Meltdown och Spectre

Peter Sundvik

Kandidatavhandling i datateknik

Handledare: Wictor Lund

Fakulteten för naturvetenskaper och teknik

Åbo Akademi

2018

Referat

Moderna processorer använder sig av tekniker som branch prediction, spekulativ exekvering och dynamisk exekvering för att maximera prestanda. I juni 2017 upptäckte forskare vid Googles Project Zero och en grupp med forskare från olika universitet och företag säkerhetssårbarheterna Meltdown och Spectre. Dessa sårbarheter utnyttjar svagheter hos de tidigare nämnda designprinciperna för att få tillgång till privat information.

I Spectre tvingar angriparen processorn att utföra spekulativa operationer som inte uppstår vid vanlig programexekvering vilket leder till att en angripare kan få tag på privat information. Meltdown utnyttjar bieffekter hos dynamisk exekvering för att läsa godtyckliga kernel minnesadresser inklusive privat information och lösenord. I den här kandidatavhandligen diskuteras principerna inom datorarkitektur som ligger bakom Meltdown och Spectre, t.ex. dynamisk exekvering och spekulativ exekvering. Fokus i avhandlingen ligger ändå vid att förstå hur Meltdown och Spectre fungerar och de åtgärder som gjorts eller borde göras för att förebygga dem.

Innehållsförteckning

1 Inledning 1

2 Datorarkitektur 2

2.1 Bakgrund 2

2.2 Pipelining 2

2.3 Instruction-level parallellism - ILP (Parallellitet på instruktionsnivå) 3

2.4 Branch prediction 4

2.5 Spekulativ exekvering 4

2.6 Dynamisk exekvering 5

2.7 Minnesisolering 5

2.8 Sidokanalsattacker 6

3 Spectre 7

3.1 Exploiting Conditional Branch Misprediction 8

3.2 Poisoning Indirect Branches 8

3.3 Åtgärder 8

4 Meltdown 8

4.1 Läsning av information 9

4.2 Överföring av information 9

4.3 Mottagning av information 9

4.4 Åtgärder 9

4.4.1 Hårdvara 9

4.4.2 KAISER 9

5 Diskussion 9

6 Källor 10

1 Inledning

[TODO]

2 Datorarkitektur

2.1 Bakgrund

[TODO]

2.2 Pipelining

Pipelining är den viktigaste tekniken som används för att åstadkomma parallellitet på instruktionsnivå i moderna processorer samt den faktor som ökar prestandan mest. Målet med pipelining är att dela upp instruktioner i flera steg som kan köras samtidigt av olika enheter i processorn. En processor utan pipeline använder normalt fem klockcykler för att utföra en instruktion. I en RISC-processor går det till på följande sätt: [3]

1. Instruktionshämtning

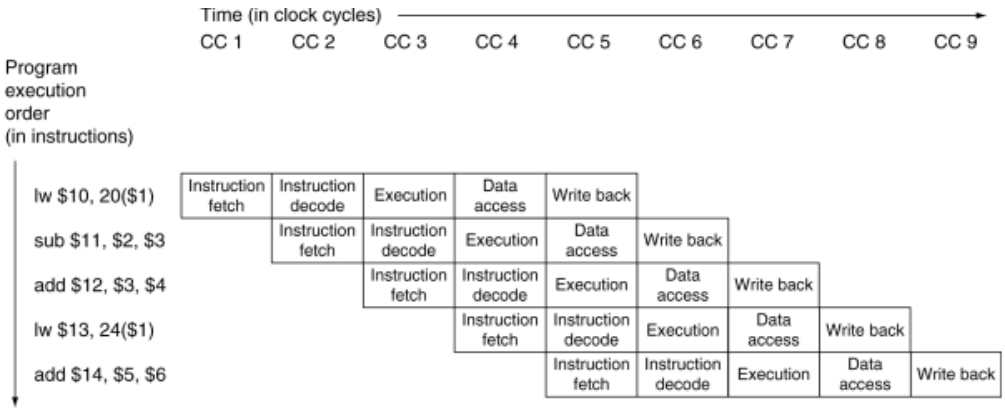
   Hämtar en instruktion från minnesadressen som finns i instruktionspekaren (program counter) och instruktionspekaren uppdateras till nästa instruktion.
2. Avkodning av instruktioner

