Holistic Defenses Against Microarchitectural Attacks

Baris Kasikci and Kevin Loughlin

University of Michigan, Ann Arbor, MI, USA

ABSTRACT

Execution context isolation is a key security requirement in personal computers, edge devices, and more importantly on multitenant computing environments such as the Cloud. It is vital that data belonging to one context (e.g., a process, enclave, or virtual machine) cannot be accessed or modified by another context without explicit permission, particularly in consideration of a remote adversary. However, the level of context isolation provided by today's systems is not well aligned with the security needs of personal users, cloud providers, and customers of cloud computing services alike. Current hardware enforces isolation at the architectural level. However recent high-profile attacks demonstrated that isolation guarantees are weak at the microarchitectural-level. To make matters worse, a lot of the defenses against microarchitectural defenses aim to protect specific side-channels rather than providing a more comprehensive solution. In this paper we describe our recent and ongoing efforts in providing holistic defenses against microarchitectural attacks.

Keywords: Security, Microarchitectural attacks

1. INTRODUCTION

Execution context isolation lies at the heart of most security policies. Mutually distrusting parties rely on the context isolation provided by the operating system and ultimately the hardware to protect the confidentiality and the integrity of their data. Unfortunately the guarantees provided by today's hardware fall short of the requirements of users.

Namely, current hardware only strongly enforces context isolation at the architectural (i.e., direct access) level, failing to prevent cross-context interactions at the microarchitectural (i.e., indirect access) level.[1–3] For instance, while architectural page table permission bits prevent software from explicitly bypassing hardware permission checks, attackers can still read or corrupt data via implicit microarchitectural behavior. Indeed, speculative execution attacks (e.g.,[4–6]) demonstrate that remote adversaries can transiently bypass architectural context isolation to leak arbitrary data from another context.

Fundamentally, microarchitectural exploits stem from a widespread problem in today's systems: commodity microarchitectures do not pervasively enforce context isolation (e.g., process-to-process). Such microarchitectures have arisen due to outdated threat models and corresponding legacy designs that form the basis of modern hardware, with microarchitectural context isolation coming as an afterthought.

Given the ongoing deluge of exploits, it is crucial to redesign edge devices, personal computers, and cloud systems with microarchitectural context isolation as a first-order principle. However, cloud providers, architects, and the systems community as a whole lack a thorough understanding of the various trade-offs (e.g., security-performance) of providing pervasive micro-architectural context isolation. The state-of-the-art approximation—MI6[7]—only considers a small portion of the design space.

In particular, the state-of-the-art only examines the ramifications of a simple flush-on-context-switch policy for select core-local microarchitecture state—i.e., the TLBs, L1 cache, and branch prediction unit (BPU)—as well as a coarse-grained set-partitioning algorithm for lower-level shared caches. Such policies incur significant performance overhead (16.4% on SPEC CPU 2006,[8] without evaluating all sources of overhead), and alternative policies (e.g., save-and-restore on context switches) are not explored.

Further author information: (Send correspondence to Baris Kasikci)
Baris Kasikci: E-mail: barisk@umich.edu, Telephone: 1 734-763-1560

More concerningly, the state of the art suffers from a major security problem. While the state of the art achieves *temporal* isolation (i.e., time-sliced resource access) for contexts executing sequentially on the same physical core, it fails to provide *spatial* isolation (i.e., resource partitioning) for contexts executing simultaneously on the same core such as hyperthreads.

Therefore, we are designing defenses that pervasively enforce microarchitectural context isolation via various policies and mechanisms, in order to evaluate the trade-offs of such isolation for cloud providers and architects alike.

In particular, our microarchitecture design prevents one security context from inferring or deterministically influencing the microarchitectural state of another without explicit permission. To achieve this goal, the processor design employs a combination of microarchitectural save-and-restore (i.e., swapping microarchitectural state to DRAM for absolute temporal isolation), dynamic partitioning (i.e., tagging microarchitectural state with a context ID for absolute spatial isolation), and per-context dynamic randomized mappings (i.e., periodic re-indexing of data for probabilistic isolation, in instances where performance and area constraints justify a relaxation in absolute isolation). We will optimize the performance of this processor using efficient microarchitectural information flow control mechanisms introduced in our recent work.[9,10] Furthermore, our proposed design will allow an administrator (e.g., the cloud provider) to tune these isolation mechanisms according to specific customer needs (e.g., performance).

