Meltdown & Spectre

Klaus Kusche

Frühjahr 2018

Inhalt

- Wiederholung Rechnerarchitektur: Architektur moderner Prozessoren (Core, MMU, Cache, Speicher)
- Meltdown & Spectre:
  - Geschichte
  - Klassifizierung
  - Funktion und Wirkung
  - Betroffene Prozessoren
  - Gegenmaßnahmen

Ziel

- Verständnis der *Funktion* der “Lücken”
- Verständnis der *Gegenmaßnahmen*
- … und ihrer *Folgen*, *Umsetzung*, …

==> Einschätzung der *Bedrohung*

==> Entscheidung über *Maßnahmen*

(die z.B. in meinem Fall *“nein”* war!)

Voraussetzungen

- ***Rechner-Architektur***
  Grundsätzliche Funktionsweise eines Prozessors
- (Assembler)
  Implementierung von Programmen
  bzw. Programmiersprachen
- Grundzüge ***Betriebssystem***
  (Speicherverwaltung,
  Unterschied normale Programme / Kernel)

Speicher (RAM)

- Folge von ***Bytes*** (Werten 0 … 255)
- Jedes Byte hat eine “Hausnummer”

  (***Adresse***, fortlaufend 0 … ein paar Mrd.)
- Physikalisch eigene Chips,
  logisch weit weg von der CPU:

  Core <==> MMU <==> Cache <==> RAM
- Viel ***langsamer*** als die CPU:

  1 Zugriff … ~200 Clocks = mehrere 100 Befehle!

Cache (1)

- ***Zwischenspeicher*** in der CPU für RAM-Daten
- Ziel: Zugriff ***schneller*** machen:

	Zugriffszeit (Clocks)
Register in der CPU:	0
L1 Cache:	2-4
L2 Cache:	10-25
L3 Cache:	20-50
RAM:	120-1000

Cache (2)

- Viel ***kleiner*** als das RAM
  (16 KB ... 16 MB im Vergleich zu 4 GB ... ???)

==> Enthält nur einen *kleinen Teil* der RAM-Daten

... und zwar die
zuletzt benutzten!

==> Neue Zugriffe *"verdrängen"*
Daten älterer Zugriffe aus dem Cache

Verwaltung: In ***Cachelines***
= Blöcken zu *32...128 aufeinanderfolgenden Bytes*

Cache (3)

Erster Zugriff auf eine Speicherstelle:

- holt die Daten ***aus dem RAM***
- kopiert betreffende Cacheline in den Cache

Weitere Zugriffe kurz darauf auf *dieselben Daten*:

- holen die Daten ***aus dem Cache***

Für Anwendungen logisch ***völlig unsichtbar***:

Ein Programm kann logisch nicht feststellen
- ob & wie viel Cache vorhanden ist
- *ob bestimmte Daten im Cache sind* oder nicht

Cache (4)

Aber:

Die CPU hat ***takt-genaue Timer***
(Auflösung ~0,25 ns)

==> ***Im Cache / nicht im Cache*** ist durch

Messung der Zugriffszeit

vom Programm aus feststellbar:

< 10 ns *im Vergleich zu* ***> 50 ns***

Cache (5)

Der Cache ist für

- alle *Programme*
- alle *VM's*
- das *Betriebssystem*

gemeinsam!

==> *Jeder "spürt" die Auswirkungen*
der Cache-Aktivitäten
aller anderen!
(als Beschleunigung / Verlangsamung seiner Zugriffe)

Core (1)

Das *“klassische”* Prozessor-Modell:

- *Befehl* holen & dekodieren
- *Operanden* holen
- *Operation* ausführen
- *Ergebnis* speichern
- *nächsten* Befehl holen …

… wäre bei gleicher Halbleiter-Technologie

einige 100 mal langsamer

als aktuelle Prozessoren wirklich sind!

Core (2)

Tricks:

- ***Pipelining*** ==> hier nicht entscheidend
- 4 ... 16 Rechenwerke (Funktionseinheiten) pro Core ("***superskalare***" CPU)

  z.B. 4 * int, 2 * float, 3 * load/store, ...

==> Ziel:

Möglichst viele Rechenwerke
möglichst ständig
(sinnvoll) "beschäftigen"!

