ASUPRA UNOR BREȘE DE SECURITATE ÎN MICROPROCESOARELE ACTUALE

Prof. univ. dr. ing. Lucian N. VINȚAN

Membru titular al Academiei de Științe Tehnice din România,
Universitatea "Lucian Blaga" din Sibiu,
E-mail: lucian.vintan@ulbsibiu.ro

REZUMAT. Acest articol analizează recent descoperitele vulnerabilităţi de securitate hardware, numite *Meltdown* şi *Spectre*, în cadrul unor microprocesoare comerciale actuale. Autorul a încercat să deducă şi să prezinte cauzele profunde, intrinseci, ale acestor vulnerabilităţi de proiectare hardware. Una dintre cauzele esenţiale constă în faptul că instrucţiunile tranzitorii – procesate *Out of Order* respectiv în mod speculativ de microprocesoarele superscalare actuale – pot aloca, din păcate, date privilegiate în cache-urile CPU. Aceste date secrete pot fi aflate ulterior de către un alt fir de execuţie al programului atacator. Autorul pune, în premieră, şi o problemă deschisă, anume dacă procesarea speculativă a unor instrucţiuni subsecvente celei care accesează cache-urile cu mapare directă, nu ar putea constitui o altă breşă de securitate în unele microprocesoare actuale.

Cuvinte cheie: Microprocesor superscalar, Procesare *Out of Order*, Predicția *branch*-urilor, Procesare speculativă, Memorie virtuală, Mecanisme hardware de asigurare a securității, Memorii *cache*

ABSTRACT. This paper analyzes the recent discovered *Meltdown* and *Spectre* present-day hardware commercial CPU security vulnerabilities. The author tries to understand and present the profound intrinsic causes of these hardware vulnerabilities. One main cause consists in the fact that transient instructions - out of order and speculative executed dynamic instructions - processed by the superscalar processors, unfortunately can allocate privileged data in the CPU's cache. Particularly, user's speculative instructions involve exception suppression. They can allocate privileged data in the CPU's cache, too. After this cache access, the secret accessed data can be learned through an attacker's thread. The author firstly emphasized an open problem, too: in the case of a L1 direct mapped cache instruction access (for example, Pentium 4 case), the corresponding subsequent speculative instructions could facilitate another CPU insecurity niche?

Keywords: Superscalar Microprocessor, Out of Order Instructions Processing, Branch Prediction, Speculative Processing, Virtual Memory, Hardware Security Mechanisms, Cache Memories

1. INTRODUCERE

Breşele de securitate ale microprocesoarelor actuale, recent semnalate public, sunt datorate, în principal, unor erori în proiectarea interfeței dintre memoria virtuală (pe funcția sa de asigurare a protecției între procesele care rulează în spectrul de privilegiu *kernel-user*) / sub-sistemul ierarhic de *cache*-uri, pe de-o parte, și procesarea speculativă (*Spectre Attacks Problem*) respectiv *out-of-order* (*Meltdown Problem*) a instrucțiunilor (prin mecanisme de tipul *Dynamic Branch Prediction,* algoritmi tip *Tomasulo, Reorder Buffer* etc.), pe de altă parte. *Meltdown* [1] („criză", „catastrofă", "accident nuclear", „topire") a fost semnalat în premieră de cercetători din mediul academic și industrial și exploatează procesarea *out of order* a instrucțiunilor în microprocesoarele superscalare actuale, pentru a accesa locații de memorie accesibile în mod normal doar în modul de lucru privilegiat. Procesarea *out of order* a instrucțiunilor permite unui proces *user* să citească date dintr-o pagină de memorie accesibilă numai în modul *kernel*. Breșa de securitate se bazează pe erori în proiectarea hardware a acestor procesoare și nu pe erori ale sistemelor de operare sau pe alte vulnerabilități software. Aceste erori afectează, spre exemplu, toate procesoarele INTEL produse din anul 2010 încoace, dar și procesoarele ARM și AMD, adică miliarde de sisteme de calcul la nivelul anului 2018, de la telefoane inteligente la cloud - servere. Această afirmație este credibilă, având în vedere că peste 80% din telefoanele inteligente conțin procesoare ARM.

Vulnerabilitatea *Spectre* ("fantomă", „fantasmă", „nălucă") se bazează pe execuțiile speculative ale instrucțiunilor, procesate în urma predicției eronate a unei instrucțiuni de salt condiționat (*branch*) [2]. Ca și *Meltdown*, se datorează tot unor erori de proiectare hardware ale procesoarelor actuale, fiind descoperită de aceiași cercetători. Independent de aceștia, o echipă de cercetători de la compania *Google* a descoperit și a semnalat erori similare, atât pentru procesarea out of order cât și pentru cea speculativă a instrucțiunilor [3]. Nu degeaba, de câțiva ani, compania *Google* își dezvoltă propriile

microprocesoare. De câțiva ani, țări precum China, Japonia, Rusia, India își dezvoltă, de asemenea, propriile microprocesoare de uz general [14]. În anul 2018, Comisia Europeană a startat programul numit *European Processor Initiative* prin care se dorește dezvoltarea unor procesoare de uz general și a aceleratoarelor aferente pentru viitoarele super-computere europene (*EuroHPC Machine* – 10k-50k procesoare). Este pentru prima dată în istoria sistemelor de calcul când se publică vulnerabilități de securitate datorate proiectării hardware a micro-procesoarelor și nu unor erori în proiectarea software sau încălcării unor cerințe software de securitate.

Scopul acestui articol este de a prezenta și analiza în mod critic principiile fundamentale ale unor asemenea breșe de securitate, la nivelul de înțelegere al unui cititor având cunoștințe de bază în domeniul sistemelor de calcul, dar fără a fi, neapărat, un specialist în acest domeniu. De asemenea, autorul a încercat ca, în urma analizei desfășurate, să extragă anumite concluzii personale referitoare la cauzele erorilor hardware de proiectare care implică vulnerabilitățile de securitate menționate și chiar să pună unele probleme deschise.

