# SecSMT: Securing SMT Processors against Contention-Based Covert Channels

Mohammadkazem Taram[†], Xida Ren[*], Ashish Venkat[*], Dean Tullsen[†]
[†]*University of California San Diego*, [*]*University of Virginia*

## Abstract

This paper presents the first comprehensive analysis of contention-based security vulnerabilities in a high-performance simultaneous mulithreaded (SMT) processor. It features a characterization of contention throughout the shared pipeline, and potential resulting leakage channels for each resource. Further, it presents a set of unified mitigation/isolation strategies that dramatically cut that leakage while preserving most of the performance of a full, insecure SMT implementation. These results lay the groundwork for considering SMT execution, with its performance benefits, a reasonable choice even for security-sensitive applications.

## 1 Introduction

The pursuit of secure computation has always featured a clear tension between performance and security. Security mitigations often come with a high performance [1] impact that can be manifested in serious environmental and economic impacts [2] if they are employed, and in disastrous security and privacy breaches [3–5] if they are not. In the context of processor architectures, this security-performance tension is only growing as new attacks appear, each exploiting a crucial performance optimization in the processor, threatening to unwind decades of architectural gains.

Microarchitectural attacks exploit different architectural features. Speculative execution [6], shared caches [7], branch prediction [8], execution units [9], and I/O throughput optimizations [10] are examples of the features that are exploited in both well-established attacks and more recent instances [6]. Turning off any of these features could be crippling to performance, so we typically seek ways to continue to enable the optimization but with higher levels of protection.

One performance optimization, however, is often switched off in the name of security, at significant performance cost: Simultaneous Multithreading (SMT) [11, 12]. SMT enables a processor core to issue instructions from multiple threads to the execution units in the same core in the same cycle. With a small investment in hardware, the processor can greatly increase the throughput/utilization of the pipeline, more effectively hiding latencies and stalls of all types. The substantial benefits of SMT have led to its widespread adoption by virtually all the major players in the high-performance processor market, i.e., Intel, AMD, IBM, and ARM.

SMT achieves its high level of execution efficiency by dynamically allocating resources to threads as they are needed, effectively utilizing resources not needed by one thread to accelerate another. Virtually every part of the pipeline is potentially shared and contended for in some way. This creates a performance coupling between co-resident threads that is an enormous challenge for security. Some have suggested turning off SMT altogether [13]. Google, as a response to MDS attacks, has disabled SMT by default on Chrome OS 74 and later [13]. OpenBSD takes a similar measure by disabling SMT by default on version 6.4 and later [14]. Red Hat has announced [15] kernels with updated controls allowing users to choose whether to disable the feature or not.

Security researchers and architecture researchers are actively working to preserve many of the optimizations recently under attack (speculative execution, caches, branch prediction, store-load forwarding, etc.). This paper makes the case that it is time to add SMT to the set of features we should preserve even for secure execution. This research seeks to identify the extent of the vulnerability of current SMT processors, and to begin to navigate the middle ground – that is, how much of SMT's performance benefits can we retain while providing vastly greater performance decoupling?

This paper seeks to provide an exhaustive evaluation of resource contention across the entire pipeline for recent offerings from both Intel and AMD. We find that those two providers already take very different approaches to the security/performance tradeoffs. Both, however, provide a number of high-bandwidth channels of potential leakage, including several never before identified. By focusing on the bandwidth of covert channels utilizing the various resources, we are able to perform a unified, systematic, and exhaustive study of pipeline resource vulnerabilities.

In a similar way, we also seek to examine more secure approaches to multithreading that can be applied in a more holistic and comprehensive manner. We identify three different approaches to partitioning that can be applied to all contended resources with slight variation, and evaluate their ability to restore the performance of a fully dynamically shared SMT processor.

These approaches include full static partitioning (the approach applied already to several pipeline resources) and two new approaches. Adaptive partitioning provides a temporary, hard partition between threads for a particular resource, but that partition can move at regular intervals to adapt to long-term program behavior, preserving some of SMT's ability to adapt to changing execution needs with minimal leakage between threads. Asymmetric SMT enables the system to prevent leakage to an untrusted thread, but without sacrificing the performance of the trusted thread. Our results show that these secure multithreading approaches provide high isolation between threads while retaining most of the performance of a dynamically shared, insecure SMT implementation.

## 2 Background and Related Work

This section provides background on modern x86 processor architectures, SMT architectures, and microarchitectural side-channels, with a focus on SMT-enabled attacks.

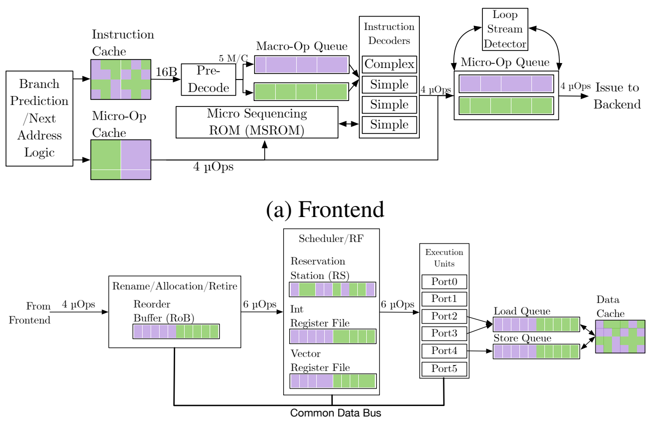
### 2.1 The x86 Pipeline Resources/Structures

Figure 1a shows a modern x86 processor's frontend. The frontend is responsible for fetching, decoding, and delivering operations to the backend. In x86 processors, this is accomplished using one of the following three methods:

*1) The legacy decode pipeline.* Each cycle, the frontend reads a 16-byte block from the instruction cache into the fetch buffer, which then feeds into the predecoder that demarcates individual variable-length x86 instructions (also called macro-ops). These macro-ops are then inserted into a small buffer (the macro-op queue). Instructions in this queue are then distributed to one of the instruction decoders, which translate each instruction into internal RISC (Reduced Instruction Set Computing)-like *micro-ops*. Modern Intel processors feature one complex decoder that is able to translate the instructions into up to four micro-ops, plus simple decoders that can only translate instructions that decompose into a single micro-op. Any instruction that translates to more than four micro-ops is handled via a Micro Sequencing ROM (MSROM). The decoded micro-ops then get queued up in a small structure called the micro-op queue – also called the instruction decode queue – until they get issued into the backend.

*2) The micro-op cache.* Due to the complexity of the decoding process, the legacy decode pipeline can be a major performance bottleneck. To alleviate this, most modern x86 processors cache decoded micro-ops in a special structure called the *micro-op cache* or the decoded stream buffer. This cache enables the frontend to bypass the slow and power-hungry legacy decode pipeline whenever the translated micro-ops of an address are already available in the micro-op cache. Due to the streaming nature of the micro-op cache, the processors often impose special placement rules [16]. For example, in Intel processors, the micro-ops of a 32-byte region of the code can be placed in the micro-op cache only if the region gets translated into less than 18 micro-ops.

*3) The loop stream detector.* Intel processors feature another optimization called a *loop stream detector* (LSD). The LSD further improves bandwidth and power consumption by bypassing both the legacy decode pipeline and the micro-op cache. The LSD identifies small loops within the micro-op queue and then locks down the micro-ops in the queue. It then delivers the micro-ops from the micro-op queue until an unexpected control flow (i.e., branch misprediction) occurs [16].



(a) Frontend

(b) Backend

Figure 1: Simplified Architecture of a Modern X86 Processor.

