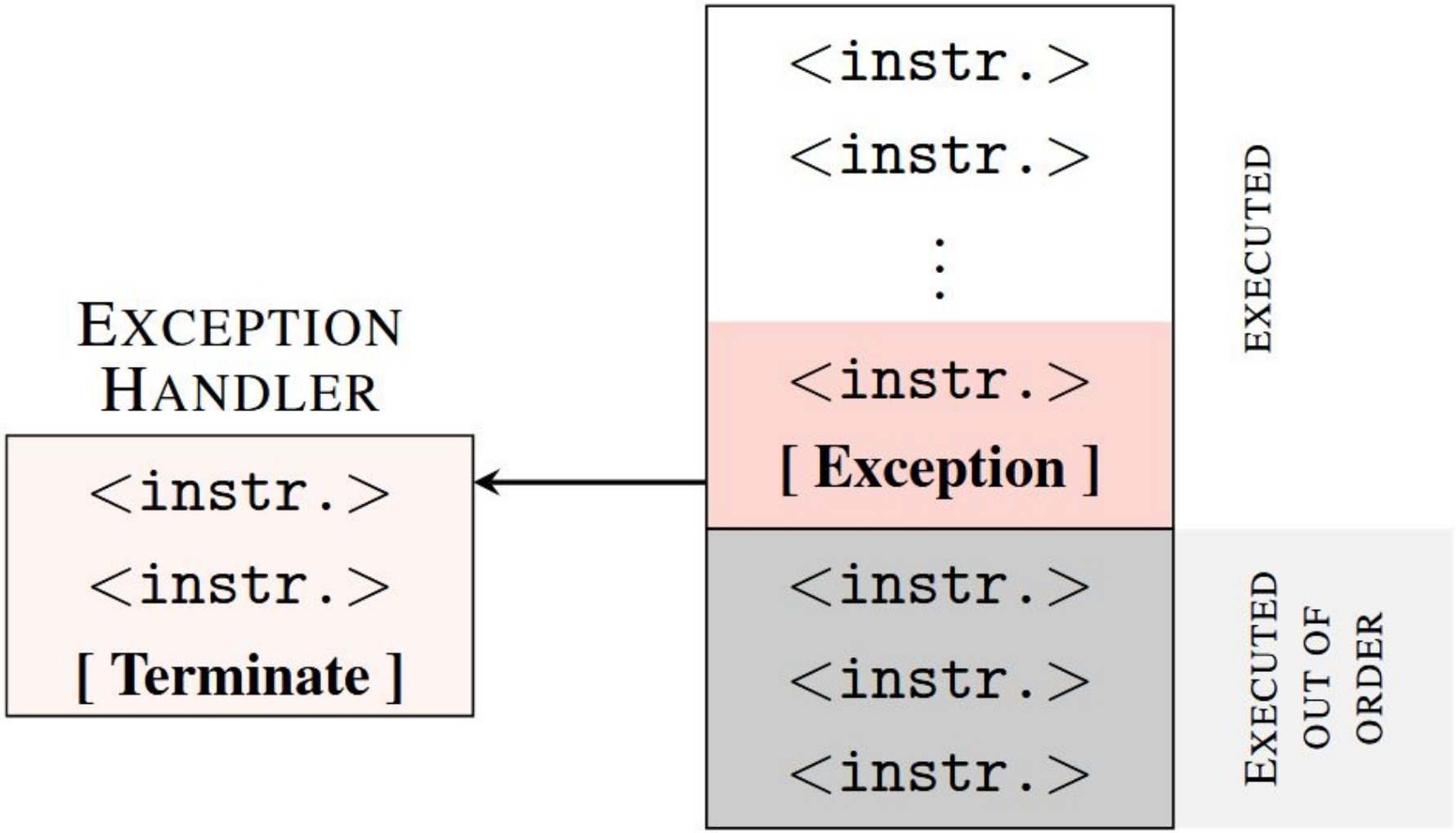
ADD RCX, 123

MOV RAX, [RCX]

może być zdekodowany i wykonany w taki sposób:



Jeżeli pierwszy MOV odwołuje się do niedostępnej lokacji pamięci, to przerwanie zostanie wywołane dopiero po wykonaniu **ostatniej** mikroinstrukcji. Efekty kolejnych instrukcji zostaną anulowane.


