# Ensino de Arquitetura e as Predições Tomadas por Desempenho com as Predições não Tomadas pela Segurança: Vulnerabilidades Meltdown e Spectre

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

***Resumo*—Na busca por desempenho não foram levados em consideração os efeitos colaterais gerados pelas correlações entre as otimizações, resultando hoje em vulnerabilidades como *Spectre* e *Meltdown* que ameaçam a segurança dos processadores fabricados nas duas últimas décadas. Este artigo apresenta uma abordagem para motivar o ensino de arquitetura de computadores para melhor compreender as vulnerabilidades, mostrando que uma visão ampliada e correlacionada de vários temas é necessária. Estes temas incluem a execução fora de ordem, a execução especulativa, as caches, os medidores em hardware, a memória virtual, dentre outros. A primeira parte deste trabalho apresenta uma proposta simplificada de ensino das otimizações para buscar desempenho e como correlacioná-las para fundamentar a explicação das vulnerabilidades. A metodologia é baseada em exemplos. A segunda parte mostra experimentos que podem ser utilizados para avaliar alguns impactos das vulnerabilidades. As grandes empresas do ramo, como Intel, AMD, ARM, entre outras, desenvolveram *patches* de correção. Porém os *patches* impactam diretamente no desempenho dos sistemas. Além da abordagem para ensino, o impacto do desempenho computacional com e sem os *patches* de segurança é ilustrado. A partir de testes feitos em *benchmarks*, foram observadas perdas de desempenho em torno de 10% para o conjunto de programas do *Linux-Bench*, podendo chegar a 16% para o conjunto de programas do *GtkPerf* a fim de garantir da proteção dos sistemas.**

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

## I. Introdução

Na busca por desempenho, decisões tomadas deixaram espaço para efeitos colaterais que tem sido explorados como vulnerabilidades de segurança. As duas falhas *Spectre* [14] e *Meltdown* [19], descobertas recentemente, têm impacto direto nas decisões de projeto a serem tomadas no desenvolvimento de novos processadores. As falhas abrem oportunidades de ensino e novos desafios no desenvolvimento de soluções de desempenho e segurança. Neste contexto, este trabalho apresenta uma abordagem de ensino com o objetivo de ilustrar e correlacionar os conceitos envolvidos para a compreensão das falhas.

De forma resumida, as falhas exploram a execução fora de ordem das instruções, a execução especulativa e a medição das diferenças de tempo de acesso entre a memória principal e a cache. Quando dados protegidos são requisitados durante uma execução especulativa, estes dados são lidos da memória, armazenados em registradores temporários e na cache. Somente quando a especulação equivocada é invalidada, as ações de segurança são tomadas. O processador retorna ao fluxo correto e anula as execuções especulativas. Porém, os dados protegidos requisitados durante o ataque podem permanecer armazenados na cache. Esta é a primeira etapa do ataque. Na segunda etapa, o código acessa os dados que podem ter sido trazidos para cache e mede o tempo de busca. Caso os dados estejam na cache, o tempo de acesso será menor e pode-se inferir seus valores, ou seja, obter acesso aos dados que deveriam estar protegidos.

As falhas foram descobertas pelo *Google Project Zero*, em 2017, o qual apontou que a maioria dos processadores que estão no mercado são vulneráveis [11]. Deste modo, as grandes empresas de serviços de nuvem, sistemas operacionais e fabricantes de processadores estão forçadas a desenvolver soluções de segurança, o que impacta na perda de desempenho [16].

Neste trabalho além de apresentar uma metodologia de ensino que apresenta os conceitos envolvidos e explorados pelas falhas, foram realizados experimentos para analisar a segurança e as perdas de desempenho. Primeiro, avaliamos um exemplo simples de execução especulativa, com objetivo de verificar as soluções implementadas nos *patches* de segurança. Em seguida, códigos de testes foram executados para analisar o desempenho antes e depois da utilização dos *patches* usando os conjuntos de teste do *GtkPerf* e do *Linux-benck*.

O presente trabalho está estruturado da seguinte forma: a Seção II contextualiza os tópicos de cache, memória virtual, execução fora de ordem e especulativa. A Seção III ilustra exemplos didáticos das falhas e suas variantes. A Seção IV mostra uma das variantes das falhas. A Seção V relata os impactos dessas vulnerabilidades no nível do sistema operacional. Por fim, a Seção VI apresenta as considerações finais.

## II. Fundamentação Teórica

Nesta seção apresentamos os tópicos em arquitetura de computadores que foram exploradas pelas falhas: memória cache, memória virtual, execução fora de ordem e execução especulativa. Não é objetivo do artigo a geração de material

didático aprofundado em cada tópico, mas sim a contextualização e correlação entre os tópicos para a compreensão dos mecanismos explorados pelos ataques.

### A. Memória Cache

Os processadores possuem uma frequência de relógio em torno de 3 GHz, o que gera um ciclo de relógio de 0,33 ns. A latência das memórias é cerca de 100 ns, portanto são necessários de 200 à 300 ciclos de relógio para buscar dados da memória RAM [6]. O uso de caches esconde a latência da RAM, provendo uma memória com alto desempenho. Com a exploração da localidade temporal e espacial dos dados e instruções, as caches conseguem reduzir o tempo médio de acesso. Por exemplo, se a taxa de acerto da cache é de 99% em apenas 1% dos casos, os dados não estarão na cache, o que obriga o processador a buscar os dados na memória. Se a latência da cache for de 4 ciclos, o tempo médio de acesso pode ser calculado por $t_{medio} = Tempo_{cache} + Taxa_{falhas} * Tempo_{RAM} = 4 + 0,01 * 300 = 7$. Portanto, a utilização da cache pode reduzir o tempo médio de 300 para 7 ciclos.

A memória cache é organizada em níveis de forma hierárquica, nos quais as caches menores e mais rápidas ficam no topo e as caches mais lentas e maiores na base para prover a velocidade do nível mais alto com a capacidade do nível mais baixo. A figura 1 ilustra os processadores atuais que possuem três níveis de cache. É importante destacar que o programador não tem controle se o dado ou a instrução está ou não nas caches. Tudo é feito de forma transparente para aumentar o desempenho. A taxa de falhas depende da localidade espacial e temporal dos dados, das dimensões da cache, do tamanho do bloco, do tipo de mapeamento e da política de substituição de dados [10].