2. PROCESSOR ISOLATION

The primary security goals of our processor design are to ensure that an adversary (1) cannot infer data from another execution context without explicit permission, and (2) cannot deterministically influence another context's execution—e.g., by controlling which instructions another context will execute.

We assume that the adversary does not have physical access to the machine. While adversaries with physical access are viable threats, remote adversaries are far more numerous and may access a vulnerable machine with relative ease, therefore posing a greater threat to cloud security.

We further assume that execution contexts are isolated at the architectural level. That is, we assume that *non-speculative* data flows (i.e., those explicitly specified by instruction set architecture semantics) are constrained by system software or architectural isolation polices and associated mechanisms. We therefore focus on preventing cross-context information flows in the microarchitecture arising from *speculative execution*.

2.1 Background: Speculative Execution Attacks

As a performance optimization, modern processors implement speculative execution in order to execute forthcoming instructions before prior instructions complete. For instance, the processor may use microarchitectural state to predict the next program counter, subsequently fetching and executing instructions from the predicted address.

In the vast majority of scenarios, the processor speculates correctly, and no context isolation violations occur. However, in certain instances, the processor mispredicts which instructions should be executed according to instruction set architecture semantics, giving rise to two potential violations of context isolation. First, the source—and therefore, effects—of a misprediction within a victim context may be induced by an attacker context. For example, modern microarchitectures do not isolate the BPU among different contexts, allowing an attacker to inject a branch target that the victim will use during speculative execution (thereby speculatively executing instructions at an attacker-chosen address). Second, the wrong-path (i.e., mis-speculated) instructions themselves may access data from another security context (e.g., a user process may read kernel data, whether intentionally or inadvertently). Thus, even though the processor does not commit wrong-path execution to architectural state, the processor's implementation of speculative execution can still violate context isolation at the microarchitectural level.

These violations becomes particularly problematic when values used during wrong-path execution can be *transmitted* through a microarchitectural side channel. Notably, numerous classes of side channels exist (e.g.,[11–13]). However, we focus on timing side channels, as they are the only class known to be remotely-exploitable (i.e., not requiring physical access to the machine).

Listing 1. Pseudocode for Meltdown.[4] A userspace attacker exploits speculative execution to access and transmit kernel data.

```
// assume array is flushed from cache
// speculatively access secret (faults)
secret_byte = *kernel_addr;
// transmit by caching dependent element
tmp = array[secret_byte * 512];
...
// later in code, after fault recovery
// infer secret via cached index
for (guess = 0; guess < 256; guess++) {
        start = timer();
        tmp = array[guess * 512];
        times[guess] = timer() - start;
}
secret = get_min_index(times);
```

In a speculative timing side channel exploit, an instruction uses a secret value (e.g., data speculatively-accessed from a kernel address) as an operand to a wrong-path instruction, wherein the wrong-path instruction yields a secret-dependent update to microarchitectural state. In particular, this update alters processor timing such that an adversary may later infer the secret value.

For clarity, a simplified example of a speculative timing side channel exploit (Meltdown,[4] disclosed by Google Project Zero) is shown in Listing 1. The attacker speculatively reads from a kernel address present in the L1 cache (line 3). Because the data access occurs speculatively (i.e., before the architectural permission check is processed), subsequent dependent instructions are able to cache a secret-dependent element from a (presumed to be pre-flushed) array (line 5). The attacker later infers the secret value by timing the accesses to each element of this array (lines 9-12), for which the (now-cached) secret-dependent value will yield the shortest access time (line 14), ultimately revealing the value of the secret.

2.2 Processor Design

To combat speculative timing side channels, our processor design prevents an adversary in one execution context from using or deterministically affecting the microarchitectural state of a victim context. Our processor design strongly isolates each microarchitectural context via a combination of microarchitectural save-and-restore (for absolute temporal isolation), dynamic partitioning (for absolute spatial isolation), and randomization (for probabilistic isolation in instances where performance and area constraints justify a relaxation in absolute isolation). Our design improves the performance of this processor—without loss of security—by incorporating efficient microarchitectural information flow control mechanisms introduced in our recent work.[9,10] Finally, our design allows the cloud provider to tune the use of these mechanisms according to customer constraints (e.g., performance).

