The Ultimate Survey: Transient Execution Attacks

Luiz Henrique Custódio Mendes Marques
Universidade Estadual de Maringá
Apucarana-PR, Brasil
luhenrique06@hotmail.com

***Abstract*—After the first transient execution attack, such attacks has grown over the years. Transient attacks arise from hardware resources, in fact from lack of security with exposes vulnerabilities. These attacks consist of two phases: (1) an transient channel for data leakage, and (2) a cache side channel for capturing user's data. The industry and the academy have developed techniques to mitigate hardware vulnerabilities, however despite the efforts it is still possible to exploit several microarchitectures. In this context, this survey presents the concepts related to transient execution attacks, describing the phases, the main works in this area and techniques to mitigate such attack.**

***Resumo*—Após a divulgação do primeiro ataque de execução transitória, esta categoria de ataques tem crescido a cada ano, visto que por meio desses ataques é possível vazar dados críticos de usuários. Ataques transitórios consistem em duas fases: (1) a execução de um canal transitório para o vazamento dos dados, e (2) a execução de um canal lateral em cache para a captura das informações. Por meio desse tipo de ataque, é possível vazar dados críticos de usuários. Tais ataques são viáveis pelo fato de aspectos de segurança serem deixados de lado durante a implementação de recursos de hardware. Devido aos potenciais danos causados, tanto a indústria quanto a academia desenvolveram técnicas para mitigar as falhas relacionadas. Apesar desses esforços, ainda é possível explorar brechas que permitem o surgimento de diferentes variantes de ataques. Este trabalho apresenta os principais conceitos relacionados a ataques de execução transitória, descrevendo suas fases de execução, os principais trabalhos da área e diferentes técnicas de mitigação.**

***Palavras-chave*—Ataque de canal lateral, execução transitória, ataque de execução transitória**

I. Introdução

Ao longo dos anos, as indústrias vêm investindo em diversas pesquisas a fim de desenvolver processadores cada vez mais rápidos e sofisticados. Porém nessa busca por desempenho alguns efeitos colaterais, que não foram levados em conta, podem surgir, resultando em diferentes vulnerabilidades que ameaçam a segurança dos processadores e assim abrindo brechas que podem colocar em risco dados sensíveis de usuários.

Diversos ataques transitórios buscam explorar vulnerabilidades em sistemas de informação, visando a obtenção do acesso à informações secretas. De fato, a implementação de otimizações, introduzidas na microarquitetura dos processadores, tais como pipeline, execução fora de ordem e execução especulativa [1], abriram brechas que podem ser exploradas. Ataques como *meltdown* [2] e *spectre* [3] demonstram que é possível forçar uma execução especulativa a fim de realizar o acesso à dados confidenciais. Ao executar uma ação incorreta de forma especulativa o processador reverte para o estado anterior, porém mantém resquícios dos dados acessados indevidamente em *buffers* da microarquitetura, que podem ser recuperados posteriormente. Conforme destaca Canella e outros [4] esses efeitos colaterais levaram a divulgação de uma série de ataques [5] [6] [7] [8] que buscam permitir a leitura de locais arbitrário da memória de *kernel* do sistema operacional (SO) ou de outros processos, possibilitando extrair dados sensíveis dos usuários sem a necessidade de uma escalada de privilégios.

Por conta dos danos que tais ataques podem causar, a indústria e a academia buscaram elaborar diferentes técnicas em nível de software [9] [10] e hardware [11] a fim de detectar e prevenir ataques transitórios. Contudo, Schwarz e outros [12] explicam que apesar das atualizações de segurança, ainda é possível explorar algumas vulnerabilidades e desta forma vazar não apenas dados da memória cache, mas também de *buffers* de armazenamento.

Este artigo tem como objetivo fornecer uma visão geral dos fundamentos necessários para compreender o funcionamento dos ataques de execução transitória (AETs). Além disto, é apresentado um resumo dos principais ataques, como também das técnicas de defesa em nível de hardware e software. Desta forma, as contribuições deste artigo são descritas a seguir.

1) Descreve o funcionamento dos AETs.
2) Apresenta os principais trabalhos da área.
3) Descreve diferentes técnicas de mitigação e detecção.
4) Discute os principais pontos em aberto em relação aos trabalhos analisados.

Este artigo inicia (Seção II) fornecendo ao leitor a definição de conceitos essenciais para o entendimento de AETs. Vale também ressaltar que um AET utiliza em uma de suas fases um ataque de canal lateral baseado em cache (ACLC). Esta categoria de ataque visa capturar dados armazenados em cache por meio da análise de tempo de acesso aos dados. Visto que, ataques transitórios buscam acessar os dados e carregá-los

em cache é fundamental compreender os ACLCs para entender o funcionamento dos ataques transitórios. Tendo isto em mente, a Seção III descreve os fundamentos de ACLCs como também apresenta os principais trabalhos e suas contramedidas. Após apresentar tais fundamentos, a Seção IV apresenta os principais AETs, descrevendo suas principais variantes e técnicas de mitigação em hardware e software. Por fim, as considerações finais são apresentadas na Seção V.

II. Fundamentos

Os AETs surgem de efeitos colaterais de otimizações em nível de hardware, as quais visam aumentar o desempenho dos processadores. Otimizações como execução fora de ordem e execução especulativa permitem a indução de falhas conhecidas como execução transitória, na qual durante um curto período de tempo é possível acessar dados confidenciais de forma arbitrária [4].

Execução fora de ordem permite a execução de instruções mais abaixo do fluxo de execução, as quais estão prontas para serem executadas devido a disponibilidade dos operandos.

Execução especulativa permite a unidade central de processamento (UCP) prever o resultado de uma possível alteração no fluxo de execução, antes de efetivamente verificar o resultado de instruções condicionais.

Execução transitória ocorre quando uma instrução é executada de forma especulativa, assim alterando o estado da microarquitetura antes da confirmação do valor predito. Ao confirmar que uma instrução foi executada pelo caminho incorreto nem todos os efeitos colaterais da microarquitetura são limpos adequadamente, permitindo assim os AETs. Qualquer instrução que cause obstrução do *pipeline* gera uma execução transitória.

A execução transitória combinada com um ACLC resulta em um AET. Durante a execução transitória dados confidenciais ficam disponíveis e podem ser carregados para *buffers* da microarquitetura, possibilitando a captura com um ataque em cache. O ACLC consiste na análise de acesso aos dados armazenados nos diferentes níveis da cache. Tal análise ocorre com a medição do tempo diferenciando *cache hits* de *cache misses* [13].

