Avaliação do Impacto das Vulnerabilidades Meltdown e Spectre em Processadores Modernos

Ana Cláudia M. P. da Costa, Kristtopher Kayo Coelho
Ricardo dos Santos Ferreira, José Augusto M. Nacif
Universidade Federal de Viçosa, Brasil
{ana.c.paraiso, kristtopher.coelho, ricardo, jnacif}@ufv.br

***Resumo*—Os processadores modernos passaram a adotar novas medidas para aperfeiçoar seu desempenho. A primeira, execução fora de ordem, permite que instruções mais abaixo do fluxo sejam executadas em paralelo com anteriores. Outra saída foi a execução especulativa, que executa instruções supondo que são realmente necessárias posteriormente. Entretanto, recentemente, foi descoberto que ambas permitem acesso a lugares na memória que não deveriam ter alcance. Assim, grandes empresas do ramo, como Intel, AMD, ARM, entre outras, desenvolveram *patchs* para corrigir estas vulnerabilidades geradas. Mesmo amenizando os impactos gerados com as tentativas de acesso aos lugares privados da memória, os *patchs* impactam diretamente no desempenho dos sistemas atuais. Este artigo apresenta um estudo sobre estas vulnerabilidades, conhecidas como Meltdown e Spectre. Verificamos o efeito e eficácia da execução especulativa e, ao obter sucesso, conseguimos calcular o impacto do desempenho computacional com e sem *patch* de segurança. A partir de testes feitos em *benchmarks*, obtemos uma perda de desempenho um pouco acima 10% em sistemas protegidos.**

***Index Terms*—Execução fora de ordem, Execução especulativa, Memória cache, Meltdown, Spectre.**

I. Introdução

Devido à alta demanda para melhores desempenhos, existem numerosos avanços em vários campos da tecnologia. Para atender a esse requisito, as indústrias vêm desenvolvendo processadores cada vez mais sofisticados e melhores em termos de taxa de transferência. Uma saída, são processadores com dois ou mais núcleos, projetados para executar instruções em paralelo. Contudo, o custo em tempo, consumo de energia e eficiência energética das operações com a memória se tornaram significantemente maiores que operações aritméticas, assim, impactando consideravelmente o desempenho [23]. Com o objetivo de melhorar o desempenho, os processadores modernos implementaram dois paradigmas importantes: execução fora de ordem e execução especulativa.

A execução fora de ordem permite que instruções mais abaixo no fluxo de instruções sejam executadas em paralelo com as anteriores [6]. Enquanto, a execução especulativa executa as instruções de forma com que sejam necessárias no futuro. Se realmente for necessário, a execução é aproveitada. Caso contrário, os resultados são descartados [12]. No entanto, ao serem utilizadas por aplicações maliciosas, acessam áreas na memória que, teoricamente, são inacessíveis a qualquer pessoa. Esta descoberta, feita pela *Google Project Zero*, em 2017, a qual apontou que grandes empresas estão vulneráveis a essas falhas de segurança [11]. Deste modo, estas empresas foram forçadas a desenvolver soluções de segurança. Consequentemente, começaram a produzir *patchs* para amenizar as consequências dessas vulnerabilidades e, impactando, significantemente, o desempenho [16].

No artigo, realizamos alguns estudos com o objetivo de analisar o impacto gerado no desempenho computacional. Em um primeiro momento, analisamos um exemplo simples de execução especulativa, com objetivo de verificar a efetividade das soluções implementadas nos *patchs* de segurança. Em seguida, realizamos testes em *benchmarks* comerciais para analisar o desempenho antes e depois da utilização dos *patchs*. Foram feitos alguns testes com as ferramentas GtkPerf e Linux-benck, calculando o tempo total de execução.

Com enfoque educacional, o artigo traz conceitos essenciais para o ensino de falhas de segurança na arquitetura de computadores. Trata-se de uma abordagem simples mas, consistente, com o auxílio de imagens, exemplos, dados quantitativos e comparações, que retratam o contexto atual do tema. Seu grande diferencial, relacionado a outros trabalhos da área, é a avaliação de perda de desempenho real proporcionado pelas estratégias corretivas adotadas por grandes empresas do ramo. Por se tratar de uma descoberta feita recentemente, em meados de 2017, ainda é algo pouco explorado, logo, não há publicações que realizam essa análise de desempenho.

O presente trabalho, divide-se em outros cinco tópicos. A seguir, Seção II, trata de conceitos importantes da arquitetura de memória, como: memória cache, execução fora de ordem e execução especulativa. A Seção III explora as descobertas recentes das falhas de segurança e suas variantes. A Seção IV mostra uma aplicação de uma das variantes, para analisar sua efetividade. Em seguida, a Seção V relata os impactos dessas vulnerabilidades nos sistemas operacionais. Por fim, a Seção VI, apresenta uma conclusão geral do trabalho.

II. Fundamentação Teórica

Esta Seção trata de conceitos gerais do funcionamento da arquitetura de memórias de CPUs modernas, servindo como base para os tópicos seguintes. Inicialmente, apresenta a estrutura das memórias cache de processadores atuais. Posteriormente, são explorados os conceitos de execução fora de ordem e execução especulativa e como são realizadas.

A. *Memória Cache*

CPUs modernas possuem em torno de 3 GHz, cada núcleo e uma latência da memória em torno de 100 ns. Assim, seriam necessários na faixa de 300 ciclos de *clock* para carregar informações da memória RAM [7]. A CPU utiliza uma memória temporária, rápida e pequena, que irá armazenar cópias dos dados localizados na memória RAM. Normalmente, partes do programa precisam ser executadas várias vezes pelo compilador e, para afirmar que essas partes dos programas devem ser mantidas em memória mais rápida para execução futura, utiliza-se o princípio de localidade [1]. Sendo localidade temporal a posição de memória que foi referenciada para processamento uma vez será encaminhada novamente em um futuro próximo. E, localidade espacial, a posição de memória que tenha sido referenciada em locais de memória vizinhos seria referida mais uma vez em curto intervalo de tempo.