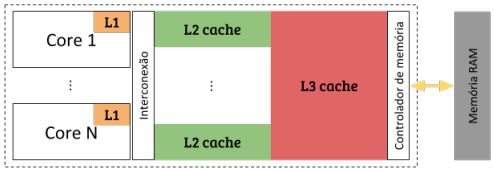


Figura 1. Hierarquia da memória.

Apesar de não ser possível garantir se o dado está ou não na cache, pode-se estimar. Este fato é utilizado por muitas bibliotecas para geração de código otimizado. As Tabelas I e II apresentam dados quantitativos de tamanho e tempo de acesso para três processadores da Intel. Pode-se observar que o tempo de acesso à cache L1 é 50 vezes menor que o tempo de acesso à memória. Ou seja, a perda de desempenho será grande se a cache for retirada para promoção de segurança ao sistema. Portanto, as soluções devem manter a cache e prover segurança.

Com relação as falhas *Spectre* e *Meltdown*, é necessário destacar alguns pontos. Conforme mencionado, o controle da cache não é acessível ao programador e, portanto, não pode-se ter certeza da presença de um determinado dado na cache. Porém, como o tamanho das caches é conhecido e na maioria

Tabela I
TAMANHO DA CACHE

| | Intel Haswell 4 núcleos | Intel Westmere 6 núcleos | Intel Skylake X 8 núcleos |
|---|---|---|---|
| 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 | 207 ciclos | 241 ciclos | 229 ciclos |

dos processadores atuais a cache L1 tem 32K e a L2 256K, muitos programadores exploram estas dimensões para otimização dos códigos. Outros recursos são os medidores de tempo em hardware que podem ser acessados pelos programadores. Estes permitem a verificação da presença ou ausência do dado na cache através da medição do tempo de acesso ao mesmo. Este recurso é uma das bases dos novos ataques.

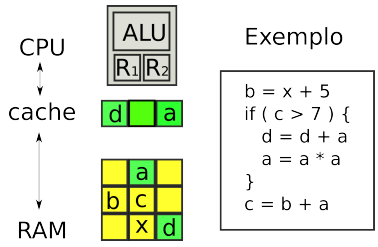


Figura 2. Exemplo de código e sistema Processador-Cache-RAM

Suponha um exemplo com alguns dados na cache e outros na memória, ilustrado na figura 2. O comando $b = x + 5$ irá demorar a finalizar a execução devido ao fato das variáveis $b$ e $x$ estarem na memória. Já a expressão $d = d + a$ pode executar rapidamente pois as variáveis $a$ e $d$ estão na cache. Como as operações com a memória possuem alta latência, o tempo de execução depende mais da localização dos dados do que das operações de cálculo realizadas. Este aspecto também é explorado pelos ataques. A estratégia do programador do ataque é a utilização de um comando condicional com variáveis que possuam grandes chances de não estarem na cache. No exemplo da figura 2, o teste $c > 7$ começa a ser executado com a variável $c$ na RAM. Supondo que o bloco $if$ seja executado especulativamente e fora de ordem para otimização de tempo, as expressões com $a$ e $d$ serão executadas antes mesmo da expressão $b = x + 5$ e da validação do teste $c > 7$. Nas próximas seções serão reforçados estes pontos e correlacionados com outras otimizações.

### B. Memória Virtual

Desde da década de 70, os computadores (servidores da época) já trabalhavam com um sistema de memória virtual. No início dos anos 80, os processadores começaram a incorporar

suporte para memória virtual em hardware. Na família x86, o suporte foi introduzido em 1985 com o lançamento do 80386, que permitiu o desenvolvimento e implementação dos sistemas operacionais multi-tarefa e multi-usuário para os computadores pessoais. A partir do suporte em hardware, a primeira versão do sistema operacional *Microsoft Windows* foi lançada também em 1985. Posteriormente, a primeira versão de Linux foi disponibilizada em 1991. Fatos que mostram a importância dos sistemas de memória virtual com suporte em hardware.

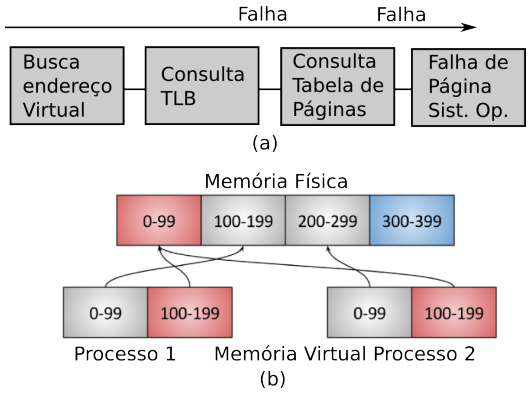


Figura 3. (a) Mapeamento de Endereços Virtuais (b) Exemplo de Mapeamento.

A memória virtual cria uma camada de abstração sobre o endereço físico. Os processos possuem seu espaço de memória a partir da posição 0 até uma posição $n$, independente da capacidade ou estrutura física do hardware. O processador, em conjunto com o sistema operacional, faz o mapeamento dos endereços virtuais em endereços reais. A Figura 3(a) mostra as principais etapas do mapeamento. Primeiro o processador consulta a TLB (*Translation Lookaside Buffer*), que é uma cache de endereços contida dentro do processador com o objetivo de mapear diretamente o endereço virtual no endereço físico. Caso o endereço virtual não esteja na TLB, um registrador do processador (CR3 no Intel X86) faz uma consulta a tabela de página em memória. Tem-se então duas situações. Se a página que contém os dados está mapeada em memória, os privilégios de acesso são verificados (permissão leitura, escrita e execução), o endereço é mapeado na TLB e segue a execução. Todo este processo é feito em hardware com suporte do processador. Caso a página não esteja na memória, ocorrerá uma falha de página, uma interrupção no nível de hardware, ao qual o sistema operacional deverá intervir e o processo pode ser retirado de execução até que o sistema operacional faça a alocação de memória física e atualização do sistema de paginação.

Após a criação da abstração de memória virtual, vários outros aspectos foram explorados para obtenção de desempenho. Para ilustração, será utilizado um exemplo simplificado proposto em [13], apresentado na Figura 3(b). O espaço de memória que apenas o sistema operacional pode acessar é ilustrado em vermelho e é mapeado nas posições físicas 0–99. Os processos dos usuário não tem acesso a área do sistema operacional. As áreas dos processos 1 e 2 do usuário estão ilustradas com a cor cinza. A Figura 3(b) ainda ilustra em azul uma área física não alocada. Podem-se observar que os dois processos possuem as posições virtuais de 0-99 mapeadas em diferentes regiões da memória física. Para um maior desempenho, a área do sistema operacional é mapeada no espaço virtual do processo, posições 100-199 para este exemplo. Estas, por sua vez, são mapeadas na mesma área da memória física pois são compartilhadas, o que evita duplicações.