III. Ataques de Canal Lateral em Cache

Para a execução de ataques transitórios são necessários ACLCs, para capturar os dados acessados de forma arbitrária durante a execução especulativa e carregados para a cache. Assim permitindo que o invasor tenha acesso a tais dados. ACLC é um ataque que depende da recuperação de dados secretos analisando a cache, em vez de explorar vulnerabilidade nos algoritmos. Este é um tipo de ataque de canal lateral (ACL) em que o invasor tem como alvo a cache do processador da vítima, para o vazamento de informações.

Canais laterais baseados em tempo têm sido explorados desde os anos 90, o primeiro artigo publicado com a proposta de usar a memória cache como um canal lateral foi escrito por Hu [14] . Ele foi o primeiro a sugerir que a diferença entre a ocorrência de um *cache miss* e um *cache hit* poderia ser usada para capturar informações sobre o estado interno do sistema.

Esses ataques comumente são divididos em três etapas. Como descreve Bazm e outros [15] no primeiro momento, o invasor preenche as linhas de cache da vítima com instruções falsas a fim de que, quando o processador procurar alguma instrução na cache este não encontre e tenha que acessar a RAM. No segundo momento, o invasor aguarda alguns ciclos de *clock* até que a vítima realize algumas operações. Por fim, o atacante acessa novamente as mesmas linhas de cache com o propósito de saber quais foram modificadas e quais estão sendo utilizadas pela vítima para a execução de alguma operação, e, desse modo, atacá-las para roubar informações, como chaves de algoritmos de criptográficos.

Aciiccmez e outros [16] demonstram como a diferença de tempo é crucial para que seja possível a realização dos ACLCs, mostrando de que maneira as falhas de cache aumentam o tempo de execução dos programas. Em um exemplo prático, os pesquisadores demonstram um ataque contra a implementação do OpenSSL RSA (*Rivest-Shamir-Adleman*), aproveitando o fato de que o cálculo de multiplicações modulares e operações quadradas utilizam funções diferentes. Essas diferentes funções deixam rastros desconformes na cache de instruções. Com isso, o atacante é capaz de identificar se a vítima realizou uma operação de multiplicação ou de raiz.

Os ataques em cache podem ser categorizados em duas classes e são baseados na medição da diferença de tempo dos *cache hit* e *cache miss*, porém em escalas diferentes.

Time-driven neste tipo de ataque, o atacante mede o tempo total da execução de operações para, assim, extrair informações confidenciais [17].

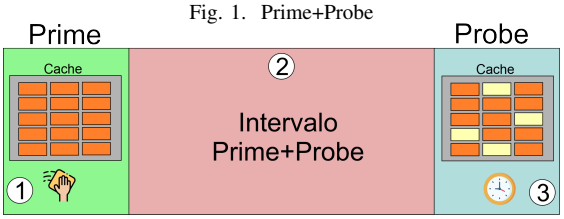
Access-driven neste tipo de ataque, o atacante monitora, em tempo de execução, o acesso da vítima a um componente específico da cache [10].

As principais técnicas que podem ser utilizadas para realizar ataques Time-driven e Acess-driven são descritas a seguir.

Prime+Probe Foi desenvolvida inicialmente por Percival [18] e Osvik e outros [19]. Esta consiste no invasor identificar quais conjuntos de cache são acessados pela vítima. Então o atacante executa um monitora-

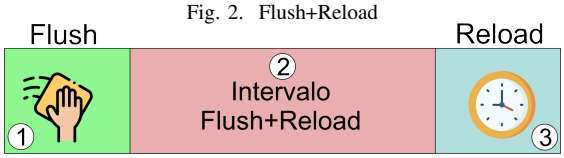
mento de uso da cache da vítima da seguinte maneira. Na fase de `prime`, o invasor preenche alguns conjuntos de cache (ou todo) com seu próprio conjunto de dados, então este aguarda um determinado tempo enquanto a vítima utiliza a cache. Na fase seguinte, `probe`, o invasor acessa os mesmos dados que foram carregados no estágio `prime`. Caso a vítima tenha carregado algum dado para as mesmas linhas de dados do invasor, este terá que esperar um longo tempo devido a um cache `miss`, por outro lado, se os dados ainda estiverem em cache, este terá uma resposta rápida, o que corresponde a um *cache hit* [20] [21].
A Figura 1 exemplifica as três fases que compõem a abordagem `Prime+Probe`, na qual inicialmente o atacante preenche a cache da vítima com os seus dados, então aguarda uma porção de tempo até que a vítima utilize as linhas de cache, e por fim acessa novamente medindo o tempo, as linhas nas quais o atacante tiver um maior tempo de resposta correspondem as linhas utilizadas pela vítima.

Fig. 1. Prime+Probe



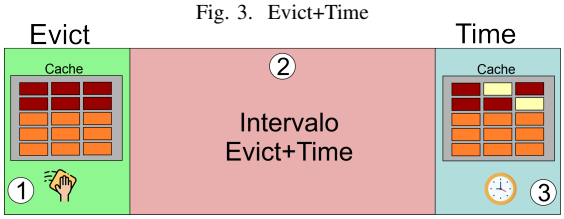
Flush+Reload Foi descrita por Gullasch e outros [22] e Yarom e Falkner [23]. Visa explorar a fraqueza de monitorar o acesso a linhas da memória em páginas compartilhadas, e é comumente utilizada para explorar o *last-level cache* (`LLC`) que é compartilhado por todos os núcleos da unidade central de processamento (`UCP`), sendo uma variante do `Prime+Probe`. O ataque consiste em 3 (três) fases: `flush, wait e reload`. Na fase `flush`, é eliminados todos os dados da linha de cache que tem a instrução de código monitorado. Então, aguarda por um período chamado de `slot time`, por fim, acessa a instrução de código monitorada e calcula o tempo de ciclos para esse acesso. A técnica `Flush+Reload` é a principal utilizada para a exploração da vulnerabilidade *meltdown*. [24] [25] [26].
A Figura 2 apresenta o plano de execução de um ataque `Flush+Reload`, inicialmente o atacante realiza a limpeza da cache utilizando a instrução `clflush`, então aguarda um intervalo de tempo até que a vítima carregue seus dados para a cache, por fim o atacante tenta novamente acessar o dados realizando a medição de tempo.

Fig. 2. Flush+Reload



Evict+Time Tem como objetivo verificar se a vítima tem um tempo médio de execução mais longa, devido às falhas na cache durante um número de tentativas. Na primeira fase, `evict`, o atacante preenche algumas linhas de cache que contém dados críticos de segurança da vítima (dados de criptografia). Então, na segunda fase, `time`, o atacante observa se a vítima usa os dados despejados. Caso esses dados sejam utilizados, uma falha na cache será ocasionada. Essa falha aumenta o tempo de execução da vítima para a operação crítica de segurança, e isso poderá ser observado pelo invasor mensurando o tempo total de execução das operações de criptografia [27].
A Figura 3 exemplifica o ataque `Evict+Time`, no qual o atacante preenche algumas porções de cache com dados criptográficos, aguarda um até que a vítima acesse essa porção, e por fim realiza a medição de tempo no acesso a esses dados os dados que tiverem o menor tempo de resposta correspondem aos dados compartilhados com a vítima.

