MELTWEB: A Transient Execution Attack to Capture Data in Fill Line Buffer

L. H. C. M. Marques and A. F. da Silva

***Abstract*—The out-of-order execution mechanism is widely used in modern processors. However, meltdown attacks exploit this mechanism to allow malicious instructions to capture sensitive data from kernel domains. This work aims to present a new meltdown attack, called Meltweb. It is a new approach in the category of privileged speculative execution attacks to leak arbitrary data into address spaces and privilege limits. Meltweb can be implemented from linear execution without the need for page faults, eliminating the need for an exception suppression mechanism, thus allowing the attack to be executed throughout the system of arbitrary code without privileges as in search engines interpretation of Javascript. To exemplify the performance of the attack, two proof-of-concept variants were developed that demonstrate the ability to perform the attack natively, as well as remotely using Javascript. The attack developed showed satisfactory results in its two variants, with 98% accuracy in capturing confidential data in the native variant. For the remote variant, the attack proved to be capable of capturing navigation data as a URL recovery with an error rate of 2.3%.**

***Index Terms*—Meltweb, Meltdown, Transient Execution Attacks.**

I. Introdução

Ao longo dos anos, as industrias vêm investindo em diversas pesquisas a fim de desenvolver processadores cada vez mais rápidos e sofisticados. Entretanto, na busca por desempenho, alguns efeitos colaterais não foram levados em conta. Ataques de execução transitória (AET) surgem como um modo eficiente de explorar vulnerabilidades presentes em sistemas de otimização de processadores, tais como *pipeline*, execução fora de ordem e execução especulativa.

Ataques *meltdown* visam explorar a execução fora de ordem, visto que em processadores vulneráveis os resultados de carregamentos não autorizados ainda são encaminhados para operações transitórias antes do lançamento de uma exceção. Isto permite a exposição de dados de diferentes *buffers* da microarquitetura, como, cache L1 [1] [2], *buffer* de arquivos de registro [3] [4] e *buffer* de preenchimento de linhas [5] [6]. Além de ataques como [7] que visam explorar o encaminhamento incorreto de instruções para injeção de dados em *buffers* da microarquitetura. Em contrapartida aos ataques *spectre*, ataques *meltdown* podem burlar os domínios de segurança e acessar dados em outros domínios arquitetônicos da memória do *kernel*.

Luiz Henrique Custódio M. Marques, Universidade Estadual de Maringá, Maringá, Paraná, Brasil e-mail:luhenrique06@hotmail.com

Anderson Faustino da Silva, Universidade Estadual de Maringá, Maringá, Paraná, Brasil e-mail:anderson@din.uem.br

Manuscript received April 19, 2025; revised August 26, 2025.

Com a divulgação das vulnerabilidades, a indústria e academia desenvolveram diferentes trabalhos com o intuito de mitigar, [8] [9], e detectar tais ataques [10] [11]. Entretanto como alerta Canella e outros [12] tais correções não são suficientes para mitigar por completo as falhas.

Neste contexto, este trabalho constrói um novo ataque da classe *meltdown*, Meltweb, capaz de obter informações arbitrárias por meio da captura de dados em transição. Para isto é apresentado um ataque que tem como alvo o *buffer* de preenchimento de linhas, capaz de escutar os dados que são transmitidos pela unidade central de processamento (UCP). Para este fim são construídos *exploits* que permitem vazar dados por meio dos limites de segurança como a *sandbox* do Javascript, processos de *kernel* e SGX (*Software Security Guard Extensions*). Em comparação ao estado da arte, Meltweb apresenta resultados semelhantes na captura de dados confidenciais em sua variante nativa com baixa taxa de falso positivos, além de apresentar resultados superiores na velocidade da captura de dados de navegação web na variante remota, sendo capaz de capturar a URL acessada pela vítima com 100% de acurácia com apenas 4 segundos de permanência na página.

A contribuição deste trabalho está no desenvolvimento de um novo ataque de execução transitória baseado na falha *meltdown*, por meio da exploração de endereços dos *buffers* de preenchimento de linhas (BPLs), sendo possível explorar tanto de forma nativa como também remotamente por meio de um ataque em navegadores. Para isso, é descrito todo o processo de desenvolvimento do ataque mostrando que o Meltweb origina-se das micro-otimizações que fazem com que a UCP atenda cargas especulativas com dados internos. Além disto, são descritos os desafios para explorar, de forma prática, a vulnerabilidade de forma remota, a fim de vazar dados da *sandbox* dos navegadores.

Especificamente o trabalho traz as contribuições a seguir.

