

Avaliação do Overhead de Desempenho Introduzido pelo KPTI no Kernel Linux

João Vitor Belmonte Rates¹, João V. F. Lima¹

¹Universidade Federal de Santa Maria,
Centro de Tecnologia, Prédio 07, Anexo B,
Avenida Roraima 1000, Santa Maria, RS, Brasil

jvrates@inf.ufsm.br, joao-vicente.lima@ufsm.br

Abstract. *This paper evaluates the performance overhead introduced by Kernel Page Table Isolation (KPTI), the Linux mitigation for the Meltdown vulnerability. We analyze the speculative-execution and cache-based side-channel mechanisms that enable Meltdown and describe how KPTI mitigates it by separating user and kernel address spaces. Using a modified version of the *slbench* microbenchmark, we measure system-call throughput and TLB behavior with KPTI enabled and disabled across three experiment sets that vary the ratio of user-space work to system calls and the Working Set Size (WSS). Results show that KPTI overhead is directly tied to system-call frequency and to the loss of TLB reuse caused by CR3 switches, reaching up to 17% in syscall-bound scenarios and vanishing as memory-access cost or user-space work comes to dominate execution time.*

Resumo. *Este artigo avalia o overhead de desempenho introduzido pelo Kernel Page Table Isolation (KPTI), mitigação adotada pelo Linux para a vulnerabilidade Meltdown. Analisam-se os mecanismos de execução especulativa e de canal lateral baseado em cache que possibilitam o ataque, bem como o funcionamento do KPTI na separação dos espaços de endereçamento de usuário e kernel. Usando uma versão modificada do microbenchmark *slbench*, mede-se a taxa de chamadas de sistema e o comportamento da TLB com KPTI ativado e desativado, em três conjuntos experimentais que variam a proporção entre trabalho em modo usuário e chamadas de sistema, além do Working Set Size (WSS). Os resultados mostram que o overhead do KPTI está diretamente associado à frequência de chamadas de sistema e à perda de reuso da TLB causada pelas trocas do registrador CR3, atingindo até 17% em cenários syscall-bound e se aproximando de zero quando o custo de acesso à memória ou o trabalho em modo usuário passa a dominar a execução.*

1. Introdução

O desempenho dos processadores modernos apoia-se fortemente em hierarquias de memória cache e execução especulativa [Hennessy and Patterson 2017]. Essas técnicas, contudo, aumentaram significativamente a complexidade das CPUs, abrindo espaço para vulnerabilidades antes restritas ao software, agora observáveis também em hardware – exploráveis, por exemplo, por meio de canais laterais baseados em cache

como o *Flush+Reload* [Yarom and Falkner 2014], capaz de identificar dados acessados por outro processo mesmo sob isolamento por virtualização, e variantes posteriores [Liu et al. 2015, Irazoqui et al. 2016].

Em 2018, a vulnerabilidade Meltdown [Lipp et al. 2018] explorou a execução especulativa fora de ordem para permitir que processos em modo usuário lessem memória do espaço de kernel, vazando o dado por meio de canal lateral de cache. A mitigação adotada pelo Linux, denominada *Kernel Page Table Isolation* (KPTI) [Corbet 2017], separa os espaços de endereçamento de usuário e kernel, eliminando a condição explorada pelo ataque. Essa separação, no entanto, implica trocas adicionais do registrador CR3 e invalidação da TLB a cada transição de modo, reduzindo o desempenho do sistema – em especial em cargas com alta frequência de chamadas de sistema.

Este trabalho analisa o overhead de desempenho causado pelo KPTI em sistemas Linux, isolando experimentalmente, por meio de benchmarks controlados, o custo das transições de modo e da perda de eficiência da TLB. O texto está organizado da seguinte forma: a Seção 2 apresenta a fundamentação teórica; a Seção 3, os trabalhos relacionados; a Seção 4 descreve o Meltdown e o KPTI; a Seção 5, a metodologia experimental; a Seção 6, os resultados obtidos; e a Seção 7 conclui o artigo.

2. Fundamentação Teórica

2.1. Memória Virtual

Memória virtual permite que cada processo opere em um espaço de endereçamento independente, traduzido para endereços físicos por meio de tabelas de páginas hierárquicas mantidas pelo kernel e percorridas a partir do endereço base indicado pelo registrador CR3 – percurso que pode envolver múltiplos acessos à memória até a obtenção do endereço físico ou a geração de uma exceção. Em arquiteturas x86 tradicionais, o mapeamento do kernel é incluído no espaço de endereçamento de todo processo – protegido por bits de privilégio nas entradas da tabela de páginas – justamente para evitar a troca completa de contexto a cada chamada de sistema. Essa decisão de projeto, motivada por desempenho, é a condição explorada pelo Meltdown e a motivação central do KPTI, discutidos na Seção 4.