Fig. 3. Evict+Time



Existem outras técnicas de `ACLCs`, menos populares como: Evict+Reload [28], Flush+Flush [29], Evict+Time [19], Prime+Abort [30], Evict+Abort [31], Cache-Collision [32].

IV. Ataques de Execução Transitória

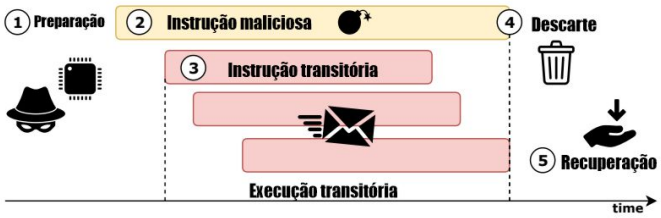
Processadores superescalares implementam a execução fora de ordem, permitindo que a `UCP` utilize diferentes unidade de execução em paralelo. Neste, o fluxo de instruções é decodificado em ordem para micro-operações mais simples que podem ser executadas assim que os operandos necessários

estiverem disponíveis [33]. Qualquer falha que ocorra durante a execução de uma instrução fora de ordem é tratada e desativada. Outra funcionalidade comum nos processadores modernos é a execução especulativa, que busca evitar o bloqueio do pipeline de uma instrução até que a ramificação condicional seja resolvida, para isso, a UCP busca prevê o resultado da condicional e continuar a execução na direção escolhida. Caso a direção predita não seja a correta todas as operações realizadas são descartadas, os resultados dessas operações são conhecidos como instruções transitórias [34].

Apesar de ocorrer o descarte das operações caso ocorra uma falha nas instruções transitórias, alguns efeitos colaterais microarquiteturais podem permanecer e não serem revertidos. Os ataques que buscam explorar esses efeitos colaterais para obter informações confidenciais são chamados de ataques de execução transitória. Tais ataques utilizam de ACLC para inferir dados secretos capturados transitoriamente do domínio da microarquitetura para o estado arquitetônico. Conforme destaca Canella [9] ataques que exploram erros de previsão são categorizados como do tipo *spectre*, enquanto ataques que exploram a falha após a execução são classificados como pertencentes do tipo *meltdown*.

A Figura 4 adaptada do trabalho de Canella [4] apresenta a visão geral de um AET. O atacante prepara a microarquitetura e executa uma instrução maliciosa a qual irá ativar algum mecanismo de execução transitória da UCP, a instrução transitória é executada e cancelada, com isso são descartadas as operações realizadas, entretanto dados confidenciais já foram carregados para o estado microarquitetural, por fim o invasor reconstrói e captura essas informações.

Fig. 4. Fases Ataque Transitório



A. Ataque Meltdown

A vulnerabilidade *meltdown* explora uma falha de projeto no mecanismo de execução fora de ordem, presente em grande parte dos processadores modernos. Tal falha permite que um processo ignore as verificações padrões de privilégio [9].

A execução fora de ordem foi descrita de forma completa por Smith e outros [35] e a partir da década de 90 esse mecanismo foi implementado em grande parte dos processadores lançados. Entretanto, Silbert e outros [36] detectaram os primeiros sinais de falhas em processadores da Intel ao observarem a presença de um canal de temporização na cache da UCP, sendo possível assim obter acesso a áreas que deveriam estar protegidas como por exemplo, registradores de ponto flutuante.

Em 2012 os sistemas da Apple receberam atualizações de segurança com a introdução do KASLR (*Address space layout randomization*) sendo uma técnica que consiste em tornar aleatório o espaço de memória no qual o kernel é mapeado, possui a capacidade de mitigar vulnerabilidades no acesso a memória. Posteriormente em 2014, o KASLR foi adotado também pelo *kernel* Linux. Lipp e outros [17] entretanto demonstram que é possível burlar os mecanismos de segurança do KASLR em processadores ARM, apresentando um ataque capaz de monitorar eventos, bem como o pressionamento de teclas em telas sensíveis ao toque.

Após diversos estudos, no início de 2018 Lipp e outros [2] descreveram uma vulnerabilidade capaz de vazar dados de microprocessadores Intel x86, além de alguns modelos de processadores ARM. Tal vulnerabilidade foi denominada de *meltdown*.

A vulnerabilidade *meltdown* explora os efeitos colaterais gerados pelo processo de execução fora de ordem, na qual a partir da leitura de locais arbitrários da memória cache, é possível extrair dados pessoais e senhas sem permissões ou necessidade de escalação de privilégios.

O ataque consiste na execução de 3 passos, como descritos a seguir.

1) O atacante seleciona o conteúdo de uma posição de memória e carrega a mesma para um registrador.
2) Uma instrução transitória acessa uma linha da memória cache, baseado no conteúdo secreto do registrador.
3) O atacante aplica uma técnica de Flush+Reload para determinar a linha de cache acessada e o segredo armazenado na posição de memória desejada.

Ao realizar estes passos diversas vezes, é possível vazar dados de diferentes posições ou até mesmo toda a memória física. Para exemplificar, 1 (um) byte, denominado segredo, é alocado em um endereço arbitrário de memória (RCX). Esse byte representa um dado no espaço de kernel que será capturado. Então o atacante inicia armazenando um vetor de 256 posições no qual cada posição comporta 4096 bytes, que corresponde ao tamanho padrão de uma página da memória virtual. Após preparar o espaço de memória, o atacante busca garantir que o vetor de dados não esta armazenados em cache a fim de forçar o processador a buscar os dados na memória RAM quando acessados. Para isso pode-se utilizar instruções específicas de processadores para limpeza de cache por exemplo a função clflush, após alocar o vetor e garantir que o mesmo não está em cache o atacante executa o código apresentado no

Algoritmo 1.

Algorithm 1: Ataque Meltdown

```
Function Meltdown():
    MOV AL, BYTE[RCX]
    CALL CLFLUSH
    SHL RAX 0xC
    MOV RBX, WORD[RBX+RAX]
    CALL FLUSHRELOAD
End Function
```

Com este algoritmo, o atacante induz uma falha ao tentar acessar o dado e carregar seu valor para o byte menos significativo do registrador `RAX`. A gerência, por parte do processador, de tal falha irá ocasionar um rastro na cache. Assim cabe ao invasor executar um `ACLC` como o ataque `Flush+Reload` para identificar quais foram as posições da cache utilizadas para armazenar o valor do byte segredo, e assim capturar o dado. Seguindo os passos descritos o invasor é capaz de ler toda a memória física do alvo byte a byte.