EXCEPTION HANDLER
<instr.>
<instr.>
[ Terminate ]
<instr.>
<instr.>
:
<instr.>
[ Exception ]
<instr.>
<instr.>
<instr.>
EXECUTED
EXECUTED OUT OF ORDER

# Przewidywanie skoków

MOV AL, byte [RCX]

SHL RAX, 0xC

JMP [RAX]

**Branch Target Buffer** - na podstawie adresu instrukcji przewiduje miejsce skoku (może to zrobić nawet przed zdekodowaniem instrukcji!).

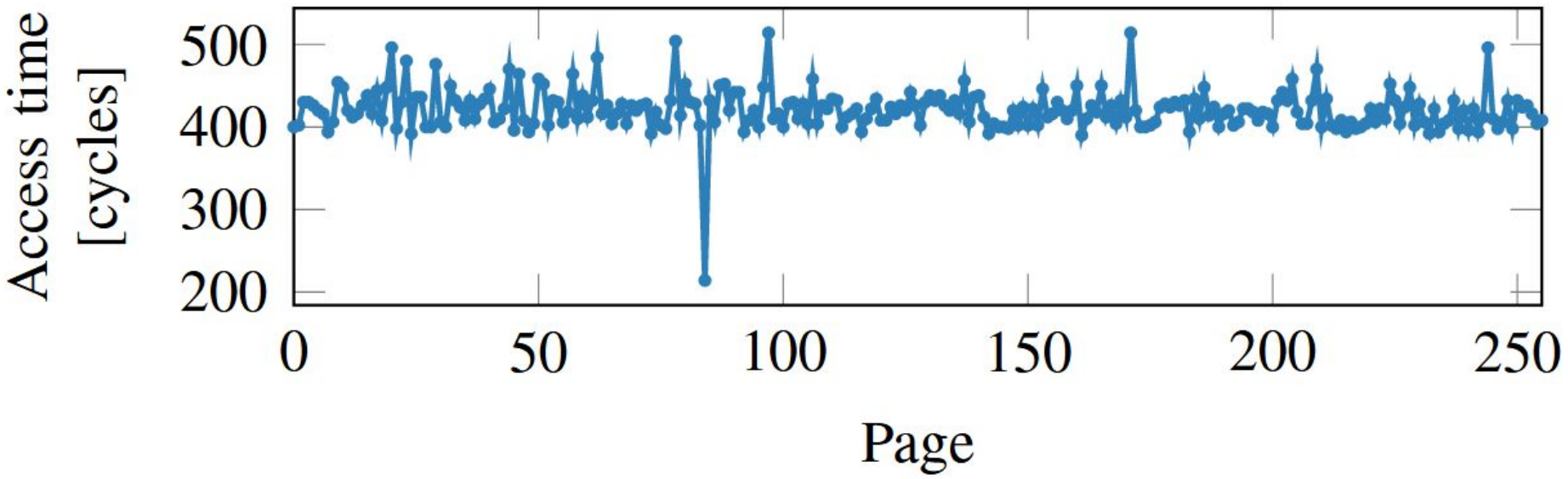
# Przesyłanie danych przez cache procesora

Założenia: mamy sobie dwa wątki A i B, które nie mogą się ze sobą komunikować (np. poprzez mechanizmy IPC). A i B mają wspólną tablicę stron, strony są w trybie tylko do odczytu. Na początku wykonuje się B, które może jednorazowo przekazać sterowanie do A (nie mamy mechanizmów synchronizujących).

Problem: A chce przesłać do B 1 bit informacji.

Rozwiązanie:

1. B czyści cache procesora instrukcją *clflush* i oddaje sterowanie do A
2. A odczytuje coś z adresu adres + 4096 * i, gdzie adres jest ustalony i znany wątkowi B, zaś i jest bitem informacji którą A chce przesłać.
3. B mierzy czas dostępu do wcześniej ustalonego adresu. Jeżeli i = 0, to ta strona pamięci była odczytana przez proces A i jej zawartość znajduje się w cache, więc czas dostępu jest stosunkowo krótki (ok. 200 cykli procesora). W p. p. czas dostępu wynosi 400-500 cykli.


Access time
[cycles]
500
400
300
200
0
50
100
150
200
250
Page

# Esencja dziury Meltdown

MOV RAX, [adres_kernela]

MOV ECX, [adres_w_userspace + RAX * 4096]

Zanim procesor wykona wszystkie mikrooperacje składające się na pierwszą instrukcję i rzuci wyjątek, prawdopodobnie wykona część drugiej instrukcji i zostawi ślad w cache. Po rzuceniu wyjątku procesor nie usunie tego śladu.