A memória cache é classificada em níveis, dependendo da velocidade e tempo de acesso. Ou seja, quanto maior a distância entre ela e a CPU, maior a velocidade e tempo de acesso. Como mostra a Figura 1, os processadores atuais possuem três níveis de cache. Ao realizar uma busca, a procura é feita diretamente ao L1, caso não esteja neste local, pesquisa no L2 e, por último, no L3. Caso, também não esteja no L3, o controlador de memória realiza a busca na RAM.

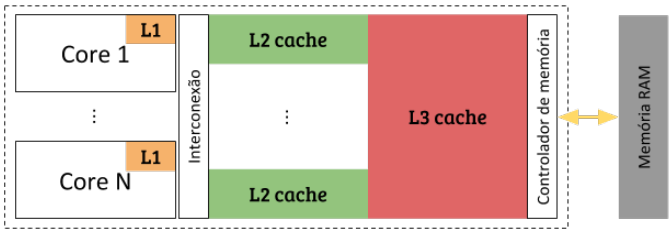

Figura 1. Hierarquia da memória.

Todas as consultas à memória RAM são tratadas por um controlador, representado por uma seta na Figura 1. Sua principal função é fornecer informações e responder para o processador sobre a leitura e gravação de dados. Outro ponto importante para obter um alto desempenho, é preencher a memória com dados que sejam cruciais. Utilizando políticas de substituição, além das de leitura e gravação, aumentará a taxa de acerto (*hit*). Mesmo com essas políticas que auxiliam para uma busca rápida e eficiente, a latência para acesso as instruções e dados da memória continua sendo o grande gargalo em questão de desempenho [10]. Uma maneira de amenizar essa perda, é a execução paralela, possibilitando assim, a execução de mais de um acesso à memória por vez. [26].

Para que o mapeamento dos dados ocorra, os mesmos são carregados da memória principal e armazenados em determinado endereço na cache, como apresentado na Figura 2. Existem três formas desse mapeamento ocorrer. A primeira, os dados são mapeados para apenas um local na cache. Esta é a maneira mais simples de ser feita e, permite-se uma busca rápida às informações. A segunda, a totalmente associativo, cada bloco pode ser colocado em qualquer lugar da cache. Logo, possui alta taxa de *hit*, porém, para encontrá-lo percorre todos os outros. Por último, a associativo por conjunto, cada bloco da memória principal é destinado a um conjunto fixo. Nesta abordagem, a sua associatividade pode ser *N-way*, em que a cache é dividida em conjunto de N linhas [19].

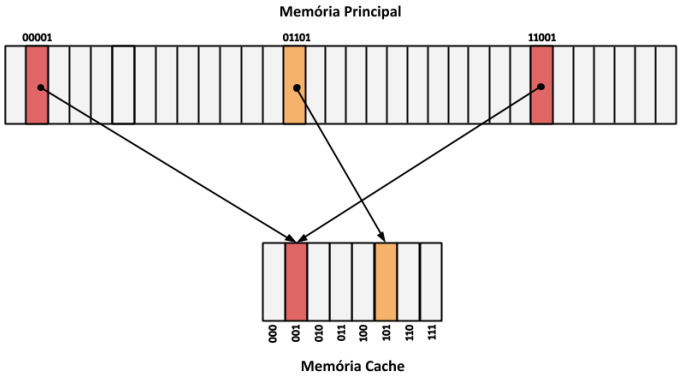

Figura 2. Mapeamento dos dados da memória.

Apresentamos dados quantitativos de três processadores atuais da Intel, relacionados aos principais conceitos descritos nesta seção. Os processadores escolhidos são: Intel i7-4770 (Haswell), Intel Xeon X5650 (Westmere) e Intel i7-7820X (Skylake X), que possuem 4, 6 e 8 núcleos, respectivamente. A Tabela I e II apresentam o tamanho de cada nível da cache e latência, nessa ordem. Assim, enfatizamos a teoria de que quanto maior seu nível, menor seu tamanho e menor seu tempo de acesso. Por fim, na Tabela III, percebe-se que todos adotam associatividade por conjunto acima de 8-*way* para o mapeamento dos dados, assim, realizam buscas rápidas.

Tabela I
TAMANHO DA CACHE

	Intel Haswell	Intel Westmere	Intel Skylake X
L1 Cache	32KB	32KB	32KB
L2 Cache	256KB	256KB	1024KB
L3 Cache	8MB	8MB	11MB

Tabela II
LATÊNCIA DA CACHE

	Intel Haswell	Intel Westmere	Intel Skylake X
L1 Cache	4 ciclos	4 ciclos	4 ciclos
L2 Cache	12 ciclos	10 ciclos	14 ciclos
L3 Cache	36 ciclos	40 ciclos	68 ciclos
RAM	36 cycles + 57 ns	40 cycles + 67 ns	79 ciclos + 50 ns

Tabela III
MAPEAMENTO DA CACHE

	Intel Haswell	Intel Westmere	Intel Skylake X
L1 Cache	8-*way*	8-*way*	8-*way*
L2 Cache	8-*way*	8-*way*	16-*way*
L3 Cache	8-*way*	16-*way*	11-*way*

B. Execução Fora de Ordem

Uma solução presente nos processadores atuais para superar a latência é a execução fora de ordem. Como, por exemplo, uma unidade de busca da memória que precisa aguardar a chegada de dados. Ao invés de travar toda a execução, os processadores modernos executam operações fora de ordem. Desta forma, exercem operações subsequentes para unidades inativas do processador [18]. Como exemplo, vamos utilizar um código simples com operações aritméticas, mostrada no Algoritmo 1.

Algorithm 1 Exemplo de Execução Fora de Ordem - Parte I.

1: $x_1 = \sqrt{a}$;
2: $x_2 = b^3$;
3: $x_3 = c * d$;
4: $x_4 = \sqrt{x_3}$;
5: $x_5 = e - f$;
6: $x_6 = a + 15$;

Caso os valores já estiverem nos registradores, são necessários 6 ciclos de *clock* para concluir a execução deste trecho de código, de modo que cada operação gaste um ciclo. Com os processadores superescalares explorando o paralelismo em nível de instrução, é possível a execução de mais de uma instrução por ciclo. Utilizaremos o exemplo apresentado no Algoritmo 1, para mostrar como ficaria a execução de mais de uma instrução por ciclo de *clock*, no Algoritmo 2.