1) Apresentação de um novo ataque de execução especulativa para UCPs Intel, capaz de permitir que atacantes, sem privilégios, realizem vazamentos de memórias por meio dos limites de segurança.
2) Demonstração da possibilidade de executar o ataque Meltweb em Javascript, e assim capturar dados de navegação.
3) Exemplificação de possíveis aplicações do ataque em cenários reais.

II. Tabalhos Relacionados

Lipp e outros [1] descrevem o *meltdown* que permite que um invasor sem privilégios possa acessar mapeamentos de

espaços de endereços privilegiados, que normalmente seriam inacessíveis, pelo bit do supervisor nas estruturas de dados de tradução. Isso se dá pois embora qualquer acesso a um endereço privilegiado acione uma condição de falha, antes de manipulá-la adequadamente, a UCP já expõe os dados à execução fora de ordem o que permite o vazamento dos dados. Atualmente a variante original do *meltdown* é mitigada em todos os processadores com as últimas atualizações de software lançadas pela Intel. Na mesma linha, o trabalho apresentado por Schwarz e outros [5] 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*. Este é o primeiro trabalho, 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 *ZombieLoad* como outros ataques contra o MDS (*Microarchitectural Data Sampling*) [3] [4], é um ataque baseado no método de análise de dados a partir de estruturas microestruturais. Entretanto, 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.

O trabalho de Canella e outros [13] 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, até então, os tipos de ataques *meltdown* conhecidos. Entretanto, após a análise dos principais métodos 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. 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.

III. MELTWEB

Considerando os atuais mecanismos de detecção e mitigação de ataques do tipo *meltdown*, o ataque Meltweb visa ser um ataque eficiente que pode ser executado de forma nativa ou remota.

Para o desenvolvimento do Meltweb, é considerado a premissa de um invasor que deseja abusar da vulnerabilidade *meltdown* para divulgar informações confidenciais, tendo como alvo um sistema baseado na arquitetura Intel executando os microcódigos mais recentes. O objetivo é que o invasor realize o ataque sem a necessidade de executar o *exploit* com privilégios no sistema alvo, sendo possível, desta forma, vazar informações entre os níveis de privilégio e espaços de endereços.

O presente trabalho apresenta duas variantes do ataque desenvolvido, uma variante nativa (Meltweb) e outra remota (Meltweb.js). O ataque consiste em induzir a UCP a realizar o carregamento ou armazenamento em algum *buffer* interno, no caso o BPL, então executar uma carga induzindo a UCP a usar especulativamente dados em transição dos BPLs, sem verificar as restrições de endereçamento, e por fim capturar os dados acessados especulativamente na memória cache por meio do um ataque evict+reload descrito por Gruss e outros [14].

A. Desenvolvimento

Para a realização do ataque Meltweb no primeiro momento, a vítima carrega um dado para um endereço específico, os BPLs, considerando que para aumentar a otimização a UCP evita carregar diretamente o dado para o cache L1D e força os carregamentos a sempre passarem pelos BPLs. Após o carregamento, é executada a instrução de falha de alinhamento consequentemente gerando uma exceção que permite o vazamento indevido dos dados por meio dos BPLs.

Durante o desenvolvimento do Meltweb, dois principais desafios foram encontrados.

1) O primeiro desafio encontrado foi obter os dados privilegiados, com usuários sem privilégios para vazar para o BPL.
2) Outro desafio é garantir que o dado que está sendo vazado é o dado alvo da vítima.

Para superar o primeiro desafio é utilizada a estratégia demonstrada por Lipp e outros [15]. Esta realiza interações com processos privilegiados de *kernel* mesmo sem privilégios, sendo assim o invasor realiza repetidamente o acesso a um arquivo privilegiado, como por exemplo o /etc/shadow no Linux. Com isso é possível induzir a vítima a ler o arquivo mesmo sem permissão e vazar o seu conteúdo. Para garantir a sincronização de dados com a vítima, ou seja, realizar o vazamento do dado alvo, é utilizada uma estratégia de despejo no qual para poder controlar os valores é realizada a retirada das entradas de cache do conjunto monitorado, no caso cache L1D. O Algoritmo 1 demonstra um exemplo de vazamento de dados em transição com Meltweb.