Core (3)

- Mehrere Befehle ***parallel*** ausführen
- ***Hyperthreading***: *1 Core* verarbeitet Befehle von *2 Programmen* gleichzeitig / abwechselnd
- Unabhängige Befehle umordnen / vorziehen ("***out-of-order***"-Architektur)
- Befehle *"auf Verdacht" / "auf Vorrat"* ausführen ("***speculative***" Execution)

Bei modernen Prozessoren:

Bis zu 200 Befehle gleichzeitig "in-flight"

Core (4)

"Befehl ausführen" und

"Ergebnis für das Programm sichtbar machen"

sind ***zwei getrennte Schritte***

(ev. zeitlich *viele Takte auseinander!!!*)

Bei "spekulativer" Ausführung,
wenn sich später herausstellt
"Spekulation war falsch":

Ergebnis / Wirkung des Befehls
wird ***nie sichtbar gemacht!***

Core (5)

Spekulative Ausführung

... ist *logisch* völlig transparent / *unsichtbar*,
d.h. vom Programm aus *nicht feststellbar*
(Terminologie Spectre: *"Transiente Befehle"*),

d.h. die Hersteller machen nichts "falsch",

... aber eine *fehlgeschlagene* spekulative Ausführung
hinterlässt durch ihre *Speicher-Lese-Zugriffe*
trotzdem *Spuren und Nebenwirkungen*:
- Daten werden ev. ***in den Cache geladen***
- MMU-Zustand ändert sich ev.

Core (6)

Sprungbefehl mit *fixem* Ziel: ***0 Clocks***

Sprungbefehl mit *variablem* Ziel: ***10-25 Clocks***

Was hat ein variables Ziel?

- *Bedingte* Sprünge: **`if`**, Schleifenbedingungen
- *Indirekte* Sprünge: **`switch`**, Interpreter & JIT's
- *Indirekte* Call-Befehle: ***Methodenaufrufe***

Core (7)

Trick: Sprungvorhersage = ***"Branch Prediction"***

- Versucht, das *Sprungziel vorherzusagen*, und beginnt am vorhergesagten Ziel mit *spekulativer Ausführung*
- Wenn sich Vorhersage als **richtig** erweist:
  - *Ergebnis* der spekulativen Befehle wird *sichtbar*
  - Sprungdauer 0 Clocks
- Wenn sich Vorhersage als **falsch** erweist:
  - Spekulativ ausgef. Befehle werden *verworfen*
  - Richtiger Sprung wird nachgeholt: 12-25 Clocks

Core (8)

Wie funktioniert die Sprungvorhersage?

U.a. großer ***"Sprungziel-Cache"***
(***"BPB"*** = *"branch prediction buffer"*):

"Wohin sprang derselbe Sprungbefehl
bei seiner vorigen Ausführung?"

==> Die Sprungvorhersage ist

"trainierbar"!

(bei genauer Kenntnis der internen BPB-Organisation kann man sogar fremde statt demselben Sprungbefehl trainieren!)

Core (9)

Der BPB ist *pro Core*, ***nicht*** programmbezogen!

==> Ein Programm kann Sprünge/Calls

- seines "*Hyperthreading-Zwillings*",
- des *Kernels*
- eines *nachfolgenden* Programmes
- theoretisch sogar des *VM-Hosts* und eines anderen *VM-Guests*

beeinflussen!

Core (10)

Der ***Microcode***

- steuert die Dekodierung und Abarbeitung von *komplexen* Maschinenbefehlen in der CPU
- ist ein *"Hardware-Programm"*
  (entfernte Ähnlichkeit mit dem Programm eines FPGA)
- wird früh *bei jedem Booten* in die CPU geladen bzw. aktualisiert (vom BIOS oder vom OS aus)

Core (11)

Microcode - Updates:

- Normalerweise: ***BIOS-Update***
- Bei *Linux* und *macOS*:
  Auch via ***Betriebssystem-Update*** möglich
  (statt BIOS-Update)
- *Microsoft*:
  - Könnte die Updates mit Windows installieren,
    macht es aber *nicht* (rechtliche Gründe?)
  - Ausnahme: *Surface: Update von Microsoft*

Core (12)

Barrier-Befehle

serialisieren bzw. verzögern

die parallele / spekulative / umgeordnete Ausführung von Befehlen

==> *Das Programm wird langsamer!*

Beispiel x86: **LFENCE** (fence = Zaun, Gatter, Hindernis)

*"Beginne alle nachfolgenden Befehle erst dann, wenn alle Lese-Befehle vor dem **LFENCE** ihre Daten fertig gelesen haben!"*

MMU (1)

"Memory Management Unit"

2 Haupt-Aufgaben:

- ***Zuordnung*** *"virtuelle Adressen"* (im Programm) ==> *"reale Adressen"* (im RAM),

  Erkennung *ungültiger* virtueller Adressen

- ***Prüfung*** der *Zugriffsrechte*:

  "Darf das Programm diese Speicherstelle lesen / schreiben/ als Befehl ausführen?"