A página virtual tem bits associados para controle de privilégios de acesso. Mesmo que a memória do sistema operacional (SO) esteja mapeada na área do processo do usuário, enquanto o processo executar em modo usuário, não será possível o acesso a parte de memória do SO. Caso o processo tente acessá-las, ocorrerá uma interrupção e o SO terminará a execução do processo. Entretanto, se o processo estiver executando em modo SO (quando executa uma chamada ao sistema operacional por exemplo), o processo terá acesso a área do SO (100-199 no exemplo). Esta técnica da mapeamento do SO no espaço de memória virtual do processo é uma prática comum utilizada nos últimos 30 anos para o aumento do desempenho, pois as chamadas do sistemas operacional, que são frequentes, compartilharão o mesmo espaço virtual, TLB, etc.

Os ataques exploram a não verificação dos bits de proteção durante uma execução especulativa e o fato do mesmo espaço de memória virtual ser compartilhado pela aplicação e o SO. Na Seção III-B é apresentada a forma como o ataque *Meltdown* explora estes aspectos.

### C. Execução Fora de Ordem

Desde da introdução dos mecanismos de escalonamento dinâmico [24] com as técnicas *scoreboard* e *tomasulo* [25] na década de 60, os processadores são capazes de executar instruções fora de ordem para aumentar seu desempenho.

**Exemplo 1:** Código Original

1: $b = \sqrt{a}$;
2: $c = b^3$; // depende da anterior
3: $c = c * d$; // depende da anterior
4: $d = \sqrt{e + f}$; // não tem dependências
5: $f = d/f$; // depende do valor de $d$ da instrução anterior
6: $f = f + 15$; // depende da anterior

Ao considerar o trecho de código ilustrado no Exemplo 1, pode-se perceber que há uma dependência entre as três primeiras instruções e entre as três últimas. No modo de execução em ordem, a segunda instrução e as demais somente executarão após o término da primeira instrução. Caso a variável $a$ não esteja nos registradores e nem na cache, a primeira instrução será lenta devido a necessidade da busca de $a$ na memória e também da operação de raiz quadrada a ser executada. Com o escalonamento dinâmico e execução fora de ordem, pode-se acelerar a execução. O trecho pode ser executado como ilustrado no Exemplo 2.

Mesmo que a primeira operação $b = \sqrt{a}$ demore em função da busca na memória, a operação $d = \sqrt{e + f}$ já pode ser executada.

**Exemplo 2:** Execução fora de ordem

1: $b = \sqrt{a}; \quad d = \sqrt{e + f};$
2: $c = b^3; \quad f = d/f;$
3: $c = c * d; \quad f = f + 15$

No caso dos processadores da Intel, a arquitetura possui um *pipeline* longo com três grandes partes: *front-end* de busca e decodificação das instruções, mecanismo de execução e um subsistema de memória, conforme ilustrado pela figura 4. As instruções são lidas pelo *front-end* da memória, decodificadas para micro-operações (µOPs). As micro-operações são então enviadas para o mecanismo de execução de maneira contínua. O processamento da instrução é implementado no mecanismo de execução. O *buffer* de reordenamento é o responsável pela alocação, renomeação e anulação de registradores em caso de erros nas execuções especulativas. As instruções podem ser executadas fora de ordem, mas, ao término, a gravação final dos registradores é executada em ordem [7].

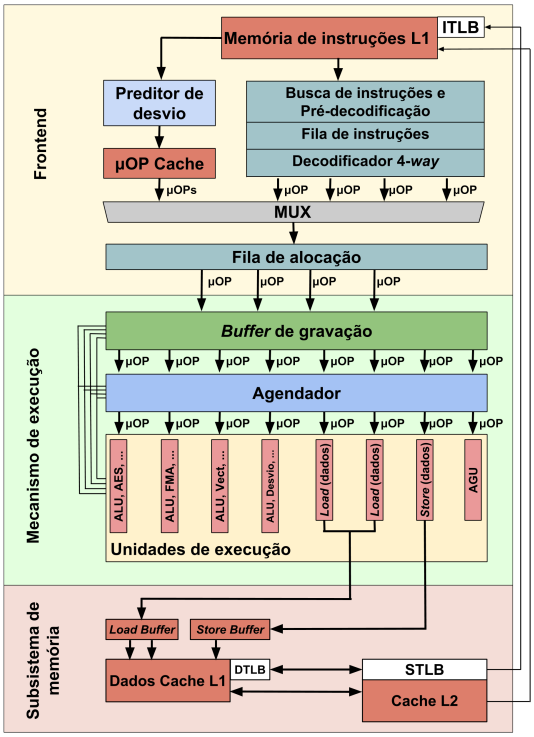


Figura 4. Diagrama de blocos de um processador Intel com mecanismos de execução fora de ordem e predição de desvios.

### D. Execução Especulativa

Na figura 4 é possível observar a presença do preditor de desvio no *Frontend* do processador Intel. Assim como a cache, os preditores são fundamentais para a garantia do desempenho dos processadores superescalares. Em média, a cada 5 instruções de código temos uma instrução de desvio condicional. Os processadores possuem um mecanismo de predição dinâmica para antecipar qual caminho seguir na presença de um desvio e, desta forma, executar trechos de instruções de forma especulativa [5]. As instruções que não possuem dependências são executadas fora de ordem com antecedência. Caso a predição seja correta, os resultados já calculados antecipadamente serão usados. Caso contrário, o *buffer* de reordenamento permite a reversão, limpando e reiniciando os registradores a partir da instrução de desvio e com prosseguimento para o caminho correto. Apesar da limpeza dos registradores, caso alguma instrução de leitura seja executada de forma especulativa, o dado requisitado por esta instrução pode ficar na cache. A princípio, apenas os registradores e memória são visíveis para os programadores. Portanto, não é um problema se um dado especulativo permanecer na cache, uma vez que não estaria visível ao programador.