2.2. Translation Lookaside Buffer (TLB)

A tradução de endereços introduz latência adicional, mitigada pela *Translation Lookaside Buffer* (TLB), um cache de traduções recentes consultado a cada acesso à memória. Um *hit* permite acesso de baixa latência; um *miss* exige um *page walk* completo pela tabela de páginas, de custo significativamente maior. As TLBs modernas organizam-se em níveis – uma L1 separada para dados (DTLB) e instruções (ITLB), e uma estrutura de segundo nível unificada (STLB/Shared TLB), de maior capacidade. Por ser uma estrutura de cache, a TLB exige invalidação sempre que a tabela de páginas ou o espaço de endereçamento mudam: em x86, isso ocorre seletivamente (instrução *invlpg*) ou de forma global, pela atualização do CR3, que descarta as traduções associadas ao espaço de endereçamento anterior. O comportamento da TLB é, portanto, determinante para o desempenho em cargas com baixa localidade espacial ou grande *Working Set Size* (WSS), tema central deste trabalho.

2.3. Canais Laterais como Técnica de Ataque

Ataques de canal lateral extraem ou inferem informações sensíveis observando efeitos colaterais da execução, como variações no tempo de resposta (*timing*) ou consumo de energia [National Institute of Standards and Technology 2009]. Os *cache-based attacks* exploram a diferença de latência entre acessos a dados presentes ou ausentes na cache. O *FLUSH+RELOAD* [Yarom and Falkner 2014], por exemplo, remove uma linha de cache compartilhada (instrução *clflush*) e mede o tempo de acesso subsequente: um acesso rápido indica que a vítima recarregou aquela linha, revelando o endereço por ela acessado – técnica usada pelos autores para extrair chaves privadas do GnuPG mesmo entre máquinas virtuais distintas que compartilham a mesma memória física. No Meltdown, esse mecanismo é o elo entre o acesso especulativo ao dado protegido e sua exfiltração: embora o valor lido nunca se torne arquiteturalmente visível, seu efeito sobre a hierarquia de cache pode ser observado e decodificado por meio de um canal lateral.

3. Trabalhos Relacionados

Diversos trabalhos avaliam o impacto do KPTI, com diferentes níveis de profundidade. [Bowen and Lupo 2020] conduzem uma análise abrangente com 47 testes do Phoronix Test Suite em 8 versões do Ubuntu, cobrindo perfis de banco de dados, compressão e renderização, mas sem investigar os mecanismos responsáveis pelos resultados observados. Com foco em ambientes de alto desempenho, [Prout et al. 2018] avaliam KPTI e Retpolines sob diferentes versões de microcode em cargas de rede, disco e computação intensiva, mostrando maior degradação em workloads com alta frequência de chamadas de sistema – porém também restritos a métricas agregadas. [Behrens et al. 2022] adotam abordagem mais analítica, medindo o número de ciclos associados à instrução *MOV CR3, reg*, mas sem explorar o custo de reconstrução da TLB após a troca de contexto, que compõe grande parte do overhead do KPTI.

Sob perspectiva microarquitetural, [Huerta and Lilja 2021] aplicam o método *Top-Down* e uma normalização por *purchasing power parity* para comparar categorias de *stall* do pipeline sob Spectre/Meltdown, mas utilizam benchmarks de alto nível (SPEC OMP2012), sem isolar TLB ou o custo de transições de modo. [Gregg 2018] utiliza contadores de desempenho (IPC, ciclos de CPU, eventos de TLB) para correlacionar o overhead do KPTI a chamadas de sistema frequentes, mas em nível agregado, sem isolar o custo específico de invalidação e reconstrução da TLB.

De forma geral, os trabalhos analisados medem o overhead de Meltdown/Spectre a partir de benchmarks de alto nível ou workloads representativos, sem isolar de forma controlada o custo direto sobre a TLB, as transições de modo e o padrão de acesso à memória – lacuna que este trabalho busca preencher por meio de uma abordagem experimental controlada.

4. Vulnerabilidade Meltdown e Mitigação KPTI

4.1. Funcionamento do Ataque

O Meltdown [Lipp et al. 2018] permite que um processo em modo usuário leia endereços do espaço de kernel, apesar dos mecanismos convencionais de proteção por nível de privilégio. O ataque explora a interação entre execução especulativa e verificação de permissões: ao processar uma instrução de leitura, o processador pode iniciar operações

dependentes antes de concluir a validação dos privilégios de acesso. Embora os resultados de uma especulação incorreta sejam descartados do estado arquitetural, eles podem alterar estruturas microarquiteturais – em particular a hierarquia de cache – cujo estado permanece observável por meio de canais laterais.