2. CUNOȘTINȚE DE BAZĂ NECESARE ÎNȚELEGERII VULNERABILITĂȚILOR

Procesarea *Out of Order* a instrucțiunilor în cadrul actualelor procesoare superscalare se bazează pe faptul că o instrucțiune poate starta execuția propriu-zisă de îndată ce valorile operanzilor sursă devin disponibile. Aceasta implică procesarea instrucțiunilor în afara ordinii lor secvențiale (*In Order*), dată de programul static. Spre exemplu, o instrucțiune rapidă, situată în program după o instrucțiune mare consumatoare de timp (acces la memorie cu *miss* în *cache*, înmulțire, împărțire etc.) și independentă de aceasta, se va executa înaintea ei, într-un asemenea procesor. Procesarea *Out of Order* a instrucțiunilor se face prin algoritmi de tip Tomasulo. **Bufferul de reordonare (*Reorder Buffer - ROB*)** este o structură hardware de tip FIFO circular, implementată în cadrul procesoarelor superscalare, în vederea implementării execuţiei *Out of Order* a instrucțiunilor-mașină, prin redenumire dinamică a resurselor hardware precum și a implementării unui mecanism precis de tratare a excepţiilor (întreruperilor). **Staţia de rezervare (SR)** este o structură de date implementată în hardware în cadrul procesoarelor superscalare, cu scopul implementării procesării *Out of Order* a instrucţiunilor. Lansarea în execuţie a unei instrucţiuni rezidente în bufferul de *pre-fetch* (*dispatch*) înseamnă mutarea acesteia în SR aferentă. De aici, instrucţiunea se lansează efectiv în execuţie (*issue*) atunci când valorile operanzilor săi sursă sunt disponibile. Prin SR și structura de date de tip ROB se efectuează o redenumire dinamică a regiştrilor, în vederea eliminării dependențelor de date de tip *Write After Read* şi *Write After Write*. De asemenea, aceste staţii accelerează execuţia programului, prin captarea agresivă a unor rezultate aşteptate de către instrucţiunile dependente *Read After Write*, aflate în stare de aşteptare în staţiile de rezervare (*forwarding, bypassing*). **Predicţia dinamică a ramificaţiilor** de program (*branches*) reprezintă procesul de predicţie, desfăşurat *run-time*, pe parcursul fazei de aducere a instrucţiunii, a următoarelor trei informaţii: dacă instrucţiunea adusă este una de ramificaţie sau nu (1), în caz că este ramificaţie, dacă se face - *Taken* sau nu - *Not Taken* (2), iar dacă se face - adresa destinaţie (*target*) la care se va face (3). Predicţia ramificaţiilor este foarte necesară în procesoarele (super –) *pipeline* – izate şi superscalare. Predictoarele de ramificație actuale sunt de tipul markovian (*Two Level Adaptive Branch Predictors*) respectiv neuronal (perceptroane simple) [4], [5]. Primul predictor neuronal de branch-uri a fost propus chiar de autorul acestui articol, în lucrarea [4]. Această lucrare a fost citată în peste 60 de articole ale unor prestigioşi autori din străinătate (până în anul 2017), inclusiv în celebrul *Meltdown Paper* [1], analizat aici. Tot autorului acestui articol i se datorează și primul predictor de *branch*-uri de tip perceptron, publicat în lucrarea [13]. Ulterior, predicția neuronală a *branch*-urilor a fost implementată în microprocesoare comerciale, precum *Oracle SPARC T4-4* (2011), *AMD Bobcat/Jaguar* (2014), *Samsung Exynos M1* (2016), *AMD Ryzen* (2017) etc. În procesoarele *INTEL* actuale instrucțiunile mașină sunt convertite automat, pe timpul rulării (faza de decodificare), în instrucțiuni de tip RISC (*Reduced Instruction Set Computer*) numite și *RISC like OPerations* (ROP) sau *micro-operations* (μOPs), care sunt mai departe procesate *out of order* prin algoritmi de tip *Tomasulo*. Preluată din lucrarea autorului acestui articol [6], Figura 1 reprezintă schema bloc a unui procesor superscalar cu procesări *Out of Order* a instrucțiunilor.

Memoria cache reprezintă un concept arhitectural care desemnează o memorie de capacitate mai mică decât cea a memoriei principale, care este mai rapidă şi mai scumpă per octet decât memoria principală, situată din punct de vedere logic între procesor şi aceasta. A fost inventată de prof. *Maurice Wilkes*, de la Universitatea din Cambridge, în anul 1965. Memoria cache este gestionată astfel încât probabilitatea statistică de accesare în cache a unei date de către procesor (rata de *hit*), să fie cât mai mare. Se diminuează deci timpul mediu de acces la memoria principală. Astfel, se reduce din "prăpastia semantică" între viteza de procesare tot mai mare a microprocesoarelor şi respectiv latenţa

prea ridicată a memoriilor principale (DRAM) actuale. Memoriile cache sunt realizate în tehnologii SRAM de mare performanţă, fiind, uzual, integrate în procesor.

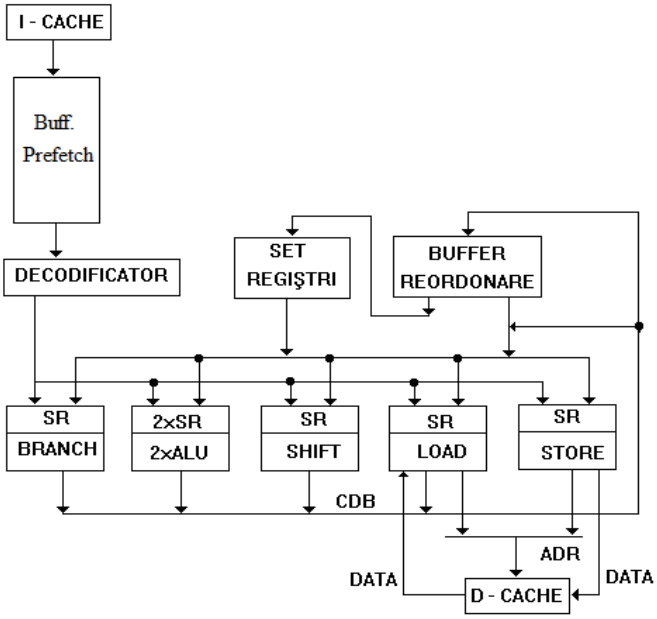

Figura 1. Schema bloc a unui microprocesor superscalar.

Memoria virtuală (MV) reprezintă o tehnică de organizare a memoriei prin intermediul căreia programatorul dispune de un spaţiu virtual de adresare de capacitate foarte mare (mai mare decât capacitatea memoriei principale) şi care, fără ca programatorul să "simtă", este mapat dinamic în memoria principală. Uzual, spaţiul virtual de adrese corespunde suportului disc magnetic / SSD (memoria secundară), programatorul având impresia, prin mecanismele de memorie virtuală, că are la dispoziţie o memorie unică, de capacitatea hard-discului şi nu de capacitatea, mult mai redusă, a memoriei principale. De asemenea, MV oferă funcţii de protecţie a accesului la memorie, prin implementarea unor drepturi de acces la memorie ale programelor. În urma comparării acestor drepturi cu nivelul de privilegiu al programelor aflate în rulare, acestea primesc, sau nu, accesul la o anumită informaţie stocată în memorie [6]. Adresele de memorie virtuale emise de procesor sunt mapate în adrese fizice de memorie. Un spaţiu virtual de memorie, mapat în memoria secundară (disc magnetic, SSD - *solid-state drive*), este împărţit în pagini, de lungime fixă (uzual 4 Ko). Maparea acestor pagini în memoria fizică (principală) se face prin intermediul unei aşa numite tabele de pagini (PT – *Page Table*), situată în memoria DRAM. Prin intermediul acestei tabele se face translatarea adresă virtuală – adresă fizică. Tot această tabelă conţine şi informaţii care asigură securitatea paginilor de memorie, prin diferite nivele de privilegiu numite şi drepturi de acces (*Read Only*, *Read/Write*, Accesibil/Neaccesibil *User* etc.) Tabela de pagini utilizată curent este pointată, prin adresa sa de bază, de un registru special (*Page Table Register* – PTR). Prin paginare, fiecare acces la o dată din memorie necesită două accese la memoria principală (DRAM): unul pentru obţinerea adresei fizice din tabela de pagini, iar celălalt pentru accesarea propriu-zisă a datei în memoria principală, cu adresa fizică anterior aflată. În vederea reducerii acestui timp de acces dublu, tabela de pagini este deseori "caşată" (memorată parţial) în CPU. Memoria cache care memorează maparea tabelei de pagini se numeşte TLB (*Translation Lookaside Buffer*). Dar o tabelă de pagini poate fi memorată şi în ierarhia de cache-uri, exact ca oricare altă locaţie din memoria principală. Cum fiecare proces (*task*) deţine propriul spaţiu virtual de memorie (fiecare proces poate accesa date din propriul său spaţiu virtual), la fiecare comutare de task-uri sistemul de operare (rulând, evident, în modul privilegiat) încarcă noul conţinut al registrului PTR, pentru a pointa spre tabela de pagini corespunzătoare noului proces activ. Spaţiul virtual de memorie al fiecărui proces este împărţit într-un sub-spaţiu *kernel* şi un altul *user*. Sub-spaţiul kernel nu poate fi accesat decât de un proces rulând în modul kernel de privilegiu. Această restricţie este implementată de sistemul de operare prin manipularea corespunzătoare a bitului Accesibil/Neaccesibil *User* din tabela de pagini [4], [5].