MMU (2)

Steuerung der MMU durch

"Page tables"

- Werden vom Betriebssystem verwaltet
- Sind *groß* (viele MB)
- Liegen im normalen *RAM*,
  mit einem eigenen *Cache* in der MMU:

  TLB = "Translation Lookaside Buffer"

MMU (3)

==> Adress-Zuordnung & Rechte-Prüfung kann

- ***schnell*** sein (wenige Clocks),
  wenn die Pagetable-Daten *schon im TLB* liegen
- ***sehr lange dauern*** (*hunderte* Clocks!)
  wenn die Pagetable-Daten *zuerst aus dem RAM*
  in den TLB geladen werden müssen

==> Eigentlicher *Daten-Zugriff* muss *warten*

==> Erkennung *"ungültig"* oder *"Rechte fehlen"*
kann lange *dauern!*

MMU (4)

Nur (?!) bei Intel-CPU's:

Adress-Zuordnung und *Rechte-Prüfung* sind

technisch & zeitlich getrennt!

Folge:

- *Daten* werden *gleich nach der Adress-Zuordnung* geladen und auch (spekulativ!) *verwendet*
- *Abbruch* wegen fehlender Rechte kommt (unter bestimmten Umständen) erst *später!*

MMU (5)

"Klassische" Pagetable-Organisation:

- Eine Pagetable *pro Prozess* (Programm)
  nur für die Speicherbereiche *dieses* Programms
- Plus eine eigene Pagetable für den *Kernel*

==> Jedes Programm "sieht"
nur seine *eigenen* Speicherbereiche

==> Zugriffe auf Betriebssystem-Daten,
fremde Daten:

"Ungültige Adresse"

MMU (6)

Das “Klassische Modell” erfordert

Zwei Pagetable-Umschaltungen
(vom Programm zum Kernel und zurück)

pro Syscall, pro Interrupt, ...

Problem:

Pagetable-Umschaltungen kosten Zeit!!!

Einige *hundert* Clocks... (bei *neuen* CPU's mit *“PCID-Feature”*),
...*viele tausend* Clocks! (bei *alten* CPU's, kompletter *TLB-Flush!*)

MMU (7)

1.) Zusätzliches *Speicher-Rechte-Bit*, HW-geprüft:

"Zugriff nur erlaubt, wenn die CPU gerade im Kernel-Modus arbeitet"

2.) In *jeder* Prozess-Pagetable wird

*der **gesamte Betriebssystem-Speicherbereich***
(und damit das gesamte RAM,
d.h. die Daten aller Programme!)

als *"gültig"* eingetragen,
aber als *"Kernel only"* markiert.

MMU (8)

==> ***Schneller!!!***

(weil *keine Pagetable-Umschaltung*
beim Wechsel in den Kernel mehr nötig ist!)

==> *"Im Prinzip" gleicher Effekt ...*

... aber im Detail nicht!

==> Jedes Programm "sieht" jetzt *alles!*

==> Zugriffe auf Betriebssystem-Daten,
fremde Daten:

"Gültige Adresse, aber Zugriff verboten"

Fragen?

Wenn nicht:
Pause!

Wer & Wann?

- *2 Teams* unabhängig voneinander:
  - Internationales Forscher-Team
    (AT / US / AU, 4 Uni's und 3 Firmen)
    (hatten schon *"Rawhammer"* gefunden)
  - Google
- Endeckung *Frühjahr 2017*
- Seit Sommer 2017 HW/SW-Herstellern bekannt
- Seit Jänner 2018 veröffentlicht
  in wissenschaftlichen Publikationen & CVE's

Klassifizierung (1)

- ***HW-basiert***, SW ist "unschuldig"

  ==> Auf *allen* Betriebssystemen
  (Windows, Linux, macOS / iOS, Android, ...)