Figure 1b shows the main components of a modern x86 out-of-order superscalar backend. First, the backend renames the architectural registers of the issued micro-ops and allocates an entry in the reorder buffer for each. They are then forwarded to the scheduler, also called reservation stations or the instruction queue, which is responsible for identifying micro-ops ready for execution (i.e., whose operands are ready) and dispatching them into available execution units. Intel groups the execution units into execution ports, and the scheduler can dispatch as many micro-ops per cycle as allowed by the superscalar width of the processor, with the restriction that only one micro-op is dispatched to a free port in any given cycle.

### 2.2 Simultaneous Multithreading

Simultaneous Multithreading (SMT) [11, 12] is an architecture that allows for multiple hardware execution contexts in a single out-of-order superscalar pipeline. In an SMT processor, instructions from multiple threads can be dispatched on any cycle. An SMT processor significantly improves the pipeline utilization since it allows continued forward progress if one thread experiences a temporary stall in the pipeline due to long latency (e.g., a cache miss), or even several tightly-dependent short latency operations. Its performance benefit comes from its ability to dynamically assign resources to the thread that needs them each cycle. But this level of sharing provides heavy exposure of one thread's performance to the characteristics of the other.

Contention is possible in various ways throughout the entire pipeline. Consider the Reservation Stations (RS). Ivy Bridge has 54 entries available for instructions to wait for their operands to become available before being eligible for execution. The number of RS entries available to a thread define the window over which the processor can look for out-of-order parallelism, and significantly impacts performance. This resource could be shared between threads in various ways, with very different security implications. RS entries could be *duplicated*, where each thread would have access to exactly 27 entries (assuming two thread contexts). The RS could be *partitioned*, where in SMT mode each thread has access to 27, but when only a single context is running, it has access to 54. In that case, the only effect one thread sees is whether the other thread is running or not. It could be fully *shared* (this was the general assumption in the original SMT literature), where RS entries are allocated to whichever thread asks for them, with the fetch unit guiding instruction entry so that neither thread would fill the structure and starve the other [12, 17]. Full sharing typically, but not always, maximizes performance; however, a thread that put significant pressure on the RS would have its performance constantly vary according to the RS utilization of the other thread, and completely saturating the RS could have a dramatic impact on the other thread's performance. An intermediate solution is *thresholding*, where either thread is allowed to use up to, say 44 entries, ensuring that at least 10 are always available to the other thread even if it's not currently using them. In theory, thresholding can leak just as much information as full sharing, but in practice it is much more difficult to successfully execute an attack because of the difficulty of keeping resource utilization right near the edge of the threshold.

In Figure 1, we can see examples of shared structures (e.g., Instruction Cache), duplicated structures (Macro-op Queue), and partitioned structures (Micro-op Queue).

## 2.3 Microarchitectural Covert/ Side Channels

The literature abounds with security attacks that leak information through a shared microarchitectural structure or feature [6, 18–20]. Shared caches [21–26], branch predictors [8, 27–30], store-to-load forwarding [31], Translation Lookaside Buffers (TLB) [32, 33], I/O throughput optimizations [10, 34, 35], and the processor execution ports [9, 36, 37] are a few examples that are exploited as a source of side-channel leakage. Most of these attacks leak information through a timing difference that is caused by contention on a shared resource. The recent transient execution attacks [6, 38–41] also rely on a microarchitectural side-channel to leak the information to the attacker's domain. Previous work also uses timing measurements to infer different processor's features such as the size of the ROB [42].

Previous work [43, 44] has used information-theoretic and mathematical approaches to study information leakage from microarchitectural channels. For example, Hunger et al. [44] develop an abstract mathematical model for microarchitectural channels, with the end goal of devising an attack detection mechanism. Our characterization study, however, targets a different goal. It aims to compare and contrast resource sharing between threads in different implementations of SMT, to then evaluate the inherent vulnerability of sharing of each resource, and to guide the design of SMT security measures.

Previous work has also shown covert and side channels constructed on GPU resources [45–47]. Naghibijouybari et al. [46] exploit contention on GPU resources to infer victim's web browsing activities. Dutta et al. [45] target Intel's integrated GPU architectures and present covert channels that put contention on the CPU-GPU bus as well as the shared last-level cache. Similar to the CPU-based channels, these channels also leak information through the timing variation caused by resource contention. However, the list of possible targets for a GPU channel is limited to co-locating kernels from two different applications on a GPU. In contrast, CPU, and in particular SMT-based channels can leak more fine-grained information from a vastly larger set of targets, and thus pose a more imminent threat.

## 2.4 SMT Covert and Side Channels

SMT, with the sheer amount of shared resources, potentially greatly expands the microarchitectural attack surface, as nearly every structure could be contended for at some level. However, while SMT facilitates the exploitation of many side channels, not all of these channels are fundamentally tied to SMT. Cache side-channels [22, 48, 49], which include the vast majority of these attacks, for example, are almost as effective in a cross-core setting [50]. Some of the SMT-specific attacks are the result of a design bug [39], not an inherent SMT issue, which can be mitigated without much performance cost in future generations. Nevertheless, the core principles of SMT have already been targeted in some of these side-channel attacks.

The SMT execution port timing channels have a long history [37, 51]. But more recently, PortSmash [9] exploits, in an end-to-end attack, the timing variation caused by the contention of the SMT threads on specific execution ports to leak the private key of a TLS server. SmotherSpectre [36] also exploits the contention on the execution ports but combines it with speculative execution to mount a transient execution attack. SmotherSpectre uses speculation to steer the execution to a gadget that includes a data-dependent branch that accesses a specific port based on the value of a secret.

Other attacks target other shared resources in SMT processors. TLBleed [32] exploits shared TLBs in SMT processors to leak a victim's memory activity. CacheBleed [23] uses contention of the sibling SMT threads on the cache banks as the source of the leakage to break a constant-time RSA implementation. Similarly, MemJam [52] targets the shared memory-dependency detection unit to leak information about memory accesses of a victim. Ren, et al. [53] develop multiple attacks exploiting the micro-op cache of Intel and AMD processors to leak information across different security domains, including colocated SMT threads. Shared branch predictors are also extensively exploited to leak secret information [8, 54, 55] and also to steer speculation to attacker-desired addresses [6].

There are also efforts to automatically construct covert channels in SMT processors. Covert Shotgun [56] proposes an automated framework to examine possible pairs of instructions in an ISA for building covert channels. Covert Shotgun compares the execution time of the instructions in single-threaded and SMT mode to detect if there is a possible covert channel. More recently, ABSynthe [57] conducts a similar study that expands Covert Shotgun to include all instructions in x86 and ARM ISAs and compares the results for a variety of architectures. While these approaches can discover covert channels in an SMT processor, the exact source of the leakage for the discovered covert channels remains unknown. This work, in contrast, characterizes different covert channels based on the actual source of leakage.

### 2.5 Side-Channel Mitigation

The research community is increasingly active on countermeasures for microarchitectural side-channels. Similar to the microachitectural attacks, the defense research also leans heavily toward mitigations for cache-based side channels [58–63]. A diverse set of strategies has been proposed to mitigate side channels. Partitioning [60, 64–67], randomization [65, 68–70], detection [67, 71–73], oblivious execution [74], and encryption [75] are among frequently suggested high-level approaches. Many of the more recent defenses focus on a mitigation in the context of speculative execution vulnerabilities [76–82].

However, much less attention is given to defenses for non-cache microarchitectural side channels. SMT-COP [83] introduces a temporal and spatial partitioning scheme for execution ports in SMT. SMT-COP also introduces a selective approach where it selectively enables and disables functional unit partitioning for some regions of the code. Unlike this work, SMT-COP's partitioning, once enabled, is entirely static, failing to take advantage of the benefits of an SMT architecture. Moreover, our study is not limited to execution ports, and we examine mitigations on a comprehensive set of pipeline resources.