Taki ślad może potem zostać odczytany przez atakującego.

# Szablon ataku

1. Załaduj do rejestru adres pamięci wirtualnej, który chcesz odczytać (np. adres w kernelu)
2. Wyczyść cache procesora, np. instrukcją *clflush*
3. Zrób forka na procesy A i B
4. Proces A czeka na zakończenie procesu B
5. Proces B wykonuje ten kawałek kodu:

```
MOV AL, byte [RCX]
SHL RAX, 0xC                    ; RAX * 4096
MOV RBX, qword [baza + RAX]
```

6. Proces B zostaje ubity, ale z dużym prawdopodobieństwem w cache jest strona o adresie baza + RAX
7. Proces A iteruje po wszystkich stronach które mógł dotknąć proces B i mierzy czas dostępu, i odczytuje w ten sposób zawartość pamięci

# Czasem odczyt nie wychodzi

Czasem sprawdzenie uprawnień do strony następuje przed pobraniem informacji z pamięci. Wówczas w rejestrze zamiast wyniku znajduje się wartość 0.

Rozwiązujemy ten problem przez powtarzanie odczytu aż do uzyskania niezerowej wartości:

```
retry:
MOV AL, byte [RCX]
SHL RAX, 0xC
JZ retry
MOV RBX, qword [RBX + RAX]
```

Jeżeli w cache nie ma żadnej informacji, to uznajemy że przesłano wartość 0. Oczywiście to założenie może prowadzić do błędów.

Błąd powstały z tego powodu wynosi ok. 0,03%

# Przepustowość wycieku można zwiększyć

W powyższych przykładach korzystamy z dość wolnego wywołania fork, by zapobiec ubiciu procesu kradnącego dane. Zamiast tego można:

- dodać swój handler do  obsługi SEGFAULTa
- wykorzystać rozszerzenie procesora TSX (transakcyjna pamięć) do zamaskowania wyjątku (dostępne od linii procesorów Broadwell)
- wykorzystać branch predictor i strażnika w postaci instrukcji skoku warunkowego do zamaskowania wyjątku

# Skutki ataku

- Odczyt z dowolnego adresu podmapowanego w pamięci wirtualnej, niezależnie od uprawnień atakującego.
- Linux i Windows mapują przestrzeń adresową kernela do tablicy stron procesu.
- W przypadku kernela Linux, cała pamięć fizyczna jest domyślnie mapowana pod adresem 0xffff 8800 0000 0000. Przy włączonym KASLR wystarczy sprawdzić ok. 1024 lokacji w pamięci by odkryć adres bazowy mapowania.
- W systemie Windows znaczne fragmenty pamięci fizycznej są mapowane do tablicy stron kernela
- Niektóre kontenery np. Docker, LXC, OpenVZ współdzielą pamięć kernela (w Linuxie przy pomocy mechanizmu przestrzeni nazw).

Wniosek: atakujący ma dostęp do pamięci kernela, pamięci fizycznej, danych innych programów i kontenerów uruchomionych u dostawców chmurowych i współdzielonych hostingów.

# Testy

Sprzęt: procesory Intela Celeron G540, Core i5-3230M, Core i5-3320M, Core i7-4790, Core i5-6200U, Core i7-6600U, Core i7-6700K (środowisko laboratoryjne), Xeon E5-2676 v3, Xeon E5-2650 v4 (w chmurze).

Scenariusze testowe:

- Zrzut pamięci z Firefoxa 56 na Ubuntu 16.10. Odczytano m. in. zapisane hasła i adres otwartej strony
- Udany atak na kernelach od 2.6.32 do 4.13.0 (bez KASLR)
- KAISER / KPTI skutecznie zapobiega atakom
- Udany atak na Windows 10
- Udane ataki z wnętrza kontenerów Docker, LXC i OpenVZ, pozwalające na odczytanie danych z kontenerów należących do innych użytkowników. Atak przeprowadzony na serwerach dostawców chmurowych

Nie udało się odtworzyć błędu na procesorach AMD i ARM, ale wg. autorów może to być możliwe.