Apesar dos esforços da indústria em mitigar o *meltdown* diversos trabalhos demonstram que ainda é possível explorar a execução fora de ordem para executar código malicioso de forma transitória sem a necessidade de qualquer tipo de permissão em nível de `SO` [5] [6] [37].

B. Ataque Spectre

Kocher e outros [3] descrevem uma nova vulnerabilidade que visa explorar o processo de execução transitória. Enquanto o *meltdown* descrito por Lipp e outros [2] explora a execução fora de ordem, o ataque *spectre* visa explorar o canal lateral presente na execução especulativa. Tal ataque, em todas as suas variantes, permite ler todo o conteúdo da memória em nível de *kernel*.

O *spectre* explora o fato de que o estado da microarquitetura não é totalmente restaurado uma vez que as instruções são executadas de forma especulativa devido ao desvio de predição, deixando assim rastros na hierarquia de cache, permitindo que um invasor possa extrair com facilidade essas informações. A vulnerabilidade *spectre*, descrita por Kocher e outros [38], apresenta 2 (duas) variações, sendo que a diferença nessas variações está na forma como a execução especulativa é induzida e controlada, como também a forma como as informações são vazadas.

Conforme descreve Koruyeh e outros [8] para que tal ataque seja bem sucedido, 2 (duas) fases precisam ser cumpridas. A primeira fase é a de configuração, na qual o invasor busca liberar a memória cache, como também treinar o preditor de desvios do processador. Para isso são realizados diversos desvios condicionais semelhantes ao que será utilizado para obter as informações confidenciais, desta forma sendo possível manipular o previsor e induzir a execução de forma especulativa pelo caminho desejado.

A segunda fase é a de exploração, nela o atacante executa uma instrução condicional que compara dados que não estão carregados na cache, assim induzindo que as instruções dentro da instrução condicional sejam executadas de forma especulativa, dado que o previsor assume como verdadeira a condição. Assim o conteúdo do laço será executado até que o processador verifique que a condição do desvio é falsa. O próximo passo é descobrir o valor armazenado na cache, da mesma forma que no *meltdown*.

O Algoritmo 2, demonstra de forma simplificada o conceito principal da primeira variante do *spectre*.

Algorithm 2: Ataque Spectre

Input: $array1size, array1, array2, x, y$
Function `spectre()`**:**
    if $x > array1size$ **then**
        $y \longleftarrow array2[array1[x]];$
End Function

Após realizar o treinamento do preditor de desvio, para garantir que o mesmo sempre especule o valor da condicional como verdadeiro. O atacante garante que o valor de `array1Size` não está em cache e assim realiza uma chamada condicional com o valor de `X` extrapolando os limites de `array1Size`. Após a `UCP` verificar que a cache não possui o valor de `array1Size` para realizar o desvio condicional, as instruções dentro do laço são executadas de forma especulativa pois o preditor de desvio assume a condição como verdadeira. Assim durante a execução especulativa é realizada a leitura do elemento apontado por `array1[x]`, que excede os limites do processo atual, acessando dados de um processo vizinho e carregando os dados do mesmo para a cache, enquanto o processador resolve o cálculo da predição que depende do valor de `array1Size`. Após a `UCP` encontrar o dado e verificar que a condição é falsa, todas as operações serão desfeitas e será reiniciado o processo, porém os dados capturados continuaram na cache, permitindo a execução de um `ACLC` para recuperar os dados. Para realizar o ataque são necessárias 3 condições.

1) A variável `x` seja controlada pelo atacante.
2) O `array1Size` não esteja armazenado em cache.
3) O `array1` esteja em cache.

Essas condições fazem com que o acesso à memória na verificação da instrução condicional seja mais lento, do que a execução especulativa ao recuperar o valor armazenado no `array1[x]`.

O processo de execução do ataque é muito semelhante ao *meltdown*. A principal diferença é que, como todo o ataque é executado dentro de um desvio condicional, não haverá exceção lançada e o programa vai apenas reverter seu estado para antes da execução da instrução, mantendo assim a cache intacta.

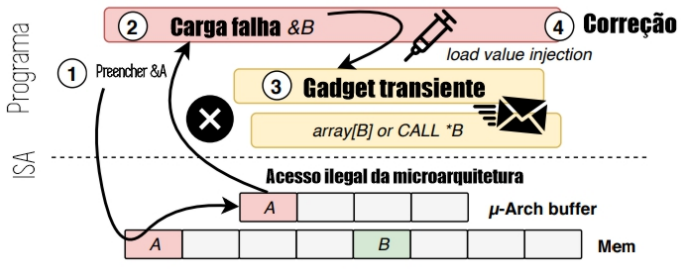
A segunda variante do ataque busca explorar desvios incondicionais seguindo a mesma linha de execução da variante com desvios condicionais, a diferença está apenas na manipulação da execução especulativa. Ao invés de manipular desvios condicionais para a execução especulativa assumir como verdadeira uma condição falsa, essa variante treina o preditor de desvio fazendo vários desvios incondicionais para um endereço onde existe um código explorável pelo *spectre*, de forma a garantir que o salto incondicional do ataque ocorra da forma desejada. Krocher e outros [3] demonstram que o `BTB` (*branch target predictor*) pode ser treinado para prever erroneamente um ramo de uma operação de salto indireto, assim o invasor precisa apenas fazer o preditor aprender esse salto indireto para levar até o dado alvo.

Após a divulgação diversos outros trabalhos foram publicados demonstrando diferentes formas de explorar a execução especulativa [7] [39] [40] [8].

C. Ataque LVI

Bulck e outros [37] descrevem um novo ataque baseado no *meltdown*, denominado como *Load Value Injection* (`LVI`)[1], tal ataque permite que os atacantes injetem valores arbitrários em certas estruturas microarquiteturais, que são utilizadas pela vítima, permitindo levar a revelação de dados sensíveis. O ataque proposto busca gerar um colapso reverso, enquanto no ataque *meltdown* é possível que o atacante leia todos os dados de um aplicativo na memória da `UCP`, o `LVI` permite que o invasor injete dados para execução na `UCP`. Tal ataque permite que o invasor possa até mesmo assumir o controle da máquina destino. Conforme demonstra a Figura 6 adaptada do trabalho de [37] o ataque pode ser dividido em 4 fases.