Absolute Temporal Isolation via Microarchitectural Save-and-Restore. Microarchitectural save-and-restore ensures that two subsequently-executing contexts cannot access or modify each other's core-local state (e.g., TLBs, private caches, and the BPU). More specifically, save-and-restore exploits the ever-improving memory speeds and capacities of modern systems to mirror time-slicing isolation mechanisms used at the architectural level. Upon a context switch (e.g., de-scheduling process P_1 and scheduling process P_2), the processor swaps core-local microarchitectural state from P_1 to memory. Because microarchitectural state need not be loaded for program correctness, P_2 may safely begin executing as soon as state from P_1 has been saved, with P_2's state loading in parallel to its execution. The processor design prioritizes loading state recognized as particularly important to performance (e.g., front-end state such as that of the BPU[7]) and can opt to only load seldomly-used state as needed (e.g., floating point unit state for primarily-arithmetic programs[14]). Ultimately, microarchitectural save-and-restore will avoid much of the performance penalty of the state-of-the-art flush-on-switch approach, for which the primary penalty arises from warming microarchitectural structures after a context switch.[7]

Absolute Spatial Isolation via Dynamic Partitioning. Dynamic partitioning ensures that two simultaneously executing contexts cannot access or modify each other's data. Thus, at a minimum, absolute isolation via dynamic partitioning necessitates extending each otherwise-unprotected entry within a resource with $\log_2 n$ context bits, where n is the number of hardware contexts which may simultaneously execute. For core-local state, typically $n = 1$ (for 2 hyperthreads, assuming subsequent contexts are temporally-isolated). If software permits shared state between hyperthreads, a second bit can denote this relationship without loss of security. Our design also weighs the performance and area costs of using additional bits to prevent the need for microarchitectural save-and-restore in common cases (e.g., a user/kernel bit for syscalls, or an address space ID tag for process-process context switches).

Probabilistic Isolation via Randomization. Given a set of performance and area constraints, achieving absolute isolation may not be feasible for all microarchitectural resources. For instance, dynamically partitioning resources via context bits does not necessarily scale to a large number of cores, particularly if data sharing is desired between an arbitrary subset of these cores (i.e., exponentially-many combinations). Furthermore, multi-core state (e.g., the last-level cache) consumes a large portion of total processor state, meaning microarchitectural save-and-restore is particularly costly for these resources. Therefore, our design exploits recent advances in randomized microarchitectural mappings (e.g.,[15,16]) to probabilistically-isolate data from each context. More specifically, our processor design is able to periodically remap select processor state to a new location, thereby decreasing the likelihood of an attacker accessing or influencing specific state from another context.

Notably, while latency requirements often preclude randomized index functions from use in core-local structures (e.g., increasing the L1 cache lookup latency from 4 to 5 cycles incurs 25% performance overhead on each access), such randomization can yield significantly lower overhead when used in longer-latency, multi-core structures like the last-level cache. Thus, using randomized mappings for such resources, as needed to satisfy user constraints is the right design decision. Ultimately, our design allows the administrator (e.g., the cloud provider) to tune the remap rate (and thus, isolation probability) based on their needs.

Non-Observability via Information Flow Control. Finally, even with temporal, spatial, and randomized isolation mechanisms at the processor's disposal, there remain instances in which such techniques are ill-suited for satisfying the user and the administrator's (e.g., cloud provider) combination of performance, area, and security needs. For instance, consider the case of parallel L1 cache and TLB accesses, where the TLB traditionally completes the associated permission checks either during or following the L1 access. To prevent cross-hyperthread data accesses in the L1 cache, the processor must either (a) increase L1 lookup latency by placing the TLB on the critical path/introducing a randomized indexing function, or (b) increase L1 area consumption by tagging each cache line with an associated context ID.

Given the premium placed on L1 performance and area in processor design, an alternative solution for this and similarly-constrained scenarios would be to mitigate cross-context *transmission* rather than *access*. While the security principle of least privilege dictates that cross-context data access should never be allowed, our prior work demonstrates that there are instances in which relaxations are possible without information leakage. Namely, NDA[9] permits (potentially cross-context) speculative data accesses but introduces microarchitectural mechanisms to delay dependent instructions. In doing so, NDA prevents the data from entering timing side channels until speculation (and thus, permission checks) resolve, while still reaping the performance benefits of speculative data accesses. Our follow-up work, DOLMA,[10] improves upon the performance of NDA without loss of security by ensuring each micro-op does not produce timing changes—and thus, is safe to execute—in the common case (e.g., an L1 hit for a load).