O ataque resume-se a seis etapas: (i) o atacante lê um endereço do espaço de kernel; (ii) antes da verificação de privilégio, o processador executa especulativamente instruções dependentes desse valor; (iii) o valor lido é usado como índice de um vetor controlado pelo atacante; (iv) o acesso ao vetor carrega uma posição específica na cache; (v) a violação de privilégio é detectada e a execução especulativa é descartada; (vi) por meio de um canal lateral baseado em cache, o atacante identifica a posição carregada e reconstrói o valor originalmente lido. A Figura 1 ilustra esse princípio: a leitura do kernel nunca se torna arquiteturalmente visível, mas o acesso ao vetor durante a janela especulativa altera o estado da cache, posteriormente sondado pelo atacante.

```
2
3 try {
4
5     // leitura ilegal de memória privilegiada
6     secret = *(byte *)kernel_address;
7
8     // uso especulativo do valor secreto
9     probe_array[secret * 4096] += 1;
10
11 } catch (Fault e) {
12
13     // após a exceção, recupera usando técnica de
14     // side-channel
15     secret = recover_from_cache(probe_array);
16 }
17
18
```

Figura 1. Representação simplificada do funcionamento do ataque Meltdown.

O ataque só é viável porque, em arquiteturas tradicionais, o kernel permanece mapeado – ainda que protegido – no espaço de endereçamento de todo processo, para evitar a troca completa de contexto a cada chamada de sistema. Assim, a tradução do endereço do kernel é válida do ponto de vista de memória; apenas a permissão de acesso é verificada, e é exatamente essa verificação que a execução especulativa contorna temporariamente. Dessa forma, o Meltdown não viola diretamente os mecanismos arquiteturais de proteção de memória: explora a diferença entre o estado arquitetural, corretamente restaurado após a falha de acesso, e o estado microarquitetural, cujos efeitos permanecem visíveis e podem ser utilizados para inferir dados protegidos.

4.2. Mitigação KPTI

De acordo com [Lipp et al. 2018], as mitigações implementadas por Linux, Windows e macOS seguiram o modelo KAISER [Gruss et al. 2017]; no kernel Linux, a técnica é denominada *Kernel Page Table Isolation* (KPTI) [Corbet 2017]. Sua ideia central é remover o mapeamento do kernel da tabela de páginas usada em modo usuário, criando duas tabelas por processo: uma, ativa em modo usuário, contendo apenas o espaço de endereçamento do processo e um pequeno conjunto de páginas do kernel necessário à transição de modo; outra, ativa em modo kernel, equivalente à estrutura tradicional utilizada antes da mitigação (Figura 2).

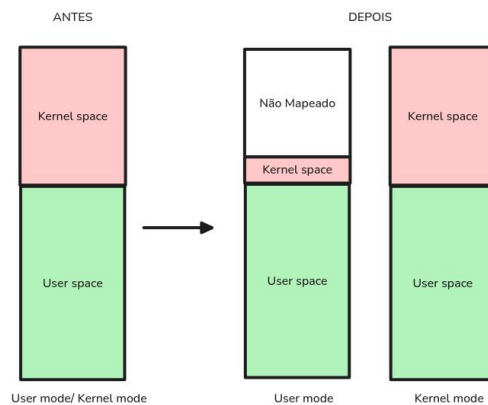


Figura 2. Organização do espaço de endereçamento antes e após a aplicação do KPTI.

A troca entre essas tabelas ocorre por meio da atualização do CR3, o que invalida as entradas da TLB associadas ao espaço de endereçamento anterior. Como essa troca ocorre duas vezes a cada transição de modo (entrada e retorno do kernel), o custo de invalidação – antes restrito a eventos pouco frequentes, como troca de processo – passa a ocorrer a cada chamada de sistema, interrupção ou exceção, elevando a taxa de *misses* e *page walks* em cargas com alta frequência de *syscalls*.

Processadores mais recentes reduzem parte desse custo por meio de *Process-Context Identifiers* (PCID), que associam um identificador ao espaço de endereçamento e permitem preservar entradas da TLB mesmo após a troca do CR3 [Intel Corporation 2026]. A arquitetura utilizada neste trabalho não dispõe de suporte a PCID (Seção 5), tornando o sistema particularmente sensível ao efeito do KPTI sobre a TLB.

5. Metodologia Experimental

5.1. Ambiente de Execução

Os experimentos foram conduzidos em uma máquina dedicada, a fim de minimizar interferências externas. O sistema possui um processador Intel Celeron N2807 (microarquitetura Silvermont), com 2 cores físicos, sem suporte a *Simultaneous Multithreading* (SMT) e sem suporte a PCID, e 4 GB de RAM com paginação de 4 KB. O sistema operacional utilizado foi o Linux 6.8.0-106-generic, compilado com suporte às mitigações da vulnerabilidade Meltdown, incluindo o KPTI. Para reduzir interferência de outros processos, o sistema foi configurado com o parâmetro de inicialização `isolcpus=1`, isolando um núcleo do escalonador. O código do benchmark foi compilado usando o GCC, versão 13.3.0, com a opção de compilação `-O0`.

5.2. Benchmarks

Para avaliar o impacto do KPTI, foi utilizada uma versão modificada do microbenchmark *slbench*¹ [Gregg 2018], originalmente proposto para analisar o custo de chamadas de

¹Disponível em: <https://brendangregg.com/blog/2018-02-09/kpti-kaiser-meltdown-performance.html>. Acesso em: jul. 2025.

sistema em conjunto com acessos à memória. A principal modificação realizada neste trabalho é a possibilidade de executar o benchmark com *Working Set Size* (WSS) igual a zero, permitindo isolar o custo das chamadas de sistema e das transições de modo.