   Avkodar instruktionen som finns i registret så att processorn vet hur många operatorer den måste hämta för att utföra instruktionen.
3. Exekvering

   Den aritmetisk logiska enheten utför operationerna som blev förberedda i den tidigare cykeln.
4. Minnes Access

   Om det var en laddnings instruktion läses data in från minnesadressen som beräknades i den tidigare cykeln.
5. Write back

   Skriver resultatet till registret, detta kan vara både från minnet i och med en laddning eller från den aritmetisk logiska enheten.



Figur 1. Diagram som illustrerar en enkel pipeline

Som figur 1 illustrerar hämtas en ny instruktion varje klockcykel i en pipeline och påbörjar exekveringen som tar fem klockcykler. Eftersom en instruktion påbörjas varje klockcykel blir prestandan fem gånger högre jämfört med en processor som inte använder pipelining.

2.3 Instruction-level parallellism - ILP (Parallellitet på instruktionsnivå)

Parallellitet på instruktionsnivå är ett begrepp som anger hur många maskininstruktioner som processorn kan exekvera samtidigt. ILP gör det möjligt för processorn att utföra flera instruktioner samtidigt, samt att även ändra ordningen på instruktionerna som exekveras. För att enkelt och effektivt öka mängden ILP används ofta loop-nivå parallellism vilket betyder att man i en loop kan utföra olika iterationer av loopen parallellt [3].

```
for (i=0; i <= 999; i = i + 1)
        x[i] = x[i] + y[i];
```

Loopen ovan adderar två listor med 1000 element. Genom att veckla upp loopen kan varje iteration utföras samtidigt med vilken annan iteration som helst. Detta sker antingen statiskt av kompilatorn eller dynamiskt med hårdvara. Detta leder till att mängden instruktioner som körs kan minskas med ungefär en fjärdedel.

2.4 Branch prediction

Branch prediction går ut på att processorn försöker förutse hur programflödet i en pipeline kommer att förgrenas vid t.ex. en villkorssats (if-else sats). Processorn läser då in den mest sannolika följande instruktionen och utför den med hjälp av spekulativ exekvering. Detta sker redan innan föregående instruktion är färdigt exekverad med följden att man sparar tid och ökar prestandan förutsatt att processorn valt rätt gren. Ifall fel gren har valts kasseras de spekulativt exekverade instruktionerna och pipelinen startas om med rätt gren [1].

Processorn använder sig av en Branch Target Buffer (BTB) som upprätthåller en mappning av adresser från nyligen exekverade branch-instruktioner till måladresserna. BTB:n används för att förutse framtida kodadresser innan branch-instruktionen avkodats [2]. Branch prediction började användas utbrett på 90-talet i t.ex. Intels Pentium processorer och används numera i de flesta högeffektiva processorer.

2.5 Spekulativ exekvering

Spekulativ exekvering eller spekulation baserat på hårdvara är en teknik som används i moderna processorer för att optimera prestanda. Genom att förutse programflödet med hjälp av branch prediction exekveras instruktioner i förväg. Syftet är att öka samtidigheten dvs. att köra så många instruktioner som möjligt samtidigt ifall resurser finns tillgängliga. Spekulativ exekvering fungerar så att processorn exekverar instruktioner innan den vet om de behövs med hjälp av ett system som förutser programflödet baserat på tidigare branch-prediktioner [3].

Denna teknik kan t.ex. användas när programflödet beror på data som inte är lagrat i cachen utan finns i det fysiska minnet. Istället för att vänta flera hundra klockcykler på att data hämtas, gissar processorn programflödets fortsättning, sparar en kopia av det nuvarande register-tillståndet och fortsätter att exekvera spekulativt [2]. När informationen processorn väntat på har hämtats från minnet kontrolleras gissningen som gjorts. Ifall gissningen var fel kasseras den spekulativa exekveringen och man går

tillbaka till den sparade kopian. Om gissningen däremot var korrekt används den spekulativa exekveringen och man får en markant prestandaökning.