Algorithm 2 Exemplo de Execução Fora de Ordem - Parte II.

1: $x_1 = \sqrt{a} \quad x_2 = b^3$;
2: $x_3 = c * d; \quad x_4 = \sqrt{x_3}$;
3: $x_5 = e - f; \quad x_6 = a + 15$;

Desta maneira, realizariam duas operações por ciclo *clock*, finalizando a execução com um tempo total de três ciclos para o mesmo conjunto de instruções. Entretanto, na segunda linha é encontrado uma dependência para os valores contidos em x_3 e x_4, caso sejam executadas ambas operações no mesmo ciclo. Lançando mão dos recursos proporcionados pela execução fora de ordem, pode-se realizar a operação x_5 antes de x_4. Isto ocorre pelo fato de não haver dependência entre x_3 e x_5. Portanto, o trecho mencionado teria o comportamento descrito no Algoritmo 3.

Algorithm 3 Exemplo de Execução Fora de Ordem - Parte III.

1: $x_1 = \sqrt{a} \quad x_2 = b^3$;
2: $x_3 = c * d; \quad x_5 = e - f$;
3: $x_4 = \sqrt{x_3}; \quad x_6 = a + 15$;

Para trabalhar com essa execução desordenada, a arquitetura Intel possui mecanismo de execução e no subsistema de memória, o *pipeline* no *front-end*, conforme ilustrado pela Figura 3. As instruções são buscadas pelo *front-end* da memória, decodificadas para microoperações (µOPs) e, enviadas para o mecanismo de execução de maneira contínua. O processamento da instrução é implementado no mecanismo de execução. Há o *buffer* de reordenamento que é o responsável pela alocação, renomeação e anulação de registradores. As µOPs são enfileiradas as operações nas portas de saída conectadas às unidades de execução. Cada unidade de execução realiza tarefas diferentes, como operações da Unidade Lógica Aritmética (ULA), da unidade de geração de endereço (AGU) ou carregamentos e armazenamentos de memória [8].

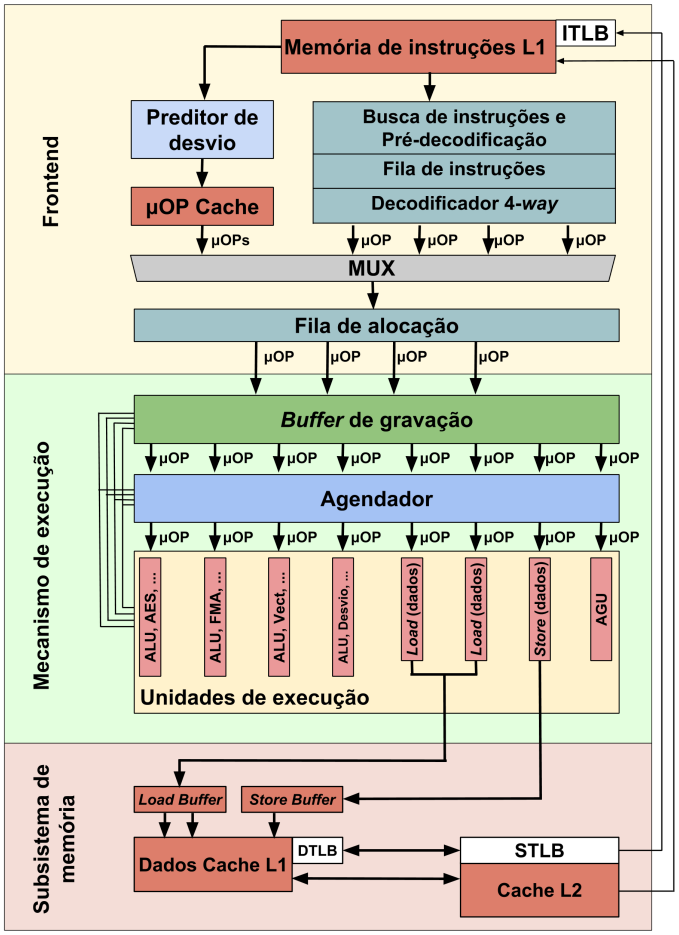

Figura 3. Execução Fora de Ordem implementada no mecanismo de execução.

Geralmente, as CPUs convencionais não executam esse fluxo de instrução linear, em casos de condicionais, por exemplo, a execução em *pipeline* fora de ordem não é suficiente para garantir o desempenho. Uma vez que o processador não sabe o que carregar até as instruções condicionais serem executadas. Então, elas possuem uma unidade de desvio de predição que realiza uma estimativa de qual instrução será executada em sequência. Preditores tentam determinar a decisão que será tomada antes que sua condição seja realmente avaliada. As instruções que não possuem dependências são executadas com antecedência e, se a previsão estiver correta, seus resultados serão usados. Caso contrário, o *buffer* de reordenamento permite a reversão, limpando-o e reiniciando-o a partir da instrução correta. Existem diferentes abordagens para esta

predição: Predição estática, o resultado é baseado apenas na própria instrução; predição dinâmica, reúne estatísticas em tempo de execução para previsão; e, previsão de um nível usa contador para registrar o último resultado. Os processadores mais modernos utilizam predição de dois níveis, os históricos dos últimos resultados, preveem padrões recorrentes [18]. Mais recentemente, predições neurais estão sendo escolhidas para integrar às arquiteturas mais atuais [24].

C. Execução Especulativa

Frequentemente, o fluxo futuro de instruções de um programa não é conhecido. Isto ocorre quando a execução fora de ordem atinge uma condicional que depende de instruções anteriores que não foram concluídas. Assim, o processador pode salvar um ponto de verificação contendo seu estado de registro atual, fazer uma previsão do caminho e, executá-las especulativamente ao longo deste caminho previsto. Se estiver correto, as instruções são retiradas na ordem de execução do programa. Se não, o processador abandona as instruções pendentes no caminho, recarrega seu ponto de verificação e, retoma o caminho correto. Quando este abandono acontece, são executadas instruções de abandono para que não fiquem visíveis para o programa. Desta forma, a execução especulativa mantém o estado lógico como se estivesse seguindo apenas o percurso certo.