Securitatea sistemelor de calcul are la bază funcţia de protecţie a memoriei virtuale. Aceasta asigură izolarea zonelor de memorie aferente unor task-uri diferite. Spre exemplu, o pagină de memorie accesibilă doar programelor privilegiate (rulând în modul kernel) este protejată prin hardware de accesul unui program utilizator (rulând în modul user), întrucât poate conţine date privilegiate (parole, chei criptare etc.) Sistemul de operare, bazat pe această funcţie de protecţie implementată în hardware, asigură izolarea spaţiilor de memorie între task-uri, atât la nivelul user – user cât şi la nivelul user – kernel (un task user să nu poată accesa zona de memorie a altuia, privilegiat). Astfel se asigură procesarea multi-tasking în sistemele de calcul. Izolarea zonelor de memorie se poate face, în principiu, printr-un bit din procesor (CPU – *Central Processing Unit*) care permite (1) / interzice (0) accesarea unei pagini kernel. Un asemenea bit nu poate fi setat decât de un program privilegiat. Acesta este automat resetat la tranziţia în modul utilizator, care se face prin intermediul unor întreruperi.

3. BREȘA DE SECURITATE *MELTDOWN*

În Figura 2 se arată un exemplu care pune în evidenţă vulnerabilitatea *Meltdown* a procesoarelor superscalare cu execuţii out of order ale in-

strucțiunilor, prin care atacatorul poate accesa zona de memorie a oricărui *task*. Ideea este ca în cadrul unui program user să se provoce un eveniment de excepție, printr-o instrucțiune declanșatoare (*trigger*) corespunzătoare (spre exemplu, o instrucțiune de acces într-o pagină privilegiată din memoria fizică). În mod normal, această declanșare a excepției va determina intrarea într-o rutină de tratare a acesteia (Rutină Tratare Excepție - RTE), care va încheia programul *user* datorită excepției de violare de privilegiu. În consecință, instrucțiunile de după instrucțiunea trigger nu ar mai trebui să se proceseze, din punct de vedere teoretic. Instrucțiunile din cadrul RTE (componentă a sistemului de operare) se vor procesa în modul privilegiat. Totuși, datorită procesării *Out of Order* (OoO) a instrucțiunilor, este posibil ca unele dintre instrucțiunile situate după instrucțiunea trigger și independente de date de aceasta să se fi procesat deja sau să fie în curs de procesare, eventual chiar în paralel cu procesarea unor instrucțiuni din RTE. Chiar și instrucțiuni subsecvente, dependente *Read After Write* de instrucțiunea trigger, pot să starteze execuția, înainte ca aceasta să se fi încheiat (faza ultimă, *Commit*). Aici apare o vulnerabilitate esențială a procesoarelor actuale! Să presupunem că aceste instrucțiuni, situate imediat după instrucțiunea trigger, utilizează data accesată de instrucțiunea trigger. Această utilizare va lăsa "urme" în resursele hardware ale microarhitecturii, după cum se va arăta în continuare, iar data privilegiată accesată de către instrucțiunea trigger, va putea fi determinată. Meltdown exploatează execuția out of order a instrucțiunilor user pe durata scurtei ferestre de timp între accesul ilegal la memorie al instrucțiunii trigger și acțiunea de terminare a programului user, declanșată prin RTE aferentă (kernel).

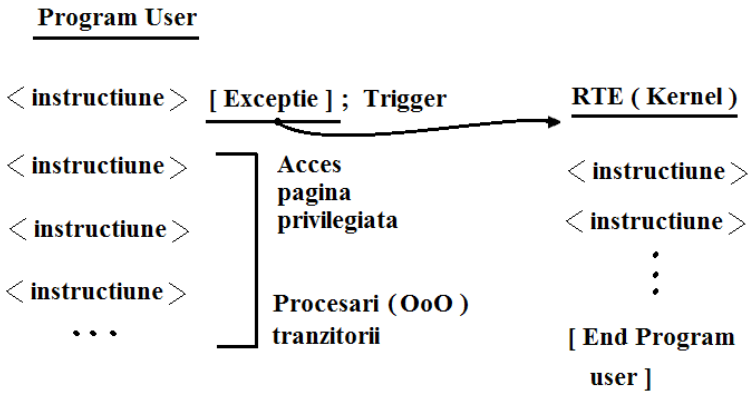

Figura 2. Principiul evidențierii breșei de securitate *Meltdown*

Desigur, datorită bufferului de reordonare care impune încheierea (*Commit*) *In Order* a instrucțiunilor, pentru asigurarea unui mecanism precis de tratare a excepțiilor, aceste instrucțiuni executate OoO nu vor afecta starea arhitecturală a procesorului (regiștrii logici, manipulabili prin instrucțiunile mașină din cadrul ISA – *Instruction Set Architecture*). Din păcate, pe parcursul procesării OoO a intrucțiunilor de după instrucțiunea trigger a excepției, numite instrucțiuni tranzitorii (*transient instructions*), conținutul locației accesate din pagina de memorie privilegiată a fost încărcat într-un cache de date, de unde va putea fi aflat, într-un mod indirect, de către atacator. În opinia autorului acestui articol, posibilitatea de a citi date privilegiate în cache prin instrucțiuni tranzitorii dintr-un program user se datorează unor erori hardware în proiectarea CPU. Dacă instrucțiunile tranzitorii din programele user nu ar afecta cache-ul de date, cel puțin pe accesul la date privilegiate din memorie, probabil că Meltdown nu ar putea acționa. Demn de subliniat, autorii [1] abordează doar problema citirii unor date privilegiate prin programe user, nu și pe cea a scrierii. Este posibil că acest fapt să se datoreze unor motive deontologice. Rămâne deocamdată o problemă deschisă posibilitatea scrierii în pagini privilegiate, prin instrucțiuni tranzitorii sau/și speculative. Pentru a se deduce valoarea acestor date privilegiate (secrete) aduse în cache și a le face vizibile, se poate folosi un atac din categoria așa numitelor "*cache side-channel*", spre exemplu unul de tipul *Flush+Reload*. Prin monitorizarea timpului necesar pentru a citi anumite date din memorie, atacatorul își poate da seama dacă data respectivă a fost reîncărcată între timp în cache de către un alt proces (fir de execuție), aparținând atacatorului. În caz afirmativ, se poate deduce valoarea datei secrete (v. în continuare). Așadar, atacurile de tip "*cache side-channel*" exploatează diferențele de timing în accesarea ierarhiei de memorie, introduse de cache-uri. Spre exemplu, tehnica de atac numită *Flush+Reload* (F+R) invalidează o anumită linie (bloc) din întreaga ierarhie de cache-uri, printr-o instrucțiune x86 de tip *clflush*. Dacă secvența din Figura 2 se iterează într-o buclă, în fiecare iterație a acesteia se poate accesa pagina următoare. Această adresare dintr-o pagină în următoarea se poate face prin intermediul RTE, care apelează din nou secvența user, incrementând înainte