Hyperspace [84] secures SGX execution against SMT channels by creating a trusted shadow SGX thread that runs on the same physical core as the main thread. Hyperspace verifies the co-location of the main and the shadow threads using a cache covert channel. The sole purpose of the shadow thread is to prevent any untrusted thread being scheduled on the same core as the main thread. Therefore, in terms of resource utilization this approach is similar to, and in some cases worse than, turning off SMT.

Xu et al. [67] propose a mitigation strategy for GPU-based covert channels. It relies on a decision tree classifier to detect a potential attack, and then enables temporal and spatial partitioning of GPU resources to mitigate the contention.

## 3 Assumptions and Threat Model

The main focus of this work is on covert channels constructed by contention on the main pipeline structures between the co-located threads on an SMT processor. These channels are the dominant form of leakage introduced by simultaneous multithreading, and until mitigated, will likely dominate other channels. We principally study covert channels, because the goal of this work is to provide, among other things, a comprehensive and systematic analysis of the vulnerability of existing SMT processors. Such an approach would not be possible if we were to try to analyze all specific side-channel attacks customized to each feature, and would become out of date quickly as new attacks are devised. By concentrating on covert channels, we can evaluate the inherent vulnerability of each feature and have some basis for comparing them and understanding where the greatest vulnerabilities lie.

Furthermore, by closing these covert channels we also close any potential side channel that exploits them, including those used in speculative execution attacks. To be successful, most speculative attacks need to effectively perform two tasks: (1) steer the execution to attacker-desired locations, and (2) leak information to the attacker domain using a covert channel. To stop such attacks, it is sufficient to prevent the latter, which we aim to do by denying unauthorized information leakage – speculative or otherwise – between SMT threads.

We, then, propose mitigations against the covert channels with the following assumptions. We assume an SMT processor on which two threads, T1 and T2, can share the pipeline. We assume T1 and T2 are running on separate processes and are prohibited from any form of direct communication, but they can run any non-privileged instruction on the SMT core. Any of the threads are allowed to make artificial contention on any of the shared pipeline resources to leak information about the usage of the other thread. We consider multiple scenarios:

- T1 and T2 are mutually untrusted: in such scenarios, any information flow from T1 to T2 and from T2 to T1 should not be allowed.
- The trust is asymmetric, i.e., T1 is untrusted, and T2 is trusted: in such scenarios, the information flow from T1 to T2 is allowed, while the information flow from T2 to T1 should be blocked.
- The threads are mutually trusted: any covert communication between T1 and T2 is allowed.

The main focus of this work is on closing timing channels engendered by resource contention between SMT threads. Therefore, this work does not consider power, voltage/frequency, and thermal channels.

## 4 Covert Channel Characterization

The goal of this paper is to make SMT processors more secure against contention-based side-channel attacks. To that end, we first conduct a rigorous analysis on current SMT processors to assess the degree to which they share each pipeline resource between threads, and to also measure the potential information leakage resulting from sharing each of these pipeline resources. This analysis then guides the design of SMT security measures (Section 5). This study also leads to the discovery of multiple previously unreported and high-bandwidth covert channels.

Table 1: Sharing Mechanism of Pipeline Resources

| | | Intel (Skylake) | | AMD (Zen2) | |
|---|---|---|---|---|---|
| | Resource | Sharing | BW (bps) | Sharing | BW (bps) |
| Front-End | L1 iCache | S | 742K | S | 1.27M |
| | Branch Target Buffer | S | 796K | S | 478K |
| | Micro-Op Cache | P | <24K | S | 604K |
| | Fetch Bandwidth | S | 1.64M | S | 833K |
| | Decode/Issue Bandwidth | S | 1.15M | S | 1.03M |
| | iTLB | P | <24K | S | 820K |
| | Micro Sequencing ROM | M | – | S | 353K |
| | Loop Stream Detector | P | <24K | – | – |
| Back-End | Reservation Station | T | – | S | 56K |
| | Reorder Buffer | P | <24K | P | – |
| | Physical Register File | P | <24K | S | 40K |
| | Execution Port | S | 1.22M | S | 715K |
| Memory | dTLB | S | 982K | S | 964K |
| | L1 dCache | S | 1.13M | S | 902K |
| | L1 dCache Read Bandwidth | S | 1.36M | S | 1.64M |
| | Load Queue | P | <24K | S | 36K |
| | Store Queue | P | <24K | P | – |

S:Shared, P:Partitioned, T:Thresholded, M: Time Multiplexed
<:Limited by the maximum switching frequency between single-threaded and SMT modes

## 4.1 Overview

The first step in our analysis is to deconstruct how the processor manages resource sharing between the SMT threads. We go through an exhaustive list of pipeline resources and reverse-engineer the sharing mechanism that the processor uses for each of the pipeline resources. We broadly categorize each pipeline resource into *statically partitioned*, *dynamically shared*, and *duplicated* resources based on our experimental analysis. We note that partitioned resources can either by spatially shared (half assigned to each thread in SMT mode) or time-multiplexed (one thread uses all resources one cycle, the other thread the next). We consider *thresholding*, where the partition is dynamic, but neither thread can completely exhaust the resource, as a special case of dynamically shared.

To reverse engineer the sharing mechanism of a pipeline resource, we craft a set of assembly instructions that create artificial contention on that resource. This set of assembly instructions needs to have four features: (1) it should saturate the structure-under-test, (2) the amount of saturation should be controllable, (3) it must not create high contention on any other pipeline resources/structures, and (4) it should put the contention on the critical path, so the effect of the contention is exposed via performance. Then we run this test code simultaneously on the sibling threads and measure the effects.

If increasing the saturation in one thread affects the execution time or the usage of the other thread, we conclude that the structure is dynamically shared. For dynamically shared resources, we can typically use the same code to construct a covert channel by selectively saturating or freeing the resource. Finally, we measure the bandwidth and error rate of the constructed covert channel.

We also explore the possibility of constructing covert channels on statically partitioned resources. Table 1 shows the list of the pipeline resources that we examine along with their discovered sharing mechanism and the achieved covert-channel bandwidth on two different implementations of SMT: AMD Zen2 and Intel Skylake. The table shows that while AMD allows most of the pipeline resources to be competitively shared, Intel Skylake employs some kind of partitioning or time multiplexing for most key pipeline resources.

In addition to Intel Skylake, we also study the sharing mechanism of pipeline resources on Ivy Bridge, an older Intel microarchitecture. Running our microbenchmarks on these microarchitectures shows that Intel uses similar sharing mechanism across these different microarchitectures. Similarly, we also examine whether any of our channels are impacted by different versions of the microcode. In an exhaustive analysis on the Ivy Bridge processor (because the older processor naturally offers updates spanning a longer time period), we observe no change in any of the discovered sharing mechanisms across all available microcode versions. Details of the methodology are discussed in Section 6. The rest of this section examines key pipeline structures, moving from front to back of the pipeline.

## 4.2 Instruction Fetch Bandwidth

To reverse-engineer the sharing policy of the instruction fetch unit in an SMT processor, we develop a microbenchmark that creates artificial contention on the instruction cache read bandwidth, while ensuring that there is no contention for the rest of the pipeline resources. To this end, we take advantage of the fact that NOP instructions have a minimal footprint, as they get eliminated early in the pipeline and consume few backend resources, if any.

However, the most commonly used x86 NOPs are 1-byte instructions. These do not suit our purposes because they saturate the decoders long before they saturate the fetch unit (which can fetch 16 NOPs per cycle). To circumvent this, we leverage a special 15-byte long NOP [16] instruction that allows us to effectively saturate the instruction cache read bandwidth, limiting the throughput of the fetch unit to just one instruction per cycle. Further, we ensure that these NOP15 instructions always miss in the micro-op cache and actually use the instruction cache read bandwidth, by using large loops of static NOP15 instructions in our microbenchmark that exceed the micro-op cache capacity.

