Spekulatiivisesta ajosta johtuvat haavoittuvuudet ja niiltä suojautuminen

Turun yliopisto
Tietotekniikan laitos
TkK-tutkielma
Tietotekniikka
Toukokuu 2025
Antti Mäkimattila

TURUN YLIOPISTO
Tietotekniikan laitos

Antti Mäkimattila: Spekulatiivisesta ajosta johtuvat haavoittuvuudet ja niiltä suojautuminen

TkK-tutkielma, 28 s.
Tietotekniikka
Toukokuu 2025

Spekulatiivinen ajo on optimointitekniikka, jolla nopeutetaan prosessorin toimintaa ja vältetään ohjelman kriittisellä polulla olevien komentojen suorituksen odottamista. Spekulatiiviseen ajoon kuuluvia menetelmiä manipuloimalla on kuitenkin mahdollista vuotaa informaatiota kohdetietokoneesta.
Tässä tutkielmassa tutustutaan viiden eri haavoittuvuuden toteutukseen ja syitä siihen, miksi haavoittuvuudet toimivat. Tutkielmassa läpikäydyt haavoittuvuudet ovat Spectre V1, Spectre V2, Meltdown, LazyFP ja TikTag. Tarkoituksena on, että lukija ymmärtää teorian hyökkäysten takana.
Tämän jälkeen tutkielmassa tutustutaan haavoittuvuuksien mahdollisiin riskeihin ja vaikutuksiin. Pohditaan tilanteita, joissa haavoittuvuuksia voidaan käyttää hyväksi, ja varotoimenpiteitä haavoittuvuuksia vastaan. Koska spekulatiivinen ajo parantaa suorituskykyä, tutkimuksessa myös selvitetään, miten varotoimenpiteet vaikuttavat prosessorien suorituskykyyn. Lopulta tutustutaan ratkaisuideoihin, jotka suojaisivat paremmin haavoittuvuuksia vastaan ja myös näiden vaikutusta suorituskykyyn.
Haavoittuvuuksien riski ei ole suuri. Tarvittavia tahoja varoitettiin etukäteen ja haavoittuvuuksia vastaan kehitettiin varotoimenpiteet ennen niiden julkaisua. Vaikka haavoittuvuudet voidaan suorittaa suurimmassa osassa tietokoneissa, tilanteita jossa haavoittuvuudet voisivat aiheuttaa enemmän haittaa kuin muut helpommin suoritetut haavoittuvuudet ovat harvinaisia. Nämä haavoittuvuudet ovat enemmän akateemisia kuin vahinkoa aiheuttavia haavoittuvuuksia.

Asiasanat: spekulatiivinen ajo, sivukanavahyökkäykset, Meltdown, Spectre, tietoturva

Sisällys

1 Johdanto

Tässä tutkielmassa tutustutaan spekulatiivisesta ajosta johtuviin haavoittuvuuksiin ja niiltä suojautumiseen. Spekulatiivinen ajo on optimointitekniikka, jolla nopeutetaan ohjelman suoritusta. Tämä menetelmä kuitenkin mahdollistaa informaation vuotamisen mahdolliselle hyökkääjälle.

Tällä hetkellä suurin saavutettu prosessorin kellotaajuus on noin 9.1 GHz. Tämä onnistui jäähdyttämällä prosessori nestemäisellä heliumilla. [1], [2] Vaikka todellisuudessa tietokoneita ei ajeta näillä nopeuksilla, niin ne pystyvät suorittamaan operaatioita hyvin nopeasti. Kaikki operaatiot eivät kuitenkaan ole samanarvoisia. Jakolasku vie enemmän prosessorisyklejä kuin yhteenlasku. Muistioperaatiot vievät vielä enemmän aikaa ja tähän vaikuttaa myös muistin etäisyys prosessorista. Jotta prosessorien hyöty saataisiin maksimoitua, on kehitetty tekniikoita, joilla lisätään prosessorien suorituskykyä. Spekulatiivinen ajo on yksi näistä tekniikoista.

Spekulatiivinen ajo pyrkii ennustamaan, miten ohjelma etenee samalla kun odotetaan toisen operaation tuloksia. Jos ennuste menee väärin, ohjelma palaa takaisin aikaisempaan tilaan ja jatkaa suoritusta. Tällä tekniikalla voi kuitenkin jäädä havaittavia sivuvaikutuksia, joita voidaan hyödyntää kohteen informaation vuotamiseen. Tällaisia haavoittuvuuksia on löydetty vuodesta 2018 lähtien aina muutama vuodessa. [3]

Tutkimuskysymykset ovat **Tk1. miten haavoittuvuudet toimivat**, **Tk2. mitkä ovat haavoittuvuuksien riskit ja vaikutukset** ja **Tk3. millaisia ratkai-**

suja on ehdotettu? Tutkielma on rajattu viiteen haavoittuvuuteen, joiden nimet ovat Spectre V1[3], Spectre V2[3], Meltdown[4], LazyFP[5] ja TikTag[6]. Jokaista haavoittuvuutta vastaan esitellään ne ratkaisut, jotka otettiin käyttöön, kun haavoittuvuudet huomattiin. Ratkaisuehdotukset on rajattu viiteen.

Hakuprosessi alkoi haavoittuvuukisen nimillä, joiden avulla löytyi tieteelliset artikkelit haavoittuvuuksista. Lisämateriaalin löytämiseen hakukantoina toimivat Web of Science ja IEEE Xplore sivustot ja hakusanoina "Spectre", "side-channel attack", "mitigation", "impact"ja "performance". Teknistä tietoa tarvittaessa Google toimi hakukoneena ja lähteinä oli muun muassa prosessorivalmistajien nettisivut.

Luvussa 2 käydään läpi tarvittavaa taustaa haavoittuvuuksien ymmärtämiseksi. Luvussa 3 esitellään haavoittuvuudet ja syyt, miksi ne toimivat. Luvussa 4 tarkastellaan haavoittuvuuksien riskejä, varotoimenpiteitä ja miten nämä varotoimenpiteet vaikuttavat prosessorien suorituskykyyn. Luvussa on myös mahdollisia ratkaisuja, jotka voivat suojata prosessoreja näiltä haavoittuvuuksilta ilman merkittävää suorituskyvyn laskua. Lopuksi kootaan asiat yhteen yhteenvedossa.

2 Tausta

Tässä luvussa tutustutaan prosessorin välimuistiin, sivukanavahyökkäyksiin ja spekulatiiviseen ajoon. Nämä kolme asiaa yhdistämällä voidaan rikkoa prosessien välinen tietoturva ja vuotaa kohteen muistia tai muuta salaista tietoa.

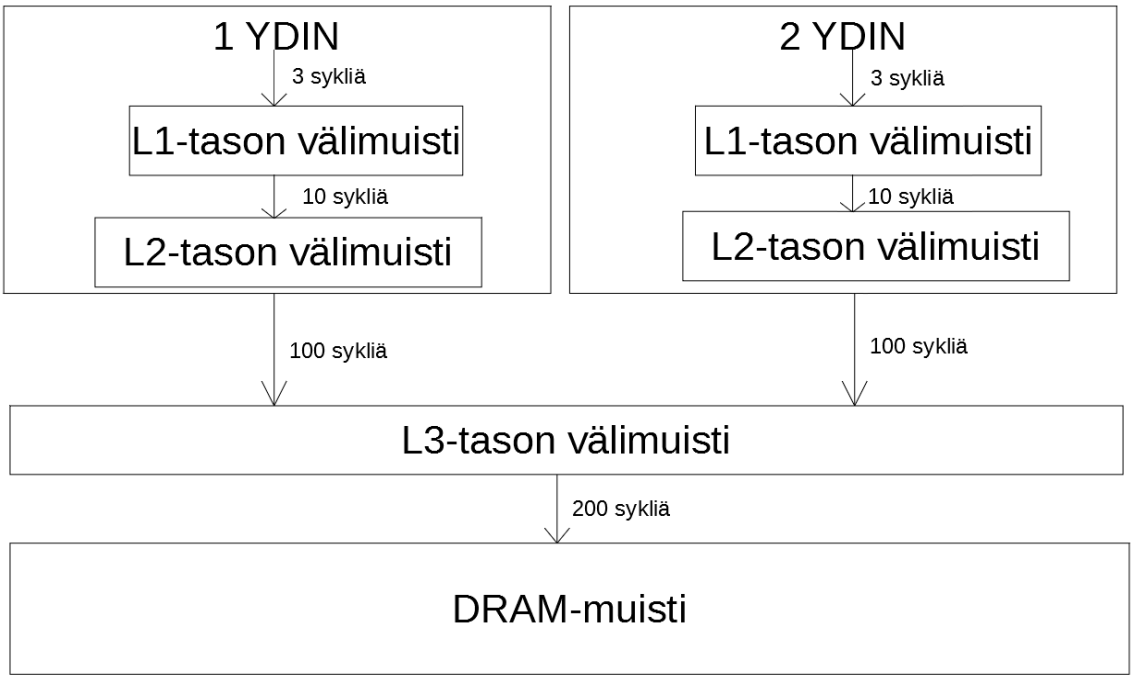
2.1 Välimuisti

Välimuisti (*eng.* cache) on prosessorin ja tietokoneen muistin välissä oleva komponentti. Välimuisti nopeuttaa tietokoneen muistioperaatioita. Välimuisti on yleensä jaettu kolmeen tasoon. Jokaisella prosessorin ytimellä on oma L1-tason ja L2-tason välimuisti ja kaikille ytimille yhteinen L3-tason välimuisti. L1-tason välimuistiin mahtuu vähän arvoja, mutta ne ovat nopeasti haettavissa. Siirtyessä alempitasoisemmille välimuisteille hakuaika lisääntyy, mutta näillä tasoilla on enemmän tilaa tallettaa dataa. Prosessori aloittaa muistiin tallennetun arvon haun käymällä järjestyksessä jokaisella eri välimuistitasolla. Jos haettua arvoa ei löydy yhdestäkään tasosta, niin vasta tämän jälkeen prosessori toteuttaa haun muistista. [7]

Tehokkuuden parantamiseksi välimuistiin ei haeta pelkästään yksittäisiä arvoja, vaan muistista haetaan suurempia kokonaisuuksia samalta muistialueelta. Ohjelmissa käytetään paljon suuria tietorakenteita, kuten listoja, joiden elementit on talletettu vierekkäisiin muistiosoitteisiin. Näin vältetään jatkuvasti yksittäisten elementtien toistuvaa hakua muistista. [7]

Prosessorin kellotaajuus ilmaisee, kuinka nopeasti prosessori voi toteuttaa eri

operaatioita. Kuitenkaan kaikki operaatiot eivät vie yhtä paljon aikaa. Esimerkiksi 2.5 GHz:n kellotaajuuden prosessori suorittaa kellosyklin 0.4 nanosekunnissa. Intelin arvojen mukaan muistiarvon hakuun menee noin 80 nanosekuntia [8]. Näistä laskettuna $80/0.4 = 200$ nähdään, että siinä ajassa kun haetaan 1 arvo muistista, olisi voitu suorittaa 200 syklin edestä komentoja [9]. Kuvasta 2.1 näkee nopeusarvot välimuistin eri tasoilta. Yarom ja Falkner löysivät samanlaisia tuloksia, kun he mittasivat L1-tason välimuistista arvon haun vievän noin 40 kellosykliä ja muistista arvon haun vievän noin 300 kellosykliä [10].