registrul pointer de memorie cu 1000H, pentru că o pagină are capacitatea de 4 K octeți. Astfel, se accesează o locație din pagina următoare. Așadar, deși accesul la informația dintr-o pagină privilegiată nu ar trebui să poată avea loc dintr-un program user, acest fapt se întâmplă datorită procesării OoO a instrucțiunilor care, iată, pot utiliza data de la acea locație privilegiată în mod abuziv, datorită unor erori de proiectare.

Conform [1], exploatarea execuției OoO a instrucțiunilor tranzitorii pentru a accesa date privilegiate (kernel) printr-o aplicație utilizator de tip Meltdown trebuie să aibă în componență blocurile de acțiuni prezentate în Figura 3. Mai întâi, trebuie să se declanșeze un eveniment de excepție datorat accesării unei date privilegiate din memorie, urmat de instrucțiuni tranzitorii (procesate OoO), care să utilizeze acele date privilegiate din memorie, în vederea identificării acestora. Evident, un asemenea acces la date privilegiate este abuziv. În mod normal, excepția declanșată prin RTE aferentă ar fi trebuit să încheie programul user (atacator). Din păcate, aceste instrucțiuni tranzitorii vor apuca să modifice starea microarhitecturii (spre ex. conținutul cache-ului de date), care va memora, într-un mod indirect, datele privilegiate accesate prin instrucțiunea trigger. Acestea erau stocate în pagini de memorie neaccesibile utilizatorului în mod normal, ci doar programelor kernel. Evident, deocamdată aceste date nu sunt disponibile în starea arhitecturală (logică) a CPU. Facem distincție între resursele microarhitecturale ale CPU, care sunt invizibile la nivelul instrucțiunilor mașinii și deci la nivelul programatorului (user) și resursele arhitecturale, logice, care sunt vizibile și manipulabile prin instrucțiuni mașină de către programator (atacator). Urmează o secvență de instrucțiuni care va transfera datele secrete în resursele hardware (regiștri logici, memorie) ale procesului utilizator, printr-un proces ingenios. Acest transfer se poate realiza, spre exemplu, prin bifurcarea (*fork*) aplicației user în două părți (fire de execuție, *threads*): un fir „părinte” – care va transfera datele secrete din resursele microarhitecturii în cele logice ale CPU, prin deducții ingenioase, și un fir „copil” – singurul care va conține instrucțiunile tranzitorii, de utilizare abuzivă a datelor privilegiate accesate anterior de instrucțiunea trigger (*fork-and-crash*). Aceste două fire de control trebuie create înainte de instrucțiunea trigger a excepției, care, de altfel, va și încheia procesarea unuia din aceste fire, anume a firului copil, prin RTE aferentă. CPU va procesa totuși instrucțiunile tranzitorii din firul copil, înainte de încheierea acestuia prin intermediul RTE. Firul părinte va continua procesarea și va restaura datele secrete din starea microarhitecturii (cache-ul de date), memorându-le în resursele logice, vizibile, ale CPU

Restaurarea datelor din resursele microarhitecturii în cele vizibile ale CPU se poate face prin tehnici de tip *Flush+Reload*. După ce instrucțiunile tranzitorii au procesat data citită dintr-o pagină privilegiată, aceasta este cașată într-un mod indirect. Atacatorul (firul părinte în cazul metodei *fork-and-crash*) poate afla prin tehnici de tip *Flush+Reload* valoarea datei privilegiate (secrete), citită de instrucțiunea trigger..

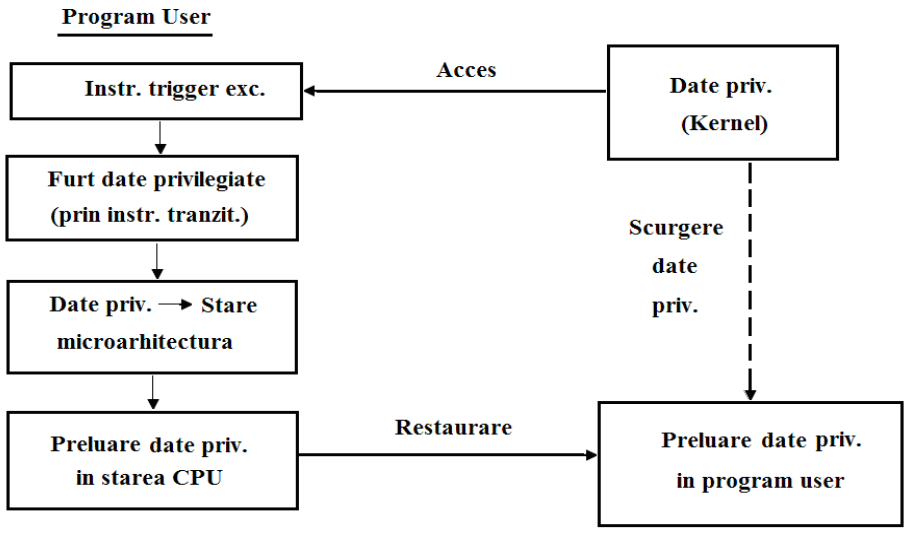

Figura 3. Schema logică a preluării datelor secrete prin breșa de securitate *Meltdown*

Așadar un atac de tip *Meltdown* constă în 3 faze succesive:

1. Se accesează și se încearcă (fără succes) încărcarea într-un registru logic al CPU, printr-o instrucțiune trigger, a unei anumite locații de memorie privilegiată, aleasă de atacator (user).

2. O instrucțiune tranzitorie subsecventă, procesată evident OoO, accesează o anumită locație

din cache-ul de date, pe baza datei secrete încărcate anterior în CPU, pe post de pointer (adresă) de memorie.

3. Atacatorul declanşează o metodă de tipul F+R (printr-un fir de control separat) pentru a afla linia (blocul) accesată din cache şi, implicit, secretul memorat la locaţia privilegiată aleasă de atacator.

Prezentăm acţiunile unui atac Meltdown pe baza unui exemplu concret, preluat din [1], pe care autorul acestui articol l-a simplificat şi l-a adaptat, în vederea unei mai bune înţelegeri a esenţei acţiunii sale. Să considerăm aşadar secvenţa de program *assembler x86* de mai jos, care rulează neprivilegiat (user):

rcx = conţine o adresă de memorie situată într-o pagină privilegiată.

rbx = conţine adresa de bază a unui aşa numit tablou de probă (TP), situat în memoria utilizator.

1. ***mov al, byte [rcx];*** În reg. acumulator *al* se doreşte introducerea valorii secrete *s* (pe un octet). Instrucţiunea aceasta va declanşa o excepţie (violare de privilegiu).