  ==> Trifft potentiell *alle* Programmier-Sprachen
  (meist C, sogar mit JavaScript erfolgreich!)
  und *alle* Anwendungen

  ==> Virenschutz usw. hilft *nicht*

  Aber: *Nicht* auf "*kleinen*" Prozessoren
  (Mikrocontroller wie Atmel, embedded ARM usw.)

Klassifizierung (2)

- ***Lokal:***

  Exploit-Code muss *am betroffenen Rechner* laufen
  (keine direkte Ausnutzung über das Netz)

  - Entweder:
    Angreifer hat *lokalen Rechner-Zugang*
  - Oder:
    JavaScript o.ä. via *Webseiten*, in Mails, ...

    ==> *indirekt doch "remote"?*

    (was ist mit Makros in Dokumenten???)

Klassifizierung (3a)

- ***Read-Only***, d.h. “Datenklau”

  Hier: *Beliebiger / aller Daten* ***aus dem RAM***

  ==> *Nicht* direkt von der *Platte*,
  nicht aus Datenbanken,
  nicht von anderen Rechnern

  ==> *Kein “Denial of Service”*
  (kein Absturz, keine Daten-Löschung, ...)

  ==> *Keine* Daten- oder Code-*Veränderung*,
  keine Änderung des Programm-Verhaltens

Klassifizierung (3b)

==> Auf Servern, Cloud-Systemen, ...:
Daten *anderer Benutzer / Programme / Kunden*

==> *Passwörter*, *Zertifikate*, *Krypto-Schlüssel*, ...
Webseiten-Inhalte & -Eingaben (z.B. TAN's)

==> Kernel-Daten, Code anderer Programme, ...
Hilft u.a., *andere Sicherheitslücken zu finden*
(Programm-Version, was liegt wo im RAM, ...)

==> Bei interpretierten Sprachen/Script-Sprachen
(Java, JavaScript, Python, ...):
Zugriff auf interne Daten des Interpreters, ...

Klassifizierung (4)

- ***"Seitenkanal-Attacke"*:**
  Das Abgreifen der Daten erfolgt nicht direkt, sondern über die *Messung physikalischer Effekte*: Stromverbrauch, Funkabstrahlung, Zeit, ...

  Hier: ***Zeit***

  Denn: Es liegt *kein Fehler* vor
  - in der *spezifizierten logischen Funktionalität*
  - oder in deren *Implementierung*

  ==> Die Hardware-Logik *"arbeitet wie definiert"*
  ==> Es gibt *kein direktes unberechtiges Auslesen*

Grundidee (1)

// Vorbereitung:
// Daten des **`char`***-Arrays* **`spy`** *aus dem Cache werfen*

// Eigentlicher Angriffscode:
// (spekulativ ausgeführt, **`x`** *und* **`y`** *werden verworfen!)*
... // je nach Angriff
`x` **`=`** *... (gewünschtes Byte aus dem Speicher)*
`y = spy[x * n];` *// mit* **`n >= 64`**
// lädt genau ***eine*** *Cacheline (64 Bytes) von* **`spy`** *in den Cache*
// trifft für jeden Wert von **`x`** *eine* ***andere Cacheline***
// (wegen **`n >= 64`** *liegen die Elemente* **`spy[x * n]`**
// mindestens 64 Bytes auseinander!)

Grundidee (2)

```
// Auswertung / Messung:
for (i = 0; i < 256; ++i) {
    // Zeitmessung Start
    y += spy[i * n];
    // Zeitmessung Ende:
    // Durchlauf mit i gleich x ist schnell,
    // weil das Element spy[i * n] schon im Cache liegt
    // alle anderen Durchläufe sind langsam!
    // ==> Wert des ursprünglich angegriffenen x ermittelbar!
}
```

Meltdown: Idee

"Rogue Data Cache Load"

Nutzt das Verhalten der ***MMU***
bei *gültigen, aber unzulässigen Zugriffen*:

Page Fault
(Programm-Abbruch)
kommt manchmal zu spät:

Erst, nachdem die "verbotenen" Daten
schon (spekulativ, unsichtbar)
gelesen & *verwendet wurden!*

Meltdown: Details

```
// Caches und TLB präparieren,
// damit das Timing passt

x = *evil_pointer;  // Löst Signal bzw. Exception aus
y = spy[x * 256];   // wird ev. trotzdem noch ausgeführt!
                    // (falls MMU nicht im TLB hat
                    // und aus dem RAM holen muss)

// Exception abfangen, Cache auswerten
```

Meltdown: Effekt

- Liest alle Daten mit *gültiger* virtueller Adresse (auch *ohne* Rechte, z.B im *Kernel-Adressbereich*), damit beliebige / alle ***RAM-Inhalte***

  = Alle *Kernel-Daten* (incl. Filesystem-Caches), alle *Daten aller Applikationen*, ...