Through performance counter measurements, we find that the Intel frontend sustains a throughput of one NOP per cycle when our microbenchmark is run in single-threaded mode. AMD Zen2 also provides the same throughput, despite enjoying an instruction cache bandwidth of 32 bytes per cycle. This is because Zen2 requires instructions that are larger than 8 bytes to be in the first 16 bytes of the fetch buffer. When our microbenchmark is run in SMT mode along with a competing SMT thread that executes the same code, the frontend throughput is exactly halved, delivering one instruction every two cycles, for each thread. However, if the code that runs on the competing thread (T2) delivers its micro-ops through the micro-op cache or the loop stream detector, T1 gains back its original one NOP per cycle throughput. This shows that the

```
SEND_ZERO:        SEND_ONE:       RECEIVER:
  MOV RAX, 100      MOV RAX, 100    TIME   #Record Time
L0:               L1:               NOP15 #15-Byte NOP
  NOP8 #8-Byte      NOP #1-Byte     ...
  ...#N<LSD         ...             NOP15 #15-Byte NOP
  NOP8 #8-Byte      NOP #1-Byte     TIME   #Record Time
  DEC RAX           DEC RAX         JMP RECEIVER
  JNZ L0            JNZ L1
```

Listing 1: Fetch Bandwidth Covert Channel on Intel Processors. The total number of micro-ops in the receiver loop is larger than the size of micro-op cache to ensure a zero percent hit rate. NOP8 and NOP15 are aliases for multi-byte NOP instructions [16].

instruction fetch bandwidth is dynamically shared between the threads as we observe a direct impact on the execution time of its sibling thread when they contend for the fetch unit.

Listing 1 shows the sender and receiver routines for a covert channel implementation that exploits the performance differences that arise due to contention for the instruction fetch bandwidth in Intel. The sender thread transmits 'zero' by executing a set of NOP instructions that are delivered by the loop stream detector or the micro-op cache, creating no fetch contention. The sender thread transmits 'one' by executing a set of regular 1-byte NOP instructions, maximizing fetch contention – note that 1-byte NOP instructions will miss in the micro-op cache as the micro-op cache of Intel processors does not cache the line if there are more than 18 micro-ops in a 32-byte region of the code [16,53]. The receiver thread then measures the execution time of long NOPs that miss in the micro-op cache to detect that contention. The total number of the micro-ops within the receiver loop is set to be larger than the size of the micro-op cache, ensuring that every instruction uses the fetch bandwidth. This covert channel, as Table 1 shows, can achieve a bandwidth as high as 1640 kbps on an Intel Skylake processor and as high as 833 kbps on Zen2.

## 4.3 Decode/Issue Bandwidth

After fetching instructions into the fetch buffer, the x86 frontend translates the instructions into micro-ops (decode) and delivers those micro-ops to the backend (issue).

To contend for these decoders, we choose regular 1-byte NOPs. Not only do they not consume backend resources, but they also do not saturate the fetch bandwidth as described above, putting decode/issue bandwidth on the critical path. Thus, when we run regular NOPs on a single thread, the decode pipeline is able to deliver 4 NOPs (micro-ops) per cycle to the backend of the processor. However, this throughput is reduced to 2 NOPs per cycle if we co-locate this thread (T1) with a sibling thread (T2) that executes the same set of NOPs, thereby contending for the decoder resources.

To construct a covert channel that exploits the decode/issue bandwidth, we need to identify the conditions upon which the processor assigns more decode bandwidth to T1. If we switch T2's instructions to large NOPs (maximum of one instruction every two cycles), T1 still observes half of its single-thread throughput, suggesting that on each cycle the instruction decoders are assigned to one thread as a whole, i.e., the decode bandwidth is time-multiplexed between the threads, rather than being statically partitioned. We note that this is consistent with the details laid out in Intel patents [85].

```
SEND_ZERO:        SEND_ONE:       RECEIVER:
  MOV RAX, 100      MOV RAX, 100    TIME #Record Time
  CLFLUSH [RCX]   L1:               NOP
  MFENCE            NOP             ...  #N>LSD
L0:                 ... #N>LSD      NOP
  #Cache Miss:      NOP             TIME #Record Time
  MOV RCX, [RCX]    DEC RAX         JMP RECEIVER
  DEC RAX           JNZ L1
  JNZ L0
```

Listing 2: Decode/Issue Bandwidth Covert Channel on Intel.

Further, while one would expect T1 to gain back its single-thread throughput if T2 does not use the legacy decode pipeline (because it is using the micro-op cache or the LSD which both bypass the decoder), we observe that even in such cases the throughput of T1 remains half of its single-threaded throughput, indicating that the decoders remain time-shared between the threads, regardless of whether the threads actually contend for them. On the other hand, if T2 is stalled due to a bottleneck in the backend (full reservation stations, for example), we observe that T2 does give up its decode slots and T1 goes back to its full original 4 NOPs per cycle.

As Listing 2 shows, to slow the backend, we exploit a data cache miss followed by a sequence of instructions that are dependent on the long-latency load. This enables a high-bandwidth covert channel on both the AMD and the Intel architectures with a bandwidth as high as 1150 kbps on Intel and 1030 kbps on AMD.  Exploiting the frontend covert channels bolsters the adversary's ability to fingerprint various activities (e.g., cache misses, micro-op cache usage pattern) of a co-located victim thread, without any cache accesses, just by measuring the execution time of NOPs.

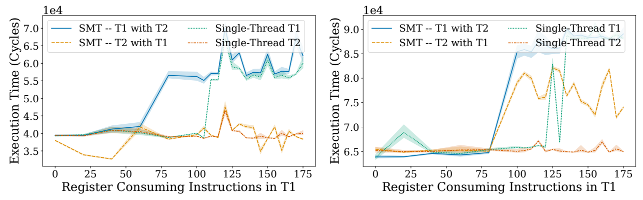
## 4.4 Register File

Next, we examine the sharing mechanism of physical register files of the Intel and AMD processors. Our register file characterization microbenchmark, shown in Listing 3, consists of two memory read instructions that always miss in the caches. Between these loads, we have a variable number of instructions that each consume a physical register, i.e., they have a destination register. When a thread's partition fills, no more instructions can be renamed, placing a limit on the window for out-of-order execution. If the second memory read is not renamed, it is then serialized (not renamed until the former commits). If there is sufficient space to rename it, the loads execute in parallel and performance is significantly improved. By increasing the number of the register-consuming MOVs we can identify the exact point where we exhaust the renaming registers before the second load instruction arrives.

The impact on the execution time is visible in our results of

```
CLFLUSH [RDI]
CLFLUSH [RSI]
MFENCE
MOV RAX, [RSI] #Long-Latency Load
MOV RBX, 88    #Consumes One Phys. Reg
...            #Use N Phys Regs
MOV RBX, 88    #Consumes One Phys. Reg
MOV RAX, [RDI] #Long-Latency Load
```

Listing 3: Microbenchmark for Making Contention on Register Files. When N is larger than available physical registers the two loads cannot be issued in parallel.



(a) Intel (b) AMD

Figure 2: Reverse Engineering the physical register file sharing mechanism.

figure 2. These figures (one for Intel, one for AMD) each show four lines, representing two threads. This more detailed result is representative of the analysis done for each of the pipeline resources, although those discussions have mostly been condensed for space reasons. Here, T1 is varied in the number of registers that are occupied before the second high-latency load, while T2 is kept constant (at 50). For Intel, when T1 runs in single-threaded mode, we see that it can use about 128 registers before performance plummets, while in SMT mode that happens at 64. Further, we see that T2 in SMT mode (dashed orange line) is relatively unaffected by the size of T1. For AMD, however, we see that T2 in SMT mode (again, dashed orange) is clearly sensitive to the varying register pressure of T1. From this, we conclude that Intel's physical register file is statically partitioned, and AMD's is dynamically shared.