Kuva 2.1: Välimuistin rakenne ja arvioitu hakunopeus sykleinä 2.5GHz:n prosessorilla ja Intelin antamilla arvoilla [8]. [9]

2.2 Sivukanavahyökkäykset

Sivukanavahyökkäys (*eng.* side-channel attack) on hyökkäystekniikka, jota käytetään, kun kohteena oleva prosessi tai algoritmi itsessään ei ole haavoittuvainen. Sivukanavahyökkäyksiä voidaan esimerkiksi käyttää kryptograafisien algoritmien mur-

tamiseen. Hyvin luodussa kryptograafisessa algoritmissa ei haittaa, vaikka hyökkääjä pääsisi käsiksi salattuun tekstiin, kunhan salausavain pysyy salassa. Salausalgoritmi voidaan kuitenkin murtaa, jos sen ohjelmisto (*eng.* software) ja laitteisto (*eng.* hardware) ei ole toteutettu riittävän hyvin. [11]

Ohjelmistoa ja laitteistoa kehitettäessä prosessiin tulee mukaan ihmiselle havaitsemattomia, mutta laitteille mahdollisesti havaittavia sivuvaikutuksia. Näistä sivuvaikutuksista voi syntyä sivukanavahyökkäyksille tarvittava sivukanava. Nämä sivukanavat eivät yleensä vuoda mitään yksittäisesti tärkeää informaatiota, mutta kun vertaa saatuja tuloksia toisiinsa, hyökkääjä pystyy paljastamaan salassa pidettyä tietoa. Tällaisia sivukanavia ovat muun muassa vaihtelut ajoituksessa funktioita suorittaessa [12], vaihtelut virrankulutuksessa [11] ja vaihtelut sähkömagneettisessa säteilyssä [13].

Yhtenä esimerkkinä sivukanavahyökkäyksestä olisi seuraava tilanne. Hyökkäyksen kohdetta pyydetään kirjoittamaan tekstiä mekaanisella näppäimistöllä ja hyökkääjä ottaa talteen ääninäytteen kirjoittamisesta. Tämän jälkeen otetaan ääninäyte siitä, kun kohde kirjautuu omalla käyttäjätunnuksella ja salasanalla johonkin palveluun. Näitä näytteitä vertaamalla hyökkääjä voi selvittää kohteen salasanan. Salasanaa kirjoittaessa se pysyy salassa, mutta näppäimistön painallukset tuottavat sivuvaikutuksena ääntä, joka toimii hyökkäyksen sivukanavana. [14]

2.3 Spekulatiivinen ajo

Spekulatiivinen ajo (*eng.* speculative execution) on yksi tärkeimmistä tekniikoista, jonka avulla lisätään prosessorin nopeutta. Spekulatiivisen ajon tarkoitus on jatkaa ohjelman suorittamista sen päätyessä tilanteeseen, jossa ei vielä tiedetä, mitä komentoja pitäisi suorittaa. Prosessori kohtaa suorituksen aikana useita haaroja (*eng.* branch) esimerkiksi ehtolauseiden takia. Se ei pysäytä ohjelman suoritusta odottaessaan haaran suunnan ratkeamista, vaan koittaa ennustaa, mihin suuntaan ohjelma

jatkaisi suoritustaan välttäen tyhjäkäyntiä (*eng.* idle). [3], [15], [16, p. 42]

Prosessori ei voi tietää, ovatko spekulatiivisesti ajetut komennot oikein. Tämän takia komentojen tulokset otetaan talteen väliaikaisesti, mutta kaikki operaatiot voidaan perua ja palauttaa prosessori aikaisempaan tilaan. Kun prosessori lopulta ratkaisee, mihin suuntaan haara etenee, mahdollisuuksia on kaksi. Jos ennuste oli oikein, talletetut tulokset otetaan todellisiksi tuloksiksi ja prosessori jatkaa tästä eteenpäin ja on näin välttänyt tyhjäkäyntiä. Jos taas ennuste oli mennyt väärin, kaikki väliaikaset tulokset hylätään, ja prosessori jatkaa ajoa oikeassa haarassa. [3], [15], [16, p. 42]

Spekulatiiviseen ajoon liittyen prosessoreilla on kaksi eri tasoista abstraktiota: arkkitehtuuri ja mikroarkkitehtuuri. Arkkitehtuuri kuvaa, miten prosessori kommunikoi eri sovellusten kanssa. Esimerkiksi kaikki Intelin prosessorit toteuttavat Intelin käskykanta-arkkitehtuurin (*eng.* Instruction Set Architecture, ISA), mutta arkkitehtuuri ei ota kantaa siihen, miten yksittäinen Intelin prosessori toimii. Mikroarkkitehtuuri on alempitasoinen abstraktio ja se kuvaa, miten prosessori toimii sisäisesti, ja toteuttaa ISA:n esittämät vaatimukset prosessorille. [17]

Spekulatiivinen ajo takaa, että prosessorin arkkitehtuurinen tila, eli sovelluksen tila, ei muutu. Tällöin rekisterien arvot ja muistissa olevat arvot pysyvät samana, vaikka ennuste olisi väärin. Prosessorin suorittamia, mutta hylättyjä komentoja, kutsutaan tilapäiseksi komennoiksi (*eng.* transient instruction). Nämä komennot saattavat aiheuttaa muutoksia mikroarkkitehtuurissa, jonka tila ei palaudu takaisin. Tähän tutkielmaan liittyen tärkeää on, että spekulatiivinen ajo voi vaikuttaa välimuistissa oleviin arvoihin. [3], [15], [16, p. 42]

3 Haavoittuvuudet

Tässä luvussa käydään läpi haavoittuvuuksia, joissa käytetään hyväksi spekulatiivisesta ajosta johtuvia sivuvaikutuksia ja vuodetaan informaatiota. Tarkasteltavaksi on valittu viisi eri haavoittuvuutta. Spectre ensimmäinen ja toinen variantti ja Meltdown nimiset haavoittuvuudet julkaistiin samaan aikaan ja ovat käytännössä ne haavoittuvuudet, josta koko kategoria sai alkunsa. Tämän jälkeen käydään läpi LazyFP niminen haavoittuvuus, joka julkaistiin noin puoli vuotta ensimmäisten haavoittuvuuksien jälkeen. LazyFP hyvä esimerkki siitä, miten suorituskyvyn parantaminen on avannut tietoturva-aukkoja. Viimeiseksi tarkastellaan haavoittuvuutta nimeltä TikTag. Se on uudempi ja siitä on nähtävissä miten hyökkäystekniikat ovat kehittyneet pidemmälle. Tämä on myös se haavoittuvuus, josta tutkielma sai ideansa.

3.1 Spectre 1 variantti

Spectren ensimmäinen variantti Spectre V1, toiselta nimeltään rajantarkastuksen ohitus (*eng.* boundary check bypass), on haavoittuvuus, joka käyttää spekulatiivista ajoa ohittaakseen ehtolauseen asettaman rajan ja näin vuotaa informaatiota. Ohjelmalistaus 1 on esimerkkitilanne, jossa voitaisiin hyödyntää Spectre V1 haavoittuvuutta. Voidaan olettaa, että x on etumerkitön kokonaisluku (*eng.* unsigned integer). *lista1* on lista tavuja, eli arvoja väliltä 0-255, ja *lista2* on toinen lista tavuja, jonka koko on 256*4096 tai 1 Megatavu. [3]

Ohjelmakoodin ensimmäisellä rivillä tarkastetaan, että muuttujaan x asetettu

Ohjelmalistaus 1 koodi jolla voitaisiin vuotaa informaatiota spekulatiivisesti.

```
[c]
if(x < lista1_koko)
    y = lista2[lista1[x] * 4096];
```

arvo on sallitun rajan sisäpuolella eli pienempi kuin *lista1_koko*. Ainakin C-kielessä tämä on tärkeää. C-kielestä käännetty koodi ei itse tarkista ajon aikana, onko annettu arvo mahdollisesti listan rajojen sisäpuolella, vaan koodin kirjoittajan on huolehdittava asiasta. *lista1[x]* on sama asia C-kielessä kuin **(lista1:n osoite + x)*. Ilman tarkastusta voitaisiin x arvoksi laittaa x = *salainen osoite – lista1:n osoite*, jolloin *lista1[x]* lukisi suoraan arvon salaisen osoitteen kohdalta ja näin vuotaisi muistia. Jos hyökkääjä ei tiedä mitään salaista muistiosoitetta, hän voi aiheuttaa segmentaatiovirheen (*eng.* segmentation fault) ja kaataa ohjelman syöttämällä x:n paikalle satunnaisen arvon. [3]