Algorithm 1: Ataque Meltweb

Input: $bufferAlvo$
Function meltweb():
  $bufferEvict \longleftarrow bufferAlvo * 1024$;
  $evict(bufferEvict)$;
  $segredo \longleftarrow *(newPage)$;
  $*ptrEntry \longleftarrow bufferEvict + (1024 * segredo)$;
  $**(ptrEntry)$;
  $evictReload(bufferEvict)$;
End Function

Inicialmente é liberado o *buffer* utilizado pelo canal lateral, para vazar os dados confidenciais obtidos especulativamente. Para isso, a UCP carrega especulativamente um valor da memória com o intuito de que seja de uma página recém-alocada, entretanto na verdade esses dados são dados em

transição dos BPLs pertencentes a um domínio de segurança arbitrariamente diferente. Quando a UCP identificar que ocorreu uma carga especulativa incorreta, todas as modificações realizadas nos registros ou na memória serão descartados e então reiniciará a execução de carregamento com o *buffer* correto. No entanto, uma vez que resquícios da carga especulativa existem no nível da microarquitetura como nas linhas de cache, é possível observar os dados vazados usando um ataque de canal lateral em cache (ACLC) como evict+reload. Assim, realizando o ACLC são acessadas todas as entradas do *buffer* para verificar se algum dos acessos é significativamente mais rápido, indicando um cache hit, identificando que a linha contém as informações vazadas. Especificamente é esperado que dois acessos sejam rápidos, visto que não apenas o correspondente as informações vazadas estarão em cache, como também os dados do *buffer* com o valor correto, e após a UCP reiniciar a execução o programa também acessa o *buffer* correto.

O algoritmo descrito é construído a partir do mecanismo de paginação por demanda para o endereço de memória carregado, fazendo com que a UCP reinicie a execução somente após manipular o evento de entrada de página e trazer uma nova recém-mapeada. Cabe ressaltar que essa estratégia não é a condição de erro, mas sim uma parte padrão do funcionamento dos sistemas operacionais (SOs). Outra estratégia que poderia ser utilizada para vazar dados em transição seria ao invés de utilizar uma página recentemente mapeada, desreferenciar um ponteiro NULL e, posteriormente, suprimir o erro. Isto demonstra que em teoria qualquer exceção de tempo de execução é o suficiente para forçar o vazamento de dados de forma especulativa.

Apesar da suposição de que os dados vazados pelo Meltweb são carregados a partir da BPL, a prova deste conceito não é trivial. Para realizar a verificação se os dados são providos dos BPLs ou de outro *buffer* da microarquitetura são realizados alguns testes a fim de medir o número de hits no BPLs. Para isso, inicialmente foi carregado um valor "segredo" conhecido em um endereço fixo, em seguida foi executado um *loop* de leitura onde a cada leitura é seguida de uma instrução do microcódigo lfence que é responsável por serializar as operações de armazenamento, então foram realizadas as operações de leitura do invasor a fim de observar se haveria algum sinal para memória WB e WT. Com isto foi possível observar que os dados vazados não foram capturados inicialmente da cache.

A variante remota Meltweb.js presume que a vítima acesse um navegador esteja nativamente ou em uma máquina virtual, onde o invasor reside em um processo vizinho ou em algum outro local da internet de forma remota. Para isso temos que o invasor foi capaz de introduzir o código do ataque de forma maliciosa em um site benigno vulnerável com injeção de código por meio de ataques como XSS, RFI ou caso o invasor comprometa o site de alguma outra forma. Outra possibilidade é que a vítima esteja navegando em um site controlado pelo invasor, portanto, em todos os casos, o código malicioso é executado no navegador da vítima.

A Figura 1 apresenta o plano de execução do Meltweb.js.

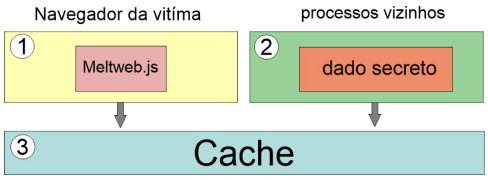

Fig. 1. Meltweb.js.

O ataque considera inicialmente que a vítima por meio do navegador acesse de forma direta a aplicação que contém a página maliciosa. Durante o acesso ocorrerá a execução especulativa no processador, permitindo a captura dos dados. A captura e monitoramento dos dados são realizados durante toda a permanência da vítima na página maliciosa.

Para o desenvolvimento desta variante é utilizado a linguagem de programação Javascript. A escolha se dá pelo fato de ser a principal utilizada no desenvolvimento de aplicações web na atualidade. Sendo que todos os navegadores modernos possuem um interpretador de Javascript, é possível demonstrar que tal abordagem em um ambiente real poderia atingir milhares de máquinas, desde que tais alvos tenham um navegador e navegue pela página maliciosa.

O ataque é projetado a partir da *sandbox* do Javascript utilizando o mecanismo *Spidermonkey*[1]. Os atuais navegadores de internet possuem webassembly como um recurso padrão, o que permite gerar com mais facilidade códigos que atendam aos requisitos para a reprodução do Meltweb.js. O Javascript possui algumas limitações que dificultam a implementação dos ataques de execução transitória, sendo assim para a realização do desenvolvimento de um ataque funcional é necessário superar alguns desafios.