O benchmark é composto por três etapas, ilustradas na Figura 3. Na etapa *SPIN* (3a), uma *thread* executa um laço puramente computacional por um intervalo de tempo fixo, servindo de referência para avaliar a variabilidade do ambiente experimental. Na etapa *POP* (3b), a região de memória do conjunto de trabalho é alocada e integralmente percorrida (um acesso por página), garantindo que todas as páginas estejam residentes antes da coleta de métricas. Na etapa principal *RUN* (3c), o laço alterna entre uma chamada de sistema – `close()`, executada de forma a falhar, preservando o custo da transição de modo sem trabalho adicional – e um conjunto de acessos à memória sobre a região alocada, com incremento definido pelo *stride* configurado, retornando ao início do conjunto de trabalho ao atingir o final da região.

```
1 // flag controlada por outra thread
2 while (g_spin) {
3     count++; // carga computacional de
               referencia
4 }
```

(a) SPIN.

```
1 void populate(char *mem, size_t memsize)
2 {
3     for (char *p = mem;
4         p < mem + memsize;
5         p += PAGE_SIZE) {
6         p[0] = 'A'; // toca cada pagina
7     }
8 }
```

(b) POP.

```
1 while (g_work) {
2     close(999);
3     count++;
4     for (j = 0; j < readmax; j++) {
5         if (memsize) {
6             junk += memp[0]; // acesso a memoria
7             memp += stride;
8             if (memp > mem + memsize)
9                 memp = mem; // reinicia no inicio do WSS
10        }
11    }
12 }
```

(c) RUN.

Figura 3. Código simplificado (em C) das três etapas do benchmark *s1bench*.

Na versão original, a etapa RUN sempre realiza acessos à memória; neste trabalho, adicionou-se a possibilidade de executá-la com WSS e número de acessos iguais a zero, isolando o custo das transições de modo sem interferência do subsistema de memória. A carga de trabalho é controlada por três parâmetros: o tamanho do conjunto de trabalho (WSS), o número de acessos à memória entre chamadas de sistema, e o *stride* dos acessos, que determina a localidade espacial e a pressão sobre a TLB. Dois conjuntos de experimentos foram conduzidos: no primeiro, o WSS foi mantido constante e o número de acessos variado; no segundo, o número de acessos foi mantido constante e o WSS variado em múltiplas ordens de grandeza.

5.3. Métricas de Desempenho

As métricas foram obtidas diretamente da saída do *s1bench*. A principal é a taxa de execução da etapa RUN (iterações por segundo); cada iteração corresponde a uma cha-

mada de sistema seguida de um conjunto de acessos à memória, sendo, portanto, representativa da taxa efetiva de chamadas de sistema sob cada configuração de carga. As taxas das etapas SPIN e POP são usadas como métricas auxiliares de estabilidade do ambiente experimental. Todas as taxas são calculadas pela razão entre o número de iterações executadas e o tempo total de cada etapa, conforme reportado pelo benchmark.

5.4. Procedimento Experimental

Cada ponto experimental corresponde a uma configuração de parâmetros, executada com KPTI ativado e desativado, permitindo comparar diretamente o impacto da mitigação. Foram realizadas 20 execuções do benchmark para cada configuração em todos os conjuntos experimentais. O procedimento previa um mínimo de cinco execuções e a realização de execuções adicionais até que a variação percentual entre as duas maiores taxas de *RUN* observadas fosse inferior a 20%; na prática, o critério foi satisfeito em todas as configurações com 20 execuções. Para as curvas de *throughput*, cada ponto representa a média aritmética das 20 observações individuais da taxa de *RUN*. As barras de erro representam intervalos de confiança de 95%.

Para reduzir o ruído experimental, um núcleo foi isolado do escalonador (*isolcpus*) e a execução foi fixada nesse núcleo por meio do utilitário *taskset*; a etapa SPIN serviu como mecanismo auxiliar de validação da estabilidade do sistema, e o utilitário *perf* foi utilizado para coletar eventos de baixo nível (ciclos de CPU, IPC, acessos à TLB, *page walks*), exclusivamente para validação e interpretação dos resultados.

Os experimentos foram organizados em três conjuntos, resumidos na Tabela 1.²

Tabela 1. Configuração dos conjuntos experimentais.

Parâmetro	Conjunto 1	Conjunto 2	Conjunto 3
WSS	0	2 MB	Variável (0 – 128 MB)
Acessos por iteração	Variável	Variável	650
Stride	4 KB (irrelevante)	4 KB	4 KB
Tempo SPIN	300 ms	300 ms	300 ms
Tempo RUN	3000 ms	3000 ms	3000 ms

A arquitetura utilizada possui uma DTLB de primeiro nível com 32 entradas e uma STLB de segundo nível com 128 entradas para páginas de 4 KB; conjuntos de trabalho acima de aproximadamente 128 páginas (512 KB) já excedem essa capacidade combinada, conforme a Tabela 2. Consequentemente, espera-se aumento progressivo de *TLB misses* e *page walks* à medida que o WSS cresce.