Toisella rivillä ohjelma lataa yhden *lista2*:n arvoista, joka on riippuvainen *lista1[x]*:n kohdalta löydetystä arvosta. Koska *lista1*:n arvojen tiedetään olevan tavuja, eli lukuarvoja väliltä 0–255, riippumatta siitä onko x:n arvo oikeasti *lista1*:n rajojen sisäpuolella, y:n arvoksi asetetaan *lista2[(0-255)*4096]* kohdalta löytyvä arvo. Lopullisesti asetetulla arvolla ei ole mitään väliä, vaan tärkeämpää on tehdä jokin muistioperaatio. [3], [4]

Prosessorin ajaessa nämä kaksi riviä lineaarisessa järjestyksessä, eli vuoron perään, ei aiheutuisi mitään ongelmia. Näin ei kuitenkaan aina tapahdu. Jos ehtolauseen tulos vie paljon aikaa, prosessori ei jää odottamaan sitä, vaan aloittaa spekulatiivisen ajon. Hyökkääjä voi hidastaa ehtolauseen tulosta esimerkiksi poistamalla välimuistista *lista1_koko*:n arvon tai tekemällä operaatioita, jotka täyttävät välimuistin muulla tiedolla ja näin ylikirjoittaa välimuistista löytyvän *lista1_koko* arvon. Samalla prosessorin haaran ennustajan (*eng.* branch predictor) voi kouluttaa olettamaan ehtolauseen olevan tosi. Koulutuksen voi tehdä helposti esimerkiksi

toistamalla 10 kertaa sallittuja arvoja ja tämän jälkeen valitsemalla muistiosoitteen, johon ei pitäisi olla pääsyä. [3]

Prosessorin ajaessa spekulatiivisesti toisen rivin komentoa *lista1[x]*:n arvo voi olla mistä tahansa muistiosoitteesta, eli päädytään aiemmassa kappaleessa mainittuun tilanteeseen. Tämä salainen arvo s on väliltä 0-255 ja prosessori voi myös spekulatiivisesti ajaa seuraavan vaiheen *lista2[s * 4096]*. Kun ehtolauseen tulos on selvinnyt ja prosessori on huomannut, että spekulaatio on väärin ja että sen tila pitäisi palauttaa aiempaan arkkitehtuuriseen tilaan, vahinko on jo tapahtunut. Yksi 256 arvosta, jonka lista2 sisältää, on tuotu välimuistiin. [3]

Lopuksi välimuistin sisältämä data pitää saada siirrettyä mikroarkkitehtuurisesta tilasta arkkitehtuuriseen tilaan. Samoilla tavoilla, miten *lista1_koko* voidaan poistaa välimuistista, voidaan myös informaatiota vuotaa sivukanavien avulla. Jos hyökkääjän käytössä on komento tyhjentää prosessorin välimuisti, kuten *cflush*-komento, voidaan käyttää huuhtele+uudelleen lataa (*eng.* flush+reload)[10] tekniikkaa. Tämän tyylisten komentojen puuttuessa tai jos hyökkääjällä ei ole oikeuksia ajaa kyseisiä komentoja, voi hän hyödyntää häädä+uudelleen lataa (*eng.* evict + reload)[18] tekniikkaa. huuhtele+uudelleen lataa tekniikalla voidaan suoraan tyhjentää halutut muistiarvot välimuistista. häädä+uudelleen lataa tekniikka sen sijaan täyttää välimuistin roskadatalla. Kun välimuistin arvot on poistettu, siirretään yksi *lista2*:n arvo välimuistiin spekulatiivisella ajolla. Tämän jälkeen käydään läpi kaikki (0-255)*4096 osoitetta ja etsitään nopeimmin latautuva arvo. Jos ohjelma pystyy suoraan laskemaan prosessorisyklien määrän, valitaan se arvo, johon kuluu vähiten syklejä. Jos se ei pysty suoraan laskemaan syklien määrää, voidaan sen sijaan mitata hakuun kuluva aika. [3]

Tämän mittauksen tuloksena pitäisi yhden mitatuista arvoista olla huomattavasti nopeammin latautunut ja tämän mitatun arvon numeron sijainti on muistista vuodetun tavun arvo. Toistamalla huuhtele tai häädä operaatioita, spekulatiivista

ajoa ja mittausta, on mahdollista nopeasti ja huomaamattomasti vuotaa muistia. Välttääkseen virheitä mittaustuloksissa hyökkääjän pitäisi käyttää suuria harppauksia. Näin voidaan välttää kappaleessa 2.1 mainitun laajan muistiosan haun, mikä sekoittaisi mittaustuloksia. [3], [4]

3.2 Spectre 2 variantti

Spectren toinen variantti Spectre V2, toiselta nimeltään haaran kohteen injektio (*eng.* branch target injection, BTI), on haavoittuvuus, joka käyttää spekulatiivista ajoa ohjatakseen hyökkäyksen kohteen ajamaan funktioita spekulatiivisesti. Erona Spectre V1:een on, että V2:n ei tarvitse löytää haavoittuvaa koodia samasta prosessista, vaan hyökkäysprosessi voi olla erillään kohdeprosessista. Hyökkäysprosessi voi vuotaa kohdeprosessin muistia myös eri säikeestä (*eng.* thread), kunhan molemmat prosessit ajetaan samassa prosessoriytimessä. [3]

Spectre V2 käyttää hyväkseen epäsuoria haaroja (*eng.* indirect branch). Näitä syntyy, kun käännetään ohjelmakoodia konekieleksi. Tällaisia haaroja ovat esimerkiksi `jmp eax` komento, joka asettaa ohjelmalaskurin (*eng.* Instruction Pointer, IP) arvoksi `eax`:sta löytyvän arvon. Koska `eax`:sta löytyvä arvo selviää vasta ajon aikana ohjelmakoodin saapuessa tähän kohtaan, on mahdollista, että prosessin suoritus pysähtyisi. Välttääkseen tyhjäkäyntiä prosessori kerää talteen viimeksi käytyjen epäsuorien haarojen hyppykohteita. Riippuen mikroarkkitehtuurista prosessorit käsittelevät talteen kerättyjä kohteita eri tavalla. Idea on kuitenkin samankaltainen kaikissa prosessoreissa. Näiden aikaisempien kohteiden avulla haaran ennustaja koittaa arvata, mihin suuntaan ohjelmakoodin pitäisi edetä. Jokaisella prosessorin ytimellä on oma haaran kohteen puskuri (*eng.* branch target buffer), mutta tämä puskuri on jaettu kaikkien ytimen säikeiden välillä. [3], [19]

Hyökkäykseen liittyen hyökkääjän pitää löytää "Spectre pienohjelma" (*eng.* Spectre gadget), joka on muutama rivi konekieltä. Pienohjelman vaatimuksena on, että se

voi tuottaa mikroarkkitehtuurisia muutoksia, kuten välimuistia muokkaavia operaatioita, kun se ajetaan spekulatiivisesti. Pienohjelmat pitää löytyä sellaisesta muistialueesta, joka on merkattu ajettavaksi (*eng.* executable). Tällaisten löytäminen voi olla vaikeaa kohdeohjelmasta, mutta tähän sopii myös jaettujen kirjastojen (*eng.* shared library) ohjelmakoodi. Jaetut kirjastot ovat ohjelmia, joita moni eri ohjelma voi käyttää samanaikaisesti. Jotta käyttöjärjestelmä ei tuhlaa muistia, monella eri ohjelmalla on pääsy jaetulle kirjastolle varattuun muistiin. Nämä ohjelmat voivat käyttää jaetun kirjaston ohjelmakoodia [18]. Yleensä ohjelmilla on muutaman megatavun verran jaettujen kirjastojen yhteistä muistia. Hyökkääjä voi analysoida jaettujen kirjastojen koodia löytyääkseen näistä sopivan pienohjelman. [3]

Ohjelmalistaus 2 Spectre pienohjelmasta esimerkki.

```
[asm]
add edi, [ebx+edx+20]
mov eax, [edi]
```

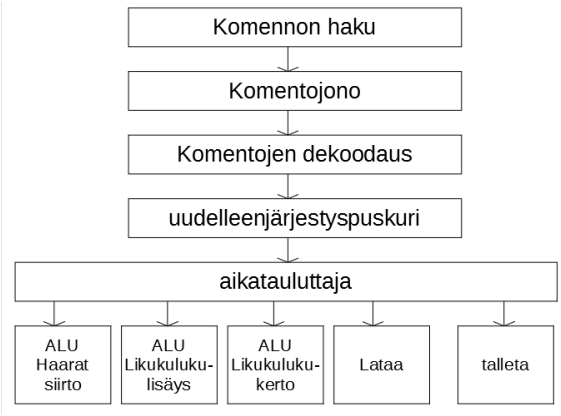
Jos hyökkääjä pystyy kontrolloimaan ohjelmalistaus 2:sta löytyvien rekisterien `edx` ja `edi` arvoja, on mahdollista vuotaa kohdeohjelman muistia. Ohjelmalistauksen ensimmäisellä rivillä lisätään `edi` rekisteriin *ebx+edx+20* osoitteesta löytyvä arvo. Jos hyökkääjä pystyy hallitsemaan `edx`:stä löytyvää arvoa ja tietää salaisen arvon *salainen* sijainnin, hyökkääjä voi asettaa `edx`:n arvoksi *edx = salainen - ebx - 20*. Tämän jälkeen `edi`:n arvoksi tulee *edi + [salainen]*. Seuraavan rivin tehtävänä on tuottaa mikroarkkitehtuurinen muutos. Rivi hakee aikaisemmin lasketun *[edi + [salainen]]* osoitteesta löytyvän arvon ja siirtää sen rekisteriin `eax`. Rivi aiheuttaa muistihaun ja näin saadaan aikaan muutos välimuistiin. Koska hyökkääjä on pystynyt ohjaamaan `edi`:n arvoa, se voi osoittaa samanlaiseen tarkastuslistaan, mitä käytettiin V1 hyökkäyksessä. Kun välimuistiin on tehty muutos, voidaan taas mitata, minkä arvon hakuun menee vähiten aikaa esimerkiksi huuhtele + uudelleen lataa tekniikalla. [3]