- Relativ schnell: > 500 KB/sec
  ==> in ein paar Stunden sind *einige GB RAM* lesbar

Meltdown: Opfer

- Alle *Intel-Core-CPU*'s seit mindestens 2008
- Intel Atom *nicht*
- AMD *nicht*
- Nur modernster ARM-Core: A75, andere nicht
- Sonst bisher nichts bekannt
- Bei ARM A15, A57 und A72:
  Ähnlicher Bug erlaubt normalen Programmen
  Zugriff auf Kernel-*Register*

Inzwischen aktiv genutzt!

Meltdown: Fix / Kurzfristig (1)

- *Kein* Hardware- / Mikrocode-Fix möglich
- *Kein* Fix auf Anwendungsebene nötig
- Im *Betriebssystem*:

Pagetables
von normalen Programmen
und Kernel wieder trennen!

==> *Komplexer Umbau!*

Meltdown: Fix / Kurzfristig (2)

Status:

- ***Intel:***
  Windows, praktisch alle Linux, aktuelles macOS:
  Endgültiger Fix schon ausgeliefert!

  (Windows-Fix legte zahlreiche Virenscanner & manche Treiber lahm, machte Probleme auf alten AMD-Proz., ...)

- ***ARM:***
  - Aktuelles Original Google Android & iOS:
    Endgültiger Fix schon ausgeliefert!
  - Linux: Beta-Fix, noch nicht in Verteilung
  - Windows Phone usw.: Nichts...

Meltdown: Fix / Kurzfristig (3)

- 0-10 % Perf.-Verlust bei *normalen Programmen*
- max. 30 % Perf.-Verlust bei
  Servern (DB/Web), *Spielen*, *I/O*-lastigem Code

  (manche Kunden beobachten auf ihren Cloud-Servern bis zu 50 % ?!)
- >= 50 % Einbruch bei *Benchmarks schneller SSD's*
  und anderen *synthetischen Tests*

  (75 % bei macOS ???)
- Abhängig vom Prozessor-Alter:
  Verlust bei alten Intel-Prozessoren
  ohne PCID-Feature deutlich *höher*!

Meltdown: Fix / Langfristig

Änderung des MMU-Verhaltens wäre *sinnvoll*
und müsste fast *"verlustfrei" möglich* sein!

==> "Verbotene" Daten *gar nicht laden*,
Adresse & Rechte *gleichzeitig* auswerten!

Aber:
Umfangreiche HW-Architektur-Änderung,
erst in *zukünftigen* Generationen,
nicht via Microcode möglich!

Spectre 1: Idee

"Bounds Check Bypass"

Nutzt die ***Sprungvorhersage***
bei *bedingten Sprüngen*:

Trainiert die Sprungvorhersage
bei einem *normalen Array-Indexgrenzen-if*,
macht dann einen Zugriff mit "*bösem*" Index

==> Prozessor macht spekulativ
den "Index ist ok"-if-Zweig
und greift auf Daten
weit außerhalb des Arrays zu!

Spectre 1: Detail

Suche irgendwo im *bestehenden* Programmcode (incl. Libraries) Code mit *folgender Struktur*:

```
if (value_from_user < arr1_size) {
   // if dauert lang, wenn arr1_size nicht im Cache ist!
   x = arr1[value_from_user];
      // greift für falsches value_from_user
      // auf jeden beliebigen Speicher zu!
   y = arr2[...x...];
   ...
}
```

Rufe ihn mit *ein paar richtigen* **value_from_user** und dann *mit einem falschen* auf!

Spectre 1: Effekt & Opfer

Liest *alle RAM-Daten* des *eigenen* Programmes (interessant z.B. bei Interpretern, Browsern)...

... auf praktisch *allen* "*großen, modernen CPU's*":

- Alle *Intel Core* und *AMD* seit "ewig" (ev. 1995?)
- *Intel Atom* seit 2014
- Fast alle "Handy- und Server-*ARM*-Prozessoren" (einzige Ausnahme: Cortex A53 Core, z.B. Raspi)
- IBM *Power*, Sun/Oracle/Fujitsu *Sparc*, MIPS, ...