We observe a similar pattern when we change the consuming instructions to instructions that consume vector, vector mask, or floating point registers. That means the same policy is applied to those register files as well. We are able to exploit the contention on physical registers in AMD to construct a covert channel with a bandwidth of 40 kbps with less than 5% error rate.

### 4.5 Reservation Station (RS)/Scheduler

To contend for reservation stations, we use a microbenchmark similar to the previous section (Listing 3); however, we use *cmp* instructions which do not consume a physical register, but are still dependent on the first load instruction, so the instructions will consume an RS entry and cannot be issued until the first load completes execution and makes the result available to dependent instructions, causing them to quickly release RS entries. If we have enough entries in the reservation station available to the thread, the second load can be issued an RS entry and then dispatched to execution in parallel with the first load. Therefore, we see a spike in the execution time when the length of the dependency chain becomes greater than the number of reservation station entries, as the two loads become serialized.

In SMT mode, we observe that Ivy Bridge does not let one thread use all of the 54 RS entries and it always limits the maximum number of allocated RS entries to 40, even when the other thread does not consume any RS entries, e.g., only executes NOPs. We refer to this type of sharing as thresholding. More details on this experiment are provided in Section A.1.

While in theory it should be possible to construct a covert channel on the 26 shared entries, it imposes considerable noise and we are not able to construct a reliable channel on the Intel processor. On the other hand, AMD Zen2 uses a shared reservation station with which we can construct a covert channel with a bandwidth of 56 kbps.

### 4.6 Reorder Buffer, Load/Store Queues

We use slightly different variations of the microbenchmark shown in Listing 3 to reverse engineer the sharing mechanism of the ROB and the load and store queues. For the ROB, we replace the register-consuming instructions with NOPs. NOPs serve our purpose to isolate the ROB because they consume ROB entries, but not reservation stations or physical registers. For the load and store queues, we leverage load or store instructions that always hit in the cache to isolate those queues. These instructions each occupy a load/store queue entry and cannot be issued until the long-latency load executes. Therefore, at some point, by increasing the number of load/store instructions, the dispatch cannot progress due to lack of load/store queue entries. This then forces the long-latency loads to be serialized. Section A.1 provides more details on these microbenchmarks. Our experiments show that the ROB, load, and store queues are all statically partitioned in Intel processors; ROB and store queue are also partitioned in AMD Zen2, but the load queue is shared in Zen2.

### 4.7 Covert Channel on Partitioned Structures

As shown in Table 1, we find that many of the pipeline resources in Intel processors, and some in AMD processors, are statically partitioned between the threads. For the sake of completeness, we investigate the potential information leakage via these partitioned resources (even though we know shared resources will be the highest bandwidth channels). We construct a covert channel where the sender goes in and out of SMT mode, allowing the receiver to observe the state of the particular resource in question. The key here is our ability to enter and exit SMT mode as quickly as possible. To this end, we examine several x86 instructions used for busy waiting.

We first look at the *pause* instruction, originally introduced in Intel's Pentium 4 processor to improve the power and performance of the spin-wait loops [16], so that the spinning thread could free resources while waiting. However, our experiments with the *pause* instruction suggest that the resources remain partitioned even in the presence of a *pause* – that is, *pause* only releases shared resources, not partitioned.

We also consider *mwait*, which is a privileged instruction that provides a hint to the processor so it can enter a specified target low-power state [16]. In fact, this does release partitioned resources, and we are able to successfully create a channel, but we do not consider this the most useful attack vector since there are so many other attacks possible for a user with privileged access.

Finally, we examine Linux's *nanosleep* system call which is a non-privileged call that deschedules the caller thread until the time specified by the user has elapsed. We observe that, on Intel processors, calling nanosleep causes the processor to unpartition resources, allowing the sibling thread to monopolize them. The nanosleep syscall causes the Linux kernel to schedule an idle task on the logical core, which then executes the aforementioned mwait instruction. This enables a covert channel for transmitting information even via statically partitioned resources. The bandwidth of this covert channel, however, is limited to the minimum latency of the nanosleep system call. As shown in Table 1, using the nanosleep system call, we can achieve a bandwidth as high as 24 kbps on the Intel processor. On the AMD processor, however, the nanosleep syscall does not cause the resources to become unpartitioned.

In Section 5, we examine multiple partitioning schemes that provide greatly increased thread isolation. However, all are still potentially vulnerable to this channel (fast enter/exit of SMT mode) if they release all resources to a single thread. Thus, for the balance of this paper, we assume a simple solution that provides a countdown timer that limits the frequency at which the pipeline can release partitioned resources, even upon exiting SMT mode.

### 4.8 Other Pipeline Resources

For completeness, we also measure most of the other pipeline resources that have been covered extensively in the literature, such as caches [86], TLBs [32], and execution ports [9]. Those results appear in Table 1, but without extensive discussion. However, there are other well-studied SMT resources such as Pattern History Table (PHT) [27] that we do not re-measure as we focus more on lesser known channels. Our mitigations, nevertheless, can be readily applied to these structures as well.

Constructing covert channels on most of the cache-like structures requires some information about internal structures of these resources such as the indexing function and associativity. Also, the knowledge of the replacement policy of a cache-like resource can greatly affect the channel bandwidth as the attacker can exploit that to minimize the size of the probe set. It is, however, still quite possible to create a high-bandwidth channel on a cache-like structure without access to detailed information on the replacement policy, as the attacker only needs to create enough accesses to cause a single eviction to the other thread's entries. For example, for any LRU-like structure (e.g., tree-PLRU, bit-PLRU) with associativity of $n$, accessing $n$ new entries guarantees an eviction to the existing entries of a particular set.

## 5 Mitigations

This section introduces a suite of mitigation schemes we examine to eliminate or reduce the leakage across the SMT pipeline. Prior work has focused on identifying and solving SMT leaks one at a time [65,70,83]. However, in keeping with our systematic characterization study of pipeline resources, we will focus on systematic mitigation solutions that can be employed broadly across each individual contended structure.

The solutions described in this section include static partitioning, adaptive partitioning, and asymmetric SMT.

### 5.1 Static Partitioning

The most basic partitioning scheme, already employed heavily, statically divides a resource into equal-sized partitions. Static partitioning can be applied in two forms: spatial or temporal. Spatial partitioning assigns a resource to a thread and that assignment does not change through time. This can only be applied to resources for which the processor has multiple instances, such as ROB entries. If the number of instances of a resource is less than the number of SMT threads (e.g., some functional units), the processor cannot statically assign the resource to a thread. In such cases, the processor employs temporal partitioning (also called time multiplexing). Temporal partitioning assigns a single resource to each thread in a round-robin fashion. These basic partitioning schemes ensure that dynamic contention cannot occur between the threads, eliminating leakage. For example, in a strictly round-robin resource, the time slot assigned to thread T1 does not depend, in any way, on the usage of thread T2. T1 only gets one out of two slots whether or not T2 uses its slot. This completely inhibits a thread from inferring any information from the usage of the other thread.

### 5.2 Adaptive Partitioning

While static partitioning can eliminate dynamic contention between SMT threads, it typically results in underutilization of pipeline resources, sacrificing overall performance – the fundamental benefit of SMT processors is allowing resources to be better utilized by dynamically assigning each resource to the context that needs it. However, we show that it is possible to gain back much of the full performance of SMT if we can adapt to the varied needs of contending threads, but more slowly. Adaptive partitioning is an on-demand partitioning scheme that allows for the dynamic reconfiguration of partition size, once per *adaptation interval*. This not only improves resource utilization and overall performance, but also limits the information leakage to at most one bit per adaptation. In fact, our results show that even a very long adaptation interval (of 100,000 cycles) can be effective in recovering much of the full performance benefits of a fully shared SMT pipeline.