Toteuttaakseen hyökkäyksen hyökkääjän täytyy käynnistää hyökkäysprosessi samassa ytimessä kuin kohdeprosessi. Hyökkäysprosessi kouluttaa haaran ennustajan ajamalla jatkuvasti epäsuoria haaroja samaan osoitteeseen, josta kohdeprosessin muistista löytyy tarvittava pienohjelma. Kohdeprosessin ollessa epäsuorassa haarassa hyökkääjän pitää varmistaa, että epäsuoran haaran kohde ei ole välimuistissa. Koska kohdeosoite ei ole muistissa, prosessi ennustaa haaran ennustajalla ennustetun kohteen ja suorittaa spekulatiivisesti väärässä osoitteessa olevan pienohjelman. Lopulta hyökkäysprosessi selvittää, mihin välimuistin osaan pienohjelma on tehnyt muutoksen. [3]

3.3 Meltdown

Meltdown on kolmas haavoittuvuus, joka julkaistiin edellisten Spectre variaatioiden kanssa. Sen toinen nimi on hallitsemattoman datan lataus välimuistiin (*eng.* Rogue data cache load). Meltdownilla on näistä kolmesta haavoittuvuudesta mahdollisuus tuottaa kaikkein eniten haittaa. Spectren variantit vaativat kohteen tarkkaa analyysiä, mutta Meltdown mahdollistaa kohteen koko muistin vuotamisen hyökkääjän valmistamalla ohjelmalla. Meltdownin erona Spectren variaatioihin on, että se voi lukea etuoikeudettomalla (*eng.* unpriviledged) ohjelmalla etuoikeutettua (*eng.* priviledged) muistia. Spectre voi lukea muistia ainoastaan samalta oikeustasolta. Meltdown käyttää hyväkseen prosessorin suorituskykyä lisäävää tekniikkaa, epäjärjestyksessä ajoa (*eng.* out-of-order execution). [4]

Yhden komennon suoritus vaatii monta vaihetta, jotka komennon on käytävä läpi. Kuvassa 3.1 vaiheita näkyy vain 5, mutta todellisuudessa lukumäärä on suurempi ja vaihtelee riippuen mikroarkkitehtuurista. Intelin Core-mikroarkkitehtuurilla on 14-vaiheinen liukuhihna (*eng.* pipeline) [16, p. 44]. Yksittäisen komennon tulosta ei kannata odottaa, vaan sitä mukaan kun liukuhihnan vaihe on suoritettu, siirretään seuraava komento liukuhihnalle. Kaikki komennot eivät ole riippuvaisia toisistaan.



Kuva 3.1: Intelin komennon mikroarkkitehtuurinen liukuhihna yksinkertaistettuna [16, p. 45].

Esimerkiksi operaatiot $X = A + B$ ja $Y = C + D$ eivät vaikuta toisiinsa. Jos sen sijaan lasketaan $Z = X + Y$, niin on odotettava, että edelliset rivit on suoritettu. Jos tässä tilanteessa, esimerkiksi muuttujan A arvo ei ole nopeasti saatavilla välimuistista, voi prosessori suorittaa $Y = C + D$ rivin, muuttamatta ohjelman lopullista tulosta. Samalla prosessorilla on tiettyihin operaatioihin erikoistuneita suoritusyksiköitä (*eng.* execution unit), kuten kuvassa 3.1 näkyy. Kunhan prosessori varmistaa, että suoritettava komento ei ole riippuvainen vielä ajamattomista komennoista, voi se suorittaa komentoja epäjärjestyksessä. [4]

Kuten luvussa 3.1 mainittiin, haettaessa satunnaisesta muistiosoitteesta arvoa, ohjelmakoodi yleensä pysähtyy ja varoittaa segmentaatiovirheestä. Tämä ei tapahdu kuitenkaan välittömästi. Kun komennot ovat uudelleenjärjestyspuskurissa (*eng.* reorder buffer), ne odottavat, että kaikki riippuvuudet ja aikaisemmat komennot ovat kelvollisia. Vasta tämän jälkeen ne voidaan poistaa jonosta (*eng.* retire) ja ottaa todellisiksi tuloksiksi. Samoin käy myös virheitä aiheuttaville komennoille. Siinä välissä, kun on suoritettu virheen tuottava komento ja sitä ei ole vielä poistettu uudelleenjärjestyspuskurista, prosessori voi suorittaa muita komentoja epäjärjestyksessä. [4]

Kuten aiemmissa haavoittuvuuksissa, nämä epäjärjestyksessä ajetut komennot

eivät jätä jälkeensä mitään arkkitehtuurisia muutoksia. Sama ei päde kuitenkaan mikroarkkitehtuuristen muutosten osalta. Meltdown tuo nämä mikroarkkitehtuuriset muutokset samalla tavalla esille hyökkääjälle kuin Spectre V1:ssä. Luomalla [256*4096] kokoisen hakulistan ja käyttämällä haettua salaista tavua *s* lista[s * 4096] arvona. Tämän jälkeen huuhtele + uudelleen lataa tekniikalla mitataan muistihakuun kuluva aika jokaisessa osoitteessa ja näin saadaan esille tavun *s* arvo. [4]

Meltdownia haavoittuvuutta toteuttaessa ohjelma kaatuisi jatkuvasti. Korkean tason kielissä löytyy *try...catch* operaatiot, joilla voidaan käsitellä poikkeuksia (*eng.* exception). Tämä ei kuitenkaan toimi matalan tason kielissä. Yhtenä hyökkääjän vaihtoehtona on luoda lapsiprosessi, joka toteuttaa haun ja kaataa ainoastaan lapsiprosessin. Tämän jälkeen on helppo toteuttaa mittaus emoprosessissa. Toisena vaihtoehtona hyökkääjä voi luoda signaalinkäsittelijän (*eng.* signal handler). Signaalinkäsittelijä on käyttäjän luoma funktio, joka suoritetaan, kun prosessori kutsuu tämän tietyn signaalin ohjelman kaatuessa. Kolmas vaihtoehto on ajaa komennot spekulatiivisesti, jolloin käytännössä toteutetaan Spectre V1 versio hyökkäyksestä. [4]

Prosessien välisen tietoturvan vuoksi prosesseilla ei ole suoraa pääsyä muistiin, vaan jokaisella prosessilla on oma virtuaalimuisti. Tämä virtuaalimuisti on jaettu sivuihin (*eng.* pages), jotka käyttöjärjestelmä käsittelee ja jakaa prosesseille tarpeen mukaan. Näillä sivuilla on erikseen merkitty tarkastettavia arvoja, kuten voiko tällä sivulla olevaa muistia suorittaa tai ainoastaan lukea tai kirjoittaa. Samalla jokaisella sivulla on bitti, joka kertoo, saako sivulla olevaa muistia käsitellä etuoikeudettomassa tilassa. Etuoikeuteton ohjelma saattaa tarvita jotakin operaatioita, joita voidaan suorittaa vain etuoikeutetussa tilassa, esimerkiksi tiedostoon kirjoittaminen. Tämän takia prosessit voivat suorittaa järjestelmäkutsuja (*eng.* syscalls). Samaan aikaan käyttöjärjestelmällä pitää olla pääsy kaikkiin fyysisiin osoitteisiin voidakseen jakaa sivuja muille prosesseille, joten koko fyysinen muisti on kartoitettu käyttöjärjestelmälle. Nopeuttaakseen järjestelmäkutsujen ajoa, koko käyttöjärjestelmä on kartoi-

tettu virtuaalimuistiin ja tätä kautta koko fyysinen muisti on kartoitettu virtuaalimuistiin. Tämä muisti on kuitenkin turvattu etuoikeusbitillä. [4]

Bitin tarkastus ei kuitenkaan ole muistihaun kriittisellä polulla, todennäköisesti siksi että se olisi hidasta, vaan tarkastus ajetaan asynkronisesti eli tulos tulee vasta jälkeenpäin. [19] Meltdownin etuna Spectreen on, että se voi lukea etuoiketettua muistia etuoikeudettomalla prosessilla. Vielä suurempana ongelmana on, että Meltdown mahdollistaa minkä tahansa muistiosoitteen arvon lukemisen ja koko fyysinen muisti on kartoitettu etuoikeudettomalle prosessille. Samalla etuoikeutetun muistin lukemisen luvan tarkastus ei tapahdu muistihaun yhteydessä. Tämä mahdollistaa kaikkien sovellusten välisten tietoturvaesteiden ohittamisen ja samalla tietokoneen koko muistin lukemisen. Ainoana esteenä on Kernelin osoitteen sijainnin satunnaistaja (Kernel Address Space Layout Randomization, KASLR), joka satunnaistaa sen osoitteen, missä kohtaa virtuaalista muistia fyysinen muisti alkaa. Tämän voi kuitenkin löytää muutamassa sekunnissa Meltdown haavoittuvuutta käyttäen. [4]

3.4 lazyFP

LazyFP tai toiselta nimeltä laiska liukuluvun tilan palautus (*eng.* lazy FP state restore), on haavoittuvuus, joka tuli ilmi edellisten haavoittuvuuksien jälkeen. Se on toteutettu käyttämällä monia samoja tekniikoita, joita käytettiin aiempien haavoittuvuuksien yhteydessä. Kuvassa 3.1 aikatauluttaja jakaa tehtäviä eri suoritusyksiköille. Yksi näistä suoritusyksiköistä on liukulukuyksikkö (*eng.* Floating-point unit, FPU), joka on erikoistunut liukulukulaskuihin. Tämä itsessään ei olisi niin haitallista, mutta FPU:n mukana tulee myös erikoistuneita rekistereitä, joita käytetään kryptografisien funktioiden, kuten AES:n, nopeuttamiseen. Haavoittuvuutta käyttämällä voidaan vuotaa esimerkiksi salausavaimia. [5]