Este tipo de execução requer que sejam feitas suposições quanto ao resultado provável das instruções de desvio. Melhores previsões, melhora o desempenho e, aumenta número de operações executada especulativamente são realizadas com sucesso. Vários componentes do processador são utilizados para essa previsão. O *Branch Target Buffer* (BTB) possui um mapeamento de endereços executados recentemente para endereços de destino. Os processadores podem usá-lo para predizer endereços futuros, mesmo antes da decodificação. Para prever se uma condicional é tomada ou não, o processador utiliza um registro de resultados recentes [14]. Analisando a sequencia de código abaixo, é possível utilizar apenas um *pipeline*:

Algorithm 4 Exemplo Execução Especulativa - Parte I.

1: $x_1 = a + b;$
2: $x_2 = x_1 + c;$
3: $x_3 = x_2 + d;$
4: **if** $x_3 = 0$ **then**
5: $\quad x_4 = e + f;$
6: $\quad x_5 = x_4 + g;$
7: $\quad x_6 = x_5 + h;$
8: **end if**

Utilizando a ideia de execução especulativa, é possível reorganizar as instruções de forma que aumente o número de *pipelines*. Assim, as operações encontradas dentro da condicional, serão executadas junto às instruções anteriores ao desvio através de operadores temporários, como mostra o código descrito no Algoritmo 4. Apesar das operações serem realizadas nos três primeiros ciclos de *clock*, os resultados obtidos serão disponibilizados apenas se a condicional for satisfeita, isto é, $x_3 = 0$.

Algorithm 5 Exemplo Execução Especulativa - Parte II.

1: $x_1 = a + b; \quad x'_4 = e + f;$
2: $x_2 = x_1 + c; \quad x'_5 = x'_4 + g;$
3: $x_3 = x_2 + d; \quad x'_6 = x'_5 + h;$
4: **if** $x_3 = 0$ **then**
5: $\quad x_4 = x'_4; \quad x_5 = x'_5; \quad x_6 = x'_6;$
6: **end if**

III. FALHA DE SEGURANÇA

As execuções apresentadas são fundamentais para obter um melhor desempenho computacional. Porém, permitem o acesso a áreas na memória que deveriam ser inacessíveis. A partir dessa abertura, algumas vulnerabilidades podem ser exploradas. Esta seção apresenta o histórico do descobrimento dessas brechas na segurança dos microprocessadores atuais e, em seguida, como são exploradas as principais variantes: Meltdown e Spectre.

A. Histórico

Desde 1995, a maioria dos processadores trabalha com execução especulativa. Foi quando pesquisadores encontraram os primeiros sinais de falhas em *chipsets* da Intel, ao ser descobertas graves vulnerabilidades, permitindo pessoas não autorizadas acessarem áreas não acessíveis [22]. Com isto, qualquer programa havia permissão para ler conteúdos protegidos. Isto ocorre, pois, este funcionamento atinge diretamente o núcleo que conecta aplicativos ao *hardware*, como processador e memória, ou seja, o *kernel*. Em 2012, a Apple adotou em seu núcleo do sistema operacional o KASLR, (*Kernel Address Space Layout Randomization*). O KASLR é técnica de segurança para exploração de vulnerabilidades da memória [15]. Em seguida, o *kernel* do Linux também sofreu essa alteração. Porém, ainda assim, especialistas de segurança consideram quase inútil sua utilização.

Após um estudo detalhado produzido pela Google, conhecido como *Google Project Zero*, divulgado, no dia 2 de janeiro de 2018, uma grave falha de segurança encontrada em chips da Intel, com brechas que atingem processadores da AMD e ARM. Estas informações foram reportadas às respectivas marcas atingidas, em junho de 2017, e foram publicadas apenas seis meses depois. Pesquisas indicam que todos os sistemas são afetados pelo Spectre, inclusive *smartphones* e servidores em nuvem. Diferente do Meltdown, que atinge apenas processadores Intel. Inicialmente, foram descobertas três variantes do problema, sendo a 1 e 2 relativo ao Spectre e a 3 ao Meltdown. Considerados sistemas que utilizam execução especulativa e predição de ramificação que permitem a divulgação não autorizada de informações com acesso de usuário local por meio uma análise do canal lateral [11].

Mais recentemente, a Microsoft e a Google, divulgaram duas novas variantes, 3a e 4. A variante 3a é similar às que foram encontradas anteriormente. O caso da variante 4,

possui uma implementação mais completa, permitindo acessar valores de memórias mais antigos ainda. Todas elas podem ser encontradas no site do *Common Vulnerabilities and Exposures* (CVE)[1], onde possui vulnerabilidades de segurança cibernética conhecidas publicamente em todo o mundo. Cada variante possui, no CVE, uma identificação única, uma descrição e referências de segurança. A seguir, as identificações referentes a cada variante:

- **Variante 1:** *Bypass* de verificação de limites (CVE-2017-5753);
- **Variante 2:** Injeção de alvo de ramificação (CVE-2017-5715);
- **Variante 3:** Leituras especulativas de dados inacessíveis (CVE-2017-5754);
- **Variante 3a:** Registro de sistema falso (CVE-2018-3640);
- **Variante 4:** Desvio de armazenamento especulativo (CVE-2018-3639).

B. Meltdown

Os efeitos colaterais da execução fora de ordem são explorados pelo Meltdown, a partir da leitura de locais arbitrários de memória do *kernel*, incluindo dados pessoais e senhas. Nos sistemas afetados, ele permite o acesso a processos ou máquinas virtuais na nuvem, sem permissões ou privilégios. O Meltdown permite ao usuário ler a memória inteira do *kernel* da máquina que ele está sendo executado, de forma simples. O mesmo explora informações de canal literal disponíveis nos processadores modernos, como em microarquiteturas da Intel, desde 2010 [18]. Suponha que um computador, sem o *patch* de segurança, tenha salvado em sua memória as informações como mostra a Figura 4 e, execute a seguinte sequência de código:

Algorithm 6 Exemplo Meltdown.