Spectre allg.: Fix (1)

Verhalten ist implizite, gewünschte Konsequenz moderner Prozessorarchitektur und "korrekt":

Code verhält sich "wie geplant"!

==> Es gibt *keinen* "Fix" für bestehenden Code, der die *Ausführung bestehenden Codes* ohne essentielle Verluste *"repariert"!*

= ohne dass das Programm *falsch* oder *zu langsam* wird!

*"Branch Prediction" komplett ausschalten wäre **viel** (> Faktor 5!) zu langsam!*

Spectre allg.: Fix (2)

Nur "*Workarounds*" möglich:

Bestehender Code muss

durch anderen Code ersetzt

werden, der *dasselbe Ergebnis* erzielt,
aber

- mit anderen *"internen Abläufen"*
  (z.B. ohne die Branch Prediction zu benutzen)
- oder mit *"expliziten Bremsen"*
  (nur dort wo wirklich nötig).

Spectre allg.: Fix (3)

1.) Falls nötig:

Per *Microcode-Update*

- *Neue Instruktionen* einführen
  (bei Intel angeblich 3, bei ARM 1)

  z.B. *"Lösche / ignoriere Sprungvorhersage"*,
  neue *Barrier-Befehle*
- Ausführung *bestehender Instruktionen*
  an neue Befehle *anpassen*

  z.B. neue Status-Bits berücksichtigen

Spectre allg.: Fix (4)

2.) *Alle Compiler* (und JIT's) *umbauen*, sodass sie für if's, virtuelle Methoden-Aufrufe, switch, ...

***anderen* Code erzeugen**

(ev. mit den "neuen" Instruktionen)

3.) *Gesamten* Code *neu kompilieren!!!*

- Betriebsystem
- Alle Anwendungen
- Alle Libraries
- ...

(bestehenden *Assembler-Code* händisch durchgehen & fixen)

Spectre allg.: Fix (5)

Aktueller Stand (Ende Jänner):

- *Microcode-Fixes* sind teilweise *verfügbar*, aber *buggy* (alle wieder zurückgezogen)!
- Modifizierte *Compiler* (gcc, llvm, msvc) sind im *Beta-Test*
- Betr. *JIT*'s gibt es *keine Info*
- Frisch compilierte *Anwendungen* sind *noch nicht* in Sicht...

==> Das wird eine Sache von Monaten / Jahren!

Spectre 1: Fix (1)

Intel (& AMD):

Theoretisch, für 100 % Schutz:

Zwischen *jedem* bed. Sprung & Speicherzugriff:

LFENCE

Effekt: Speicherzugriffs-Befehl muss *warten*,
bis die if-Bedingung fertig ausgewertet ist.

LFENCE gibt's schon länger
==> *kein* Microcode-Update nötig!

Spectre 1: Fix (2)

Performance-Auswirkungen:

Ja (weniger spekulative “Vorausarbeit”),
katastrophal: ***› 60 % langsamer***

==› *Nicht bei jedem* bedingten Sprung einbauen!

==› Compiler muss überlegen:

“Wo brauche ich das LFENCE, wo nicht?”

==› Abwägung Performace / Sicherheit!

1. Versuch von MSVC:
Performance-Verlust gering, aber viele Lücken blieben offen!

Spectre 1: Fix (3)

ARM:

2 *verschiedene* Fixes, je nach Core-Typ:

- ***Neuer*** *Barrier-Befehl* **CSDB** (ähnlich Intel)

  ==> *Microcode-Update* nötig!

- *Conditional-Move-Befehl*
  statt normalem Move-Befehl verwenden
  (wirkt angeblich auch bei AMD, nicht Intel?)

==> Für *universellen* Code (für alle Cores)
beides gleichzeitig einbauen!

Spectre II: Idee

"Branch Target Injection"

Nutzt die ***Sprungvorhersage***
bei *indirekten Sprüngen und Calls*:

Trainiert die *Sprungvorhersage*
eines *fremden* Sprungbefehls
(oder des eigenen Befehls bei *Hyperthreading*),
um ein *Codestück* anzuspringen,
das mit dem if/Call *gar nichts zu tun* hat!