Aina kun suoritettava prosessi vaihtuu, prosessin koko arkkitehtuurinen tila täytyy ottaa talteen. Operaatio vaatii useita muistioperaatioita, jotka vievät aikaa. Sa-

ma käy myös FPU:n tilan kanssa. Kaikki prosessit eivät kuitenkaan käytä FPU:ta. Suorituskyvyn lisäämiseksi on kehitetty laiska FPU:n tilan vaihto (*eng.* lazy FPU context switching). Tekniikan idea on varata tietylle prosessille FPU. Jos ytimessä ajettava prosessi vaihtuu, käyttöjärjestelmä ottaa FPU:n pois käytöstä. Kun prosessori suorittaa jonkin FPU:ta käyttävän komennon, kun se on pois käytöstä, prosessori tuottaa laite ei käytössä (*eng.* device not available, DnA) poikkeuksen ja tarkastaa onko ajossa oleva prosessi FPU:n omistaja. Jos prosessi omistaa FPU:n, se otetaan käyttöön ja prosessi jatkaa suoritusta. Jos se ei omista FPU:ta, sen tila talletetaan muistiin ja muistista tuodaan ajettavan prosessin oma FPU:n tila ja jatkaa suoritusta. [5]

Ohjelmalistaus 3 Koodiesimerkki lazyFP haavoittuvuudesta

```
[asm]
movq rax, xmm0
and rax, 1
shl rax, 12
mov dword [rbx + rax + 4], 0
```

Ohjelmalistaus 3 vuotaa `xmm0` rekisteristä yhden bitin dataa. Ensimmäinen rivi hakee `xmm0` rekisterin arvon ja asettaa sen `rax` rekisteriin. Muut rivit tuottavat mikroarkkitehtuurisen sivuvaikutuksen käyttämällä *and*- ja shift-operaatioita ja suorittamalla muistioperaation johonkin muistiosoitteeseen. Komennon ajo aiheuttaa samalla poikkeuksen, jos prosessi ei omista FPU:ta. Haavoittuvuus toimii samalla tavoin kuin Meltdown. Siinä välissä, kun prosessori suorittaa komennon ja käsittelee poikkeustilanteen, prosessori ehtii suorittaa muut komennot epäjärjestyksessä. [5]

Bitin vuotaminen muuttaa hyökkääjäprosessin FPU:n omistajaksi. Jotta kohdeprosessin koko FPU:n tila voidaan vuotaa, pitää välttää DnA poikkeuksen käsittely. Mahdollisia tapoja toteuttaa tämä välttäminen on suorittaa ennen poikkeusta tuottavaa komentoa jokin toinen komento, joka myös tuottaa poikkeuksen. Kun tätä edellistä poikkeusta käsitellään, prosessori hylkää kaikki tämän poikkeuksen jälkeis-

ten komentojen tulokset, mutta on ehtinyt suorittaa nämä rivit epäjärjestyksessä. Toinen tapa on suorittaa komennot spekulatiivisesti. Tällöin vältetään kokonaan poikkeusten käsittely ja palautetaan hyökkääjäprosessin tila sellaiseksi, että voidaan nopeasti toteuttaa seuraavan bitin vuotaminen. Toistamalla jatkuvasti bitin vuotamista ja DnA poikkeuksen välttämistä, voidaan nopeasti vuotaa koko kohdeprosessin FPU:n tila. [5]

3.5 TikTag

TikTag on uudempi haavoittuvuus, joka löydettiin 2023 loppupuolella ja julkaistiin 2024 kesäkuussa. Sillä ei voi suoraan vuotaa kohteen muistia. Sen sijaan spekulatiivista ajoa käyttäen on mahdollista murtaa ARMv8.5-A arkkitehtuurissa käyttöön otettu muistinleimauslaajennus (*eng.* Memory tag extension, MTE). MTE:n tehtävänä on estää erilaisien muistia korruptoivien ohjelmien hyväksikäyttöä, joiden avulla voi aiheuttaa haavoittuvuuksia, kuten puskurin ylivuoto (*eng.* buffer overflow) tai käytä vapautuksen jälkeen (*eng.* use-after-free). [6]

MTE:n ideana on antaa kaikille osoittimille oma leima. Koska ohjelmat suoritetaan virtuaalimuistissa, niiden arvo voi olla vapaasti valittu. MTE käyttää neljää ensimmäistä osoittimen bittiä avaimena. Näin on käytössä 16 eri leimaa. Muistiosoitteille annetaan myös sama leima. Prosessori tarkistaa muistiarvoja hakiessa, että osoittimen leiman arvo on sama, kuin haetun muistiosoitteen vaatima leima. Jos se on sama, ohjelma jatkaa normaalia suoritustaan. Jos se on eri, prosessori tunnistaa laittoman hakuoperaation ja tuottaa poikkeuksen. [20]

TikTag haavoittuvuuden ideana on vuotaa tämä muistiosoitteen oikea leima ja näin ohittaa koko suojausmenetelmä. Hyökkääjän onnistuessa vuotamaan leimat, voi hän jälleen suorittaa muistikorruptio haavoittuvuuden. Nämä haavoittuvuudet voivat olla suurempi uhka kuin edelliset muistia vuotavat haavoittuvuudet. TikTag tarvitsee tietynlaisen pienohjelmarakenteen. [6]

Ohjelmalistaus 4 koodi jolla voitaisiin toteuttaa TikTag haavoittuvuus

```
[c]
if(hidas_haku)
    *arvaus;
    *testaus;
```

Ohjelmalistauksen 4 ensimmäisellä rivillä varmistetaan, että seuraavat rivit toteutetaan spekulatiivisesti. Vaatimuksena on jälleen, että *hidas_haku* muuttuja ei löydy välimuistista, esimerkiksi huuhtele + uudelleen lataa tekniikalla. *arvaus* on osoitin, joka osoittaa johonkin kohteen muistiosoitteeseen. Tällä osoitteella on kuitenkin jokin leima, jota hyökkääjä ei voi tietää. Tämän takia hyökkääjä asettaa *arvaus* osoittimen leimaksi jonkin satunnaisen arvon. Seuraavalla rivillä oleva *testaus* osoitin on taas osoite, jonka tiedetään olevan oikein ja hyökkääjä voi mitata, kuinka kauan osoitteen hakuun menee. [6]

Pienohjelman suoritettua on mahdollista selvittää, oliko arvattu leima oikein vai väärin. Jos leima ei ollut oikein, prosessori suorittaa vain *arvaus*:n osoittaman arvon haun, mutta *testaus* osoittaman arvo ei löydy välimuistista ja mitattu aika on hidas. Jos leima oli taas oikein, prosessori suorittaa molemmat rivit spekulatiivisesti ja tällöin *testaus*:n osoittama arvo löytyy välimuistista ja haku on tällöin nopea. Käymällä kaikki 16 mahdollista leiman arvoa, voi hyökkääjä selvittää minkä tahansa muistiosoitteen leiman ja näin jatkaa hyökkäystä muistikorruptio haavoittuvuudella. Näin hyökkääjä on välttänyt kokonaan MTE suojauksen. Samalla koska käskyt ovat suoritettu spekulatiivisesti, hyökkääjä voi todennäköisesti välttää systeemin ylläpitäjän huomion, joka seuraisi mahdollisista leimauksen rikkovista poikkeuksista. [6]

4 Riskit ja vaikutukset

Tässä luvussa käydään läpi, kuinka merkittävän riskin haavoittuvuudet tuovat. Tämän jälkeen tarkastellaan varotoimenpiteitä haavoittuvuuksia vastaan ja millaisia vaikutuksia näistä varotoimenpiteistä on seurannut prosessorien suorituskykyyn. Lopuksi käydään läpi ehdotettuja korjausratkaisuja haavoittuvuuksiin.

4.1 Riskit

Spekulatiivisesta ajosta seuraavat haavoittuvuuksien riskit ovat laajat, mutta samalla lievät. Kyseessä ei ole yksittäisten sovelluksien haavoittuvuudet, jotka seuraisivat huonosta ohjelmakoodista ja jotka olisi helppo korjata. Ongelmana on, että spekulatiivisesta ajosta johtuvat haavoittuvuudet on kaikkiin prosessoreihin sisäänrakennettu ominaisuus. Spectren eri variaatioita on havaittu Intelin ja AMD:n prosessoreissa. Samoin Meltdownin uskotaan vaikuttaneen kaikkiin Intelin prosessoreihin, jotka on valmistettu 20 vuotta ennen haavoittuvuuden julkaisua. [21]

Tutkielmassa käydyistä haavoittuvuuksista Meltdownilla on selvästi mahdollisuus tehdä kaikkein eniten haittaa. Se mahdollistaa etuoikeudettoman sovelluksen lukea kohteen muistia ilman mitään erikoisvaatimuksia. [4] Toiseksi suurin riski on LazyFP haavoittuvuus. Vaikka muistin määrä, joka on mahdollista vuotaa on pienempi, LazyFP:n avulla voi keskittyä tarkoituksella salassapidettyyn tietoon. Muita haavoittuvuuksia hyödyntäen muistia vuotaessa satunnaisesta sijainnista, voi olla vaikea löytää hyödyllistä informaatiota. Muistiavaruuden kokoluokka on 2^{64}, joten

satunnainen haku vie aikaa. Luvussa 3.4 mainittiin, että FPU:ta käytetään kryptografisessa algoritmeissa. Hyökkääjä pystyy kohdistamaan informaation vuotamisen paljon tarkemmin ja tärkeämpään tietoon. [5]

Spectren eri variaatiot ovat pieniriskisempiä haavoittuvuuksia. Ne vaativat kohdeohjelman tarkkaa analyysia ja manipulaatiota. Samalla ne voivat vuotaa vain samalta oikeustasolta muistia. Kuitenkin näitäkin ominaisuuksia voidaan käyttää hyväksi. Selaimet ovat hyvä kohde Spectre hyökkäyksille. Selaimen javascript käännetään ajonaikaisella kääntäjällä (*eng.* just-in-time compiler, JIT) konekieleksi, jota prosessori ymmärtää. Näin on mahdollista vuotaa kohteen selaimen muistia. [3] Toinen hyvä kohde on Linux käyttöjärjestelmän kernelistä. Googlen tietoturva-analyytikkoryhmä Project Zero onnistui vuotamaan etuoikeutettua muistia käyttäen Spectren ensimmäistä ja toista variaatioita. [19]

TikTag haavoittuvuus ei yksin ole kovinkaan haitallinen. Vaikka se murtaa MTE-suojauksen, vaikutus ei ole suuri, jos hyökkääjä ei pysty käyttämään muistiosoitteen leimaa hyväkseen. Toisaalta jos haavoittuvassa ohjelmasta löytyy lisäksi muistikorruptio haavoittuvuus, on mahdollista aiheuttaa paljon vakavampaa vahinkoa. MTE suojelee esimerkiksi puskurin ylivuotoja vastaan. Jos hyökkääjä saa suoritettua puskurin ylivuodon, voi se johtaa etäkoodin ajoon (*eng.* remote code execution). [22] TikTag voi siis olla osana hyökkäysketjua, joka johtaa vakavampiin seurauksiin kuin muut Spectre variaatiot.