2. ***shl rax, 0xC;*** Înmulţeşte *s* cu 2, de 12 ori. Reg. *rax* (de fapt o locaţie din bufferul de reordonare, nu chiar registrul arhitectural *rax*) va conţine $s*2^{12} = s*1000H=s*4096$.

3. ***mov rbx, qword [rbx + rax];*** Caşează implicit un bloc de date situat în pagina numărul *s* a tabloului de probă! Este singurul bloc de date din tabloul de probă, caşat! (Rezultatul nu va ajunge în *rbx*.)

Tabloul de probă (TP)

PG. 1 (4 KO)	PG. 2 (4 KO)	...	PG. *S* (4 KO)	...	PG. 256 (4 KO)

Se consideră că tabloul de probă are o lungime de 256x4096 octeţi, deci conţine 256 de pagini succesive de memorie. Evident, acesta este situat în memoria utilizator, fiind creat de atacator. De asemenea, se consideră că nicio locaţie din cadrul acestui tablou nu este caşată (programatorul atacator se poate asigura, şi se va asigura de acest fapt). Instrucţiunea 1 face un acces indirect registru (pointer *rcx*) la o locaţie de memorie privilegiată, deci un acces ilegal. Această acţiune va declanşa o excepţie de tip violare de privilegiu (dar mai târziu, în faza ultimă, cea numită *Commit*, a instrucţiunii), pentru că nivelul de privilegiu curent al acestui program user nu permite accesul la o pagina *kernel*. Datorită procesării *Out of Order*, implementată prin algoritmi de tip Tomasulo, anumite faze intermediare ale instrucţiunilor tranzitorii 2 şi 3 se procesează practic în paralel cu ultimele faze ale instrucţiunii 1. Instrucţiunile tranzitorii 2 şi 3, aflate în staţii de rezervare în urma fazei de decodificare şi *dispatch* (mai exact, micro-instrucţiunile RISC în care aceste instrucţiuni x86 au fost convertite automat în faza de decodificare, dar uneori exactitatea excesivă tulbură înţelegerea adevărului...) preiau data privilegiată (secretă, pe un octet) de pe *Common Data Bus*-ul CPU [6], dată pusă aici de către instrucţiunea 1, imediat după faza de citire din memoria de date. Astfel, aceste două instrucţiuni (2, 3) se vor executa cvasi-simultan. După ce instrucţiunea 1 se încheie (în faza sa ultimă, numită *Commit*, realizând apariţia violării de privilegiu, se va încheia fără să scrie valoarea *s*, situată în bufferul de reordonare, în registrul *rax*), se declanşează o excepţie de tip violare de privilegiu, care goleşte resursele microarhitecturii (structuri *pipeline*, buffer de reordonare etc.) pentru a elimina rezultatele instrucţiunilor tranzitorii (2 şi 3) şi a altora subsecvente, procesate OoO. Din nefericire, este posibil ca instrucţiunea tranzitorie 3 să fi caşat deja locaţia de memorie de la adresa *[rbx + rax]*, înainte de încheierea instrucţiunii 1 (care se face în faza sa ultimă, numită *Commit*, când şi verifică dacă a generat o excepţie; va constata că a generat, dar prea târziu!). Din păcate, instrucţiunile tranzitorii, ca şi cele speculative de altfel, pot să aloce în cache date privilegiate şi apoi să le utilizeze prin intermediul altor instrucţiuni user. Aceasta reprezintă o altă eroare majoră de proiectare a microprocesoarelor actuale.

Instrucţiunea 2 înmulţeşte valoarea secretă *(s)* citită de instrucţiunea 1 cu constanta 2^{12} (capacitatea unei pagini de memorie, 4 K octeţi = 4096 octeţi), registrul acumulator *rax* conţinând acum *s*1000H* (mai exact, nu chiar registrul *rax* va conţine această valoare, ci locaţia corespunzătoare din bufferul de reordonare. Registrul *rax* a fost redenumit în faza de decodificare a instrucţiunii cu locaţia din coada bufferului de reordonare, conform [6], Par. 4.4. Pentru simplificarea expunerii însă, vom considera că valoarea respectivă este disponibilă în registrul arhitectural *rax*.) Astfel, *[rbx + rax]=[rbx + s*1000H]* va pointa, prin instrucţiunea 3, la o dată din pagina numărul *s* din cadrul tabloului de probă. În instrucţiunea 3 se citeşte un cuvânt din tabloul de probă situat la adresa *[rbx + s*1000H]*, adică în pagina numărul *s* a tabloului. (Din nou, pentru exactitate, rezultatul acestei instrucţiuni speculative nu ajunge în *rbx*, ci doar în locaţia din bufferul de reordonare corespunzătoare lui *rbx*.) Practic, valoarea secretă *s* a devenit, din dată, o adresă de memorie. Rolul acestei instrucţiuni 3 este exclusiv de a aloca în cache blocul aferent datei citite din pagina numărul *s* a tabloului de probă. Va fi singurul bloc de date caşat din tot tabloul de probă! Acum, spre exemplu, programul atacator poate citi, printr-

un alt fir de control (care rulează, evident, tot la nivel utilizator), utilizând eventual o metodă de tipul *Flush+Reload*, în mod succesiv, primul bloc al fiecăreia dintre cele 256 de pagini ale tabloului de probă (prin citirea primului bloc din fiecare pagină, octet cu octet, adică utilizând un pointer la memorie de genul *[rbx + k*1000H], k=0, 1, 2, ..., 255*, cores-punzător fiecărui bloc citit). Aşadar, cele 256 de accese (citiri) la blocurile din tabloul de probă sunt "împrăștiate" din 4 în 4 Ko în spațiul adreselor de memorie. Din cele 256 de accese succesive la aceste 256 de blocuri (câte unul la începutul fiecărei pagini din cadrul *TP*), doar unul va dura mai puțin (cel care a generat *hit* pe citirea primului octet din respectivul bloc). Prin măsurarea timpului de acces aferent fiecăruia dintre cele 256 de accese la aceste 256 de blocuri din tabloul de probă, atacatorul poate afla numărul paginii care a generat *hit* în cache (acest acces va fi cel mai scurt, desfășurându-se 100% cu *hit* în cache; celelalte 255 de accese la blocurile TP, fiind cu câte un *miss* în cache – pe primul octet aferent blocului accesat, vor dura semnificativ mai mult). Numărul paginii din tabloul de probă care generează *hit* în cache (cel mai scurt timp de acces) reprezintă tocmai valoarea datei secrete (*s*)! Iterând secvența de instrucțiuni 1-3 pentru diferite valori conținute în registrul *rcx*, atacatorul poate citi orice locație privilegiată (*memory dumping*). De remarcat că instrucțiunile tranzitorii nu afectează starea arhitecturală a CPU, adică regiștrii săi logici, ceea ce este corect. În ciuda acestui fapt, prin "urmele" lăsate în cache-ul de date de către aceste instrucțiuni, valoarea secretă s-a putut afla.