1: $secreto = lerCaractere(1000)$;
2: $x_1 = 3 + 2$;
3: $caracteres = [' A', ' B', ' C', ..., ' Z']$;
4: $caracteres[secreto]$;

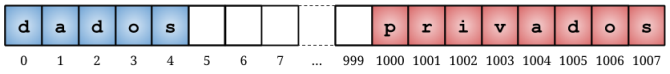

Figura 4. Dados da memória.

A Figura 4, apresenta um esquema de funcionamento da memória, no qual os dados estão salvos até a posição 4 e, a partir da posição 1000, possui informações restritas, que usuários não deveriam ter acesso. A primeira linha do código, do Algoritmo 6, tenta ler um carácter que se encontra nas informações privadas, usando seu endereço. Consequentemente, ele efetua uma exceção, seu código será executado e sua informação salva na memória cache. Em seguida, ele realiza o cálculo de x_1 e cria um vetor que possui todas as letras do alfabeto. Desta forma, é possível analisar cada posição do vetor até encontrar a informação contida na posição 1000. Neste caso, $secreto = p$, a informação considerada privada, torna-se acessível. Portanto, para revelar as outras posições, o código é executado com $lerCaractere(1001)$, retornando a informação contida da posição ao lado, $secreto = r$. Assim, executa sucessivamente, até alcançar o 1007.

[1]https://cve.mitre.org/

C. Spectre

O método utilizado pelos processadores para economizar tempo e melhorar desempenho, a execução especulativa, é explorado pela vulnerabilidade Spectre. Isto é feito de forma que ele aproveita esta especulação para injetar um código malicioso. Da mesma maneira que insere um comando para induzir o processador decifrar uma informação específica, ele retorna um comando que quebra a segurança, permitindo que o invasor tenha acesso a qualquer informação [14].

O primeiro passo para que este ataque ocorra, é fazer com que operações sejam executadas de forma que, posteriormente, uma previsão especulativa acesse dados inexploráveis. Assim, realizam operações para induzir que esta execução ocorra. Uma opção, é fazer com que a cache libere um valor necessário para determinar o destino de uma instrução da ramificação. Além disso, também pode preparar o canal lateral para extrair informações da vítima. Os ataques de canais laterais utilizam técnicas Prime+Probe, que tentam descobrir um conjunto de blocos da memória [25], para atingirem o nível L1 da cache [27].

Em seguida, o processador executa especulativamente instruções que transferem informações confidenciais para um canal lateral da microarquitetura, isto pode solicitar que a vítima execute uma ação. Outra forma, é o invasor aproveitar desta execução para obter informação através do seu próprio código. Embora os dados possam ser fornecidos por canais secundários da memória, ainda é possível que uma operação seja feita para modificar o estado da cache e expor seu valor [14].

Um exemplo de como isso pode ocorrer, é o Algoritmo 7, onde se encontra um conjunto de dados com informações quaisquer. Posteriormente, cria-se um valor de entrada, muito superior ao tamanho no conjunto de dados e define uma condicional, que será satisfeita apenas se o tamanho do conjunto de dados for maior que o valor estipulado. Como este algoritmo realiza a execução especulativa, o código presente dentro da condicional será realizado. Ao executá-lo, paralelamente, será executado um código que exponha os valores inacessíveis da cache, como mostra o Algoritmo 8.

Algorithm 7 Exemplo Spectre - Parte I.

1: $dados = [1, 2, 3, 4]$;
2: $entrada = 1000$;
3: **if** $entrada < tamanho(dados)$ **then**
4: $\quad dados[1000]$;
5: **end if**

```c
#include "libkdump.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

const char *strings[] = {
    "Generating witty test message...",
    "Go ahead with the real exploit if you dare",
    "Please wait while we steal your secrets...",
    "Don't panic..."};

int main(int argc, char *argv[]) {
  libkdump_config_t config;
  config = libkdump_get_autoconfig();
  libkdump_init(config);

  srand(time(NULL));
  const char *test = strings[rand() % (sizeof(
      strings) / sizeof(strings[0]))];
  int index = 0;

  printf("Expect: \x1b[32;1m%s\x1b[0m\n", test);
  printf("   Got: \x1b[33;1m");
  while (index < strlen(test)) {
    int value = libkdump_read((size_t)(test + index)
        );
    printf("%c", value);
    fflush(stdout);
    index++;
  }

  printf("\x1b[0m\n");
  libkdump_cleanup();

  return 0;
}
```

Figura 5. Exemplo simples de execução especulativa IV.

```
Expect: Please wait while we steal your secrets...
   Got: Please wait while we steal your secrets...
```

Figura 6. Exemplo da saída do algoritmo.

O código cria um conjunto com todos os caracteres e faz a leitura da primeira posição. Como mostra na Figura 4, a posição de número 1000 contém a letra p. Sendo assim, o código executará a linha 2, do Algoritmo 8, de todas as posições até a encontrar a letra p, no caso, a 15^a. Enquanto o Algoritmo 7 é executado com $entrada$ até 1007, o Algoritmo 8 executa até a posição de cada caractere, consecutivamente. Assim ao finalizar, toda a informação privada estará disponível para o invasor.

Algorithm 8 Exemplo Spectre - Parte II.

1: $caracteres = [' a', ' b', ' c', ..., ' z'];$
2: $caracteres[0];$
3: $caracteres[secreto];$

IV. Estudo de Caso

Para verificarmos o potencial efeito e visualizarmos a efetividade da execução especulativa, executamos os algoritmos apresentados por [14]. Nesta etapa de avaliação utilizou o sistema operacional Linux sem a devida proteção proporcionada pelos *patchs* de correção. Posteriormente, o sistema foi atualizado, tornando-o invulnerável para as variantes de ataques Spectre citadas neste artigo.

O trabalho apresentado em [14] lista detalhes referentes aos ataques Spectre em alto nível. Deste modo, é possível verificar visualmente como a execução especulativa quebra a barreira de segurança dos processadores, podendo assim vazar informações sigilosas. Para que o ataque ocorra de forma controlada, os autores criaram e tornaram públicos[2] alguns programas simples os quais abrigam dados secretos. Para ter acesso a esses dados, outra aplicação maliciosa explora os recursos de execução especulativa para ter acesso a todo endereço de memória da máquina, inclusive os endereços contendo as informações privados.