Fig. 5. Ataque LVI



[1]https://lviattack.eu/

1) Na primeira fase o atacante prepara a microarquitetura preenchendo um `buffer` oculto com um valor controlado por ele.
2) E em um segundo momento é induzida uma falha no programa da vítima evocando assim uma execução transitória.
3) Durante a execução transitória o valor do invasor é injetado temporariamente em um *gadget* permitindo que instruções maliciosas sejam executadas. Dependendo do código *gadget* o `LVI`, pode codificar dados secretos ou servir como um controle de fluxo de dados para facilitar a exploração do dispositivo em um segundo estágio, como por exemplo utilizar um dado como ponteiro.
4) Na quarta fase os cálculos realizados durante a execução transitória são descartados entretanto rastros foram deixados no estado de microarquitetura da `UCP`, que posteriormente podem ser recuperados por meio de canais laterais.

O Algoritmo 3 demonstra a base do ataque `LVI` apresentado por Bulk e outros [37].

Algorithm 3: Ataque LVI

Input: $valorMalicioso$
Function `callVitima()`**:**
  $*argCopy \longleftarrow valorMalicioso$;
  $array[**ponteiroConfiavel * 4096]$;
End Function

O código é semelhante ao do ataque *spectre*, porém sem a necessidade da indução de erros no preditor. Inicialmente o invasor instância um valor de 64 bits (`valorMalicioso`) em um espaço de memória confiável (por exemplo, na pilha) que será passado como argumento na chamada da função transitória, (na linha 2). Essa operação é utilizada para trazer um valor controlado pelo invasor para algum *buffer* da microarquitetura. Posteriormente, um ponteiro para ponteiro (`ponteiroConfiavel`) é desreferenciado (linha 3), no qual o mesmo é multiplicado por 4096 a fim de evitar o *prefetch*, assim impedindo que dados de páginas vizinhas sejam carregados para o espaço da microarquitetura. Então o atacante presume que após a desreferencia do ponteiro a vítima sofre uma falha de página ou microcódigo. A falha faz com que a `UCP` encaminhe incorretamente o valor (`valorMalicioso`) do invasor que foi anteriormente trazido para o *buffer* de armazenamento como em um ataque do tipo *meltdown*. Assim o invasor consegue substituir o valor pretendido arquitetonicamente no endereço `*ponteiroConfiavel` com seu próprio valor escolhido. Por fim o código do *gadget* utiliza temporariamente

valorMalicioso como o endereço base para uma desreferência de ponteiro de segundo nível e usa o resultado como um índice em uma tabela de pesquisa.

A atual prova de conceito do ataque LVI é executado de forma nativa, sugerindo que para a sua execução seja necessário o acesso local por meio de um *malware*. No entanto, Bulck e outros [37] alertam que um ataque remoto também é possível via Javascript, semelhante ao ataque *spectre*. Atualmente, apenas UCPs da Intel são afetados pelos novos ataques LVI, entretanto os pesquisadores não descartam que UCP da AMD e ARM também possam ser afetados. Conforme declara Gruss e outros [41] em princípio, qualquer processador vulnerável a vazamento de dados por *meltdown* também seria vulnerável a injeção de dados por LVI.

Apesar da prova de conceito apresentada por Bulck e outros [37], a Intel alega que devido aos inúmeros requisitos complexos que devem ser satisfeitos para a implementação de um ataque do LVI, o mesmo não é prático para uma exploração no mundo real, sendo apenas uma ameaça teórica. A Intel também aponta que os desenvolvedores devem ter cautela ao aplicar métodos de mitigação para o LVI.

D. Ataque ZombieLoad

Schwarz e outros [12] busca demonstrar que apesar das correções de hardware implementadas pela Intel nos novos processadores, estes ainda continuam vulneráveis a ataques do tipo *meltdown*. Para isso os autores apresentam um novo tipo de ataque baseado na falha, denominado *ZombieLoad Attack*[2] . Este é o primeiro a demonstrar a possibilidade de vazamento de dados carregados e armazenados recentemente em núcleos lógicos, sendo possível realizá-lo mesmo em processadores resistentes a *meltdown*. O ataque apresentou eficiência em capturar informações de diferentes níveis dos anéis de privilégio dos SO assim se demonstrando um ataque extremamente poderoso, visando a lógica de preenchimento de *buffers*.

O *ZombieLoad* é um AET, ou seja, com este é possível vazar informações inacessíveis por meio do estado micro arquitetônico do processador a partir de instruções que nunca são confirmadas. Como outros ataques no *microtectural data sampling* (MDS), o *Zombieload* é um ataque baseado no método de análise de dados a partir de estruturas microestruturais. O ataque consiste em o atacante liberar uma quantidade de dados ao processador, a fim do mesmo não conseguir entender ou processar os dados enviados, assim este é forçado a pedir ajuda aos microcódigos do processador para evitar uma exceção ou um travamento. Então, quando a microarquitetura solicita assistência de microcódigo, ela pode primeiro ler valores obsoletos antes de ser reemitida eventualmente. Assim como em qualquer ataque do tipo *meltdown*, isso abre uma janela de execução transitória, na qual esse valor pode ser usado para cálculos subsequentes, permitindo então que um invasor possa codificar este valor vazado em elementos microarquiteturas, como a cache.

Conforme descreve os autores, diferentemente de outros ataques, com o *ZombieLoad*, não é possível selecionar o valor a vazar com base em um endereço específico, sendo que o ataque simplesmente vaza qualquer valor atualmente carregado ou armazenado pelo núcleo físico da UCP. Entretanto os autores destacam que apesar de isto parecer uma limitação, isso abre um novo campo de AET baseado em amostragens de dados. Além disso o ataque proposto considera todos os limites de privilégios e não se limita a um específico como em ataques anteriores.

Schwarz e outros [12] apresenta 3 diferentes variantes do ataque, sendo a primeira uma configuração do *ZombieLoad* que não depende de nenhum recurso específico da UCP, sendo necessário apenas um endereço virtual do *kernel*. Na segunda variante são introduzidas cargas zumbis, que eliminam a necessidade de um mapeamento de *kernel* necessitando apenas de uma página física acessível ao usuário por um endereço virtual. Na terceira variante, semelhante a variante 1, não é necessário nenhum recurso específico e é acionada a partir de uma rodada pela tabela de páginas assistidas por microcódigos. Os testes desenvolvidos apresentam resultados satisfatórios em todos os cenários testados, demonstrando que com o *ZombieLoad* é possível observar valores de todos os processos em execução no mesmo núcleo lógico da UCP. Além disso, também foi possível observar a captura de amostra de todos os dados carregados ou armazenados por qualquer aplicativo no núcleo físico atual da UCP. Os autores também demonstram a capacidade do ataque em capturar dados de navegação, para isso foi adaptado o ataque para detectar sequências de bytes específicas dos dados carregados. Com isso demonstram que é possível realizar a detecção de palavras chaves pesquisadas nos navegadores, além da recuperação do localizador uniforme de recursos (URL) para monitorar o comportamento de navegação da vítima em tempo real, com taxas de falso-positivos abaixo de 12%.