A execução fora de ordem e de forma especulativa foram exploradas pelos ataques. Dois aspectos devem ser destacados. Primeiro: é possível conhecer os princípios do preditor de desvio e executar trechos de códigos para "viciar" o preditor gerando uma grande probabilidade de tomar a decisão por um determinado caminho. Por exemplo, o ataque pode executar vários trechos com um comando condicional que verifica se o índice $i$ para acessar um vetor é menor que o tamanho do vetor, por exemplo $if(i < V.tamanho())$. Se o comando for verdadeiro várias vezes, existe uma grande possibilidade da especulação ficar "viciada" em verdadeiro. Segundo aspecto: se algum valor na expressão condicional não estiver nos registradores e nem memória cache, o comando irá demorar a ser verificado e todo o trecho dentro do $if$ pode ser executado de forma especulativa.

Para acelerar ainda mais a especulação, como nada será gravado nos registradores definitivos durante a execução especulativa, pois os valores são mantidos no *buffer* em registradores temporários, a segurança pode ser relaxada e deixar de verificar as proteções de acesso a memória, pois os testes de segurança serão realizados na efetivação da execução especulativa.

**Exemplo 3:** Execução Especulativa

1: char vetor[] = {1,2,3,4};
2: ...
3: $i = \sqrt{a}$
4: **if** $i < vetor.tamanho()$ **then**
$x = vetor[k]; \; y = vetor[i];$

O trecho de código do Exemplo 3 ilustra um comando condicional. Suponha que a variável $a$ não está na cache e que o comando condicional seja executado especulativamente. O valor de $i$ só será calculado após vários ciclos de relógio. Durante a execução especulativa, se o valor de $k$ em tempo de execução for propositalmente fora dos limites de tamanho alocado para o vetor, leituras em posições indevidas podem ser iniciadas durante a especulação. Porém, o processador não gera uma exceção durante a especulação. Gera apenas quando finaliza em ordem as instruções. Outra situação acontece caso o valor de $a$ estiver nos registradores, porém o tamanho do

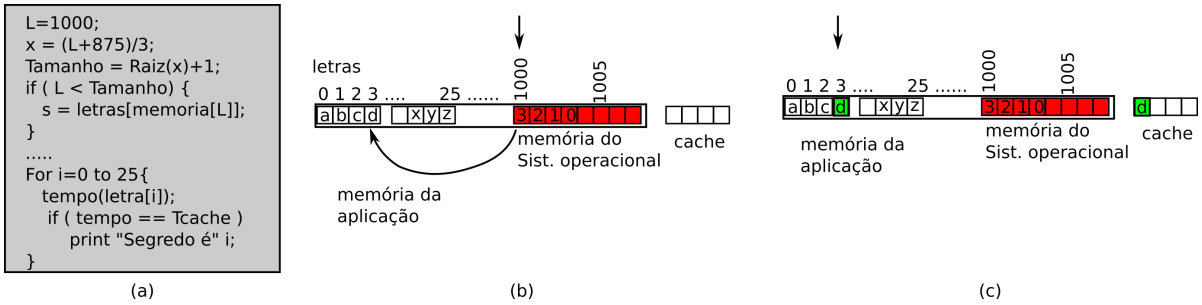


Figura 5. Exemplo de ataque: (a) Trecho de código; (b) Configuração inicial; (c) Após a execução especulativa que deixa um rastro na cache

vetor não está armazenado nos registradores e nem na cache. O teste será executado especulativamente, enquanto o tamanho é buscado na memória. Neste caso, o valor de $i$ também pode violar o tamanho do vetor até que a especulação seja resolvida.

### E. Exemplo de Ataque

Nesta seção é apresentado um exemplo para a explicação de como correlacionar os conceitos e efetuar um ataque. A Figura 5(a) ilustra um trecho de código. A Figura 5(b) ilustra a configuração da memória antes da execução com duas áreas. Para área na cor branca, a aplicação possui acesso aos dados. A outra área, na cor vermelha, é protegida pois pertencente ao SO, onde está armazenado um segredo. O objetivo do exemplo é mostrar como o ataque irá fazer acesso a um dado da área protegida através de um rastro deixado na cache. Na área da aplicação tem-se o vetor *letras* que é usado como instrumento para registrar o acesso à área protegida, através do uso da indireção. O vetor contem todos os caracteres, iniciando com a letra 'a' armazenada na posição 0. A posição 3 contém a letra 'd'. Suponha que o SO armazene a senha "3210" na posição 1000. O ataque irá explorar a execução especulativa e a execução fora-de-ordem. Supondo que o corpo do comando $if$ seja executado especulativamente, devido ao fato da variável $Tamanho$ demorar a ser calculada. Ao executar de modo especulativo o comando $s = letras[memoria[L]]$, como $L = 1000$ e a $memoria[1000]$ tem o valor 3, portanto, o elemento $letra[3]$ será armazenado em cache (letra $d$ para exemplo). Após finalizar o cálculo da variável $Tamanho$, o trecho do comando condicional é anulado, os registradores são restaurados e a execução continua com a sequência do código. Porém o conteúdo da cache não é apagado. A segunda parte do ataque executará com a cache contendo a variável($letra[3] = d$) como ilustrado na Figura 5(c). O comando $For$ irá percorrer o vetor $Letras$. Suponha que a função $Tempo$ meça o tempo de acesso a memória. Para todos os valores, o tempo de acesso será lento pois serão buscados da memória, exceto para $i = 3$ que já está na cache, ou seja. quando o valor de $i$ for igual a 3, o acesso será rápido, portanto o ataque pode inferir que o primeiro número da senha é 3. Ou seja, um efeito colateral gerado pelo rastro na cache, a possibilidade de medir o tempo de acesso, juntamente com a execução especulativa e fora de ordem, pode produzir um ataque. Este exemplo foi bem simplificado para apresentar as estratégias de ataque. Na prática deve-se incluir os mecanismos de memória virtual, a organização em blocos da cache e muitos outros detalhes.

## III. Falhas de Segurança

### A. Histórico

A execução especulativa foi introduzida nos processadores comerciais na década de 80 e, desde 1995, a maioria dos processadores possuem essa funcionalidade. Nesta epóca, foram detectados os primeiros sinais de falhas em *chipsets* da Intel, que permitiam acesso a áreas que deveriam estar protegidas como registradores de ponto flutuante de outros processos, registradores de controle de segmentos, dentre outros [23]. Ou seja, em um cenário de execução concorrente, um processo poderia ter permissão de leitura de conteúdos protegidos de outro processo. Em 2012, a Apple adotou medidas de segurança em seu núcleo do sistema operacional com a introdução do KASLR, (*Kernel Address Space Layout Randomization*). O KASLR é a técnica de segurança para proteger de vulnerabilidades no acesso à memória [15]. Em seguida, o *kernel* do Linux também sofreu essa alteração.