Vaikka edellä mainitut haavoittuvuudet voivat johtaa tietovuotoon, ei yksittäisten henkilöiden tarvitse kuitenkaan olla kovinkaan huolissaan näistä haavoittuvuuksista. Nämä kaikki haavoittuvuudet vaativat vähintään, että hyökkääjä voi suorittaa etuoikeudetonta koodia kohdetietokoneessa. Jos hyökkääjällä on pääsy yksityiseen tietokoneeseen tällä tasolla, Spectren eri variaatioilla ei ole enää väliä, vaan hyökkääjä pystyy saavuttamaan samat asiat helpommin. Suurimpana riskinä yksityisiin tietokoneisiin ovat selainpohjaiset Spectre hyökkäykset, joita voisi suorittaa

javascriptin avulla, mutta selaimiin on tehty varotoimia näiden välttämiseksi. [3], [6] Todellisena kohteena Spectre hyökkäyksille ovat palvelimet ja pilvipalvelut. Jos hyökkääjällä on pääsy palvelimelle, on mahdollista vuotaa palvelimen prosesseista informaatiota. Pilvipalvelimilla toimivat virtuaalikoneet ovat toinen mahdollinen kohde Spectre hyökkäyksille. Ne tarjoavat alustan, jossa on mahdollista suorittaa omaa koodia, joten hyökkäys on toteutettavissa.

4.2 Toimenpiteet haavoittuvuuksia vastaan

Helpoimpana ratkaisuna olisi vain kokonaan poistaa käytöstä spekulatiivinen ajo ja Meltdownin yhteydessä myös epäjärjestyksessä ajo. Tällaiset ratkaisut ovat kuitenkin kaikkein huonoimmat ja vaikutus suorituskyvyn heikkenemiseen olisi huomattava. [4] Tämän takia on kehitetty muita ratkaisuja, joilla estetään tiedon vuotaminen, mutta pyritään säästämään mahdollisimman paljon suorituskyvystä.

Haavoittuvuuksista LazyFP on kaikkein helpoin välttää. Koska se perustuu optimointitekniikkaan, joka ohjelman tilan vaihtuessa säästää FPU:n tilan, ratkaisuna on pakottaa prosessori vaihtamaan FPU:n tila innokkaasti (*eng.* eager). [5] Laiskaan tilanpalautukseen on vaikea palata ilman, että spekulatiivisesta ajosta ei tule haittaa, joten tämä optimointitekniikka ei ole suositeltavissa.

Meltdownin suurimpana ongelmana on, että se pystyy lukemaan etuoikeudettomalla ohjelmalla etuoikeutettua muistia ja koko fyysinen muisti löytyi jokaisen ohjelman virtuaalimuistista. Ratkaisuna tähän oli eristää koko fyysisen muistin kartoittaminen virtuaalimuistiin. Tätä varten oli jo kehitetty konseptina KAISER, joka kehitettiin eteenpäin ja nimettiin kernelin sivujen eristäminen (*eng.* kernel page-table isolation, KPTI). KPTI ideana on eristää kernelin sivut virtuaalimuistista niin, että vain minimimäärä on kartoitettu virtuaalimuistiin, ilman että se vaikuttaisi ohjelman toimintaan. [23] Tämä ratkaisu myös vahingossa estää Meltdownin. Yhteys ratkaisun ja Meltdownin välillä oli, että Gruss ym. kehittivät KAISER systeemin

osallistuivat myös Meltdown haavoittuvuuden löytämiseen. [4]

Spectre V1 ratkaisuna toimii este-komennot. Intel arkkitehtuurista löytyy LFENCE komento, joka estää ohjelman suorituksen spekulatiivisesti, kunnes kaikki ennen LFENCE komentoa suoritetut komennot on poistettu suoritusjonosta. Tätä komentoa ei kuitenkaan kannata käyttää liian vapaasti, vaan valita ne kohdat, joissa on mahdollista vuotaa muistia. Muutoin komennon runsan käyttö voi vaikuttaa huomattavasti suorituskykyyn. [15] Tätä ratkaisua käytetään myös Linuxin kernelin kehitykseen, jossa koodianalyysillä ja skannaustyökaluilla pyritään löytämään mahdolliset vuotokohdat. [24] Myös TikTag perustuu tällaiseen spekulatiiviseen ajoon, joten sitä vastaan suositellaan käyttämään spekulatiivisen ajon esteitä. [6]

Spectren V2 haavoittuvuutta vastaan tarjotaan kaksi ratkaisua. Ensimmäisenä on uusi rajapinta prosessorin ja sovellusten väliin. Tällä rajapinnalla tarjotaan ratkaisuja, joilla voidaan estää epäsuorien haarojen ennustaminen. Rajapinta tarjoaa kolme ominaisuutta: 1) Epäsuoran haaran rajoitettu spekulaation (*eng.* Indirect Branch Restricted Speculation, IBRS), joka rajoittaa spekulatiivista ajoa epäsuoriin haaroihin [25], 2) Yksittäisen säikeen epäsuoran haaran ennustaja (*eng.* Single Thread Indirect Branch Predictors, STIBP), joka estää epäsuoran ennustamisen ohjauksen toisesta säikeestä [26], 3) Epäsuoran haaran ennustajan este (*eng.* Indirect Branch Predictor Barrier, IBPB), joka estää aikaisemmin suoritetun ohjelman vaikuttamasta myöhempiä epäsuoran haaran ennusteita [27].

Ohjelmalistaus 5 Retpolinen rakenne.

```
[asm]
call alusta;
kaappa_spekulaatio:
 pause;
 jmp kaappa_spekulaatio;
alusta:
 mov rsp, rax;
 ret;
```

Toisena ratkaisuna on Googlen kehittämä paluu trampoliini(*eng.* return trampoline, retpoline). Retpolinen idea on korvata epäsuorat hypyt uudella komentosarjalla, mikä estää hyökkääjän ohjaamasta spekulatiivisen haaran sijainnin. Ohjelmalistaus 5:ssa kutsutaan alusta-funktiota. Funktio asettaa `rsp`:n eli funktion paluusijainnin tilalle `rax` rekisteristä löytyvän epäsuoran haaran kohteen. Jos `rax` rekisteristä ei löydy välittömästi arvoa, ohjelma etenee spekulatiivisesti seuraavaan ret komentoon. Ret komento hyppää ensimmäisen rivin call komennon jälkeiselle riville kaappa_spekulaatio. Spekulatiivinen ajo jää jumiin tähän pause → jmp kaappa_spekulaatio silmukkaan. Kun lopulta `rax` arvo on ratkaistu ja ohjelman oikea suoritus voi jatkua, ret komento hyppää `rsp` rekisterissä olevaan sijaintiin. Näin on suoritettu epäsuora hyppy, mutta estetty mahdollisuus manipuloida spekulatiivisen ajon suuntaa. [28]

4.3 Varotoimien vaikutukset suorituskykyyn

Varotoimien eivät vaikuta prosessorien suorituskykyyn yhtä paljon kuin kokonaan spekulatiivisen ajon käytöstä pois ottaminen. Kuitenkin kun uusia Spectre haavoittuvuuksia löytyy ja ne vaikuttavat samaan prosessoriarkkitehtuuriin, niin vaikutus on kuitenkin suuri. Yksi varotoimenpide voi vaikuttaa vain muutaman prosentin, mutta 20 eri haavoittuvuutta, jotka kaikki vaikuttavat vain muutaman prosentin, tuovat huomattavaa haittaa suorituskykyyn.

Red Hat on yritys, joka tarjoaa Linuxia yrityskäyttöön. Tällaiset Linux järjestelmät voivat olla hyvä kohde käyttää Spectreä. Kun ensimmäiset haavoittuvuudet julkaistiin 2018 ja näille luotiin korjaukset, niin yrityksen mukaan vaikutukset suorituskykyyn olivat 1–20 % välillä. Tämä on myöhemmin parantunut noin 1–8 % välille. [29]

Varotoimien vaikutukset tulevat esiin, kun suorittaa paljon järjestelmäkutsuja. Eräässä testissä luotiin 1000 TCP yhteyttä ja lähetettiin yksi tavu. Spectren va-

rotoimenpiteiden mitattiin hidastavan prosessia noin 15 %. Samassa tutkimuksessa testattiin myös kirjoitusnopeutta levylle. Varotoimien hidastivat prosessia noin 50 %. [30]

Toisessa tutkimuksessa testattiin Windows-laitteiden varotoimenpiteiden vaikutusta kryptografisien algoritmien suorituskykyyn. Kryptografiset algoritmit ovat kriittinen kohde, joita vastaan haavoittuvuuksia saatetaan käyttää. Testien tuloksina oli noin 3-4 % hidastuminen eri hajautusfunktiolla (*eng.* hash function) ja 7,5 % hidastus eri AES-salausalgoritmin asetuksilla. [31]

Haavoittuvuudet voivat tulla myös uudelleen esille. Vuonna 2022 Spectre V2:sta kehitettiin uusi versio haaran historian injektio (*eng.* Branch history injection, BHI), joka ohittaa joitakin suojaustoimenpiteitä. [32] Tämä taas pakotti käyttämään enemmän retpoline rakenteita. Fawad Murtaza on kirjoittanut blogipostauksen Notebookcheck-sivulle, jonka mukaan haavoittuvuus on aiheuttanut 14 % - 35 % Intelin prosessoreissa ja noin 10 % vaikutuksen AMD:n prosessoreissa. [33]

4.4 Mahdollisia ratkaisuja

Haavoittuvuuksia vastaan on kehitetty monia erilaisia ratkaisuja. Ratkaisut voivat joko pienentää hyökkäyspinta-alaa (*eng.* attack surface) tai tunnistaa mahdollisesti ajossa olevia Spectre-hyökkäyksiä. Ongelmana on, että ratkaisujen pitäisi mahdollisimman hyvin estää Spectre-hyökkäykset ja samalla vaikuttaa mahdollisimman vähän prosessorin suorituskykyyn. Muuten voisi vain poistaa spekulatiivinen ajo kokonaan pois päältä, mutta vaikutus suorituskykyyn olisi suuri. Tässä muutama esimerkki tarjotuista ratkaisuista.