E. Ataque Echoload

O trabalho de Canella e outros [9] surge com uma nova abordagem de ataque capaz de burlar os sistemas de segurança presente nos novos processadores da Intel. Os processadores a partir da linha *cascade lake* receberam atualizações de hardware capazes de mitigar todos os tipos de ataques *meltdown* conhecidos. Entretanto, após a análise dos principais métodos

[2]https://zombieloadattack.com/

de prevenção, os autores apresentam uma nova técnica denominada *echoLoad*, que é capaz de detectar endereços com suporte físico de aplicações sem privilégios, quebrando o KASLR e assim burlando os mecanismos de prevenção de *meltdown* e MDS. O *echoLoad* em suma, explora os efeitos colaterais relacionados ao *meltdown* para atacar o KASLR. Sendo que o ataque interrompe o KASLR, independentemente do SO, das defesas de software e das atualizações de microcódigos. A ideia geral é distinguir se o acesso a um endereço de *kernel* no domínio de execução transitória, leva a uma paralisação conforme pode ser conferido no Algoritmo 4.

Algorithm 4: Ataque Echoload

Input: $mem, *address$
Function `Echoload()`**:**
  if $transientBegin()$ **then**
    $*(volatilechar*)(mem + *address)$;
  if $flushReload(mem)$ **then**
    return addressMapped;
  else
    return addressNotMapped;
End Function

Inicialmente o atacante induz a execução transitória provocando uma falha ou especificação incorreta. Caso o acesso executado pare, o endereço do usuário não poderá ser calculado antes que a execução transitória seja interrompida. Caso contrário, o endereço do usuário é desreferenciado e, portanto, armazenado em cache antes da interrupção da execução. Com isso, após a execução transitória o atacante investiga para descobrir os blocos utilizados pelo usuário por meio da técnica Flush+Reload. Se o endereço do usuário estiver em cache, o endereço é válido, ou seja, com suporte físico, caso contrário, o endereço não é válido, assim retornado que o não foi mapeado.

Em linhas gerais o ataque *echoload* explora como função de falha da execução transitória um canal do tipo *spectre*, isso permite que os invasores manipulem o estado da unidade de previsão de ramificação e abusem da ramificação errônea para vazar dados arbitrários dentro do espaço de endereços acessível por meio de canais laterais. Porém esta premissa por si só é útil apenas em cenários de fuga de *sandbox*, sendo assim necessário a implementação de *gadgets* que serão executados especulativamente capazes de divulgar espaços da memória de maneira cruzada.

Os testes foram realizados em processadores domésticos e de servidores, como: Intel Core i9-9900k, Intel *Xeon Silver* 4208, Intel *Cascade Lake* (Google Cloud). O *echoload* nos testes realizados em processadores com as correções do *meltdown* obteve sucesso para distinguir as páginas utilizadas pelo *kernel* em todos os casos, com uma taxa de 0% de falsos positivos. Os autores também demonstram a prova de conceito do primeiro ataque remoto *meltdown* desenvolvido em Javascript, capaz de extrair informações de plataformas com sistema operacional x86 de 32 bits, que ainda não estão amplamente protegidos contra a falha. Ambos os ataques demonstrados foram capazes de extrair dados confidenciais nos ambientes testados. Isto indica que o *echoLoad* é um ataque muito confiável podendo ser executado até um processador de última geração.

F. Contramedidas

Diante dos impactos do *meltdown*, grandes empresas de servidores em nuvem, SOs e fabricantes de processadores foram forçados a desenvolverem soluções de segurança. Entretanto tais soluções podem reduzir significativamente o desempenho dos processadores especialmente em computadores mais antigos, apresentando perdas entre 2% e 30% [4].

Meltdown ignora o isolamento imposto pelo hardware, não havendo à vulnerabilidade de software, sendo assim qualquer correção via software não mitiga por completo a falha. Uma contramedida trivial seria desabilitar o mecanismo de execução fora de ordem, entretanto causaria um impacto direto no desempenho dos processadores modernos. Segundo Lipp e outros [2] uma possível mitigação seria realizar a verificação de permissões antes da busca do registro alvo, todavia isso poderia causar uma sobrecarga significativa para cada busca da memória, dado que haveria uma parada até que a verificação da permissão fosse concluída. Uma das primeiras correções aplicadas pela indústria foram as atualizações de microcódigos com o intuito de diminuir a precisão do temporizador, além de pequenas modificações no mecanismo da execução fora de ordem. Entretanto, a maioria dos AETs não podem ser corrigidos com atualizações de microcódigos. Nos processadores mais recentes, a Intel introduziu mitigações de hardware que visam mitigar de forma completa ataques *Meltdown*.

Os SOs Linux foram os primeiros a implementar mitigações baseadas em software. Essas mitigações buscam introduzir uma separação mais rigorosa do espaço de *kernel* e usuário, assim impedindo a manipulação e acesso a dados fora do anel de permissão do usuário. O mecanismo implementado para isso é KAISER[3], que consiste em uma modificação no *kernel* de forma que ele não seja mapeado para o espaço de usuário. Esta modificação foi inicialmente proposta por Schwarz e outros [12] para evitar ataques de canal lateral. No entanto constatou-se que ela pode também impedir o *meltdown*, pois garante que não haja mapeamento válido para o espaço do *kernel* ou para a memória física disponível no espaço do usuário. Apesar do KAISER ser uma abordagem viável,

[3]https://github.com/IAIK/KAISER

sua implementação não é possível para todas as arquiteturas, uma vez que no design da arquitetura x86, vários locais de memória do *kernel* ainda precisam ser mapeados para o espaço do usuário, deixando uma superfície de ataque residual para o meltdown. Os sistemas operacionais da Apple e Microsoft também receberam atualizações baseada no `KAISER` chamada `LAZARUS` descrita por Gens e outros [5] que consiste no mudança do padrão de mapeamento de páginas na troca de contexto do `SO`.

Trabalhos como de Bilal [42] propõe a utilização de modelos de aprendizagem para detecção em tempo real de ataques do tipo *meltdown*. A abordagem consiste em utilizar contadores de hardware e eventos de software para monitorar atividades relacionadas à execução especulativa, previsão de ramificações e interferências em cache. Segundo os autores esses eventos produzem padrões distintos quando o sistema está sobre ataque, sendo que o modelo proposto é capaz de identificar ataques com uma precisão de 99%. Apesar da eficácia garantida pelo autor, mitigações com aprendizagem de máquina ainda não foram incorporadas pela indústria.