Após um estudo detalhado produzido pela Google, conhecido como *Google Project Zero*, divulgado no dia 2 de janeiro de 2018, graves falhas de segurança dos processadores da Intel foram divulgadas. As falhas também atingem os processadores da AMD e da ARM. Estas informações foram primeiramente reportadas as empresas fabricantes dos processadores em junho de 2017 e só foram publicadas seis meses depois. Pesquisas indicam que todos os sistemas são afetados pela falha de segurança *Spectre*, inclusive os *smartphones* e os servidores em nuvem. Diferente da falha *Meltdown*, que atinge apenas processadores da Intel. Inicialmente, três variantes do problema foram descobertas, sendo as variantes 1 e 2 relativas a falha *Spectre* e a variante 3 relativa a falha *Meltdown* [11].

Mais recentemente, a Microsoft e a Google divulgaram a descoberta das variantes 3a e 4. A variante 3a é similar às que foram encontradas anteriormente. O caso da variante 4, possui uma implementação mais complexa e permite acessar mais posições de memória. Não é objetivo deste artigo detalhar todos os aspectos das falhas. Recomenda-se aos leitores a busca por mais informações no site do *Common Vulnerabilities and Exposures* (CVE) [11]. Cada variante possui uma identificação única no CVE, uma descrição e referências de segurança. A seguir, as identificações referentes a cada variante:

- **Variante 1:** Ignorando as verificações de limites (*bounds check bypass*, CVE-2017-5753);
- **Variante 2:** Injeção destinos de desvios (*branch target injection* CVE-2017-5715);
- **Variante 3:** Fuga de informação em leituras na cache de dados (*Rouge Data Cache Load* CVE-2017-5754);
- **Variante 3a:** Fuga de informação nos Registradores do Sistema (*Rogue System Register Read* CVE-2018-3640);
- **Variante 4:** Violação especulativa de Dados ( *Speculative Store Bypass* CVE-2018-3639).

A diferença básica entre as falhas *Spectre* e *Meltdown* é que no *Spectre* é feita uma exploração de leituras indevidas de dados de outros processos em um mesmo nível de privilégios de acesso, enquanto que no *Meltdown* a exploração é feita na leitura de dados para os quais o processo não possui privilégios para acessar.

### B. Meltdown

Os efeitos colaterais da execução fora de ordem são explorados pelo *Meltdown*, a partir da leitura de locais arbitrários de memória, sem permissão, do núcleo (*kernel*) do sistema operacional ou de outros processos, incluindo dados pessoais e senhas. Os sistemas afetados podem ser também máquinas virtuais na nuvem. O *Meltdown* permite ao usuário ler a memória inteira do *kernel* da máquina de uma forma simples. O mesmo explora as informações dos efeitos colaterais ou "canais laterais"(*side-channel*) dos processadores das últimas duas décadas, como por exemplo os processadores com a microarquiteturas da Intel desde 2010 [19]. Suponha um sistema sem a correção de segurança executando o código em C do Exemplo 4.

**Exemplo 4:** Meltdown.

```
1: char MeusDados[256] = {0};
2: ...
3: segredo = *(char*)1000;
4: char c = MeusDados[segredo];
5: // varredura do vetor MeusDados
```

No Exemplo 4, o vetor *MeusDados* é um vetor do processo no qual não existe violação de acesso. Todos os valores foram inicializados com o valor 0. Suponha também que nenhum elemento do vetor *MeusDados* esteja na cache (linha 2, omitidos para simplificar o exemplo) e que o processo não tenha privilégios de acesso às posições de memória acima do endereço 1000, por ser uma área restrita do sistema operacional (SO). Na linha 3, o processador executará uma instrução de *load* na área restrita ao SO. Mas isto só será confirmado após consulta na tabela de páginas e finalização da execução da instrução no último estágio do pipeline. Com a execução fora de ordem, o processador inicia em paralelo a execução da linha 4, até que o *load* da linha 3 termine de percorrer o pipeline. Ao finalizar a linha 3, uma exceção é gerada e os resultados temporários da execução fora-de-ordem da linha 4 em diante são descartados. Entretanto, se a vetor *MeusDados* não estava na cache, a linha 4 irá gerar uma falha de acesso a cache e o processador disparará especulativamente uma requisição de leitura na memória principal na posição *MeusDados[segredo]*, onde *segredo=Memoria[1000]*. Das linhas 5 em diante, o ataque fará uma varredura do vetor *MeusDados* e medirá o tempo de acesso. Se o elemento $i$ estiver na cache, então $i = segredo$ e assim o valor de uma área restrita é descoberto. O ataque pode ser repetido para as outras posições $1001, 1002, \ldots$ e assim varrer toda a memória protegida do SO.

```
uint8_t* MeusDados = new uint8_t[256 * 4096];
// ... Trecho para retirar MeusDados da cache
uint8_t MemoriaSO = *(uint8\_t*) (EnderecoSO);
uint64_t MemoriaSO_final= MemoriaSO * 4096;
uint8_t teste = MeusDados[MemoriaSO_final];
// ... Detecta falha de pagina
// ... Varredura de MeusDados para ver os trecho em
    cache
```

Figura 6. Exemplo a Vulnerabilidade *Meltdown* com varredura da área do sistema operacional.