A Figura IV apresenta o código em linguagem C de um destes testes. Este algoritmo contém dados secretos armazenados em *strings[]*. Seu objetivo é escolher de forma direta uma frase secreta entre as possíveis, de modo aleatório e apresentá-la. Em seguida, indiretamente, o programa consegue montar e retorna a mesma frase como apresentado na Figura IV. Entretanto, para que a mesma frase pudesse ser encontrada, o programa buscou caractere a caractere, acessando o bloco de memória onde a mesma se encontrava referenciada. Desta maneira, devido ao acesso indevido ao conteúdo da memória do programa, o vazamento de informações torna-se viável.

A abordagem citada na Figura IV é simples e exibe o conteúdo referente à recuperação de dados secretos em um mesmo programa. Contudo, outras variações de programas apresentam abordagens mais complexas. As quais possibilitam que uma aplicação maliciosa acesse endereços de memória reservado a outros programas. Há ainda atividades maliciosas capazes de encontrar o endereço de randomização KASLR, o que proporciona identificar com precisão onde estão armazenados os dados protegidos pelo sistema operacional.

[2]https://github.com/IAIK/meltdown

Além de verificar e avaliar a capacidade invasiva proporcionada pela execução especulativa, este estudo de caso tem como objetivo secundário, validar a efetividade das soluções implementadas em *patchs* de correções. Portanto, os resultados invasivos obtidos com execução dos algoritmos, no sistema sem proteção, foram reiterados com o sistema atualizado. Com isto, foi possível verificar que apenas o algoritmo apresentado na Figura IV obteve sucesso, uma vez que a execução especulativa ocorre apenas no endereço do processo. As demais tentativas de ataques avaliadas, tais como obter o endereço KASLR, e acessar o conteúdo reservado a um processo distinto à aplicação maliciosa, não obtiveram êxito. Outro dado relevante em relação à avaliação, é a aferição da leitura confiável em memória. Onde, nos testes sem proteção apresentavam uma taxa de confiabilidade de leitura 99.9%, contra 0.39% com o sistema atualizado. Assim, comprova-se a eficiência proporcionada pelas correções disponibilizadas nas atualizações do sistema operacional.

V. Impactos no Desempenho dos Sistemas Operacionais

Algumas empresas se pronunciaram sobre medidas tomadas para amenizar o impacto dessas vulnerabilidades. As providências estão relatadas a seguir, a partir de informações retiradas dos relatórios de segurança divulgados em seus respectivos sites. Em seguida, realizamos uma análise de desempenho, com o objetivo de mensurar a perda real devido às atualizações de seguranças.

A. Correções

Esta classe de ataques de canal lateral, incluindo suas variantes, permitem que usuários ou aplicações maliciosas tenham acesso à dados sigilosos gerenciados pelo sistema operacional. As correções ou atualizações com intuito de amenizar o impacto de tentativas de acesso indevido envolvem alguns passos discretos. Estes passos e suas aplicações variam conforme a marca e modelo da arquitetura do microprocessador. Cada microprocessador pode ser vulnerável a alguma variante do ataque. Os *patchs* de correção trouxeram juntamente com o benefício da proteção e segurança, um impacto no desempenho dos sistemas atuais [16]. Grandes empresas divulgaram comunicados oficiais, relatando estarem cientes das vulnerabilidades e o impacto gerado por elas. Segue alguns feitos realizados por essas grandes companhias:

Intel: Disse estar trabalhando com outras empresas (AMD, ARM e fornecedores de sistemas operacionais) e, começou a fornecer atualizações de *software* e *firmware* para amenizar as consequências dessas vulnerabilidades. Afirmou que, ao contrário de alguns relatórios, qualquer impacto no desempenho depende da carga de trabalho e configuração da plataforma. Para usuário comum, não deve ser significativo, apenas para máquinas potentes e servidores são relevantes [13].

AMD: Uma das variantes do Spectre pode ser retirada através de correções de software e representa um impacto insignificante na performance, segundo a organização. No caso do Meltdown o risco é quase zero, pois seus componentes não estão sujeitos a essa falha de segurança [2].

ARM: A companhia informou que os núcleos Cortex-M, geralmente empregados em soluções da Internet das Coisas são imunes ao Spectre. Todavia, alguns núcleos da família Cortex-A, que equipam sistemas da Qualcomm, Samsung e TSMC estão na lista de componentes vulneráveis. A organização comprometeu, a partir de julho de 2018, disponibilizar atualizações para parte dos atingidos [5].

Microsoft: A empresa declarou que a maior parte da estrutura do Azure foi atualizada para solucionar essas vulnerabilidades. Para maioria dos clientes, o impacto no desempenho é imperceptível, apenas o desempenho de rede é notório em alguns casos [21]. Quanto à linha Xbox One, o vice-presidente corporativo afirmou que a arquitetura Jaguar presente na CPU de seus consoles é imune a ambas vulnerabilidades [20].

Linux: O sistema conta com uma correção de *kernel* para o Meltdown[3]. Ao presumir que todos os chips de 32 bits são inseguros e passíveis de rodar o *patch*, força a correção inclusive nos processadores AMD.

Apple: Informou que todos os seus dispositivos são afetados pelo Meltdown e/ou Spectre, exceto o Apple Watch. Entretanto, afirma que não existe nenhum caso de invasão detectada. Mesmo assim, a companhia liberou *patchs* de correção contra Meltdown nas atualizações do iOS 11.2, macOS 10.13.2 e tvOS 11.2 e contra o Spectre no Safari 11.0.2 [4].

Google: Foi inserida uma correção do Spectre para a linha Nexus/Pixel nos *patchs* de segurança recentes. No entanto, as correções em outros dispositivos Android caberá aos fabricantes implementá-las por conta própria [3]. O Google Chrome incorpora mais correções a partir da versão 64. Chromebooks e outros dispositivos que rodam o Chrome OS versões 3.18 e 4.4 estão com o *Kernel Page Table Isolation* (KPTI) corrigidos. Os mais antigos serão atualizados posteriormente e os demais produtos e serviços seguem sem serem afetados [9].