# Testy wydajnościowe

Testy wydajnościowe przeprowadzono na procesorze Intel Core i7-6700K. Polegał on na przeczytaniu 12 MB pamięci kernela.

| Średnia prędkość odczytu | Błąd | Sposób obsługi wyjątków |
|---|---|---|
| 503 KB/s | 0,02% | TSX |
| 123 KB/s | 0,03% | Przechwytywanie sygnału |

# KPTI - ochrona przed Meltdown w Linuksie

Pomysł polega na stworzeniu dwóch osobnych tablic stron, po jednej dla kernela i procesu użytkownika, oraz ich zmienianie przy zmianie kontekstu.

W praktyce wystarczy tylko trzymać osobno tablice najwyższego poziomu (Page Global Directory), wpisy na niższych poziomach mogą być współdzielone. W przestrzeni procesu trzeba trzymać drobne fragmenty kernela: obsługę pułapek, niemaskowalnych przerwań, wywołań systemowych i zmiany PGD.

Szacuje się, że po aktywowaniu KPTI system działa średnio o ok. 5% wolniej, w niektórych przypadkach aż o 30%.

KPTI jest dostępne od jądra 4.15, jest on też dostępny w niektórych starszych jądrach.

# Ochrona sprzętowa

Główną przyczyną podatności Meltdown jest niedostatecznie szybka weryfikacja uprawnień dostępu do strony przez procesor.

Autorzy pracy zaproponowali, by przy pomocy specjalnego rejestru, np. CR4 włączać dodatkową ochronę pamięci. Ochrona ta by sprawiała, że niezależnie od wpisów w tablicy stron górna połowa przestrzeni adresowej byłaby dostępna tylko z poziomu Ring 0 (czyli  kontekstu kernela).

# Spectre

Po wprowadzeniu KPTI i podobnych, atak Meltdown nie działa. Problemem jest to, że proces atakujący nie ma podmapowanych stron należących do ofiary.

Nowy schemat ataku będzie opierał się na znalezieniu ofiary, która ma dostęp do interesujących nas danych i na którą można w jakiś sposób wpływać. Ofiarami mogą być:

- Kernel
- Silnik JavaScript w przeglądarce
- Silnik SQL bazy danych ?
- Biblioteka, np. PAM, ntdll.dll ?
- ...

# Szablon ataku

1. Znalezienie ofiary posiadającej dostęp do interesującego nas fragmentu pamięci
2. Wyszukanie w kodzie ofiary następującego fragmentu (x jest kontrolowany przez atakującego):

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

3. Doprowadzenie do sytuacji, w której elementy zaznaczone na niebiesko są w cache, tych na czerwono nie ma, a branch predictor jest tak wytrenowany, że skoczy do środka ifa.
4. Wykonanie tak znalezionego gadgetu.
5. Odczytanie wartości array1[x] zapisanej w cache.

# Co jeśli nie mamy dostępu do wartości w cache?

Często się zdarza, że program atakujący nie ma dostępu do tablicy w której zapisana jest odczytana wartość (bo np. jest w kernelu). W takim przypadku autorzy pracy proponują następującą sztuczkę:

1. Stwórz w swojej przestrzeni adresowej tablicę rozmiaru pamięci cache
2. Odczytaj całą tablicę prowadząc do umieszczenia jej w cache
3. Przeprowadź atak Spectre
4. Zobacz, które fragmenty tablicy wyleciały z cache. Istnieje korelacja między adresem brakującej strony, a adresem strony odczytanej podczas ataku Spectre (z adresu tej strony można odczytać wykradzioną wartość)

# Ewaluacja

Procesory: między innymi Intel Ivy Bridge (i7-3630QM), Intel Haswell (i7-4650U), Intel Skylake (nieustalony Xeon w chmurze Google), AMD Ryzen.

Co ciekawsze testy:

- atak na funkcję znajdującą się w tym samym procesie
- odczytywanie danych z dowolnego obszaru pamięci Chrome poprzez JavaScript
- odkrycie, że pamięć branch predictora jest współdzielona między procesami

# Ochrona przed atakiem

Można zaproponować kilka rozwiązań:

- Czyszczenie cache przy każdej zmianie kontekstu (nie stosowane w praktyce)
- Bariery pamięci (specjalne instrukcje ARM, mfence i lfence u Intela)
- Uniemożliwienie precyzyjnego mierzenia czasu (stosowane w Chrome)
- Separacja wrażliwych danych między procesami (w przeglądarce każda karta to osobny proces nie współdzielący wrażliwych danych, takich jak ciasteczka sesji)
- Sztuczki w asemblerze utrudniające działanie branch predictora (RETpolina stosowana w LLVM)

Niestety żadne z nich nie rozwiązuje w całości problemu.