Our adaptive partitioning design augments each resource with a set of counters: (1) the current size of the partitions, (2) counters that track the number of "full" events that each thread encounters, and (3) a countdown timer until the next adaptation interval. Note that in our experiments we assume a single countdown timer for all resources to help simplify presentation of the results. It also takes three parameters that are set at design time: adaptation interval, adaptation step, and adaptation threshold. Figure 3a shows an example for adaptive partitioning of the instruction queue. For every adaptation period, we select a thread that has faced more full events in that period. We then increase the size of the partition of the selected thread by the adaptation step size. We increase the partition size only if the new partition size is still smaller than the adaptation threshold.

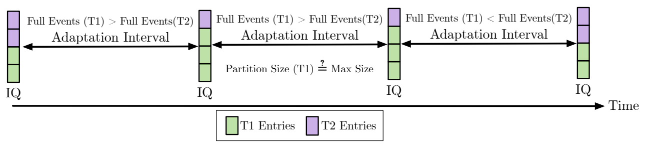
We can also apply adaptive partitioning for temporally shared resources, deviating from a completely symmetric round-robin assignment. For example, we might adaptively modify the assignment process for a resource that alternates between two threads each cycle, in such a way that the resource is assigned to a more hungry thread two out of every three cycles. If it needs more, we could again alter the policy such that the resource is assigned to the hungry thread three out of four cycles. At each adaptation interval, we examine the number of full events of each thread and increase the share of the thread with the highest full event count. If that thread is more resource hungry, we increase its count (subject to the threshold), otherwise we decrease its count. Figure 3b shows an example of adaptive temporal partitioning.

Adaptive partitioning severely restricts the leakage. Now, for each resource, the attacker can only leak at most approximately 1 bit (adaptation without threshold) per adaptation interval (100,000 cycles). For example, an attacker can probe the size of its own reservation station in two consecutive adaptation intervals. If the RS shrunk, the attacker infers that the victim's RS usage exceeds that of the attacker. This is orders of magnitude below known channels across cores on non-SMT processors.

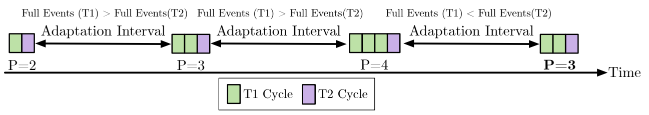
While adaptive partitioning limits leakage to a single bit per interval, the values of the counters could potentially be exposed when a thread crosses protection domain boundaries, and it is therefore critical to reset all such counters at domain crossings. Resetting the adaptation counters stops the current context's behavior from affecting that of the next context. By doing so, we might miss one opportunity to adapt. But we find that in steady-state, partition sizes do not change dramatically, and the performance effect of missing one adaptation opportunity is minimal.

## 5.3 Asymmetric SMT

While the adaptive partitioning scheme can help recover a significant chunk of the performance lost due to partitioning, it is still restrictive as it limits the pipeline resource utilization


Full Events (T1) > Full Events(T2)
Adaptation Interval
Full Events (T1) > Full Events(T2)
Adaptation Interval
Partition Size (T1) $\overset{?}{=}$ Max Size
Full Events (T1) < Full Events(T2)
Adaptation Interval
IQ
IQ
IQ
IQ
Time
T1 Entries
T2 Entries

(a) Spatial


Full Events (T1) > Full Events(T2)
Adaptation Interval
Full Events (T1) > Full Events(T2)
Adaptation Interval
Full Events (T1) < Full Events(T2)
Adaptation Interval
P=2
P=3
P=4
P=3
Time
T1 Cycle
T2 Cycle

(b) Temporal

Figure 3: Adaptive Partitioning Examples.

even in scenarios where at least one of the threads running on the processor are trusted or when no sensitive code is running and cross-thread information leakage may not be of concern. In this section, we describe a mitigation called *Asymmetric SMT* that allows two threads with asymmetric trust levels to securely share resources in an SMT processor, while preventing unauthorized information leakage from a higher security domain to a lower security domain.

With Asymmetric SMT, then, assuming active threads T_H and T_L, where T_H is at a higher security level than T_L, we have the ability to block the leakage from T_H to T_L, but allow leakage from T_L to T_H. An example where this is useful is sandboxing in web browsers. While it is not secure to leak information from the browser thread to the sandbox thread, it is safe to leak information from the sandbox to the browser thread. Similarly, it might be safe to leak information from a user process to a kernel process, from a guest virtual machine to the hypervisor, etc. Asymmetric SMT enables lost resource utilization due to partitioning, but the only beneficiary is the trusted thread. The performance of trusted threads is on the critical path for many applications, such as a web browser that runs untrusted Javascript. A study by the Google v8 team [87] shows that only 20% of Chrome's page processing time is spent in running untrusted Javascript code, while the rest is spent in trusted browser code that performs tasks such as parsing, compilation, and garbage collection. Therefore, by improving the performance of the trusted part of the execution, Asymmetric SMT can significantly impact overall performance.

The key to Asymmetric SMT is that it allows the trusted thread to *borrow* unused pipeline resources from the untrusted user, only in cases where it can *instantly* return the resource when the untrusted thread needs it back. Fortunately, this instantaneous return is possible for many of the pipeline resources as the out-of-order pipeline is already well-equipped with mechanisms to deal with various squash events.

The rest of this section discusses how Asymmetric SMT can be enabled for various pipeline resources. We categorize the resources into stateful (e.g., ROB), stateless (e.g., functional units), and cache-like resources.

### 5.3.1 Stateful Resources

We refer to the resources that hold the transient execution state of a thread's instructions across multiple cycles and then get released, as stateful resources – this includes physical registers, IQ entries, ROB entries, and load/store queue entries. Asymmetric SMT allows the borrowing thread to use a free unused entry from the other thread's partition. Here, we use Physical Register File (PRF) as an example.

Figure 4 depicts two example scenarios for borrowing a physical register. In the initial state, three out of eight entries are assigned to T_H (assuming PRF already uses an adaptive partitioning scheme). Note that Asymmetric SMT is orthogonal to the other two partitioning schemes (static and adaptive) and can be added to either. Once Asymmetric SMT receives a request from the trusted thread, T_H, which has exhausted all of the entries in its partition, it checks if T_H can borrow an unused entry from T_L. Asymmetric SMT permits borrowing only if the number of T_L's free entries is larger than a threshold (MIN_FREE), i.e., it always leaves some free slack registers. This condition is helpful at reducing the number of expensive squashes which results when T_L runs out of resources and one must be freed immediately by T_H.

Figure 4a shows the common-case scenario where we commit one of T_H's instructions before T_L's partition gets full. Note that in this case we are not borrowing a specific register, but rather allowing T_H's count to exceed its threshold. Thus, any T_H instruction that commits and frees a register will restore it to T_L. Figure 4b shows a scenario where T_L takes up all of the free entries in its partition before T_H gets the chance to return the borrowed register. In such a scenario, Asymmetric SMT immediately returns the borrowed register to T_L. It selects the youngest T_H instruction that holds a physical register and assigns that register to T_L. T_H then needs to squash that instruction and all of its subsequent instructions and restart execution from there.

Similarly, it is possible to allow a trusted thread to borrow instruction queue and load/store queue entries. During issue, if the trusted partition is full but the untrusted partition has more than MIN_FREE free entries, the trusted thread can borrow unused entries. If the untrusted partition becomes full before the trusted partition returns the borrowed instruction queue entry, the trusted thread should immediately return the borrowed entry.

Not all stateful resources are suitable for borrowing. For example, we can only allow borrowing of a limited number of ROB entries due to the instantaneous return requirement. If we borrow from the ROB more entries than what we can free per cycle (retire bandwidth), then the borrowing becomes visible to the owner thread and that leads to information leakage.