1) A primeira limitação é o fato que atualmente o webassembly suporta apenas índices de 32 bits.
2) Outro limitante é a falta de instruções de limpeza do cache como clflush, assim impedindo a utilização de abordagens de ACLC como flush+reload.
3) Para evitar ACLC os interpretadores presentes nos navegadores diminuem a precisão dos temporizadores, tornando assim mais difícil a medição de tempo na realização das ações.

Para contornar tais desafios, durante o desenvolvimento do Meltweb.js, é utilizado a mesma metodologia aplicada por [13], a qual utiliza uma execução especulativa como expressão de falha. Assim possibilitando a implementação da prova de conceito[2]. A primeira fase do desenvolvimento é composta pelo desafio de implementar uma técnica capaz de suprir a falta da função clflush. Para tal, é implementada uma variação da técnica descrita por [14] na implementação de um ataque evict+reload, pois a mesma não necessita de instruções de limpeza do cache. Conforme demonstrado por Gruss e outros [14], a falta da instrução clflush não impede a implementação de um ataque eficiente em cache, sendo possível adotar uma estratégia de evict que consiste em preencher todo o espaço da cache com dados aleatórios, de endereços congruentes, tendo o efeito semelhante ao da

[1]Spidermonkey versão 45, presente no Firefox
[2]Esta prova compreende o teste de para vazar 00B

limpeza de cache, uma vez que os dados despejados na cache não são utilizados por nenhum processo da vítima. O maior desafio de implementar uma abordagem de `evict` é a influência causada pela política de substituição de cache utilizada pela `UCP`, visto que a mesma pode influenciar o tamanho do conjunto de despejo e o padrão de acesso necessário para construir uma abordagem eficiente, assim uma estratégia que é eficaz para uma política provavelmente será ineficaz para outra. Para a implementação do `Meltweb` foi utilizado uma estratégia capaz de encontrar a melhor remoção de cache de forma totalmente automatizada. O Algoritmo 2 demonstra a implementação de um conjunto de despejo.

Algorithm 2: Estratégia de Despejo

```
Function evict():
    EvictBuffer ⟵ 8192 * 1024;
    offset ⟵ 64;
    rodadas ⟵ 1000;
    for i ← 0; to rodadas do
        for j ← 0; to (evictBuffer/offset) do
            current ⟵ j * offset;
End Function
```

Ao executar diversas vezes o Algoritmo 2, é criado gradualmente conjuntos de despejo capazes de cobrir a maior parte do cache. Este algoritmo só não é capaz de preencher toda a cache, pelo fato do mesmo não substituir as linhas que são acessadas pelo mecanismo de tempo de execução do Javascript. Como o Javascript não permite a utilização de ponteiros não é possível identificar quais conjuntos de cache correspondem aos de despejo específico. Além disso, executar o algoritmo diversas vezes no mesmo sistema resultará em um mapeamento diferente para cada execução. Para encontrar o conjunto de despejo, Hund e outros [16] sugere acessar um *buffer* de despejo de memória física de 8MB para assim conseguir invalidar todos os conjuntos de cache. Entretanto, essa abordagem não é possível ser realizada em modo de usuário. Para suprir esse desafio, a abordagem é alocar uma matriz de bytes de 8MB de memória virtual por meio do Javascript. Experimentos indicam a necessidade de realizar 1000 interações de subconjuntos cada vez com um endereço diferente, permitindo o invasor a identificar os endereços que residem em cada conjunto de cache e assim criar a estrutura de dados do conjunto de despejo.

Outro desafio é a medição de tempo, visto que os temporizadores de alta resolução integrados nos navegadores foram desativados como parte das mitigações contra `ACLC`. Entretanto a literatura apresenta técnicas que visam suprir a falta de tais temporizadores. Canella e outros [13] para contornar a falta da instrução `rdtscp`, para a medição de alta resolução temporal, utilizam o recurso `web workers`[3] que permite a criação de um *thread* separado que diminui repetidamente um valor em um local de memória compartilhada, permitindo assim produzir um cronômetro que fornece a resolução suficiente para o desenvolvimento do ataque. Frigo e outros [17] apresentam a possibilidade de gerar um contador de tempo baseado em `GPUs`, sendo possível executar até mesmo em dispositivos móveis. Entretanto, a estratégia utiliza em `Meltweb.js` é reabilitar o temporizador do `spidermonkey performance.new()` para retornar a saída de `rdtscp`, o Algoritmo 3 apresenta a medição do tempo de acesso aos dados em cache.

[3]Recurso disponível no `HTML5`.