Para ataques *spectre*, assim como para o ataque *meltdown*, a solução mais trivial para esta falha está em desativar o mecanismo de execução especulativa. No entanto, esta solução se demonstra inviável, pois comprometeria o desempenho do sistema. Alternativamente, a solução de software é a utilização da instrução `lfense` que bloqueia o uso da execução especulativa. Para contornar parte da falha a Intel e `AMD` implementaram o *Indirect Branch Restricted Speculation* (`IBRS`) que busca impedir que códigos sem privilégios possam influenciar a previsão de desvio em seções de código com privilégio, outra medida foi a restrição do compartilhamento do mecanismo de previsão de desvio entre núcleos, assim impedindo que um processo possa influenciar na previsão de desvio de outro nos núcleos vizinhos. Kocher e outros [3] propõem impedir que os dados sejam mapeados para a microarquitetura até que a operação executada de forma especulativa verifique os acessos, rastreando os dados carregados durante a execução e impedindo o seu uso em operações subsequentes.

V. Considerações Finais

Apesar da execução especulativa aumentar o desempenho dos processadores modernos, tal mecanismo permite a execução de códigos maliciosos que visam capturar dados confidenciais de usuários. Fica evidente pelos trabalhos aqui apresentados que existe uma variedade de técnicas que podem ser utilizadas para vazamento de informações, as quais tornam-se mais efetivos ano após ano.

Novos ataques transitórios surgem todos os dias, explorando diferentes *buffers* da microarquitetura. Mecanismos de defesa apropriados geralmente não estão disponíveis ou não podem ser implementados facilmente. Embora existam diversos trabalhos propondo contramedidas e essas estejam prontamente disponíveis, é necessário o esforço dos desenvolvedores de hardware para implementar em larga escala mecanismos de defesa, buscando formas de evitar que instruções de falha sejam capazes de serem executadas de forma especulativa.

Pode-se notar que os atuais trabalhos, ainda apresentam certa deficiência em demonstrar a praticidade dos ataques realizados, visto que alguns dos ataques mencionados funcionam apenas em uma arquitetura e/ou software específico. Além da falta de informações sobre os ambientes utilizados para a realização dos testes. Assim, ainda existe uma grande carência de exemplos convincentes. Portanto, apesar dos esforços da indústria e da acadêmia, ainda é possível capturar dados por meio da execução transitória. Desta forma, é necessário que pesquisas futuras pensem cuidadosamente sobre novas formas de mitigação para os ataques já divulgados como também variantes futuras.

Referências

[1] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. New York, USA: Elsevier, 2011.

[2] LIPP, M. SCHWARZ, M. GRUSS, D. PRESCHER, T. HASS, W. FOGH, A. HORN, J. MANGARD, S. KOCHER, P. GENKIN, D. YAROM, Y. HAMBURG, "Meltdown: Reading kernel memory from user space," *In: Proceedings of the Usenix Security Symposium*, pp. 973–990, 06 2018.

[3] KOCHER, P GENKIN, D GRUSS, D HASS, W HAMBURG, M LIPP, MANGARD, S PRESCHER, T SCHWARZ, M YAROM Y, "Spectre attacks: Exploiting speculative execution." *In Proceedings on Security and Privacy Symposium*, pp. 1–19, 06 2019.

[4] CANELLA, C., VAN BULCK, J., SCHWARZ, M., LIPP, M., VON BERG, B., ORTNER, P., PIESSENS, F., EVTYUSHKIN, D., and GRUSS, D., "A systematic evaluation of transient execution attacks and defenses." in *Proceedings of the Conference on Computer and Communications Security*. New York, NY, USA: USENIX, 2020, pp. 943–959.

[5] CANELLA, CLAUDIO AND GENKIN, DANIEL AND GINER, LUKAS AND GRUSS, DANIEL AND LIPP, MORITZ AND MINKIN, MARINA AND MOGHIMI, DANIEL AND PIESSENS, FRANK AND SCHWARZ, MICHAEL AND SUNAR, BERK AND VAN BULCK, JO AND YAROM, YUVAL, "Fallout: Leaking data on meltdown-resistant cpus," in *Proceedings of the Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2019, p. 769–784.

[6] VAN BULCK, JO AND MINKIN, MARINA AND WEISSE, OFIR AND GENKIN, DANIEL AND KASIKCI, BARIS AND PIESSENS, FRANK AND SILBERSTEIN, MARK AND WENISCH, THOMAS F AND YAROM, YUVAL AND STRACKX, RAOUL , "Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution," in *Security Symposium Security*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 991–1008.

[7] CHEN, GUOXING AND CHEN, SANCHUAN AND XIAO, YUAN AND ZHANG, YINQIAN AND LIN, ZHIQIANG AND LAI, TEN H, "Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution," in *European Symposium on Security and Privacy*, IEEE. New York: IEEE, 2019, pp. 142–157.

[8] KORUYEH, ESMAEIL MOHAMMADIAN AND KHASAWNEH, KHALED N AND SONG, CHENGYU AND ABU-GHAZALEH, NAEL , "Spectre returns! speculation attacks using the return stack buffer," in *Workshop on Offensive Technologies*. New York: IEEE, 2018.

[9] CANELLA, C., SCHWARZ, M. HAUBENWALLNER, M. SCWARZL, M. GRUSS, D., "Kaslr: Break it, fix it, repeat," *Proceedings of the Asia Conference on Computer and Communications Security*, 2020.

[10] LAPID and WOOL, "Cache-attacks on the arm trustzone implementations of aes-256 and aes-256-gcm via gpu-based analysis," *International Association for Cryptologic Research*, vol. 2018, p. 621, 2018.

[11] COSTAN and DEVADAS, "Intel sgx explained," *International Association for Cryptologic Research*, p. 86, 2016.

[12] SCHWARZ, M e MICHAEL, LIPP, M e MORITZ e MOGHIMI, DANIEL, BULCK, V e JO, STECKLINA, J, PRESCHER, T e GRUSS,D, "Zombieload: Cross-privilege-boundary data sampling." *In Proceedings Conference on Computer and Communications Security*, pp. 1–19, 06 2019.

[13] ZHANG, Y., "Cache side channels: State of the art and research opportunities," in *Proceedings of the Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2017, p. 2617–2619.

[14] W. HU, "Lattice scheduling and covert channels," in *Proceedings of the Symposium on Security and Privacy*. USA: IEEE Computer Society, 1992, p. 52.

[15] BAZM, M. SAUTEREAU, T. LACOSTE, SUDHOLT, MENAUD, "Cache-Based Side-Channel Attacks Detection through Intel Cache Monitoring Technology and Hardware Performance Counters," in *In: Proceedings of the International Conference on Fog and Mobile Edge Computing*. Barcelona, Spain: IEEE, 2018, pp. 1–6.