2.3 Implementation and Evaluation.

For rapid implementation and exploration of the design space, we are pursuing a basic prototype of our speculative, out-of-order processor in the cycle-accurate gem5 microarchitectural simulator.[17] Our evaluation targets are a wide range of single-threaded[18] and multi-threaded[19] benchmark applications, as well as additional CPU- and memory-intensive workloads from a variety of different suites.[20] We will augment the performance and security statistics provided by these simulators with area and energy measurements from McPAT.[21,22] To better understand the design trade-offs of microarchitectural context isolation, we will then port a subset of these policies to a RISC-V prototype implementation, building atop the open-source BOOM core.[23]

REFERENCES

[1] Yu, J., Hsiung, L., El Hajj, M., and Fletcher, C. W., "Data oblivious isa extensions for side channel-resistant and high performance computing.," in [*NDSS*], (2019).

[2] Razavi, K. and Trivedi, A., "Stratus: Clouds with microarchitectural resource management," in [*HotCloud*], (2020).

[3] Gandhi, V. and Mickens, J., "Rethinking isolation mechanisms for datacenter multitenancy," in [*HotCloud*], (2020).

[4] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., et al., "Meltdown: Reading kernel memory from user space," in [*USENIX Security*], (2018).

[5] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., et al., "Spectre attacks: Exploiting speculative execution," in [*IEEE S&P*], (2019).

[6] Weisse, O., Van Bulck, J., Minkin, M., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Strackx, R., Wenisch, T. F., and Yarom, Y., "Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution," *Technical Report* (2018).

[7] Bourgeat, T., Lebedev, I., Wright, A., Zhang, S., Devadas, S., et al., "Mi6: Secure enclaves in a speculative out-of-order processor," in [*MICRO*], (2019).

[8] Henning, J. L., "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH CAN* (2006).

[9] Weisse, O., Neal, I., Loughlin, K., Wenisch, T., and Kasikci, B., "NDA: Preventing Speculative Execution Attacks at Their Source," in [*MICRO*], (2019).

[10] Loughlin, K., Neal, I., Ma, J., Tsai, E., Weisse, O., Narayanasamy, S., and Kasikci, B., "Dolma: Securing speculation with the principle of transient non-observability," in [*USENIX Security*], (2021). To appear.

[11] Osvik, D. A., Shamir, A., and Tromer, E., "Cache attacks and countermeasures: The case of aes," in [*CT-RSA*], (2006).

[12] Kocher, P., Jaffe, J., and Jun, B., "Differential power analysis," in [*CRYPTO*], (1999).

[13] Nazari, A., Sehatbakhsh, N., Alam, M., Zajic, A., and Prvulovic, M., "Eddie: Em-based detection of deviations in program execution," in [*ISCA*], (2017).

[14] Stecklina, J. and Prescher, T., "LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels," *arXiv preprint arXiv:1806.07480* (2018).

[15] Bourgeat, T., Drean, J., Yang, Y., Tsai, L., Emer, J., and Yan, M., "Casa: End-to-end quantitative security analysis of randomly mapped caches," in [*MICRO*], (2020).

[16] Purnal, A., Giner, L., Gruss, D., and Verbauwhede, I., "Systematic analysis of randomization-based protected cache architectures," in [*S&P*], (2021). To appear.

[17] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., et al., "The gem5 simulator," *ACM SIGARCH CAN* (2011).

[18] SPEC, "Standard Performance Evaluation Corporation SPEC CPU 2017." spec.org/cpu2017/.

[19] Bienia, C., Kumar, S., Singh, J. P., and Li, K., "The parsec benchmark suite: Characterization and architectural implications," in [*PACT*], (2008).

[20] Ghose, S., Li, T., Hajinazar, N., Cali, D. S., and Mutlu, O., "Demystifying complex workload-dram interactions: An experimental study," in [*ACM SIGMETRICS*], (2019).

[21] Li, S., Ahn, J. H., Strong, R. D., Brockman, J. B., Tullsen, D. M., and Jouppi, N. P., "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in [*MICRO*], (2009).

[22] Xi, S. L., Jacobson, H., Bose, P., Wei, G., and Brooks, D., "Quantifying sources of error in mcpat and potential impacts on architectural studies," in [*HPCA*], (2015).

[23] Celio, C., Zhao, J., Gonzalez, A., and Korpan, B., "Riscv-boom documentation," (2019).