Ehdollinen spekulaatio (*eng.* Conditional speculation) on tekniikka, joka kehitettiin Spectre V1, V2, V4 ja SpectrePrime[34] haavoittuvuuksia vastaan. Sen idea on merkitä ne välimuistiin vaikuttavat komennot, jotka ovat riippuvaisia jostakin aiemmasta tuloksesta. Tällöin prosessori odottaisi, että riippuvaisuus on ratkaistu

ja tämän jälkeen jatkaa suoritusta. Tällöin prosessori voisi edelleen suorittaa spekulatiivisesti turvallisia komentoja, mutta välttäisi vaaralliset spekulaatiot. Tällainen ratkaisu kuitenkin laskisi suorituskykyä keskimäärin 53,6 % ja pahimmillaan 146,8 %. Tuomalla prosessiin suodattimia, jotka sallivat joidenkin komentojen suorituksen, mutta voivat päästää läpi myös Spectre pienohjelmia, jolloin suorituskyvyn lasku olisi keskimäärin vain 6,8 %. [35]

SpecShieldin idea on estää spekulatiivisesti saadun tiedon päätymisen sivukanavalle. SpecShield sallii spekulatiivisen ajon hakevan tiedon muistista, mutta ei päästä sitä eteenpäin linjastossa tietoon riippuviin komentoihin. SpecShield sallii tiedon etenemisen, vasta kun komento on varmasti suoritettu ja poistettu uudelleenjärjestyspuskurista. Tämä hidasti suorituskykyä 73 %. Selvittämällä missä vaiheessa voi tiedon lähettää eteenpäin ilman spekulaatiota saatiin suorituskyky parannettua vain 21 % menetykseen. Lopulta määrittämällä erikseen korkean ja matalan riskin komentoja saavutettiin noin 10 % hidastus. [36]

Ohjausvirran eheyden (*eng.* Control-flow integrity, CFI) tarkoitus on estää ohjelman kulun kaappaamisen. Se määrittää kohteet, johon ohjelma voi mennä laskemalla etukäteen ohjausvirtagraafin (*eng.* Control-flow graph). SpecCFI:n idea on käyttää samaa periaatetta myös spekulatiivisesti selvitettyihin sijainteihin. Se suojaa Spectre-hyökkäyksiltä, joiden kohde on haaran ennustaja, antamalla haaroille oman CFI leiman. Jos spekuloidun kohteen leima ei vastaa haaran leimaa, prosessori ei suorita spekuloitua koodia. [37]

HidFix on tekniikka, joka valvoo välimuistin tilaa ja palauttaa sen kun prosessorin konteksti vaihtuu. Spectre-hyökkäykset käyttävät välimuistia sivukanavana, joten se pitää aina ennen vuotamista alustaa tunnettuun tilaan. Vaikka kohde aiheuttaisi muutoksen alustetussa välimuistissa kuten huuhtele + lataa uudelleen tekniikassa, HidFix huuhtelee välimuistin uudelleen kontekstin vaihtuessa palauttaen sen alkuperäiseen tilaan. Tekniikka ei aiheuta huomattavaa suorituskyvyn menetystä ajaessa

samaa prosessia, mutta tutkimuksessa ei mitata, miten välimuistin jatkuva palautus vaikuttaisi samanaikaisesti muiden prosessien ajoon. [38]

Tietokoneet keräävät jatkuvasti informaatiota ohjelmien suorituksesta. Laitteiston suorituskyvyn laskurien (*eng.* Hardware performance counters) tai Intelin suorituskyvyn laskuri monitorien (*eng.* Performance counter monitor) numerot kasvavat, kun prosessorissa aiheutuu joitakin tapahtumia (*eng.* event). Spectre-hyökkäyksiä ajaessa tapahtuu monia helposti havaittavia tapahtumia kuten huuhteluoperaatiot tai väärän haaran ennustus (*eng.* branch misspredictions). Käyttämällä koneoppimisalgoritmeja, voi tunnistaa milloin Spectre-hyökkäys on ajossa. [39], [40]

Vaikka vaihtoehdot ovat hyviä, ei silti olla päästy eroon kaikista Spectre-haavoittuvuuksista. Voi olla tapauksia, jotka ratkaisut eivät ole ottaneet huomioon tai löydetään uusi tapa vuotaa muistia. Joillekin ratkaisuille voi käydä kuten InvisiSpec:lle [41], jonka idea oli tehdä spekulatiivisesti tehdyt muutokset näkyväksi vasta kun se oli turvallista. Myöhemmin kuitenkin löydettin tapa ohittaa tämä suojausmenetelmä. [42]

5 Yhteenveto

Tietoturva ei ole helppo aihe ja mihin tahansa sovellukseen tai laitteistoon voi jäädä tietoturva-aukkoja. Tutkimuskysymys 1 kysyy miten haavoittuvuudet toimivat. Tutkimuksessa käydyistä haavoittuvuuksista ainoastaan Meltdownia ja LazyFP:ta voidaan ajatella todellisina virheinä prosessorien ohjelmoinnissa. Spectre V1 ja V2 johtuvat siitä, miten spekulatiivinen ajo on suunniteltu. Muistioperaatiot ovat paljon aikaa vieviä komentoja, joten niiden etukäteen toteuttaminen on juuri sitä, mihin spekulatiivinen ajo koittaa vaikuttaa vähentämällä näihin kuluvaa aikaa. Ongelma syntyy vasta siinä, kun hyökkääjä pystyy manipuloimaan näitä prosessorin näkymättömiä optimointimenetelmiä. Kun koitetaan estää Spectre V1 ja V2 kaltaisia haavoittuvuuksia, prosessori lopettaa spekulaation virhetilanteessa, mikä on aivan looginen ratkaisu haavoittuvuuksien tullessa ilmi. TikTag on kuitenkin seurauksena tästä suojausmenetelmästä.

Tutkimuskysymys 2 kysyy haavoittuvuuksien riskeistä ja vaikutuksista. Haavoittuvuuksien riski ei ole ollut suuri, koska niistä ilmoitettiin etukäteen prosessorien valmistajille ja käyttöjärjestelmien ylläpitäjille. Näin saatiin kehitettyä etukäteen varotoimenpiteet, jolloin esimerkiksi Meltdownia ei voitu hyväksikäyttää päivitetyissä järjestelmissä. Haavoittuvuudet ovat erikoisia siinä, että jos jostakin prosessorista löydetään uusi spekulatiivista ajoa hyödyntävä haavoittuvuus, se voi vaikuttaa hyvin laajasti eri tietokoneisiin. Tämä kyseinen haavoittuvuus voi toimia monessa eri prosessorisukupolvessa ja vaikuttaa miljooniin laitteisiin. Samalla kuitenkaan suu-

rinta osaa käyttäjistä ei tarvitse olla huolissaan haavoittuvuuksista. Haavoittuvuus vaatii vähintään mahdollisuuden suorittaa ohjelmakoodia samassa prosessorissa, jolloin suurin huoli ei ole enää mahdolliset Spectre-haavoittuvuudet. Samasta syystä ei haluta luopua kokonaan spekulatiivisesta ajosta. Sillä ei ole paljon väliä kuinka nopea prosessori on, jos se odottaa jatkuvasti tuloksia jatkaakseen ohjelman suoritusta. Tämä suorituskyvyn lasku toisi paljon enemmän haittaa prosessorien valmistajille ja käyttäjille, kuin nämä haavoittuvuudet itsessään.

Lopulta jäljelle jäävät ne, joiden tarvitsee ottaa nämä haavoittuvuudet tosissaan. Jotkin palvelut tarjoavat laskentatehoa tai virtuaalikoneita, joissa voidaan suorittaa ohjelmakoodia. Voi olla myös, että palvelun täytyy noudattaa joitakin määräyksiä, jotka rajoittavat riskien sallimista. Tällaisten tahojen pitää olla enemmän selvillä asioista ja mahdollisista riskeistä, mutta tämä kuuluu myös työnkuvaan.

Tutkimuskysymys 3 kysyy millaisia ratkaisuja on ehdotettu haavoittuvuuksia vastaan. Ratkaisut ovat käytännössä spekulatiivisen ajon ominaisuuksien rajaaminen tai reaaliaikinen valvonta. Ominaisuuksia rajaavat ratkaisut voivat kokonaan estää haavoittuvuudet, mutta liian agressiivinen rajaus vaikuttaa suorituskykyyn huomattavasti ja ei ole hyvä ratkaisu. Tämän takia ratkaisuissa sallitaan, että jokin haavoittuva koodi saattaa päästä läpi, jolloin suorituskyvyn lasku on tarpeeksi pieni. Reaaliaikaisessa valvonta johtaisi siihen, että ajetaan ohjelmakoodia, joka valvoisi suoritettavaa koodia. Tällöin ajettavan koodin määrä kaksinkertaistuisi ja tällä olisi selkeä vaikutus suorituskykyyn.