4. BREȘA DE SECURITATE *SPECTRE*

O altă metodă pentru furtul datelor privilegiate prin intermediul unor programe utilizator (care, evident, pot să apeleze şi rutine *kernel*), se bazează pe execuția speculativă a instrucțiunilor de pe ramura eronat predicționată (în mod intenționat!) a unei instrucțiuni de salt condiționat (*branch*). Astfel, aceste instrucțiuni speculative (tranzitorii şi ele) din cadrul atacului utilizator vor accesa, spre exemplu, date protejate dintr-o pagină privilegiată de memorie. Să presupunem că instrucțiunile acestea se vor executa imediat după ce condiția de salt a fost predicționată pe valoarea logică *True*. Dar atacatorul se va asigura că această condiție de salt nu va putea avea niciodată această valoare de adevăr în momentul atacului (ea, în realitate, va avea întotdeauna valoarea *False* pe parcursul procesării atacului). Instrucțiunile procesate speculativ de pe ramura *True* vor citi date inaccesibile în mod normal dintr-o pagină privilegiată (similar cu cele *Out of Order* din cazul *Meltdown*) şi nu vor declanșa nicio excepție, din păcate, fiind doar speculative (*Exception Suppression*). Aceste instrucțiuni speculative nu vor ajunge niciodată în faza finala a procesării lor (*Commit*), fază în care se ia în considerare o eventuală excepție, din cauza faptului că, odată constatată predicția eronată, acțiunile acestor in-strucțiuni se anulează (prin procesul *branch recovery*). Din păcate, ele au reușit deja să încarce date secrete în *cache*! Interzicerea execuției speculative a instrucțiunilor de citire din memorie ar putea fi o soluție (pentru că instrucțiunile de scriere în memorie nu sunt speculative şi nici *OoO*, din motive evidente). Apoi, după ce CPU va constata că a predicționat eronat *branch*-ul, va declanșa un proces de restaurare a stării de dinainte de predicție (*branch recovery* – din păcate nu va invalida şi locațiile afectate din *cache*-ul de date prin in-strucțiuni de citire din memorie) şi apoi va procesa instrucțiunile de pe ramura *False*. Acestea vor prelua datele secrete din resursele microarhitecturii şi le vor transfera în starea vizibilă a procesorului, similar cu ceea ce făcea firul părinte din strategia *fork-and-crash*, anterior descrisă. Figura 4 prezintă într-un mod sugestiv aceste acțiuni.

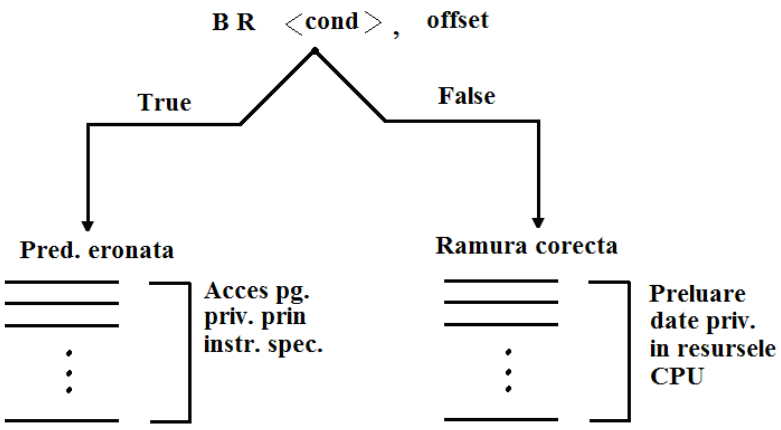

Figura 4. Principiul evidențierii breșei de securitate *Spectre* (*Exception Suppression*).

Modul detaliat de acțiune al atacurilor bazate pe accesarea unor resurse privilegiate ale CPU de către instrucțiunile speculative sunt prezentate într-un alt articol de mare impact, inclusiv emoțional, anume [2]. Așadar, o altă eroare majoră de proiectare constă în faptul că este posibil ca instrucțiunile user speculative să citească date privilegiate, fără declanșarea unei excepții. Faptul că nu se declanșează excepția este datorat instrucțiunilor speculative, care nu apucă să se încheie (faza *Commit*) din cauza procesului de *branch recovery*, care le anulează acțiunile, deși au reușit să încarce date secrete în *cache*. Excepțiile se pot declanșa doar în faza *Commit* în implementările actuale, așa că eroarea aceasta este, oarecum, „de înțeles".

Preluat din [2], prezentăm principiul unui atac de tip *Spectre*, bazat deci pe execuția speculativă a instrucțiunilor aferente unui branch predicționat eronat. Să considerăm secvența de program de mai jos, aparținând unui program *kernel* (funcție de bibliotecă, spre exemplu *syscall – call gate*), apelabil de către utilizator. (Același exemplu este dat și de cercetătorii de la *Google* în lucrarea [3].)

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

În mod normal, o valoare a lui *x* în afara limitei impuse ar declanșa o excepție. Din păcate, acest fapt nu se întâmplă dacă accesul se face prin instrucțiuni speculative. Să considerăm că *array1[x]* este adresa de memorie a unui octet secret *s* (dată privilegiată), situat în memoria *kernel*. Atacatorul determină în prealabil ca variabilele *array1_size* și *array2* (adresa de bază a tabloului de 64 Kocteți numit *array2*) să nu fie prezente în *cache*, dar octetul *s* să fie prezent, ceea ce este posibil conform [2], pentru că funcția *kernel* respectivă poate fi determinată să acceseze octetul *s* în mod legal. Valorile anterioare ale lui *x* au fost furnizate de atacator așa încât condiția să fi fost evaluată pe adevărat, de mai multe ori succesiv. Astfel, s-a antrenat predictorul de *branch*-uri ca să prezică ramura *True*, cea pe care se face asignarea variabilei *y*. Citirea variabilei *array1_size* se va face cu *miss* în *cache*, deci citirea acesteia din DRAM va dura multe zeci (chiar sute) de tacte CPU. În momentul în care atacatorul furnizează brusc o valoarea a lui *x* în afara domeniului *array1_size*, predictorul de *branch*-uri va predicționa eronat valoarea de adevăr a condiției (pe *True*, din inerție!) și, în consecință, CPU va procesa speculativ asignarea *y = array2[array1[x] * 256]*. Prin urmare, se citește rapid, cu hit în cache, valoarea octetului secret *s* de la adresa *array1[x]*. Apoi, procesarea speculativă va declanșa o altă citire din memorie, de data aceasta "lungă", de la adresa *array2[s * 256]*, doar pentru ca data respectivă să se aloce în cache, în urma *miss*-ului! Cât timp se va accesa DRAM-ul în vederea acestei citiri consumatoare de timp, se va fi încheiat citirea variabilei *array1_size*, ceea ce va determina un proces de *branch recovery* (CPU a realizat predicția eronată). Din păcate, este însă tardiv, pentru că data de la adresa *array2[s*256]* s-a alocat deja în cache! Dacă un alt fir de control va citi iterativ locațiile de memorie de la adresele *array2[n * 256]*, *n= 0, 1, …, 255,* va constata că doar o citire este rapidă (cu hit în cache), cea pentru *n=s*, iar celelalte 255 de citiri sunt lente (cu miss în cache). Mai precis, spre exemplu, pentru a afla valoarea secretă, atacatorul poate apela din nou funcția aceasta *kernel* cu o valoare *x'* corectă de astă dată. Dacă *array1[x']=s*, accesul la tabloul *array2[s * 256]* va fi rapid, pentru că această locație din tabloul *array2* de memorie este deja alocată în cache. În caz contrar, accesul va fi mai lent, pentru că celelalte locații ale tabloului *array2[array1[x] * 256]* nu sunt alocate în cache (detalii în [2] pg. 6). Practic s-a aflat valoarea lui *s*, pe baza metodei F+R, ca și în cazul *Meltdown*, anterior explicat. Autorii au sesizat vulnerabilități de tip *Spectre* la procesoare INTEL, AMD și ARM.