No Conjunto 1, o WSS é fixado em 0, eliminando acessos à memória durante a fase RUN e isolando o custo das transições entre modo usuário e kernel. No Conjunto 2, o WSS é fixado em 2 MB e o número de acessos por iteração varia de 0 a aproximadamente 200.000, permitindo analisar a interação entre o custo de transição de modo e o custo

²Valores completos dos parâmetros variáveis. Conjuntos 1 e 2 (acessos por iteração): 0, 1, 8, 16, 32, 64, 128, 256, 384, 512, 768, 1024, 1536, 2048, 3072, 4096, 6144, 8192, 12288, 16384, 24576, 32768, 49152, 65536, 98304, 131072, 196608. Conjunto 3 (WSS em bytes): 0, 4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288, 1048576, 2097152, 4194304, 8388608, 16777216, 33554432, 67108864, 134217728.

Tabela 2. Relação entre WSS e número de páginas.

WSS	Número de páginas (4 KB)
4 KB	1
128 KB	32
512 KB	128
1 MB	256
2 MB	512
128 MB	32768

de acesso à memória. No Conjunto 3, o número de acessos é fixado em 650 e o WSS varia de 0 a aproximadamente 128 MB, com *stride* de 4 KB (uma página por acesso), maximizando a pressão sobre a TLB e o mecanismo de *page walk*.

Por fim, o impacto do KPTI é quantificado pelo overhead relativo entre as execuções com a mitigação desativada e ativada (Equação 1), em que R_{no_pti} e R_{pti} representam o throughput obtido com KPTI desativado e ativado, respectivamente. Valores positivos indicam perda de desempenho; valores próximos de zero indicam impacto reduzido, independentemente do throughput absoluto de cada execução.

$$\text{Overhead} = \frac{R_{no_pti} - R_{pti}}{R_{no_pti}} \times 100 \quad (1)$$

6. Resultados Experimentais

Os resultados a seguir são analisados tanto em termos de throughput absoluto (taxa de chamadas de sistema) quanto do overhead relativo do KPTI (Equação 1), o que permite distinguir a queda de desempenho causada pela mitigação daquela decorrente da variação dos parâmetros do benchmark. Os resultados são organizados nos três conjuntos experimentais descritos na Seção 5.

6.1. Baseline de chamadas de sistema (WSS = 0)

No primeiro experimento, o WSS é definido como 0 e o laço principal executa apenas chamadas de sistema consecutivas. Variando o número de iterações computacionais em espaço de usuário entre cada chamada de sistema, altera-se a frequência das transições para o modo kernel, permitindo isolar o impacto direto do KPTI. A Figura 4 mostra os resultados.

Conforme a Figura 4a, a taxa de *syscalls*/segundo é máxima para poucas iterações em *userspace* e cai progressivamente à medida que esse trabalho aumenta; entre 10^1 e 10^2 iterações, as curvas com e sem KPTI convergem e permanecem semelhantes a partir deste ponto. O gráfico de overhead (Figura 4b) evidencia esse efeito com mais clareza: com poucas instruções por iteração (e, portanto, alta taxa de chamadas de sistema), o overhead é significativo, atingindo cerca de 17% em um cenário dominado quase exclusivamente por *syscalls*. Esse comportamento decorre do custo adicional que o KPTI impõe a cada transição de modo: a troca do CR3 invalida as entradas da TLB, exigindo sua reconstrução em acessos subsequentes e aumentando o número de ciclos gastos em tradução de endereços.

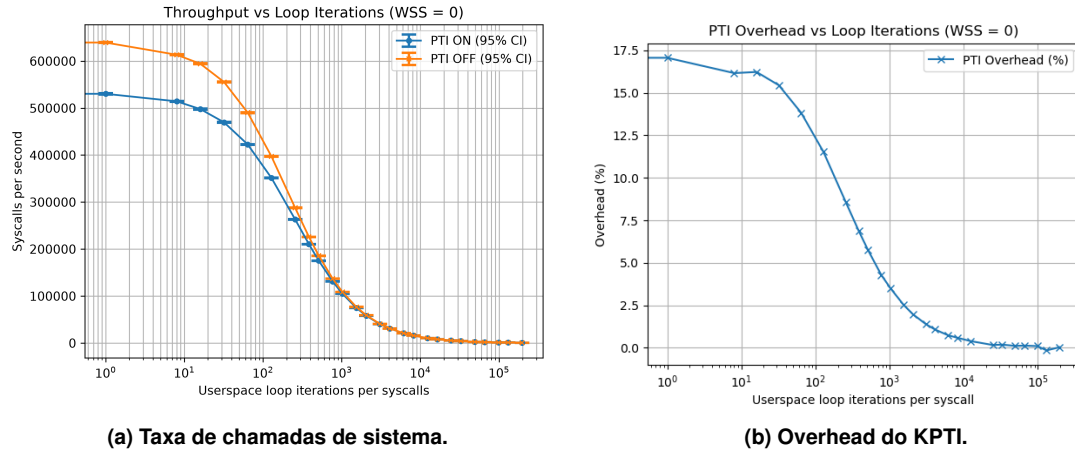


Figura 4. Resultados experimentais em função das iterações do laço de usuário com WSS = 0.