A Figura 6 ilustra um segundo exemplo de ataque *Meltdown*, apresentado em [13], semelhante ao anterior porém com mais detalhes sobre a memória virtual. Primeiro, o vetor $MeusDados$ é declarado na área de memória do processo em execução. Depois, o código busca garantir que o vetor não estará armazenado na cache. Uma das formas de fazer isto é usar instruções específicas dos processadores para limpar a cache. O próximo passo do ataque é a leitura de um dado no espaço de endereçamento do SO. Como já mencionado na seção II-B e na Figura 3(b), a memória virtual e as tabelas de páginas dos SO atuais são usualmente mapeadas no espaço virtual do processo. A princípio, cada página possui suas permissões e o processo, em modo usuário, não pode acessar a área do SO, qualquer acesso deste tipo irá gerar uma falha. A terceira linha do exemplo da Figura 6 é um exemplo de tentativa de acesso a área do SO. Entretanto, o processador executa fora-de-ordem as linhas 3,4 e 5, até que as instruções da linha 3 atinjam o final do pipeline e abortem a execução. Na linha 4, um byte de dados (cujo valor varia entre 0 e 255) é multiplicado pelo tamanho da página do sistema, tipicamente 4096. Semelhante ao exemplo anterior, o valor retornado é usado para acessar um elemento do vetor $MeusDados$ (linha 5). A multiplicação por 4096 evita que a CPU execute ações de busca antecipada de dados (*prefetcher*). Apesar da CPU abortar a execução em andamento das linhas 4 e 5, o elemento do vetor $MeusDados$ foi provavelmente armazenado na cache. Semelhante ao trecho de código ilustrado na Figura 5(a) onde o vetor *letras* é percorrido pelo comando *For*, na Figura 6 devemos acrescentar um laço para varrer o vetor $MeusDados$ das posições de 0 à 255 com medição do tempo de acesso a cache. A posição que for detectada na cache corresponde ao valor do byte lido da área do SO na linha 3. Esta técnica possibilita ler toda a memória física do computador, byte por byte. Assim pode-se questionar a serventia dos bits de proteção da tabela de página e como um processo em modo usuário

pode conseguir acesso a uma área restrita do SO. A resposta é simples, isto é um *bug* dos processadores da Intel, ao buscar desempenho. Ao recordar que todo acesso a memória virtual passa primeiro pela TLB (Figura 3), que pode conter os bits de permissão. Porém, os processadores da Intel não armazenam estes bits na TLB. Processadores da AMD e ARM fazem a verificação na TLB e portanto são imunes ao *Meltdown*, exceto um processador da linha ARM.

Uma solução para *Meltdown* é a separação das áreas de endereçamento virtual do SO e do processo, conforme pode ser observado na Figura 7, para correção dos efeitos do compartilhamento do mapeamento conjunto da Figura 3(b). Esta técnica é conhecida com isolamento da tabela de página do núcleo do SO (*Kernel Page Table Isolation (KPTI)* e provê mais segurança, mas implica em uma perda de desempenho ao evitar o compartilhamento.

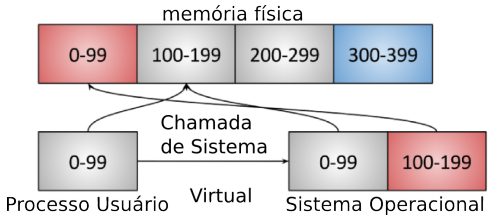


Figura 7. Separar a área virtual do SO e do Processo

### C. Spectre

Enquanto o *Meltdown* (variante 3) explora a execução fora-de-ordem, o *Spectre* (variante 1) [14] explora a execução especulativa. O modo de execução do processador pode até ser *em-ordem*. Os ataques utilizam técnicas *Prime+Probe*, que tentam descobrir um conjunto de blocos da memória [26] que foram buscados para cache $L_1$ [27].

**Exemplo 5:** Spectre - Execução Especulativa

```
1: ...
2: if ( p < TamanhoVetor) {
3:         valor = MeusDados[vetor[p]];
4: }
5: ...
```

O Exemplo 5 ilustra um trecho de código bem simplificado para explicar os princípios do ataque *Spectre*. Na linha 2, o comando condicional verifica se o valor de $p$ viola o tamanho do vetor para garantir que não seja executado um acesso indevido além da área alocada para o vetor. Primeiramente, o ataque (linha 1, omitidos para simplificar o exemplo) executará várias vezes comandos condicionais similares com valores de $p$ dentro da faixa do vetor para o treinamento do preditor de desvios com um grande probabilidade de sempre executar o condicional. O próximo passo do ataque é dar continuidade agora com $p$ violando os limites do vetor, porém executado especulativamente. Além de treinar o preditor, o ataque também supõe que o valor da variável *TamanhoVetor* não esteja na cache. Portanto ao executar a linha 2, o processador irá especular, enquanto busca a variável *TamanhoVetor* na memória. O processador executará, então, especulativamente o comando da linha 3, *valor = MeusDados[vetor[p]]* com acesso malicioso ao *vetor[p]*. Assim, o elemento apontado pelo *vetor[p]* será buscado para cache durante o ataque, enquanto o processador resolve o cálculo da predição que depende da variável *TamanhoVetor*, também sendo lida. Quando o processador verifica a condição que invalida a especulação, o resultado da execução especulativa *valor = MeusDados[vetor[p]]* estará na cache. E, como descrito nos exemplos anteriores, o ataque continua varrendo o vetor *MeusDados* para descobrir qual elemento foi buscado para cache e portanto determinar o valor do segredo *vetor[p]*.

## IV. Estudo de Caso

Nesta seção serão ilustrados casos reais nos quais é possível observar os efeitos das vulnerabilidades. Nas seções anteriores, o objetivo foi a revisão dos principais conceitos de arquitetura para a compreensão do funcionamento dos ataques e como correlacioná-los. Os ataques motivam o ensino da disciplina de arquitetura de computadores com exemplos atuais de problemas com grande impacto nos quais vemos a importância de correlacionar as otimizações e compreender as entrelinhas. Para a verificação das vulnerabilidades, foram executados alguns exemplos apresentados em [14]. As execuções foram realizadas no sistema operacional Linux sem a devida proteção proporcionada pelas correções de segurança (*patches*). 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 viola as barreiras de segurança dos processadores, o que permite o acesso a informações sigilosas. Para que o ataque ocorra de forma controlada, os autores disponibilizaram publicamente [22] alguns exemplos de códigos relativamente simples. Nos códigos, para acessar os dados de um processo, outra aplicação maliciosa é executada em conjunto e explora os recursos de execução especulativa. Portanto, a aplicação que realiza o ataque pode ter acesso a todo endereço de memória da máquina, inclusive os endereços que contém as informações privadas do código exemplo.

A figura 8 apresenta o código em linguagem C para um teste simples para introduzir as estratégias de ataque, pois a recuperação de dados secretos ocorre em um único processo com vetor que se conhece o endereço base. Apesar dos outros exemplos serem mais complexos, a violação exemplificada dentro de um mesmo processo, uma aplicação maliciosa pode acessar endereços de memória reservado a outros programas. O recurso utilizado no exemplo é a biblioteca *libkdump* desenvolvida pela equipe que descobriu as vulnerabilidades. Todas as leituras realizadas através da *libkdump* permitem ao processo ler qualquer parte da memória, o que inclui a área do sistema operacional e a memória física nos sistemas sem proteção em execução nos processadores vulneráveis as falhas.