Algorithm 3: Captura do tempo cache hit/miss

```
Input: bufferEvict
Function EvictReload():
    for i ← 0; to 256 do
        tempoInicial ⟵ ciclos();
        *(bufferEvict + 1024 * i);
        tempoFinal ⟵ ciclos() − tempoInicial;
        if tempoFinal < tempoInicial then
            ++resultado[i]
End Function
```

Para descobrir quais linhas foram utilizadas para armazenar o dado secreto capturado durante a execução transitória é realizada a medição de tempo para cada uma das entradas do `bufferEvict` e comparado o tempo inicial com o tempo final, assim as linhas que apresentarem o menor tempo de resposta indicam ser a linha na qual está armazenado os dados vazados. O *exploit* do canal lateral `evict+reload` é escrito completamente em Javascript, enquanto a instrução de falha para a execução transiente é escrita diretamente em `webassembly` permitindo executar a mesma diretamente mesmo fora da *sandbox* do Javascript.

Para o ataque proposto ser capaz de monitorar os dados de navegação web e identificar quais `URLs` estão sendo utilizadas pela vítima por meio de caracteres vazados nos *buffers* arbitrários, é adotada uma estratégia de implementação de *wordlists*. [5] sugere uma abordagem mais indireta na qual no domínio transitório é detectado uma substring "www." dentro dos dados vazados, e após encontrar o valor correspondente é vazado os caracteres após o "www." para o domínio arquitetônico por meio da cache. O `Meltweb` utiliza de uma *wordlist*, a fim de aumentar a velocidade em identificar a `URL` acessada pela vítima. A técnica consiste em filtrar os caracteres no domínio transitório vazando os mesmos para a cache e então comparar os caracteres vazados com os pertencentes à *wordlist*. Este processo realiza comparações até que se encontre a combinação completa com algum item da lista. Nesta abordagem não é necessário vazar toda a `URL`, dado que o algoritmo é capaz de identificar qual o site correspondente a partir de apenas alguns caracteres.

IV. Resultados

Para demonstrar a eficiência do ataque proposto, além de mensurar o nível de vazamento de informações que podem ser obtidos em cada processador, os experimentos são realizados em diferentes máquinas. Os ambientes de testes utilizados são os descritos a seguir.

1) Processador Intel *Core* I7 - 7700 com frequência 3.6GHz e arquitetura x86-64 e 32 GB de memória `RAM`.

2) Processador Intel *Core* I5 - 9400f com frequência 4.1GHz e arquitetura x86 64 e 12 GB de memória RAM.
3) Processador Intel *Core* I3 3110M, Frequência 2.4GHz arquitetura x86-64 e 8GB de memória RAM.
4) Máquina em nuvem Azure Cloud: Processador Intel Xeon Platinum 8171M com frequência 2.60GHz e arquitetura x86 64 e 16 GB de memória RAM.

A avaliação consiste em executar as variantes dos ataques para carregar um valor conhecido A em um endereço de memória fixo específico. O carregamento e execução dos ataques foram realizados em 2 blocos de rodadas, o primeiro de 1000 rodadas e um segundo de 10000 rodadas, sendo que para cada rodada é capturada a precisão na recuperação do dado e o tempo de execução. Cada experimento é executado 10 vezes, registrando o número de vezes que foram vazados os dados corretos (A) para cada uma das variantes de ataques. Após a captura, são calculadas as proporções de positivos e falsos positivos para cada ataque.

Para a variante remota, os testes medem a capacidade do ataque em capturar dados de navegação web, mas especificamente capturar a URL que está sendo acessada pelo alvo. O teste realizado consiste em acessar a mesma URL 1000 vezes sequencialmente para cada site, enquanto o ataque é executado em uma aba vizinha do navegador. Desta forma, é possível observar quantos acessos são necessários para capturar a URL alvo, além de ser possível medir o número de falsos positivos. Outra questão avaliada é a capacidade do ataque em capturar a URL com apenas 1 acesso, visto que sites modernos realizam centenas de requisições durante a permanência no mesmo. Para isso são executados *exploits* e então realizado um único acesso a um site específico e mantido a permanência no mesmo durante 2 intervalos de tempo. No primeiro cada ataque é executado durante 1 minuto, enquanto no segundo durante 30s. Assim como no outro cenário, cada experimento é repetido 10 vezes.

A Tabela I apresenta os trabalhos aos quais o Meltweb foi submetido para comparação, a seleção dos ataques visam comparar o ataque proposto com o atual estado da arte em ataques de execução transitória comparando com as versões nativas e remotas, os trabalhos escolhidos para a análise mostram-se relevantes dado que eles são capazes de burlar os atuais mecanismos de prevenção e detecção de ataques *meltdown* aplicados nos processadores e SOs modernos.