Como esse custo é essencialmente fixo por chamada de sistema, ele é progressivamente amortizado pelo maior volume de trabalho em *userspace*: a partir de aproximadamente 100 instruções por iteração observa-se redução acentuada do overhead, que se aproxima de 0% acima de 10.000 instruções, quando o tempo de execução passa a ser dominado pelo processamento em modo usuário. Esses resultados confirmam que o impacto do KPTI é fortemente dependente da frequência de chamadas de sistema, sendo mais pronunciado em cenários *syscall-bound*.

6.2. Interação entre chamadas de sistema e memória (WSS = 2MB)

No segundo conjunto, o WSS é fixado em 2 MB, de modo que cada iteração do laço principal realiza acessos sequenciais à memória com *stride* de 4 KB (uma página distinta por acesso), impondo pressão direta sobre a TLB. A Figura 5 mostra os resultados.

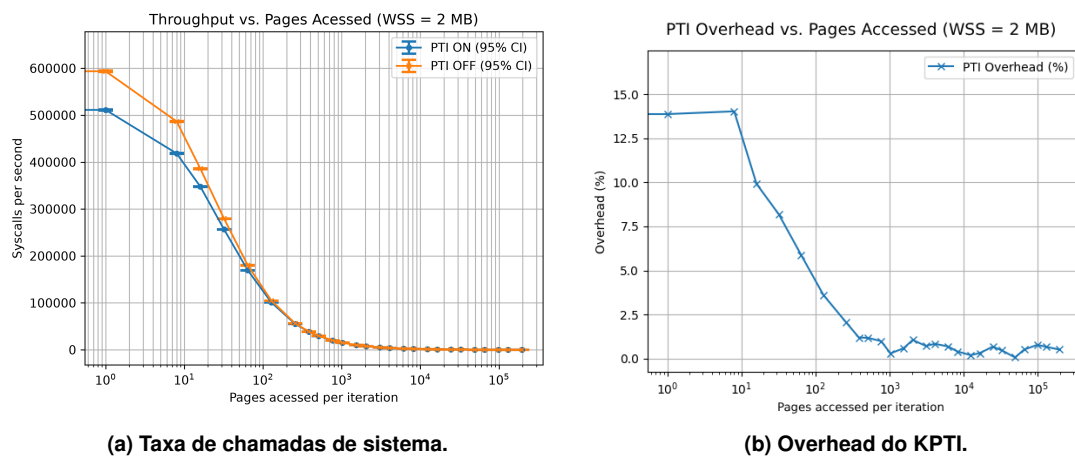


Figura 5. Resultados experimentais em função das iterações do laço de usuário com WSS = 2MB.

O *throughput* (Figura 5a) cai de forma mais acentuada já nas primeiras iterações do que no caso WSS = 0, e a convergência entre as configurações com e sem KPTI ocorre

mais cedo, a partir de aproximadamente 10^2 acessos por iteração – reflexo de *misses* reais de TLB introduzidos pelos próprios acessos à memória, e não apenas pelo custo de transição de modo. O overhead inicial (Figura 5b) é da ordem de 14%, mas cai rapidamente: a partir de aproximadamente 16 acessos por iteração (16 páginas distintas), a configuração sem KPTI já apresenta *misses* de TLB, e o desempenho passa a ser dominado pela tradução de endereços, aproximando o overhead de zero independentemente do KPTI.

A Figura 6 confirma essa leitura por meio do percentual de ciclos em atividade da DTLB, coletado via *perf*: mesmo com poucos acessos, a configuração com KPTI apresenta percentual significativamente maior, resultado das invalidações da TLB a cada transição de modo, enquanto, sem KPTI, o mapeamento do kernel permanece estável, preservando as entradas da TLB. À medida que o número de acessos cresce, o percentual de ciclos em DTLB aumenta em ambas as configurações, convergindo quando o regime passa a ser dominado por *misses* reais em vez do custo de invalidação do KPTI.

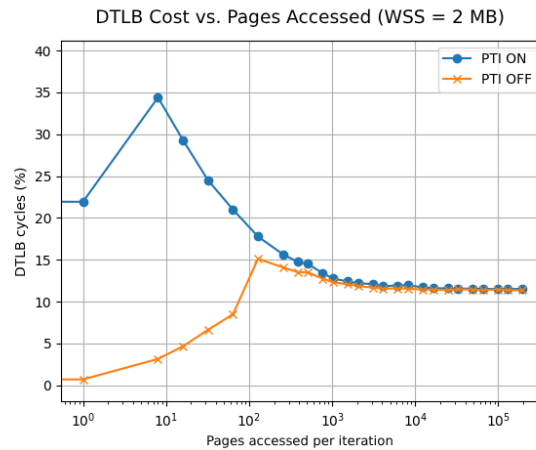


Figura 6. Custo da DTLB em ciclos (%) em função das páginas acessadas por iteração (WSS = 2 MB).