Este algoritmo possui dados “secretos” armazenados no vetor *strings[]*. O objetivo é escolher uma frase secreta entre as possíveis, de modo aleatório e apresentá-la. Em seguida, o comando *libkdump_read* permite ler qualquer endereço de memória indicado como parâmetro. Deste modo, o programa consegue de maneira indireta reconstruir e retornar exatamente a mesma frase, caractere por caractere como apresentado na Figura 9.

Uma atividade didática é inserir algumas alterações no código para realizar um experimento. Por exemplo, a Figura 10 sugere algumas modificações. Para tal, foi inserida uma nova variável que contém uma cadeia de carácteres "secretos", denominada *segredos[]* e contém a palavra *secrets*. A variável *test* aponta para a primeira posição do vetor *strings[]*. Logo em seguida são impressos o conteúdo apontado pela variável *test*. No trecho especulativo, usa-se ponteiro *test* para acessar uma área fora dos limites do vetor *strings[]*. Ao subtrair 8 posições (7 caracteres mais o caracter 0 para indicar o final da cadeia "secrets") com o auxílio da biblioteca *libkdump*, indiretamente é acessada a área do vetor *segredos[]* através do ponteiro do vetor *strings[]*, como mostrado na Figura 11. Portanto, deste modo é feito o acesso a um endereço fora dos limites de um vetor e os dados pertencentes a outro vetor são resgatados. Após a correção da vulnerabilidade, não é possível o acesso indireto ao vetor *segredos[]*.

Os demais exemplos disponibilizado em [22] também são utilizados para a verificação da efetividade das soluções implementadas pelos *patches*. Após a instalação da correção no sistema, o exemplo de código da Figura 8 obteve acesso a memória uma vez que a execução especulativa ocorre apenas no endereço do processo. Porém, os demais exemplos de ataques avaliados, tais como a obtenção do endereço KASLR, e o acesso ao conteúdo reservado a um processo distinto à aplicação maliciosa, não obtiveram êxito.

## V. Impactos no Desempenho dos Sistemas Operacionais

Nesta seção são apresentadas as ações propostas pelas grandes empresas do setor e uma análise das perdas de desempenho.

### A. Correções de Segurança

As correções ou atualizações de segurança propostas tem um impacto negativo no desempenho [16] e variam conforme a marca/modelo do processador. Cada processador pode ser vulnerável a alguma variante do ataque. A seguir são sumarizados alguns pontos apontados pelas grandes empresas:

***Intel:*** Em trabalho conjunto com outras empresas (AMD, ARM e fornecedores de sistemas operacionais) e começou a fornecer atualizações de *software* e *firmware* para amenizar as consequências das vulnerabilidades. Afirma que, ao contrário de alguns relatórios, qualquer impacto no desempenho depende da carga de trabalho e configuração da plataforma. Para os usuários de computadores pessoais não há impacto significativo, apenas para servidores de alto desempenho [12].

```
#include "libkdump.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 8. Exemplo simples de execução especulativa.

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

Figura 9. Exemplo da saída do algoritmo.

```
//inserir
const char *segredos[] = {"secrets"};

//alterar
const char *test = strings[0];

while (index < strlen(segredos[0])) {
    int value = libkdump_read((size_t)((test-8) +
        index));
```

Figura 10. Alterações para causar violação no acesso à memória.