### 5.3.2 Stateless Resources

Examples of stateless resources include execution ports, functional units, fetch bandwidth, and commit bandwidth. Asymmetric SMT works well with stateless resources, as borrowing a stateless resource will not ever require an expensive squash. Here we use execution ports to describe Asymmetric SMT in stateless resources.

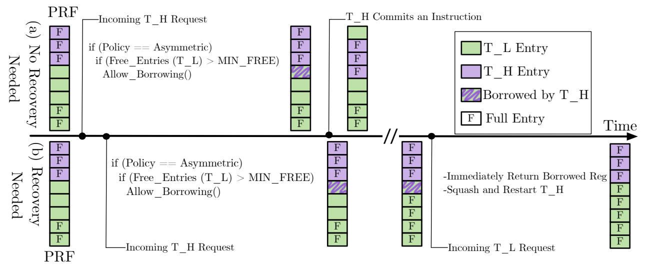


Figure 4: Borrowing a Physical Register in Asymmetric SMT. It shows the state of the physical register file over time for two scenarios.

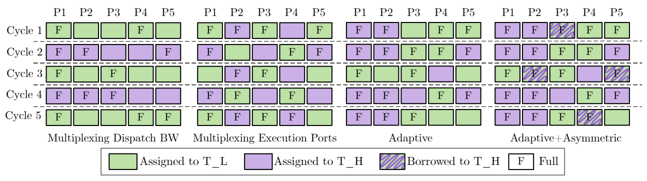


Figure 5: Partitioning Schemes for Execution Units/Ports. Our adaptive partitioning and Asymmetric SMT architecture can reclaim the unused execution slots caused partitioning.

Figure 5 illustrates different temporal partitioning (multiplexing) schemes for execution ports. We can assign the whole dispatch bandwidth to one thread each cycle, which reverts the pipeline to fine-grain multithreading [88] and sacrifices the benefits of SMT in eliminating horizontal waste [89]. A better approach is to multiplex individual execution ports each cycle, which can either be partitioned evenly or unevenly using our adaptive partitioning methodology. However, even with adaptive partitioning, there are cycles where T_L cannot fully utilize the ports that are assigned to it. In such cases, Asymmetric SMT utilizes the unused ports and borrows them for T_H. The scheduler for Asymmetric SMT, at each cycle, first tries to schedule all the ready instructions of T_L to the execution ports that are assigned to T_L at that cycle, then it assigns any unused T_L slots to T_H and schedules T_H's ready instructions. Note that for brevity, we do not show the non-pipelined functional units in Figure 5. Non-pipelined functional units (e.g., Intel's divider unit) are also borrowable, but as soon as we receive a request from the owner thread, we abort the execution of the borrowing thread and immediately start execution of the owner instruction. Most of the functional units in the Intel and AMD processors are pipelined.

### 5.3.3 Cache-like Structures

Cache-like structures are a special type of stateful resource. In the context of Asymmetric SMT, the major difference between these and other stateful resources is that caches, once warmed-up, do not have empty entries. Structures that fit in this category are the micro-op cache, the private TLBs, and the private data and instruction caches.

For these structures, Asymmetric SMT cannot enable borrowing of blocks without leaving a visible effect. To address this, we introduce a mechanism that invalidates entries in the untrusted (owner) thread's partition. This invalidation mechanism must be deterministic and independent of any requests by the trusted thread. Therefore, it does not leak information.

Similar to some of the methods proposed in architecture literature on cache dead block prediction [90–92], we dynamically calculate the average reuse distance of cache blocks of the untrusted partition. Then, we multiply that average with a static parameter (*distance_coefficient* $> 1$) and use that as a threshold to distinguish between live and dead cache blocks. If the last access to a cache block is greater than the threshold, we invalidate that cache block so that it can be lent to the other thread. Again, only cache accesses of the untrusted thread can influence this invalidation, so it does not leak information about the access pattern of the other thread. We use a simple dead block detection mechanism that is only influenced by accesses, but more sophisticated dead block elimination methods can also be used [90–92]. Unlike other stateful resources, in caches returning a borrowed cache line does not incur any "squash" cost – it is a simple eviction. As soon as we receive an access from the owner thread, we invalidate one of the borrowing thread's cache lines (based on the cache replacement policy) and return it to the owner partition. For a write-back cache, we only allow a borrowed line to become dirty if the cache features a write-out-buffer (WOB) [93] – a buffer that handles the writes down to the memory hierarchy – so that we can guarantee the process of returning a dirty borrowed line is still instantaneous. In addition, we do not allow the number of dirty borrowed lines to grow larger than the size of the WOB.

#### 5.3.4 Overheads

Overall, we find that borrowing from stateless resources (functional units and fetch bandwidth) does not impose significant overheads on the processor pipeline. In terms of area overhead, we only need to (1) make sure that instructions are tagged with one bit of thread ID across the pipeline (already necessary for other reasons), and (2) add very simple logic that checks if borrowing of a specific resource is allowed in each cycle, i.e., it checks that the current instruction belongs to the borrowing (trusted) thread and also there is no demand for the resource from the untrusted thread. This simple logic, as shown in Section A.2, does not impose any overhead on the cycle time, and has negligible power and area overheads. For stateless resources, Asymmetric SMT only uses an unused resource which will be lost if not utilized by borrowing. Therefore, the performance effect will always be positive. Borrowing stateful resources, however, may require additional flushing if the untrusted thread requests a borrowed resource. While this flushing imposes performance overhead to the borrowing thread, our results show that the improved utilization brought by borrowing greatly outstrips the flushing costs.

Table 2: Architecture Detail for the Baseline x86 Core

| **Baseline Processor** | | | |
|---|---|---|---|
| Frequency | 3.3 GHz | I cache | 32 KB, 8 way |
| Fetch Width | 16 B | D cache | 32 KB, 8 way |
| Fetch Policy | IQ count | Retire Width | 8 uops |
| Issue Width | 8 uops | Decode Width | 5 uops |
| INT/FP Regfile | 186/144 regs | IQ | 97 entries |
| LQ/SQ Size | 64/36 entries | Functional | Int ALU(4), Mult(1), |
| ROB Size | 224 entries | Units | FPALU/Mult(2) |

#### 5.3.5 More than Two Threads

While SMT implementations with more than two threads are not common, our asymmetric SMT can be extended to those processors as well. In those cases, we can define a security lattice between trust domains and allow threads with higher trust levels to borrow from threads with lower trust levels. For example, a kernel thread can borrow from both a sandbox thread and a browser thread, while a browser thread can only borrow from the sandbox.

#### 5.3.6 HW/SW Interface

Asymmetric SMT requires knowledge of the trust level of active threads running on the SMT processor. We envision two possible modes of operation for Asymmetric SMT. In the first mode, Asymmetric SMT relies on existing privilege levels. That is, without any software change, Asymmetric SMT enables the kernel to borrow resources from a user thread, or let the hypervisor borrow resources from any of the guest threads. The second mode allows the software to specify more fine-grain trust levels for the SMT threads. Thus, Asymmetric SMT needs to add and maintain new control registers that represent the trust level of an active thread. Privileged software would update the control register via a privileged instruction (e.g., x86's *wrmsr*). The second mode also does not require extensive software changes as all the modifications required will be contained in the thread/process creation logic.

#### 5.3.7 Quality of Service (QoS)

In addition to the evident security use case, Asymmetric SMT can also be leveraged for better and more versatile performance isolation (i.e., better QoS guarantees). The OS could mark a latency-critical thread as a high security/priority thread. Asymmetric SMT, then, improves QoS by assigning more resources to the latency-sensitive job with guaranteed (performance) protection of other jobs.