2.6 Dynamisk exekvering

Dynamisk exekvering är en kombination av tre tidigare nämnda tekniker, branch prediction för att välja vilken instruktion som ska exekveras, spekulativ exekvering för att utföra instruktionerna förrän man vet om de behövs och dynamisk skedulering för att schemalägga olika kombinationer av block [3]. Dynamisk exekvering är en optimeringsteknik som introduceras av IBM på 1960-talet men började användas utbrett på 1990-talet i t.ex. Intels P6 arkitektur som lanserades 1995, P6 arkitekturen användes även som grund för arkitekturer som t.ex. Core och Haswell.

Tekniken baserar sig på Tomasulos algoritm som beskriver dynamisk skedulering av instruktioner, vilket i sin tur möjliggör dynamisk exekvering och effektivare användning av flera olika exekveringsenheter. Dynamisk exekvering gör det möjligt för en processor att undvika fördröjningar som uppstår när data saknas för att utföra en operation. Istället för att processorn är inaktiv så länge den väntar på att informationen hämtas, exekveras följande instruktion. Ifall instruktionen kan köras omedelbart av andra exekveringsenheter i processorn som inte är upptagna.

2.7 Minnesisolering

För att isolera processer från varandra använder processorer en virtuell minnesrymd där virtuella minnesadresser översätts till fysiska minnesadresser. Den virtuella adressrymden är indelad i en uppsättning av sidor som individuellt kan mappas till det fysiska minnet via en översättningstabell. Översättningstabellen som är i användning finns i ett specifikt register i processorn och när en annan process körs uppdateras registret med dess tabell så att varje process får en egen virtuell adressrymd. Detta görs för att säkerställa att en process endast kan få tillgång till data från den egna adressrymden [4]. Varje virtuell adressrymd är i sin tur uppdelad i en användare och en kernel del. Användardelen kan accesseras av programmet som körs medan kernelminnet endast kan accesseras av operativsystemet. Eftersom kerneln

måste utföra operationer på användarens sidor finns fysiskt minne mappat i kerneln, vilket kan utnyttjas för att få tillgång till privat information via en sido-kanal.

2.8 Sidokanalsattacker

En sidokanalsattack är en attack där information samlas in genom att observera ett system, t.ex. genom att mäta arkitektoniska egenskaper hos systemet [5]. Tekniker som t.ex. spekulativ exekvering förutser program-beteende för att förbättra prestanda. För att uppnå målet upprätthålls tillstånd som beror på tidigare exekveringar och det antas att framtida exekveringar är liknande eller relaterat till tidigare beteenden. När flera program körs på samma hårdvara endera samtidigt eller genom tidsdelning kan ändringar i tillstånd orsakat av ett program påverka andra program. Detta i sin tur kan resultera i informationsläckage från ett program till ett annat [2].

I Meltdown och Spectre används teknikerna Flush+Reload och Evict+Reload för att läcka information. I dessa attacker börjar angriparen med att kasta bort ett cacheblock som delas med offret från cachen. Efter att ha låtit offret exekvera en stund mäter angriparen hur lång tid det tar att utföra en minnesläsning till adressen som motsvarar det bortkastade cacheblocket. Om offret hade tillgång till det övervakade cacheblocket finns data i cachen och läsningen blir snabb. Däremot om offret inte hade tillgång till cacheblocket blir läsningen långsam. Således kan angriparen genom att mäta läsnings-tiden får reda på om offret hade tillgång till det övervakade cacheblocket mellan bortkastningen och läsningen [7].

Skillnaden mellan de två teknikerna är sättet på vilket det övervakade cacheblocket tas bort ur cachen. I Flush+Reload attacken används en maskininstruktion t.ex. x86 instruktionen *clflush* för att kasta bort blocket. Medans det i Evict+Reload attacken sker genom att tvinga fram en konflikt på cache-raden där blocket finns. Detta kan göras t.ex. genom att accessera andra minnesadresser som i sin tur blir sparade i cachen. Detta leder till att påföljande accesser blir bortkastade på grund av cachens begränsade utrymme.