TABLE I
ATAQUES DE COMPARAÇÃO

Ataques	Abordagem	Autor	Ano
LVI	Nativo	Bulk e outros [7]	2020
Echoload	Nativo/Remoto	Canella e outros [13]	2020
Zombieload	Nativo	Schwarz e outros [5]	2019
Foreshadow	Nativo	Bulk e outros [6]	2018
Splitspectre	Remoto	Mambretti e outros [18]	2018

A Tabela II apresenta os ambientes utilizados para avaliar a capacidade do Meltweb, em capturar dados sem ser detectado pelos mecanismos de segurança. As duas versões do ataque são avaliadas em sistemas Linux x86-64 e x86, assim demonstrando a sua efetividade em diferentes arquiteturas. A marcação OK indica que o ataque é efetivo no ambiente, sendo capaz de capturar dados sem a interferência de qualquer mecanismo de prevenção.

TABLE II
EFETIVIDADE DO MELTWEB

UCP	Arquitetura	S.O	Meltweb	Meltweb.js
I3-3110M	Ivy Bridge	Linux x86-64	OK	-
I3-3110M	Ivy Bridge	Linux x86	OK	OK
I5-9400F	Coffee Lake	Linux x86-64	OK	-
I5-9400F	Coffee Lake	Linux x86	OK	OK
I7-7700	Kaby Lake	Linux x86-64	OK	-
I7-7700	Kaby Lake	Linux x86	OK	OK
Xeon 8171m	Sky Lake	Linux x86-64	-	-

A variante Meltweb é eficaz na maioria dos sistemas testados, sendo capaz de burlar todas as mitigações de software presente nos sistemas operacionais alvo. Apenas no Xeon, o Meltweb não é capaz de capturar os dados. Isso se deve pelas atualizações mais recentes da Intel para processadores de servidores. Como esperado a variante, Meltweb.js só é eficaz em sistemas com arquitetura de 32 bits. Isso se dá pela limitação atual do Javascript, que não permite a indexação de vetores de 64 bits. Segundo a Mozilla [19], a próxima versão do Javascript será capaz de realizar a indexação em 64 bits, sendo possível atualizar o ataque para sistemas atuais.

A Figura 2 apresenta a relação entre o tempo de execução do ataque e a precisão na captura do dado em 1000 execuções, indicando que o tempo gasto na execução está intimamente ligado com a acurácia da captura dos dados. Execuções muito rápidas, como por exemplo tempos menores que 0.3ms, representam que o ataque não obteve resultado em capturar os dados no estado transitório.

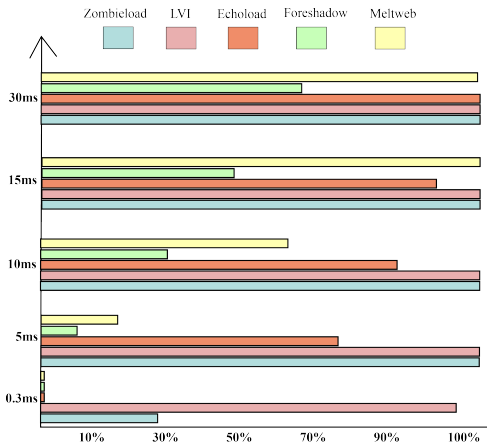

Fig. 2. Desempenho em 1000 rodadas de execução.

É possível observar que o Meltweb leva em média 10ms na capturara dos dados, com 70% de acurácia, e 15ms, com 98% de acurácia, sem a necessidade de qualquer pré requisito para a sua execução. Apesar dos bons resultados, em nenhuma rodada o Meltweb foi capaz de alcançar 100% de acurácia. O ataque LVI apresenta o melhor resultado, entregando uma acurácia de 98% com uma média de tempo de execução de apenas 3ms e 100% com 5ms. O ataque foreshadow possui o pior desempenho, com um tempo médio de 30 ms e uma taxa de 86% de acurácia.

A Figura 3 apresenta as médias das rodadas de 10000 execuções. É possível observar que não existe alterações consideráveis no tempo de execução do ataque, porém nenhum ataque é capaz de demonstrar 100% de acurácia mesmo aumentando a quantidade de iterações. O `Meltweb` possui um tempo de execução superior ao dos ataques `zombieload` e `LVI`. Isso se deve pela abordagem escolhida para a recuperação dos dados em cache, uma vez que o `Meltweb` utiliza a abordagem `evict+reload`, a qual demonstrou ser mais lenta em relação às abordagens que utilizam `flush+reload`.