==> Ein anderes Programm (bzw. der zweite Thread)
führt später den "falschen" Code spekulativ aus!

Spectre II: Details

Man muss im Zielprogramm wieder
ein *vorhandenes Codestück* suchen,
das *abhängig* von einem *bestimmten Wert*
auf *irgendwelche anderen Daten zugreift*
(und dabei Spuren im Cache hinterlässt).

Die Adresse dieses Codestückes
trainiert man in die Sprungvorhersage…

Extremfall:
Google hat den BPF-Interpreter im Linux-Kernel dazu gebracht,
beliebigen (bösartigen) BPF-Code auszuführen.
Der BPF-Code hat dann Zieldaten gelesen und den Cache befüllt.

Spectre II: Effekt & Opfer

Liest *alle RAM-Daten* eines *fremden* Programmes oder des *Kernels*.

- Alle *Intel Core* seit ??? (sehr modellabhängig!)
- AMD: *"Theoretisch ja, aber …"* (trotzdem Patch?)
- *Intel Atom* nein?
- Fast alle "Handy- und Server-*ARM*-Prozessoren" (einzige Ausnahme: Cortex A53 Core, z.B. Raspi)
- Andere: unbekannt

Ausnutzung erfordert *komplexe Analyse* pro CPU!

Spectre II: Fix

- Viel *Chaos und Verwirrung*
- Viel (teils geheimes) *"Snake Oil"*
- Noch *weit* entfernt von endgültiger Lösung
- Intel: Bisher *keine* Lösung mit *Barrier-Befehlen?* (ARM: ???)

==> Derzeit keine oder nur *vorläufige Patches*, Probleme mit *Nebenwirkungen*

==> *Zwei* (oder mehr) unabhängige *Lösungsansätze*, grundverschieden!

Spectre II: Fix Variante 1 (1)

"Retpolines" =
"Return trampolines"

Statt indirektem Sprung/Call-Befehl: (stark vereinfacht!)

- Zieladresse in ein Register legen
- *Fixe Hilfs-Funktion* aufrufen
- Hilfsfunktion *ersetzt Return-Adresse am Stack*
  durch Zieladresse
- … und macht *Return*
  ==> Return springt zum ursprünglichen Ziel!

Spectre II: Fix Variante 1 (2)

- Nutzt *"Return Prediction"* statt Branch Prediction
- Kommt *ohne neue Maschinenbefehle* aus
  ==> Kein μCode-Update nötig
- Braucht dafür *neue Compiler-Versionen*

Aber ***kostet Performance***:

- Mehr Befehle
- "Return Prediction" sagt *immer* falsch voraus:
  Der Code enthält dort eine zusätzliche *Warteschleife*,
  die angesprungen und *spekulativ ausgeführt* wird,
  bis der Spekulations-Fehler erkannt
  und das echte Ziel angesprungen wird

Spectre II: Fix Variante 1 (3)

- Auf *älteren* Intel-Prozessoren:

  Funktioniert, *mittlere* Performance-Verluste

- Auf *neueren* Intel-Prozessoren:

  - Relativ *hohe* Performance-Verluste!

  - *Bei/ab Skylake*: Auch Return ist Spectre-*anfällig*!
    (nutzt ab 16 Returns nacheinander die Branch Prediction
    ==> Linux entwickelt Workaround)

  - HW hat möglicherweise Problem mit "Mißbrauch" des Return
    (ev. manchmal *Absturz*? ev. bisher unbekannter *echter HW-Bug*?)

  ==> *Trotzdem **neuer Microcode** nötig?!*

Spectre II: Fix Variante 2 (1)

Zusätzliche, ***neu eingeführte Befehle***
zur Kontrolle / Abschaltung / Leerung
der ***Sprungvorhersage***
(bei Intel: 3 Stück)

Wo sind die Befehle nötig?

- Auf jeden Fall *im Betriebssystem*
  (bei jedem *Syscall*, bei jedem *Prozesswechsel*)
  (gegen Angriffe auf das Betriebssystem und teilw. fremde Programme)
- Auch in den *Anwendungen* bei jedem einzelnen indirekten Call/Sprung-Befehl
  (für Schutz gegen alle Varianten)

Spectre II: Fix Variante 2 (2)

- *Microcode-Update* Voraussetzung,
  sonst Absturz wegen *"unbekanntem Befehl"*
- Erster Microcode-Update wurde *zurückgezogen*,
  machte Prozessoren *instabil*
- Auf *neueren* Intel-Prozessoren:
  Mittlere bis hohe Performance-Verluste
  (angeblich > 4000 Clocks bei Prozesswechsel?)
- Auf *älteren* Intel-Prozessoren:
  Sehr hohe Performance-Verluste!