Autorul acestui articol crede că durata de procesare a ramurii speculative poate fi semnificativ prelungită, în beneficiul atacatorului, dacă acesta ar avea "norocul" unei secvențe de cod de genul:

array1_size =mem_miss[i*j/k]; asignare mare consumatoare de timp (*miss, MUL/DIV*...)
if (x < array1_size)
y = array2[array1[x] * 256];

În acest exemplu, asignarea variabilei *array1_size* este astfel construită încât să consume mult timp, datorită unui acces cu miss în cache la o variabilă din memorie - anume *mem_miss[i*j/k]* - pointată prin operații (instrucțiuni mașină, în *background*) mari consumatoare de timp și ele (spre ex. instrucțiuni de înmulțire/împărțire – *MUL / DIV*). Cum banch-ul (*if (x < array1_size)*) va fi predicționat eronat, procesarea instrucțiunilor de pe ramura speculativă va dura mult, mai precis până în momentul în care se va afla valoarea variabilei *array1_size*. Abia în acel moment se va declanșa procesul de *branch recovery*, dar va fi tardiv.

În cazul cache-urilor cu mapare directă (nivelul L1 cache), citirea unei date se poate face speculativ, în paralel cu procesul de comparare a *tag*-urilor (cazul procesorului Pentium 4). Valoarea citită speculativ va ajunge la toate instrucțiunile dependente din stațiile de rezervare și va fi utilizată în continuare de către acestea. Compararea *tag*-urilor

se va face mai târziu în *pipeline*. În caz de *miss* în L1 cache, numai instrucțiunile dependente de data citită speculativ din cache vor fi re-executate (*selective re-issue*). Oare procesarea speculativă a instrucțiunilor subsecvente în acest caz, nu ar putea constitui o altă breșă de securitate în unele microprocesoare actuale?

Metode agresive de procesare speculativă a instrucțiunilor, precum cele bazate pe tehnici de tipul *Dynamic Value Prediction* spre exemplu [7], în a căror dezvoltare și promovare a încercat să se implice și autorul acestui articol, probabil că au șanse ceva mai mici să se implementeze în procesoarele comerciale, datorită breșelor de securitate de tip Spectre, descoperite recent. Complexitatea excesivă impune instrumente adecvate de proiectare și optimizare, inclusiv dintre cele bazate pe tehnici rafinate de învățare automată, precum sunt cele prezentate în [8], [9], [10].

Există și alte tipuri de erori hardware similare, deja semnalate. Spre exemplu, în [12] se exploatează posibilitatea ca un core să invalideze o intrare din cache-ul unui alt core prin intermediul unor instrucțiuni speculative, în cadrul unui sistem construit în jurul microprocesorului *Intel Core i7 (4cores, 2.4 GHz)*.

5. CONCLUZII

În urma acestui studiu, autorul a ajuns la următoarele concluzii personale:

- Posibilitatea de a aloca date privilegiate în cache, prin instrucțiuni tranzitorii user, se datorează unor erori hardware în proiectarea CPU. Posibilitatea scrierii în pagini privilegiate, prin instrucțiuni tranzitorii sau/și speculative, nu ar trebui să fie posibilă în cazul unei proiectări corecte (instrucțiunile de scriere în memorie nu ar trebui să se execute în mod tranzitoriu, ci doar *in order*).

- Din păcate, instrucțiunile tranzitorii, ca și cele speculative de altfel, pot să aloce în cache date privilegiate și apoi să le utilizeze prin intermediul altor instrucțiuni user. Aceasta reprezintă o altă eroare majoră de proiectare a microprocesoarelor actuale.

- Instrucțiunile tranzitorii nu afectează starea arhitecturală a CPU, adică regiștrii săi logici, ceea ce este corect. În ciuda acestui fapt, prin "urmele" lăsate în cache-ul de date de către aceste instrucțiuni, valoarea secretă s-a putut afla.

- Altă eroare de proiectare constă în faptul că este posibil ca instrucțiunile (inclusiv *user*) speculative să citească date privilegiate, fără declanșarea unei excepții. Faptul că nu se declanșează excepția este datorat instrucțiunilor speculative, care nu apucă să se încheie din cauza procesului de *branch recovery*, care le anulează acțiunile, deși au reușit să încarce date secrete în cache. Interzicerea execuției speculative sau *Out of Order* a instrucțiunilor de citire din memorie ar putea fi o soluție (cu reduceri de performanță însă), în opinia autorului acestui articol (pentru că instrucțiunile de scriere în memorie nu sunt speculative și nici *Out of Order*, din motive evidente). O altă soluție posibilă, în opinia autorului acestui articol, doar în cazul procesării speculative, ar impune ca procesul de *branch recovery* să invalideze locațiile afectate din *cache*-ul de date prin instrucțiunile speculative de citire din memorie.

- În cazul cache-urilor cu mapare directă (nivelul L1 cache), citirea unei date se poate face speculativ, în paralel cu procesul de comparare a *tag*-urilor (cazul procesorului Pentium 4). Valoarea citită speculativ va ajunge la toate instrucțiunile dependente din stațiile de rezervare și va fi utilizată în continuare de către acestea. Compararea *tag*-urilor se va face mai târziu în *pipeline*. Oare procesarea speculativă a instrucțiunilor subsecvente în acest caz, nu ar putea constitui o altă breșă de securitate în unele microprocesoare actuale?

- Metode agresive de procesare speculativă a instrucțiunilor, precum cele bazate pe tehnici de tipul *Dynamic Value Prediction* spre exemplu [7], în a căror dezvoltare și promovare a încercat să se implice și autorul acestui articol, probabil că au șanse ceva mai mici să se implementeze în viitorul imediat în procesoarele comerciale, datorită breșelor de securitate de tip *Spectre*. Prin asemenea tehnici, valoarea unei instrucțiuni poate fi predicționată încă din faza de aducere a acestei instrucțiuni, iar instrucțiunile subsecvente dependente de această valoare se vor procesa speculativ. Proiectarea trebuie făcută corect, astfel încât aceste instrucțiuni speculative să nu poată accesa date privilegiate.

- Aceste vulnerabilități au arătat că actuala interfață ISA utilizată între proiectarea hardware și cea software este prea abstractă, fiind insuficientă pentru asigurarea securității, a siguranței în funcționare. Această interfață trebuie extinsă cu elemente referitoare la *timing*-ul instrucțiunilor mașină și cu alte detalii de procesare microarhitecturală.

- În domeniul arhitecturii sistemelor de calcul sunt esențiale conceptele statistice de vecinătate temporală, spațială respectiv de vecinătate a valorilor instrucțiunilor (*temporal, spatial and value locality*),

de predicție și speculație, de secvențialitate, concurență și paralelism, de virtualizare etc. Erorile de proiectare *hardware* menționate au apărut la interfața acțiunilor dintre aceste concepte fundamentale. **Cum s-ar putea formaliza aceste concepte fundamentale în vederea dezvoltării unei științe deductive axiomatice a sistemelor de calcul?** Iată o problemă deschisă, de incontestabil interes...