Outros desenvolvedores de sistemas tornaram público alguns dados referentes às perdas de desempenho causadas pelas estratégias de proteção implementadas. As informações iniciais e especulações apontavam que a perda de desempenho seria em torno de 30%. Entretanto com analises mais detalhas das soluções aplicadas nota-se que o efeito real é consideravelmente inferior. A RedHat (desenvolvedora de uma distribuição Linux) divulgou dados referente ao impacto no desempenho, causado pela implantação das soluções [17]. As primeiras estratégias causaram uma perda de desempenho entre 1 e 20%. A empresa tem melhorado e monitorado constantemente as atualizações de *softwares* corretivos. Deste modo seus últimos resultados divulgados apontam para uma perda de desempenho melhor, estando na faixa de 1 a 8% de perda. Estas informações são mensuradas a partir de testes em *Benchmarks* comerciais. Estes valores podem variar conforme a carga de trabalho adicionada.

B. Análise de desempenho

Com intuito de validar as informações anunciadas pelas empresas desenvolvedoras, e mensurar a perda de desempenho real proporcionado pelas atualizações de seguranças, testes de *benchmark* foram realizados antes e depois da atualização do núcleo do sistema operacional. A máquina utilizada para testes possui processador Intel Core I7 de terceira geração com 8 núcleos e *clock* de 2.10 GHz. Possui ainda, 6GB de memória RAM além de 64K de cache L1, 256K de L2 e 6144K de cache L3. O sistema operacional instalado nesta máquina é o Deepin 15.5, o qual é baseado na distribuição Debian. Esta versão de sistema operacional antes da atualização do núcleo do sistema operacional portava a versão Linux 4.9.0-deepin12-amd64. Com esta versão foram coletados os dados de desempenho sem os *patchs* de segurança. Em seguida, a versão do núcleo do sistema foi atualizada de modo a prover os mecanismos de segurança, sendo esta a versão a Linux 4.14.0-deepin2-amd64.

Duas ferramentas específicas para *Benchmarks* foram escolhidas para avaliar o desempenho e medir os tempos de execução gastos em execução de tarefas específicas. Os testes foram feitos utilizando o GtkPerf e o Linux-Bench. Para a

[3]https://lkml.org/lkml/2017/12/4/709

apresentação dos dados referentes a tempo, foram coletados 10 amostras e calculada a média. Os resultados referentes a analise de desempenho medido pelo Linux-Bench é apresentados na Tabela IV, onde a primeira coluna expõe os testes realizados e as duas colunas seguintes apresentam o tempo, em segundos, gasto por cada aplicação com e sem a adição dos mecanismos de proteção, consecutivamente. A quarta coluna expõe em porcentagem o quanto as estratégias afetaram o desempenho do sistema como um todo.

Tabela IV
IMPACTO NO DESEMPENHO COMPUTACIONAL AFERIDO COM AUXILIO DA FERRAMENTA LINUX-BENCH.

Benchmarks	Sem *patch* (s)	Com *patch* (s)	Impacto (%)
Blowfish	2,29	2,40	5,12
CryptoHash	651,03	725,45	11,43
Fibonacci	0,80	0,76	-5,03
N-Queens	7,22	7,22	0,11
FPU FFT	1,29	1,45	12,57
FPU Raytracing	1,95	1,91	-1,90
GPU Drawing	93,78	97,38	3,84
Tempo total	**758,34**	**836,58**	**10,32**

Tabela V
IMPACTO NO DESEMPENHO COMPUTACIONAL AFERIDO COM AUXILIO DA FERRAMENTA GTKPERF.

Benchmarks	Sem *patch* (ms)	Com *patch* (ms)	Impacto(%)
Entry	0,03	0,03	0,00
ComboBox	0,70	0,83	18,57
ComboBoxEntry	0,61	0,72	18,03
SpinButton	0,10	0,11	10,00
ProgressBar	0,12	0,13	8,33
ToggleButton	0,19	0,22	15,79
CheckButton	0,09	0,12	33,33
RadioButton	0,12	0,14	16,67
TextView-AddTex	0,15	0,15	0,00
TextView-Scroll	0,01	0,11	1000,00
DrawArea-Line	0,73	0,68	-6,85
DrawArea-Circ	0,79	0,83	5,06
DrawArea-Text	0,17	0,13	-23,53
DrawArea-Pixb	0,07	0,07	0,00
Tempo total	**3,88**	**4,27**	**10,05**

Nota-se que para o conjunto de teste apresentado na Tabela IV a perda de desempenho máxima foi em torno de 12.5% para o algoritmo de *FPU FFT*. Entretanto, outras aplicações não foram afetadas, como no caso da *FPU Raytracing*, a qual teve um desempenho de aproximadamente 2% melhor, ou até mesmo a função de *Fibonacci* com desempenho na faixa de 5% superior, uma vez que este algoritmo não explora os recursos de computação especulativa. Deste modo, avaliando o tempo total da execução de todos os métodos do pacote *Linux-Bench*, com e sem proteção à falha, obtivemos uma perda de desempenho um pouco superior a 10% para o sistema protegido.

Para o conjunto de testes fornecidos pela ferramenta GtkPerf, os resultados foram semelhantes aos apresentados anteriormente. Mesmo que *Benchmarks* tenham apresentado uma variação maior em alguns testes. Na média, computando o tempo total de execução, temos uma perda de desempenho próxima a 10%. Deste modo, obtemos dados condizentes com os informados no início desta seção. Os resultados obtidos com o auxílio da ferramenta GtkPerf encontram-se na Tabela V, a qual possui a mesma disposição da Tabela IV.

VI. CONCLUSÃO E TRABALHOS FUTUROS

O presente artigo analisou conceitos fundamentais sobre as medidas tomadas por processadores modernos para obter um melhor desempenho e seus efeitos colaterais. Logo, foram apresentadas as principais falhas de segurança presentes nos sistemas atuais: Meltdown e Spectre. Descrevemos o funcionamento de cada uma dessas vulnerabilidades e a forma como são encontradas. Além disso, verificamos a efetividade da execução especulativa através de um estudo de caso realizado no sistema operacional Linux sem *patch*. A partir do sucesso obtido por esse estudo, realizamos testes de *benchmarks*, antes e depois da atualização do núcleo do sistema operacional.