```
Expect: Generating witty test message...
   Got: secrets
```

Figura 11. Saída com violação de memória

***AMD:*** Uma das variantes do *Spectre* pode ser corrigida por software com um impacto insignificante. O risco do *Meltdown* é pequeno, pois seus processadores não são vulneráveis [1].

***ARM:*** Os núcleos Cortex-M, geralmente empregados em soluções da IoT, não são vulneráveis ao *Spectre*. Alguns núcleos Cortex-A, que equipam sistemas da Qualcomm, Samsung e TSMC são vulneráveis. A ARM se comprometeu, a partir de julho de 2018, disponibilizar atualizações [4].

***Microsoft:*** A maior parte da estrutura de nuvem do Azure já foi atualizada. Para a maioria dos clientes, o impacto no desempenho é imperceptível, apenas uma perda de desempe-

nho de rede ocorre em alguns casos [21]. Quanto à linha *Xbox One*, a arquitetura Jaguar do processador de seus consoles é imune a ambas vulnerabilidades [20].

***Linux:*** O sistema conta com uma correção de *kernel* para o Meltdown. O *patch* pode ser aplicado a todos os processadores de 32 bits, que a princípio são inseguros, é sugerido a execução da correção inclusive nos processadores AMD.

***Apple:*** A Apple informou que todos os seus dispositivos podem ser afetados pelo Meltdown e/ou Spectre, exceto o Apple Watch, apesar da não existência de casos de invasão até o momento. Mesmo assim, a companhia liberou *patches* para a variante *Meltdown* nas atualizações dos sistemas operacionais iOS, macOS e tvOS e para a falha *Spectre* para o navegador Safari [3].

***Google:*** Uma correção para *Spectre* foi inserida na linha Nexus/Pixel nos *patches* de segurança recentes. No entanto, as correções para outros dispositivos Android são de responsabilidade dos fabricantes [2]. O navegador *Chrome* v64 incorpora as correções. Chromebooks e outros dispositivos que rodam o Chrome OS versões 3.18 e 4.4 estão com o *Kernel Page Table Isolation* (KPTI) corrigido. Os sistemas mais antigos serão atualizados posteriormente [8].

Outros desenvolvedores tornaram públicos dados referentes às perdas de desempenho causadas pelas estratégias de proteção. As informações iniciais apontavam uma perda de desempenho de 30%. Entretanto após analises mais detalhadas nota-se que o efeito real é consideravelmente inferior. A RedHat divulgou dados referente ao impacto no desempenho [17]. As primeiras estratégias causaram uma perda de desempenho entre 1% e 20%. Seus últimos resultados divulgados apontam para uma perda de desempenho menor na faixa de 1 à 8%. Estas informações foram medidas com um conjunto de *Benchmarks* e podem variar conforme a carga de trabalho.

### *B. Análise da Perda de Desempenho*

Tabela III
IMPACTO NO DESEMPENHO COMPUTACIONAL AFERIDO COM O CONJUNTO DE TESTE LINUX-BENCH.

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

Esta seção apresenta testes para avaliação da perda de desempenho real proporcionado pelas atualizações de seguranças. Os testes foram executados antes e depois da atualização do núcleo do sistema operacional. A máquina utilizada para testes possui processador Intel Core I7-3612QM 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 utilizado foi o Deepin 15.5 que é baseado na distribuição Debian. A versão do núcleo do sistema operacional antes da atualização era versão Linux 4.9.0-deepin12-amd64. Esta versão foi utilizada na coleta de dados de desempenho sem os *patches* de segurança. Em seguida, a versão do núcleo do sistema foi atualizada via interface por meio das atualizações automáticas disponibilizadas pelos desenvolvedores. Entretanto esta atualização pode ser realizada via terminal, a partir do comando: *$sudo apt-get dist-upgrade*. A versão atualizada (Linux 4.14.0-deepin2-amd64) provê os mecanismos de segurança. A verificação se o *patch* está ativo pode ser realizada através do comando *$grep cpu_insecure /proc/cpuinfo && echo "patched| echo "unpatched"*.

As ferramentas GtkPerf [9] e Linux-Bench [18] possuem um conjunto de códigos de testes ( *benchmarks*) utilizados para avaliar o desempenho de processadores e sistemas. Estes conjuntos de teste foram utilizadas para avaliar o desempenho com base na aferição dos tempos de execução. O GtkPerf tem o objetivo de testar o desempenho da parte gráfica de um computador e é composto por série de aplicações gráficas comuns. Os códigos de testes executam tarefas como abrir caixas de diálogo, simular cliques do mouse, navegar por textos e desenhar uma série de imagens de diversas cores e formatos na tela. Durante a execução destes testes, um medidor interno, já inserido no código, é responsável por mensurar o tempo de execução. Os resultados do GtkPerf exibem o tempo total gasto para a execução de todas as tarefas e também o tempo para cada atividade separadamente. Por outro lado, o Linux-Bench é um *script* simples o qual fornece um nível básico de funcionalidades para a execução de programas de teste. Seu foco principal é o desempenho do processador, com a execução de algoritmos de criptografia, operações de ponto flutuante e aplicações de vídeo além do algoritmo de solução do problema n-rainhas. O tempo gasto na execução é aferido com base em instrumentação do próprio código dos testes e apresentados ao final de cada execução.

Com o objetivo de apresentar dados referentes ao tempo de execução, foram coletadas 10 amostras e calculada a média. Os resultados referentes as análises de desempenho medidas para o *Linux-Bench* são apresentados na Tabela III, onde a primeira coluna expõe os testes realizados e as duas colunas seguintes apresentam o tempo de execução (em segundos) gasto por cada aplicação com e sem a adição dos mecanismos de proteção, respectivamente. A quarta coluna calcula o impacto na perda de desempenho do sistema, ou seja, o aumento no tempo de execução após a correção do sistema contra as vulnerabilidades.

Para o conjunto de teste apresentado na Tabela III, a maior perda de desempenho foi em torno de 12.5% para o *FPU FFT*. Para avaliação do tempo total da execução de todos os métodos do pacote *Linux-Bench*, com e sem proteção à falha, obteve-se uma perda de desempenho um pouco superior a 10% para o sistema protegido.

Para o conjunto de testes fornecidos pela ferramenta GtkPerf ( Tabela IV), os resultados foram semelhantes aos apresentados anteriormente. Mesmo que os testes tenham apresentado uma variação maior. Na média, o tempo total de execução teve uma perda de desempenho próxima a 16%.

Tabela IV
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-Circ | 0,79 | 0,83 | 5,06 |
| DrawArea-Pixb | 0,07 | 0,07 | 0,00 |
| **Tempo total** | **2,98** | **3,46** | **16,107** |

## VI. CONCLUSÃO

As duas novas vulnerabilidades *Meltdown* e *Spectre* descobertas recentemente levantaram um ponto importante onde a busca por desempenho pode ter efeitos colaterais e um alto custo para repará-la, podendo até ser necessário substituir todos os processadores produzidos nos últimos vinte anos. Este trabalho mostra a importância de ensinar várias otimizações de arquitetura de computadores e correlacioná-las para mostrar os perigos e as vulnerabilidades que elas podem gerar quando exploradas de forma maliciosa. O ensino não pode isolar cada ponto (cache, memória virtual, execução fora-de-ordem), deve apresentar os temas e explorar exemplos como as vulnerabilidades descobertas para motivar os alunos a compreensão dos tópicos e suas relações para buscar soluções que sejam robustas aos efeitos colaterais de otimizações combinadas. O trabalho é um texto introdutório apontando referências para aprofundamento no assunto.

Nos testes efetuados neste trabalho com e sem os *patches* de correção, com o uso das ferramentas Linux-Bench e GtkPerf, foram verificadas perdas de desempenho em torno de 10% e 16% respectivamente. Finalmente, se estas ou futuras vulnerabilidades forem intrínsecas do hardware, apenas a substituição dos processadores poderá prover segurança.

Novas variantes derivadas das falhas *Meltdown* e *Spectre* explorando recursos de redes, retorno de chamadas, instruções escondidas estão sendo divulgadas e abrem espaço para trabalhos futuros de como correlacionar com o ensino de outros conceitos de arquitetura computadores.

## VII. AGRADECIMENTOS

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001. Esta pesquisa também foi suportada pelo CNPq, Fapemig, Funarbe, Nvidia and Intel.

## REFERÊNCIAS

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

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

[3] 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.

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

[5] H. P. Baranda, J. C. Penha, and R. Ferreira. Implementacao de um preditor de desvio no mips 5 estagios. *International Journal of Computer Architecture Education*, 6, 2018.

[6] 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.

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

[8] 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.

[9] GtkPerf. Gtkperf benchmark. Disponível em: http://gtkperf.sourceforge.net/. Acessado em: 02/08/2018, 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] 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.

[13] M. Klein. Meltdown and spectre, explained, 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] Linux-Bench. Linux-bench benchmark. Disponível em: http://linux-bench.com/about/. Acessado em: 02/08/2018, 2018.

[19] 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.

[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] I. of Applied Information Processing and Communications. Meltdown proof-of-concept, 2018.

[23] 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.

[24] J. E. Smith. Dynamic instruction scheduling and the astronautics zs-1. *Computer*, 22(7):21–35, July 1989.

[25] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, Jan 1967.

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

[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.