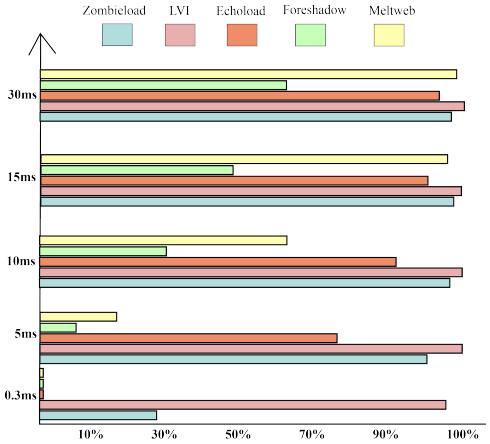

Fig. 3. Desempenho em 10000 rodadas de execução.

A Tabela III apresenta o comparativo de desempenho do ataque `Meltweb.js` para capturar dados de navegação de usuários.

TABLE III
RESULTADOS DE TEMPO E TAXA DE ERRO

Ataque	Tempo	Taxa de erro
Meltweb.js	38.4 kbits/s	2.3%
Echoload.js	215 kbit/s	0.09%
Splitspectre	362 kbit/s	4%
Zombieload	28.8 kbit/s	0%

Vale ressaltar que o ataque `Zombieload` apesar de apresentar uma variante capaz de capturar dados de navegação é executado de forma nativa.

O ataque `Meltweb.js` apresenta uma taxa de falsos positivos de 2.3%. Isso se deve pelo fato do algoritmo utilizar uma lista de palavras chaves carregada pelo atacante, e a partir desta ser realizada uma comparação dos primeiros caracteres com os sites presentes na lista, a fim de aumentar a velocidade. Entretanto essa abordagem apresentou um ataque não tão consiste comparado as abordagens utilizadas por outros ataques. Por exemplo, o `zombieload` que captura todos os caracteres da `string` do domínio filtrando a partir do "www.". Vale ressaltar que o `Meltweb.js` pode ser estendido para capturar outros dados além de `URL`s, por exemplo para capturar números de cartões de crédito, credenciais digitadas em formulários entre outros.

A Figura 4 apresenta os resultados da capacidade da captura, de `URL`, utilizando apenas um único acesso e mantendo a permanência no site.

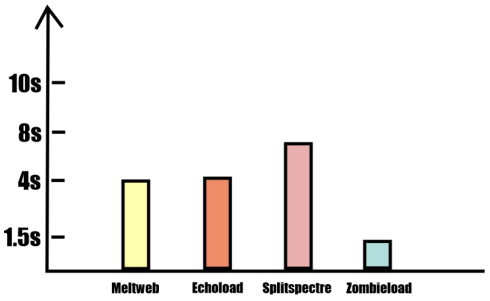

Fig. 4. Velocidade na captura de URLs.

Os resultados indicam que o `Meltweb.js` é um ataque efetivo em um cenário real. Este necessita em média de 4 segundos, de permanência na página alvo, para capturar com 100% de acurácia a `URL` acessada. Como usuários realizam diversas requisições durante a permanência na mesma página, o `Meltweb` é capaz de capturar quais páginas a vítima já tinha aberto antes mesmo de abrir a página maliciosa. Por sua vez, o `Meltweb.js` apresenta resultados semelhantes aos demais ataques da literatura, sendo que apenas o ataque `zombieload` é mais rápido na captura da `URL`. Isso se deve pelo fato do `zombieload` executar de forma nativa, o que permite um ataque visivelmente mais rápido.

Os resultados demonstram a limitação da variante remota em ser efetiva apenas em sistemas operacionais de 32 bits. Dado que atualmente grande parte dos usuários utilizam sistemas 64 bits, o ataque não é efetivo em um cenário real. Além da limitação da arquitetura do sistema operacional, a variante web, é eficaz apenas no Firefox, sendo ineficiente em outros navegadores como Google Chrome. Isso se deve pelo fato do navegador da Google implementar o isolamento de processos para cada página aberta, impedindo assim a quebra do mecanismo de *sandbox*.

V. TRABALHO FUTUROS

Trabalhos futuros podem explorar pontos que não foram abordados durante o desenvolvimento deste trabalho, como:

1) Com a atualização do Javascript, para suporte a índices de 64bits, é possível estender o `Meltweb` para sistemas operacionais 64 bits. Assim, possibilitando o ataque a atingir uma gama maior de cenários.
2) Alterar o método de vazamento de `URL`s utilizada no ataque, para capturar todos os caracteres das `URL`s. Visto que, atualmente o `Meltweb` utiliza *wordlists* para captura de `URL`s.
3) Estender o ataque explorar outros navegadores web.
4) Demonstrar a efetividade do `Meltweb` em arquiteturas `ARM`. Assim, estendendo o ataque para sistemas embarcados e smartphones.