- Vulnerabilitățile microprocesoarelor constau în interacțiunea complexă și subtilă între mai multe module ale CPU, nicidecum în erori de proiectare la nivelul unui anumit modul hardware. Fiecare din aceste module au fost proiectate "ca la carte". Complexitatea sistemelor de calcul actuale depășește puterea umană de înțelegere. Rezultă că este nevoie de metode automate de evaluare și verificare formală a proiectelor, al căror scop este de a demonstra matematic securitatea implementării [11].

- *Last but not least*... de admirat ingeniozitatea, inspirația, talentul și intuiția adâncă, toate caracteristici umane inefabile, a celor care au descoperit aceste vulnerabilități majore în proiectarea microprocesoarelor actuale [1, 2, 3]. Aceste descoperiri sunt fertile, pentru că îmbogățesc înțelegerea noastră asupra procesării programelor pe sistemele de calcul și, în consecință, conduc la îmbunătățirea performanței și securității acestor sisteme.

Procesoarele generației următoare vor trebui proiectate din punct de vedere hardware astfel încât, erori precum cele semnalate anterior, să nu mai poată să apară. Deocamdată s-au dezvoltat anumite produse software (spre exemplu *KAISER* [1], în cazul *Meltdown*) care pot stopa, din păcate doar parțial, atacurile bazate pe asemenea vulnerabilități intrinseci microprocesoarelor actuale. Așadar, soluțiile software la problemele hardware descrise anterior, pot ameliora problemele de securitate semnalate, dar nu le pot elimina complet. În acest scop, este necesară o re-proiectare a acestor procesoare, bazată pe noi metode, care să asigure corectitudinea procesărilor *out of order* respectiv speculative. Este necesară inclusiv perfecționarea metodelor de verificare formală și de optimizare multi-obiectiv. Dar asemenea soluții hardware necesită timp îndelungat până la implementarea lor în noile microprocesoare comerciale (câțiva ani, deși Intel a anunțat că va lansa noi procesoare fără asemenea erori la finele lui 2018). Desigur, problema este una deschisă și de mare complexitate. O soluție trivială, de moment, dar inacceptabilă datorită scăderii dramatice a performanței, ar consta în dezactivarea (selectivă a) procesărilor *Out of Order* și speculative ale instrucțiunilor. Compromisul între securitate și performanță trebuie re-evaluat, în vederea determinării optimului. Securitatea trebuie să devină un obiectiv al proiectării hardware, deși definirea și cuantificarea acesteia pe parcursul explorării spațiului parametrilor și proiectării reprezintă momentan probleme deschise. Oricum, complexitatea excesivă este tot mai greu de stapânit si, iată, se plătește!

REFERINȚE BIBLIOGRAFICE

[1] Lipp Moritz, et al, *Meltdown*, arXiv preprint arXiv:1801.01207, January 2018, disponibil online la https://meltdownattack.com/meltdown.pdf (accesat la 17.01.2018)

[2] Kocher Paul, et al, *Spectre Attacks: Exploiting Speculative Execution*, arXiv preprint arXiv:1801.01203January 2018, disponibil online la https://arxiv.org/pdf/1801.01203.pdf (accesat la 17.01.2018)

[3] Horn Jann, *Reading privileged memory with a side-channel*, 3 ianuarie 2018, disponibil online la https://googleprojectzero.blogspot.ro/2018/01/reading-privileged-memory-with-side.html (accesat la 17.01.2018)

[4] Vințan N. Lucian, *Towards a High Performance Neural Branch Predictor*, Proceedings of The International Joint Conference on Neural Networks, pp. 868 – 873, vol. 2, Washington DC, USA, IEEE, 10-16 July 1999

[5] Hennessy J., Patterson D., *Computer Architecture: A Quantitative Approach*, 6-th Edition, Morgan Kaufmann (Elsevier), 2018

[6] Vințan N. Lucian, *Fundamente ale arhitecturii microprocesoarelor*, Editura Matrix Rom, București, 2016

[7] Vințan N. Lucian, *Prediction Techniques in Advanced Computing Architectures*, Matrix Rom Publishing House, Bucharest, 2007

[8] Vințan L., Chiș R., Md. Ali Ismail, Coțofană C., *Improving Computing Systems Automatic Multi-Objective Optimization through Meta-Optimization*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Volume 35, Issue 7, pp. 1125-1129, July 2016

[9] Jahr R., Calborean H., Vințan L., Ungerer T., *Finding Near-Perfect Parameters for Hardware and Code Optimizations with Automatic Multi-Objective Design Space Explorations*, Concurrency and Computation: Practice and Experience, Volume 27, Issue 9, pp. 2196-2214, John Wiley & Sons, 2015

[10] Jahr R., Calborean H., Vințan L., Ungerer T., *Boosting Design Space Explorations with Existing or Automatically Learned Knowledge*, The 16-th International GI/ITG Conference on Measurement, Modelling and Evaluation of Computing Systems and Dependability and Fault Tolerance, March 19-21, 2012

[11] Heiser G., Yarom Y., *Insecure by design – lessons from the Meltdown and Spectre debacle*, disponibil online la https://theconversation.com/insecure-by-design-lessons-from-the-meltdown-and-spectre-debacle-90629, 4th February 2018

[12] Trippel C., et al, *MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols*, Submitted on 11 Feb 2018, disponibil online la https://arxiv.org/abs/1802.03802

[13] Vințan L., *Towards a Powerful Dynamic Branch Predictor*, Romanian Journal of Information Science and Technology (ROMJIST), vol. 3, nr. 3, pg. 287-301, ISSN: 1453-8245, Romanian Academy, Bucharest, 2000

[14] Marc Duranton et al, *HiPEAC Vision 2017*, ISBN 978-90-9030182-2, January 2017

Despre autor

Prof. univ. dr. ing. Lucian N. VINȚAN
Membru titular al Academiei de Științe Tehnice din România

Este expert în arhitectura sistemelor de calcul, optimizare multi-obiectiv și metode de *text-mining*. A publicat 7 monografii științifice (Editura Academiei Române, Editura Tehnică etc.) și peste 160 de articole științifice, inclusiv în reviste (*Thomson Reuters) Web of Science (WoS, Clarivate Analytics)* prestigioase, dintre care se menționează *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Information Sciences* (Elsevier), *Concurrency and Computation: Practice and Experience* (John Wiley & Sons), *Journal of Systems Architecture* (Elsevier), *IET Computers & Digital Techniques (*UK), *Microprocessors and Microsystems* (Elsevier) etc., precum și în conferințe internaționale de referință (*IEEE, ACM*), din România, SUA, Marea Britanie, Italia, Germania, Spania, China etc. Peste 40 dintre acestea sunt indexate / cotate *WoS*. Lucrările sale au înregistrat până în prezent peste 750 de citări internaționale independente, în publicații de certă valoare științifică (spre ex. în revista *IEEE Transactions on Computers*). În anul 2005 i s-a acordat Premiul "*Tudor Tănăsescu*" al Academiei Române.