6.3. Impacto do WSS e TLB (WSS variável)

No terceiro conjunto, o número de instruções de *userspace* por iteração é fixado em 650 e o WSS varia de 0 a 128 MB, com *stride* de 4 KB (cada acesso toca uma página distinta), permitindo observar diretamente o efeito da capacidade da TLB sobre o desempenho, com e sem KPTI. A Figura 7 apresenta a frequência de *page walks* por instrução (7a) e o percentual de ciclos em atividade da DTLB (7b) em função do número de páginas do conjunto de trabalho.

Três regimes são observados. Para WSS reduzido, a taxa de *page walks* é baixa – a maior parte das traduções é resolvida pela TLB –, com pequenas oscilações (e.g., em torno de 4 páginas) atribuídas à baixa reutilização e à maior sensibilidade à latência em regimes de baixo paralelismo de acessos, e não à capacidade da TLB. Por volta de 256 páginas (aproximadamente 1 MB, equivalente à soma das capacidades da DTLB de 32 e da STLBI de 128 entradas para páginas de 4 KB [Intel Corporation 2016]), a frequência de *page walks* cresce abruptamente: o número de páginas acessadas excede a capacidade

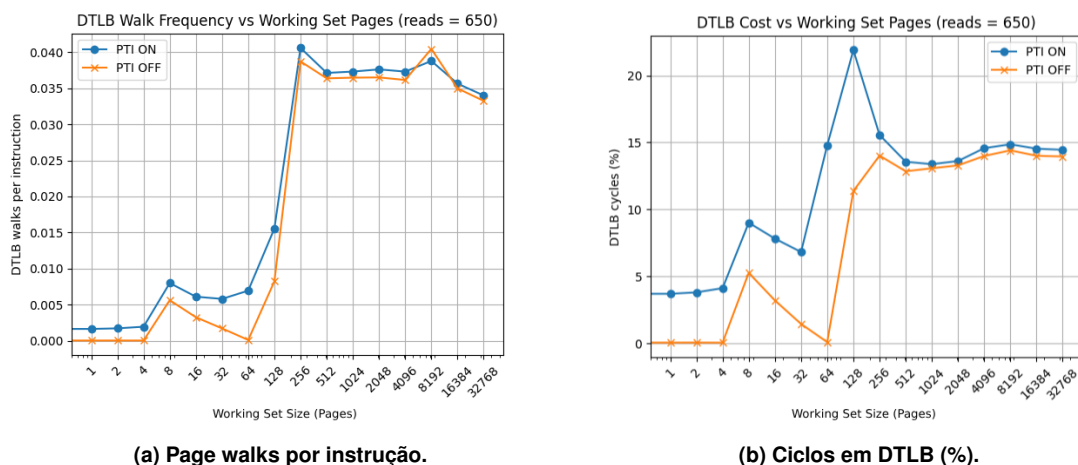


Figura 7. Comportamento da TLB em função do número de páginas do conjunto de trabalho (reads = 650).

de tradução e o paralelismo de *misses* simultâneos suportado pela microarquitetura, e a latência dos *page walks* deixa de ser amortizada. Para WSS elevado, a curva se estabiliza em um regime dominado por *misses* de TLB, no qual as diferenças entre KPTI ativo e inativo tornam-se pouco significativas, pois praticamente todo acesso requer tradução via *page walk*.

O percentual de ciclos em DTLB (Figura 7b) reforça essa leitura: na região inicial, a configuração sem KPTI apresenta consumo praticamente nulo – alta reutilização das traduções –, enquanto com KPTI o percentual já é significativo desde o início, devido às invalidações recorrentes a cada transição de modo. Na região intermediária (abaixo de 256 páginas), a configuração sem KPTI ainda apresenta baixa atividade de DTLB, mas com KPTI o custo já cresce de forma acentuada, evidenciando que a mitigação antecipa o impacto da tradução de endereços mesmo antes de a capacidade da TLB ser plenamente excedida. Acima de aproximadamente 1 MB, as duas configurações convergem, passando a ser dominadas por *misses* reais de TLB independentemente do KPTI.

Em síntese, observam-se três regimes de execução: um regime inicial, dominado pelo custo de transição de modo, no qual o impacto relativo do KPTI é maior; um regime intermediário, de interação entre invalidação da TLB e *misses* reais, no qual a diferença absoluta de desempenho é mais pronunciada; e um regime dominado por memória, no qual o overhead do KPTI torna-se pouco relevante frente ao custo inerente de tradução de endereços. Esse padrão é consistente com a organização em múltiplos níveis de TLB e com o paralelismo limitado de acessos à memória da microarquitetura Silvermont, sugerindo que aplicações *syscall-bound* sofrem maior impacto do KPTI, enquanto aplicações *memory-bound* tendem a ser menos afetadas.