VI. CONCLUSÕES

O presente trabalho apresentou a prova de conceito de uma nova variante de ataques do tipo *meltdown*, visando vazar dados do *buffer* de preenchimento de linhas. O `Meltweb` permite que um invasor vaze valores carregados recentemente pela `UCP`, a partir do espaço de usuário sem a necessidade

de permissões de *kernel*. O ataque proposto é capaz de escapar da *sandbox* do Javascript, por meio de sua variante remota, permitindo capturar dados de navegação. O `Meltweb` é efetivo em sistemas com todas as mitigações de software contra ataques *meltdown* ativadas, sendo possível explorar a vulnerabilidade em processadores de diferentes gerações da Intel, incluindo os de nona geração.

No geral os resultados demonstram grande potencial em ambas as variantes apresentadas. Na captura de dados, em sua variante nativa, é possível capturar dados em transição com uma acurácia de 98% com apenas 15 ms, e em sua versão remota com uma taxa de 38.4 kbits/s com erro de 2.3%. Portanto, o `Meltweb` apresenta resultados semelhantes ao estado da arte em ataques de execução transitório.

REFERENCES

[1] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, *et al.*, "Meltdown: Reading kernel memory from user space," *In: Proceedings of the Usenix Security Symposium*, pp. 973–990, 06 2018.

[2] J. Wampler, I. Martiny, and E. Wustrow, "Exspectre: Hiding malware in speculative execution.," in *Proceedings of the Network and Distributed System Security Symposium*, 2019.

[3] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, *et al.*, "Fallout: Leaking data on meltdown-resistant cpus," in *Proceedings of the Conference on Computer and Communications Security*, (New York, NY, USA), p. 769–784, Association for Computing Machinery, 2019.

[4] J. Stecklina and T. Prescher, "Lazyfp: Leaking fpu register state using microarchitectural side-channels," *arXiv preprint*, 2018.

[5] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling.," *In Proceedings Conference on Computer and Communications Security*, pp. 1–19, 06 2019.

[6] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lippi, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens, "Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution," in *Security Symposium Security*, (New York, NY, USA), pp. 991–1008, Association for Computing Machinery, 2018.

[7] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lippi, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," in *Symposium on Security and Privacy*, (New York, NY, USA), Association for Computing Machinery, 2020.

[8] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," in *Proceedings of the International Symposium on Microarchitecture*, pp. 974–987, IEEE, 2018.

[9] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *Proceedings of the International Symposium on Microarchitecture*, pp. 428–441, IEEE, 2018.

[10] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Safespec: Banishing the spectre of a meltdown with leakage-free speculation," in *Proceedings of the Design Automation Conference*, pp. 1–6, IEEE, 2019.

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

[12] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses.," in *Proceedings of the Conference on Computer and Communications Security*, (New York, NY, USA), pp. 943–959, USENIX, 2020.

[13] C. Canella, M. Schwarz, M. Haubenwallner, M. Schwarzl, and D. Gruss, "Kaslr: Break it, fix it, repeat," *Proceedings of the Asia Conference on Computer and Communications Security*, 2020.

[14] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," p. 897–912, 2015.

[15] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Armageddon: Cache attacks on mobile devices," in *Proceedings of the USENIX Conference on Security Symposium*, (USA), p. 549–564, Proceedings of the USENIX Conference on Security Symposium, 2016.

[16] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *Proceedings Symposium on Security and Privacy*, (New York, NY, USA), pp. 191–205, IEEE, 2013.

[17] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand pwning unit: Accelerating microarchitectural attacks with the gpu," in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 195–210, IEEE, 2018.

[18] A. Mambretti, M. Neugschwandtner, A. Sorniotti, E. Kirda, W. Robertson, and A. Kurmus, "Let's not speculate: Discovering and analyzing speculative execution attacks," *IBM Research*, 2018.

[19] "www.mozilla.org,"

Luiz Henrique Custódio Mendes Marques é mestre em Ciência da Computação pela Universidade Estadual e Maringá, Paraná/Brasil. Atualmente cursa Doutorado em Ciência da Computação pela mesma universidade (UEM). Seus interesses de pesquisa envolvem ataques de execução transitória, ataques de canal lateral e arquitetura de computadores.

Anderson Faustino da Silva é doutor em Engenharia de Sistemas e Computação pela COPPE/Universidade Federal do Rio de Janeiro, Rio de Janeiro/Brasil. Atualmente e professor associado da Universidade Estadual de Maringá, Paraná/Brasil. Seus interesses de pesquisa envolvem compiladores, arquitetura de computadores e programação paralela.