Os ataques apresentados permitem que usuários ou aplicações maliciosas tenham acesso a dados restritos, preocupando grandes companhias, como Intel, ARM e AMD, que rapidamente se manifestaram. A maioria dessas organizações anunciou ser vulneráveis a esses ataques e que adotaram medidas de proteção e segurança. Entretanto, essas providências tomadas exigem atualizações que colocam em risco o desempenho computacional. Algumas companhias alegaram que o impacto gerado é imperceptível, como a RedHat, que em seus últimos resultados divulgados apontaram uma perda entre 1 e 8%. A fim de validar essas informações, realizamos testes com e sem *patchs* de correção, nas ferramentas GtkPerf e Linux-Bench, obtendo a perda um pouco maior de 10% nos dois casos.

Portanto, mesmo com o propósito de melhorar o desempenho, alguns processadores modernos tiveram perdas ao tentarem se proteger desses ataques. O próximo objetivo da pesquisa é abranger a análise realizada em outras ferramentas específicas para *benchmarks* e, em outros sistemas. Espera-se ainda relizar estudos de caso os quais abordam as novas variantes dos ataque Meltdown e Spectre. Desta forma, será possível realizar uma comparação gradativa do impacto causado por essas vulnerabilidades em cenários distintos. Avaliando ainda o efeito causados pelas evoluções das soluções implementadas, obtendo resultados ainda mais consistentes.

REFERÊNCIAS

[1] J. Ahmed, M. Y. Siyal, S. Najam, and Z. Najam. Multiprocessors and cache memory. In *Fuzzy Logic Based Power-Efficient Real-Time Multi-Core System*, pages 1–15. Springer, 2017.

[2] AMD. Amd processor security updates. Disponível em: https://www.amd.com/en/corporate/security-updates. Acessado em: 17/06/2018, 2018.

[3] Android. Android security bulletin—january 2018. Disponível em: https://source.android.com/security/bulletin/2018-01-01. Acessado em: 17/06/2018, 2018.

[4] Apple. About speculative execution vulnerabilities in arm-based and intel cpus. Disponível em: https://support.apple.com/en-us/HT208394. Acessado em: 17/06/2018, 2018.

[5] ARM Developer. Speculative processor vulnerability. Disponível em: https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability. Acessado em: 17/06/2018, 2018.

[6] D. Baudisch and K. Schneider. Evaluation of speculation in out-of-order execution of synchronous dataflow networks. *International Journal of Parallel Programming*, 43(1):86–129, 2015.

[7] J. Charles, P. Jassi, N. S. Ananth, A. Sadat, and A. Fedorova. Evaluation of the intel® core™ i7 turbo boost feature. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 188–197. IEEE, 2009.

[8] I. Coorporation. Intel 64 and ia-32 architectures optimization reference manual, 2009.

[9] Google. Today's cpu vulnerability: what you need to know. Disponível em: https://security.googleblog.com/2018/01/todays-cpu-vulnerability-what-you-need.html. Acessado em: 17/06/2018, 2018.

[10] M. Hashemi, E. Ebrahimi, O. Mutlu, Y. N. Patt, et al. Accelerating dependent cache misses with an enhanced memory controller. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 444–455. IEEE Press, 2016.

[11] J. Horn. Reading privileged memory with a side-channel. *Project Zero*, 3, 2018.

[12] X. Huang, L. Zhang, R. Li, L. Wan, and K. Li. Novel heuristic speculative execution strategies in heterogeneous distributed environments. *Computers & Electrical Engineering*, 50:166–179, 2016.

[13] Intel. Intel responds to security research findings. Disponível em: https://newsroom.intel.com/news/intel-responds-to-security-research-findings/. Acessado em: 17/06/2018, 2018.

[14] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, Jan. 2018.

[15] J. Levin. *Mac OS X and IOS Internals: To the Apple's Core*. John Wiley & Sons, 2012.

[16] R. H. E. Linux. Kernel Side-Channel Attacks - cve-2017-5754 cve-2017-5753 cve-2017-5715, 2018.

[17] R. H. E. Linux. Speculative Execution Exploit Performance Impacts - describing the performance impacts to security patches for cve-2017-5754 cve-2017-5753 and cve-2017-5715, 2018.

[18] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *ArXiv e-prints*, Jan. 2018.

[19] E. Malathy and C. S. Thirumalai. Review on non-linear set associative cache design. *IJPT*, 8:5320–5330, 2016.

[20] Microsoft. Windows client guidance for it pros to protect against speculative execution side-channel vulnerabilities. Disponível em: https://support.microsoft.com/en-us/help/4073119/protect-against-speculative-execution-side-channel-vulnerabilities-in. Acessado em: 17/06/2018, 2018.

[21] Microsoft Azure. Securing azure customers from cpu vulnerability. Disponível em: https://azure.microsoft.com/en-us/blog/securing-azure-customers-from-cpu-vulnerability/. Acessado em: 17/06/2018, 2018.

[22] O. Sibert, P. A. Porras, and R. Lindell. The intel 80/spl times/86 processor architecture: pitfalls for secure systems. In *Security and Privacy, 1995. Proceedings., 1995 IEEE Symposium on*, pages 211–222. IEEE, 1995.

[23] D. Šidlauskas and C. S. Jensen. Spatial joins in main memory: Implementation matters! *Proceedings of the VLDB Endowment*, 8(1):97–100, 2014.

[24] E. Teran, Z. Wang, and D. A. Jiménez. Perceptron learning for reuse prediction. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–12. IEEE, 2016.

[25] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[26] D. Wang, G. Joshi, and G. Wornell. Efficient task replication for fast response times in parallel computation. In *ACM SIGMETRICS Performance Evaluation Review*, volume 42, pages 599–600. ACM, 2014.

[27] Y. Yarom and N. Benger. Recovering openssl ecdsa nonces using the flush+ reload cache side-channel attack. *IACR Cryptology ePrint Archive*, 2014:140, 2014.