  ==> *Langsamer als Variante 1!*

Spectre II: Fix Variante 2 (3)

Auf AMD:

- *Nachrüstbarkeit* via μCode *fraglich*
  (auf welchen Prozessoren?)!!!
- Bisher *keine Aussagen* über Performance!

  (dafür läuft *Fix Variante 1* auf AMD
  angeblich *ziemlich performant*!)

  ==> AMD hätte lieber Variante 1

Auf ARM:

- Anscheinend auch Variante 2 (neuer Befehl: BTB leeren)

Spectre II: Fix Variante 3

Auf AMD hilft auch ein einfaches **LFENCE**:

Verhindert das Problem nicht grundsätzlich,
aber macht den *Spekulations-Zeitraum so kurz*,
dass sich keine Ausnutzung mehr ausgeht.

Aber:

- Geschwindigkeitsauswirkungen *unklar*
- *Keine Aussagen von Intel*
  betr. Wirksamkeit auf Intel-Prozessoren

Spectre II: Fix

- Microsoft / *Windows* setzt anscheinend nur auf *Variante 2*
- *Linux* und Google / *Android* wollen hingegen primär *Variante 1* implementieren?

  (ganz böse Mail von Linus Torvalds...)

  (zum Schutz von VM's stellenweise *zusätzlich Variante 2* nötig!)
- AMD möchte nur Variante 1 (oder 3),
  Intel möchte nur Variante 2

==> Prozessoren, Compiler und BIOS müssen wohl *beides* unterstützen?

Spectre allg.: Weitere Zugriffe

"Draufgaben" von *Google* für Spectre I / II auf *Linux*:

- Spectre I & moderner Intel Core:
  User-Programm liest *Kernel-RAM*
- Spectre I & AMD:
  Ebenso, aber nur wenn BPF-JIT im Kernel aktiv
  (BPF-JIT: Firewall-Regelcode-Interpreter)
- Spectre II & moderner Intel Core:
  Root-Programm im *VM Guest*
  liest *Kernel-RAM* des *VM Host* (!)

... und da kommt sicher noch mehr!

Logistisches Problem

Eigentlich bräuchte man
- *verschiedene* Betriebssystem-Versionen
- *verschiedene* Anwendungs-.exe's und -.dll's

- für *jede einzelne* Intel-/ARM-Prozessor-*Generation*
- für AMD-Prozessoren
- für Systeme *mit* / ohne BIOS-Update

==> Logistisch nicht beherrschbar???

==> "Auf Verdacht" doppelt Behebungen einbauen!

==> Noch mehr Performance-Problem!

Anwendungs-Änderungen

Z.B. in Chrome, Firefox, ...:

- JavaScript-*Timerfunktionen*
  viel *ungenauer* gemacht (wegen *Cache-Messung*)!
- Sprach-Feature *"SharedArrayBuffer" entfernt*

==> Kein Performance-Verlust, aber *Feature-Verlust*!

- Ein *separater Prozess* pro Webseite / pro Tab
  (aus Sicherheitsgründen allgemein empfohlen!)

==> Mehr *Speicherbedarf*, etwas *Laufzeit-Overhead*

Literatur (1)

- Publikationen *TU Graz*:

  https://meltdownattack.com/
  https://meltdownattack.com/meltdown.pdf
  https://spectreattack.com/spectre.pdf

- Publikation *Google*:

  https://googleprojectzero.blogspot.de/2018/01/reading-privileged-memory-with-side.html

- CVE's:

  CVE-2017-5753, CVE-2017-5715, CVE-2017-5754

Literatur (2)

- Intel:

  https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf

- ARM:

  https://developer.arm.com/support/security-update/download-the-whitepaper
  https://developer.arm.com/support/security-update

Literatur (3)

- AMD:

  https://www.amd.com/en/corporate/speculative-execution

  http://developer.amd.com/wordpress/media/2013/12/Managing-Speculation-on-AMD-Processors.pdf

- “Retpolines”:

  https://support.google.com/faqs/answer/7625886

“The end”

Fragen?