[16] ACIICCMEZ, BRUMLEY, GRABHER, "New results on instruction cache attacks," in *Proceedings of the International Conference on Cryptographic Hardware and Embedded Systems*. Berlin, Heidelberg: Springer-Verlag, 2010, p. 110–124.

[17] LIPP, M. GRUSS, D. SPREITZER, D. MAURICE, S. MANGARD, "Armageddon: Cache attacks on mobile devices," in *Proceedings of the USENIX Conference on Security Symposium*. USA: Proceedings of the USENIX Conference on Security Symposium, 2016, p. 549–564.

[18] PERCIVAL, C, "Cache missing for fun and profit," *In: PROCEEDINGS OF THE BSD Linux conference, Washington. Anais... Washington: ACM, 2005*, 2005.

[19] OSVIK, D. SHAMIR, A. TROMER, E., "Cache attacks and countermeasures: The case of aes." in *In: Proceedings of the Pointcheval Topics in Cryptology*. San Jose, CA: Spring, 01 2006, pp. 1–20.

[20] MUSHTAQ, M. AKRAM, A. BHATTI, M. K. RAIS, R. LAPOTRE, V. GOGNIAT, "Run-time detection of prime + probe side-channel attack on aes encryption algorithm," *In: Proceedings of the Global Information Infrastructure and Networking Symposium*, pp. 1–5, 2018.

[21] OREN,Y. KEMERLIS, V. SETHUMADHAVAN, S. KEROMYTIS, ANGELOS D., "The spy in the sandbox: Practical cache attacks in java and their implications," in *Proceedings of the Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2015, p. 1406–1418.

[22] GULMEZOGLU, INCI, IRAZOKI, EISENBARTH, SUNAR, "Cross-vm cache attacks on aes," *Transactions on Multi-Scale Computing Systems*, vol. 2, pp. 1–1, 04 2016.

[23] Y. YAROM and K. FALKNER, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," *Cryptology ePrint Archive*, vol. 2013, p. 448, 2013.

[24] LIU, F. YAROM, Y. GE, Q. HEISER, G. LEE, R., "Last-level cache side-channel attacks are practical," in *Proceedings of the Symposium on Security and Privacy*. USA: IEEE Computer Society, 2015, p. 605–622.

[25] YAROM and BENGER, "Recovering openssl ecdsa nonces using the flush+reload cache side-channel attack," *International Association for Cryptologic Research*, vol. 2014, p. 140, 2014.

[26] ZHOU, P. WANG, T. LI, G. ZHANG, ZHAO, "Analysis on the parameter selection method for flush+reload based cache timing attack on rsa," *China Communications*, vol. 12, no. 6, pp. 33–45, 2015.

[27] PANDA, "Fooling the sense of cross-core last-level cache eviction based attacker by prefetching common sense," *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 138–150, 2019.

[28] GRUSS, D. SPREITZER, R.MANGARD, S., "Cache template attacks: Automating attacks on inclusive last-level caches," p. 897–912, 2015.

[29] GOLDER, A. DAS, D. DANIAL, D. GHOSH, S. SEN, S. RAYCHOWDHURY, "Practical approaches toward deep-learning-based cross-device power side-channel attack," *In: Proceedings of the Transactions on Very Large Scale Integration Systems*, vol. 27, no. 12, pp. 2720–2733, 2019.

[30] DISSELKOEN, C. KOHLBRENNER, D. PORTER, L. TULLSEN, D., "Prime+abort: A timer-free high-precision l3 cache attack using intel tsx," in *Proceedings of the USENIX Conference on Security Symposium*. USA: USENIX Association, 2017, p. 51–67.

[31] GRUSS, LETTNER, SCHUSTER, OHRIMENKO, HALLER, COSTA, "Strong and efficient cache side-channel protection using hardware transactional memory," in *Proceedings of the USENIX Conference on Security Symposium*. USA: USENIX Association, 2017, p. 217–233.

[32] BOGDANOV, EISENBARTH, PAAR, WIENECKE, "Differential cache-collision timing attacks on aes with applications to embedded cpus," in *Proceedings of the International Conference on Topics in Cryptology*. Berlin, Heidelberg: Springer-Verlag, 2010, p. 235–251.

[33] FOG, A, "The microarchitecture of intel, amd and via cpus: An optimization guide for assembly programmers and compiler maker," *Copenhagen University College of Engineering*, pp. 190–193, 2012.

[34] ESFAHANI, SOLEIMANY, AREF, "Practical implementation of a new flush+ reload side channel attack on aes," 2020.

[35] SMITH, J. PLESZKUN, REW, "Implementing precise interrupts in pipelined processors," *In: PROCEEDINGS OF THE IEEE SYMPOSIUM ON SECURITY AND PRIVACY*, vol. 37, no. 5, pp. 562–573, 1988.

[36] SILBERT, O. PORRAS, LINDELL, R., "The intel 80/spl times/86 processor architecture: pitfalls for secure systems," *In: PROCEEDINGS OF THE IEEE SYMPOSIUM ON SECURITY AND PRIVACY*, no. 11, pp. 11–22, 1995.

[37] VAN BULCK, JO AND MINKIN, MARINA AND WEISSE, OFIR AND GENKIN, DANIEL AND KASIKCI, BARIS AND PIESSENS, FRANK AND SILBERSTEIN, MARK AND WENISCH, THOMAS F AND YAROM, YUVAL AND STRACKX, RAOUL, "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," in *Symposium on Security and Privacy*. New York, NY, USA: Association for Computing Machinery, 2020.

[38] KOCHER, K. PAUL, C., "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Proceedings of the Annual International Cryptology Conference on Advances in Cryptology*. Berlin, Heidelberg: Springer-Verlag, 1996, p. 104–113.

[39] KIRIANSKY, VLADIMIR AND WALDSPURGER, CARL, "Speculative buffer overflows: Attacks and defenses," *European Symposium on Security and Privacy*, 2018.

[40] EVTYUSHKIN, DMITRY AND RILEY, RYAN AND ABU-GHAZALEH, NAEL CSE AND ECE AND PONOMAREV, DMITRY, "Branchscope: A new side-channel attack on directional branch predictor," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 693–707, 2018.

[41] GRUSS, D. HANSEN, D. GREGG, B, "Kernel isolation: From an academic idea to an efficient patch for every computer," *Proceedings of the Conference on Security Symposium*, vol. 43, 2018.

[42] A. BILAL, "Real time detection of spectre and meltdown attacks using machine learning." *Proceedings of the Conference on Security Symposium*, 2020.