Spekulatiivisien ajon haavoittuvuuksien tutkiminen on jatkuvaa. Vuosittain on löydetty uusia versioita haavoittuvuuksista, tekniikoita suojaamaan haavoittuvuuksilta ja tapoja ohittaa nämä suojausmenetelmät. Täydellistä ratkaisua ei ole näkyvissä ainakaan lähiaikoina, mutta nykyiset suojausmenetelmät tekevät jo vaikeasti toteutettavasta haavoittuvuudesta vieläkin vaikeamman.

Lähdeluettelo

[1] elmor, *9117.75 MHz with Core i9 14900KS (8P) @ 9117.8 MHz*. url: https://hwbot.org/submission/5508265_elmor_cpu_frequency_core_i9_14900ks_(8p)_9117.75_mhz.

[2] Elmorlabs, *First 9.1 GHz CPU (overclocked 14900KS)*, Youtube video CPU:n ajosta. url: https://www.youtube.com/watch?v=qr26jxPIDm0.

[3] P. Kocher et al., "Spectre Attacks: Exploiting Speculative Execution", teoksessa *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, s. 1–19. DOI: 10.1109/SP.2019.00002.

[4] M. Lipp et al., "Meltdown: Reading Kernel Memory from User Space", teoksessa *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD: USENIX Association, elokuu 2018, s. 973–990, ISBN: 978-1-939133-04-5. url: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[5] J. Stecklina ja T. Prescher, *LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels*, 2018. arXiv: 1806.07480 [cs.OS]. url: https://arxiv.org/abs/1806.07480.

[6] J. Kim et al., *TikTag: Breaking ARM's Memory Tagging Extension with Speculative Execution*, 2024. arXiv: 2406.08719 [cs.CR]. url: https://arxiv.org/abs/2406.08719.

[7] C. Su ja Q. Zeng, "Survey of CPU Cache-Based Side-Channel Attacks: Systematic Analysis, Security Models, and Countermeasures", *Security and Communication Networks*, vol. 2021, nro 1, s. 5 559 552, 2021. url: https://onlinelibrary.wiley.com/doi/abs/10.1155/2021/5559552.

[8] B. Brett, *Memory performance in a Nutshell*, kesäkuu 2016. url: https://www.intel.com/content/www/us/en/developer/articles/technical/memory-performance-in-a-nutshell.html.

[9] A. Fogh, *Negative result: Reading kernel memory from user mode*, heinäkuu 2017. url: https://cyber.wtf/2017/07/28/negative-result-reading-kernel-memory-from-user-mode/.

[10] Y. Yarom ja K. Falkner, "FLUSH plus RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack", teoksessa *PROCEEDINGS OF THE 23RD USENIX SECURITY SYMPOSIUM*, USENIX Assoc, BERKELEY: USENIX ASSOC, 2014, s. 719–732, ISBN: 978-1-931971-15-7.

[11] P. C. Kocher, J. Jaffe ja B. Jun, "Differential Power Analysis", teoksessa *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, sarja CRYPTO '99, Berlin, Heidelberg: Springer-Verlag, 1999, s. 388–397, ISBN: 3540663479.

[12] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems", teoksessa *Advances in Cryptology—CRYPTO'96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings 16*, Springer, 1996, s. 104–113.

[13] D. Agrawal, B. Archambeault, J. Rao ja P. Rohatgi, "The EM side-channel(s)", teoksessa *CRYPTOGRAPHIC HARDWARE AND EMBEDDED SYSTEMS - CHES 2002*, B. Kaliski, C. Koc ja C. Paar, toim., sarja Lecture Notes in Computer Science, 4th International Workshop on Cryptographic Hardware and

Embedded Systems, REDWOOD SHORES, CA, AUG 13-15, 2002, vol. 2523, 2002, s. 29–45, ISBN: 3-540-00409-2.

[14] A. Taheritajar, Z. M. Harris ja R. Rahaeimehr, *A Survey on Acoustic Side Channel Attacks on Keyboards*, 2023. arXiv: 2309.11012 [cs.CR]. url: https://arxiv.org/abs/2309.11012.

[15] *Intel Analysis of Speculative Execution Side Channels*, Intel, toukokuu 2018. url: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[16] *Intel® 64 and IA-32 Architectures Software Developer's Manual*, volume 1, v085, Intel, lokakuu 2024. url: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[17] *Architecture and micro-architecture*, Luettu: 25.10.2024. url: https://developer.arm.com/documentation/102404/0201/Architecture-and-micro-architecture.

[18] D. Gruss, R. Spreitzer ja S. Mangard, "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches", teoksessa *24th USENIX Security Symposium (USENIX Security 15)*, Washington, D.C.: USENIX Association, elokuu 2015, s. 897–912, ISBN: 978-1-939133-11-3. url: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/gruss.

[19] J. Horn, *Reading privileged memory with a side-channel*, Luettu: 6.11.2024, tammikuu 2018. url: https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html.

[20] *Memory Tagging Extension*, 2019. url: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

[21] A. Efe ja M. O. Güngör, "The impact of meltdown and Spectre attacks", *International Journal of Multidisciplinary Studies and Innovative Technologies*, vol. 3, nro 1, s. 38–43, 2019.

[22] A. One, "Smashing the Stack for Fun and Profit", *Phrack*, vol. 7, nro 49, marraskuu 1996. url: http://www.phrack.com/issues.html?issue=49&id=14.

[23] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice ja S. Mangard, "Kaslr is dead: long live kaslr", teoksessa *Engineering Secure Software and Systems: 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings 9*, Springer, 2017, s. 161–176.

[24] the linux kernel, *Spectre side Channels*, Luettu: 27.11.2024. url: https://docs.kernel.org/admin-guide/hw-vuln/spectre.html.

[25] Intel, *Indirect Branch Restricted Speculation*, maaliskuu 2018. url: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html.

[26] Intel, *Single Thread Indirect Branch Predictors*, maaliskuu 2018. url: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/single-thread-indirect-branch-predictors.html.

[27] Intel, *Indirect Branch Predictor Barrier*, maaliskuu 2018. url: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-predictor-barrier.html.

[28] P. Turner, *Retpoline: a software construct for preventing branch-target-injection*, Luettu: 27.11.2024. url: https://support.google.com/faqs/answer/7625886.

[29] *Speculative Execution Exploit Performance Impacts - Describing the performance impacts to security patches for CVE-2017-5754 CVE-2017-5753 and CVE-2017-5715*, maaliskuu 2019. url: https://access.redhat.com/articles/3307751.

[30] A. Prout et al., "Measuring the Impact of Spectre and Meltdown", teoksessa *2018 IEEE High Performance extreme Computing Conference (HPEC)*, 2018, s. 1–5. DOI: 10.1109/HPEC.2018.8547554.

[31] O. Alhubaiti ja E.-S. M. El-Alfy, "Impact of Spectre/Meltdown Kernel Patches on Crypto-Algorithms on Windows Platforms", teoksessa *2019 International Conference on Innovation and Intelligence for Informatics, Computing, and Technologies (3ICT)*, 2019, s. 1–6. DOI: 10.1109/3ICT.2019.8910282.

[32] E. Barberis, P. Frigo, M. Muench, H. Bos ja C. Giuffrida, "Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks", teoksessa *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA: USENIX Association, elokuu 2022, s. 971–988, ISBN: 978-1-939133-31-1. url: https://www.usenix.org/conference/usenixsecurity22/presentation/barberis.

[33] F. Murtaza, *Spectre-v2 mitigation wreaks havoc on the performance of some Intel CPUs as AMD chips come out largely unscathed*, maaliskuu 2022. url: https://www.notebookcheck.net/Spectre-v2-mitigation-wreaks-havoc-on-the-performance-of-some-Intel-CPUs-as-AMD-chips-come-out-largely-unscathed.607925.0.html.

[34] C. Trippel, D. Lustig ja M. Martonosi, *MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols*, 2018. arXiv: 1802.03802 [cs.CR]. url: https://arxiv.org/abs/1802.03802.

[35] P. Li, L. Zhao, R. Hou, L. Zhang ja D. Meng, "Conditional Speculation: An Effective Approach to Safeguard Out-of-Order Execution Against Spectre Attacks", teoksessa *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, s. 264–276. DOI: 10.1109/HPCA.2019.00043.

[36] K. Barber, A. Bacha, L. Zhou, Y. Zhang ja R. Teodorescu, "SpecShield: Shielding Speculative Data from Microarchitectural Covert Channels", teoksessa *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2019, s. 151–164. DOI: 10.1109/PACT.2019.00020.

[37] E. M. Koruyeh, S. Haji Amin Shirazi, K. N. Khasawneh, C. Song ja N. Abu-Ghazaleh, "SpecCFI: Mitigating Spectre Attacks using CFI Informed Speculation", teoksessa *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, s. 39–53. DOI: 10.1109/SP40000.2020.00033.

[38] A. Pashrashid, A. Hajiabadi ja T. E. Carlson, "HidFix: Efficient Mitigation of Cache-Based Spectre Attacks Through Hidden Rollbacks", teoksessa *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2023, s. 1–9. DOI: 10.1109/ICCAD57390.2023.10323979.

[39] J. Cho, T. Kim, T. Kim ja Y. Shin, "Real-Time Detection on Cache Side Channel Attacks using Performance Counter Monitor", teoksessa *2019 International Conference on Information and Communication Technology Convergence (ICTC)*, 2019, s. 175–177. DOI: 10.1109/ICTC46691.2019.8939797.

[40] Z. Tong, Z. Zhu, Y. Zhang, Y. Liu ja D. Meng, "KSM: Killer of Spectre and Meltdown Attacks", teoksessa *2024 27th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, 2024, s. 157–162. DOI: 10.1109/CSCWD61410.2024.10580256.

[41] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher ja J. Torrellas, "InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy", teoksessa *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, s. 428–441. DOI: 10.1109/MICRO.2018.00042.

[42] M. Behnia et al., *Speculative Interference Attacks: Breaking Invisible Speculation Schemes*, 2021. arXiv: 2007.11818 [cs.AR]. url: https://arxiv.org/abs/2007.11818.