7. Considerações Finais

Este artigo analisou a vulnerabilidade Meltdown e a mitigação KPTI no kernel Linux, avaliando experimentalmente seu impacto de desempenho por meio de benchmarks controlados que isolam a taxa de chamadas de sistema, o acesso à memória e o tamanho do conjunto de trabalho (WSS). Os resultados mostram que o overhead do KPTI é proporcional à taxa de chamadas de sistema – uma maior proporção de trabalho em modo usuário

implica menor penalidade de desempenho – e que, ao variar o WSS com número fixo de acessos, o principal fator determinante do overhead passa a ser o custo adicional imposto sobre a TLB pelas invalidações e reconstruções decorrentes da troca do CR3 a cada transição de modo. Em cenários dominados por acesso à memória, esse impacto relativo se reduz, uma vez que o custo de tradução de endereços passa a ser dominado por *misses* reais de TLB, independentemente da presença do KPTI.

Os intervalos de confiança de 95% apresentados nos gráficos de *throughput* quantificam a incerteza da média amostral observada nas 20 execuções de cada configuração. Sua interpretação, contudo, deve considerar as limitações do procedimento experimental. As execuções foram realizadas sequencialmente na mesma máquina e não em sessões experimentais independentes; portanto, não se pode descartar dependência temporal entre observações ou efeitos sistemáticos compartilhados, e os intervalos não capturam necessariamente toda a variabilidade entre sessões, sistemas ou plataformas. Além disso, a avaliação utiliza um único processador Intel Celeron N2807, da microarquitetura Silvermont, sem suporte a PCID, e cargas sintéticas baseadas no *slbench* modificado. Assim, os resultados caracterizam o comportamento observado nessa plataforma e nessas configurações controladas, mas não devem ser generalizados diretamente para outras microarquiteturas, processadores com PCID ou aplicações reais sem avaliação adicional.

Como trabalhos futuros, sugere-se estender a análise a outras arquiteturas de processadores – em especial com suporte a PCID – e a outras técnicas de mitigação de vulnerabilidades de execução especulativa, bem como avaliar cargas de trabalho reais em ambientes de produção.

Referências

- Behrens, J., Belay, A., and Kaashoek, M. F. (2022). Performance evolution of mitigating transient execution attacks. In *Anais do Seventeenth European Conference on Computer Systems (EuroSys '22)*, pages 251–265, New York, NY, USA. Association for Computing Machinery.
- Bowen, L. and Lupo, C. (2020). The performance cost of software-based security mitigations. In *Anais do ACM/SPEC International Conference on Performance Engineering (ICPE '20)*, pages 210–217, New York, NY, USA. Association for Computing Machinery.
- Corbet, J. (2017). The current state of kernel page-table isolation. Publicado em: 20 dez. 2017. Acesso em: 27 abr. 2026.
- Gregg, B. (2018). Kpti/kaiser meltdown initial performance regressions. Acesso em: 26 abr. 2026.
- Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., and Mangard, S. (2017). Kaslr is dead: Long live kaslr. In *Anais do International Symposium on Engineering Secure Software and Systems*, pages 161–176. Springer.
- Hennessy, J. L. and Patterson, D. A. (2017). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Boston, 6 edition.
- Huerta, Y. A. and Lilja, D. J. (2021). Revisiting the effects of the spectre and meltdown patches using the top-down microarchitectural method and purchasing power

parity theory. *BenchCouncil Transactions on Benchmarks, Standards and Evaluations*, 1(1):100011.

Intel Corporation (2016). *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation, Santa Clara, CA. Section: Intel Atom Microarchitecture (Silvermont/Goldmont).

Intel Corporation (2026). *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, Santa Clara, CA.

Irazoqui, G., Inci, M. S., Eisenbarth, T., and Sunar, B. (2016). Flush+flush: A fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer.

Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. (2018). Meltdown: Reading kernel memory from user space. In *Anais do 27th USENIX Security Symposium (USENIX Security '18)*, pages 973–990. USENIX Association.

Liu, F., Yarom, Y., Ge, Q., Heiser, G., and Lee, R. B. (2015). Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy (SP)*, pages 605–622. IEEE.

National Institute of Standards and Technology (2009). Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. Physical Security Testing Workshop. Acesso em: 27 abr. 2026.

Prout, A., Arcand, W., Bestor, D., Bergeron, B., Byun, C., Gadepally, V., Houle, M., Hubbell, M., Jones, M., Klein, A., Michaleas, P., Milechin, L., Mullen, J., Rosa, A., Samsi, S., Yee, C., Reuther, A., and Kepner, J. (2018). Measuring the impact of spectre and meltdown. In *Anais do 2018 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–5, Waltham, MA, USA. IEEE.

Yarom, Y. and Falkner, K. (2014). Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *Anais do 23rd USENIX Security Symposium (USENIX Security '14)*, pages 719–732. USENIX Association.