3 Spectre

Det här kapitlet behandlar attacken Spectre och fokus ligger vid att förklara hur Spectre fungerar. Spectre är en attack där processorn blir ”lurad” att utföra en spekulativ exekvering som inte uppstår vid normal program-exekvering, detta leder till att en angripare kan få tillgång till privat information via en sido-kanal. På grund av att de flesta moderna processor använder sig av spekulativ exekvering är Spectre ett allvarligt hot [2].

I de flesta pall påbörjas attacken med en förberedande fas där angriparen utför operationer som gör det möjligt att senare utnyttja felaktiga spekulativa exekveringar. Fasen innehåller också steg som förorsakar spekulativ exekvering. Angriparen utför en målinriktad minnesläsning som får processorn att ge ut ett värde från cachen som krävs för att bestämma målet för en förgrening i programflödet. I denna fas förbereds även sido-kanalen som används för att utvinna offrets information, dvs. flush eller evict delen av en flush+reload eller evict+reload attack [7].

I den andra fasen exekverar processorn spekulativt instruktioner som leder till att privat information från offret överförs till en sidokanal. Det här kan förorsakas genom att angriparen gör en förfrågning till offret att utföra en specifik handling t.ex. att köra ett systemanrop. Påföljande spekulativa exekvering läser en minnesadress vald av angriparen och därefter utförs en minnesoperation som modifierar cachen så att ett värde avslöjas [2].

I den sista fasen utvinns den privata informationen genom att använda flush+reload eller evict+reload. Denna process mäter hur lång tid det tar att läsa från minnesadressen i cacheblocket som observeras.

3.1 Exploiting Conditional Branch Misprediction

3.2 Poisoning Indirect Branches

3.3 Åtgärder

4 Meltdown

Meltdown är en attack som gör det möjligt att övervinna minnesisolering genom att förse en process med ett sätt att läsa hela kernelminnet på maskinen som det körs på. Detta leder till att allt fysiskt minne mappat i kerneln också kan accesseras. Meltdown utnyttjar bieffekter hos dynamisk exekvering som används i många moderna processorer vilket leder till att Meltdown fungerar på de flesta operativsystem.

Meltdown består av tre steg:

Steg 1: Angriparen väljer en minnesadress som den inte kan komma åt, vars innehåll laddas in i ett register.

Steg 2: En access till ett cacheblock körs spekulativt baserat på innehållet i registret.

Steg 3: Angriparen använder Flush+Reload för att bestämma det accesserade cacheblocket och således den privata information som finns i minnesadressen.

Genom att upprepa dessa steg för olika minnesadresser kan angriparen generera en kernel minnesdump och en fullständig minnesdump [4].

4.1 Läsning av information

4.2 Överföring av information

4.3 Mottagning av information

4.4 Åtgärder

4.4.1 Hårdvara

4.4.2 KAISER

5 Diskussion

6 Källor

[1] Jann Horn. Project Zero. (3.1.2018) ”Reading privileged memory with a side-channel”. Tillgänglig: https://googleprojectzero.blogspot.fi/2018/01/reading-privileged-memory-with-side.html (Hämtad 14.2.2018)

[2] Paul Kocher. Et.al. (2017) ”Spectre Attacks: Exploiting Speculative Execution”. Tillgänglig: https://spectreattack.com/spectre.pdf (Hämtad 25.2.2018)

[3] John L. Hennessy. David A. Patterson. (2012). ”Computer Architecture A Quantitative Approach” Femte Upplagan. Morgan Kaufmann.

[4] Moritz Lipp. Et.al. (2017) ”Meltdown”. Tillgänglig: https://meltdownattack.com/meltdown.pdf (Hämtad 27.2.2018)

[5] Intel Corporation. (2018) ”Intel Analysis of Speculative Execution Side Channels”. Tillgänglig: https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf (Hämtad 27.2.2018)

[6] Nikolay A. Simakov. Et.al. (2018) ”Effect of Meltdown and Spectre Patches on the Performance of HPC Applications”. Tillgänglig: https://arxiv.org/pdf/1801.04329.pdf (Hämtad 27.2.2018)

[7] Yuval Yarom. Katrina Falkner. (2014) ”Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack”. Tillgänglig: https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-yarom.pdf (Hämtad 31.3.2018)