## 6 Methodology

This section details the experimental methodology for performance evaluation of the proposed partitioning schemes, including the Asymmetric SMT architecture. We also discuss the methodology we use for our covert-channel characterization framework presented in Section 4.

For performance evaluation, we model our partitioning scheme and Asymmetric SMT architecture in the gem5 v20 architectural simulator [94]. We add full support for SMT in gem5's out-of-order CPU model. We choose the parameters of our baseline architectures to resemble an Intel Skylake, except that, for a more intuitive comparison of partitioning schemes, our baseline architecture dynamically shares all the pipeline resources between the threads. The other exception is that the assignment of the functional units to the execution ports are slightly different than Skylake and more closely resembles AMD processors where floating-point and integer units do not share a single port. Table 2 describes the detailed architectural configuration.

To characterize performance, we use the C and C++ benchmarks from the SPEC CPU2017 suite. These benchmarks are compiled at the -O3 optimization level using the LLVM compiler. Following the prevalent methodology for creating accurate and representative simulation points [95–97], we use PinPlay [98] and Simpoint [99] to select representative regions for simulation. For each of these benchmarks, we select the Simpoint region with the highest weight – the most representative region. We then make one multi-threaded checkpoint for every possible pair of the benchmark programs by combining their selected Simpoints. We run each pair twice: In the first experiment, we simulate until we complete 100 Million instructions from the first thread, then swap the threads and repeat. For example, we will run *lbm* and *perl* together twice. The effect of *perl* on *lbm* performance will be factored into the *lbm* bars in our graphs, and the effect of *lbm* on *perl* will appear in the *perl* bars.

The speedup numbers of different schemes are calculated as the ratio of the combined *instruction per cycle (IPC)* for each pair of the programs over the combined IPC of that pair of programs in the dynamically shared processor. Thus, for the Asymmetric SMT results, it accounts for both the sped-up trusted thread and the unaffected untrusted thread in the overall results. This is also equivalent to weighted speedup [100], a well-established performance metric for multiprogram workloads, where in this case the baseline is that thread's performance in a dynamically shared SMT processor. Weighted speedup more accurately reflects useful performance gains, and avoids over-rewarding speedup of high-IPC threads. For adaptive partitioning we use at least 100,000-cycle adaptation intervals, unless otherwise noted. The performance bars that represent Asymmetric SMT are the results of applying Asymmetric SMT together with adaptive partitioning.

In addition to the SPEC benchmarks, to evaluate Asymmetric SMT in a more realistic scenario, we model a setting that resembles the computation of web browsers: running untrusted Javascript codes on one thread and sensitive cryptography computations on the other. On the first thread, we run Javascript programs from SunSpider [101] benchmarks on Duktape [102] Javascript engine, and on the second thread, we run *RSA* and *AES GSM* benchmarks from Wolf SSL v4.5.0 [103].

We evaluate the performance of our proposed mitigations by applying them to a wide range of pipeline resources, including the Instruction Queue, the Load Queue, the Store Queue, the integer and vector physical register files, the ROB (Adaptive only), the instruction and data TLBs, the instruction and data caches, Branch Target Buffer, fetch and decode

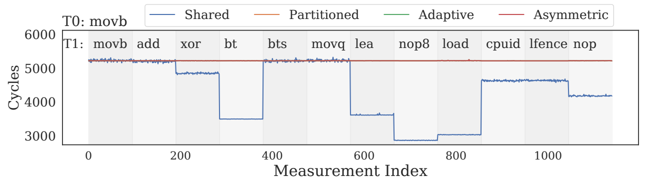


Figure 6: Covert Channels between a Spy (T0) and a Trojan (T1) Thread. In a fully shared pipeline the instructions executed on T1 have a clear effect on T0's execution time. *Partitioned*, *Adaptive* and *Asymmetric* are always constant and share the same straight line.

bandwidth, commit bandwidth, and the execution units.

We use Verilog HDL to implement different partitioning schemes on an example structure (Dispatch Unit). To that end, We use Synopsis Design Compiler Q-2019.12-SP5-3 with the 45 *nm* NanGate standard cell library [104] to synthesize and obtain timing, area, and power information. The results of this analysis are discussed in Section A.2.

For covert-channel characterization experiments we build our microbenchmarks on Agner's test infrastructure [105]. We run our experiments on various processors. Specifically, we use AMD Ryzen Threadripper 3960X (Zen2), Intel Xeon E3-1230 (Ivy Bridge), and Intel Core i7-6770HQ (Skylake). To measure the bandwidth and error rate of the covert channels, we transfer a pseudorandom bit sequence which is generated using a 15-bit wide linear feedback shift register (LFSR). This allows us to identify various errors that might happen during the transmission, including bit loss, multiple insertions of bits, or bit swaps [106]. To estimate the error rate, we use Levenshtein edit distance between the sent and received data for the pseudorandom bit sequence.

## 7 Results

This section characterizes our mitigation strategies. We first present the security evaluation, followed by performance.

### 7.1 Security Evaluation

To show the effectiveness of our mitigation strategies in stopping covert channels, we perform a study similar to Covert Shotgun [56]. In this experiment, a spy thread constantly executes one type of instruction and measures its timing. The trojan thread tries to send a signal by executing different instructions, thereby varying the contention on various pipeline resources. Figure 6 shows the results for just one instance of this experiment where the spy thread constantly executes the movb instruction. In a fully shared pipeline, the timings of the spy process can be clearly influenced by the trojan's instructions. An attacker, therefore, can pick any pair of instructions that show a different effect on the spy process to create a covert channel. When we enable any of our mitigation strategies, the spy thread timings become constant, effectively stopping all identified covert channels. For adaptive partitioning, all measurements are within one adaptation period. For Asymmetric SMT, the trust levels are set so that only the tro-

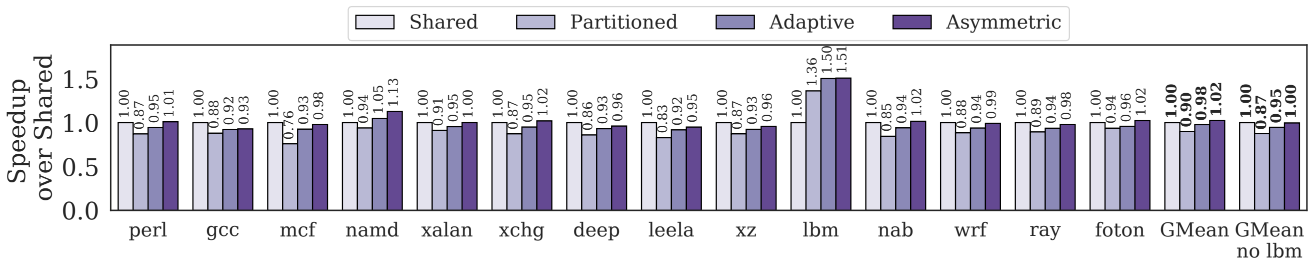


Figure 7: Performance of the Proposed Schemes.

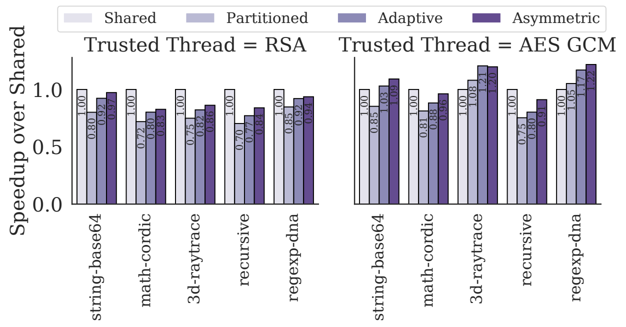


Figure 8: Running Trusted Cryptography Computation with Untrusted JavaScript Code.

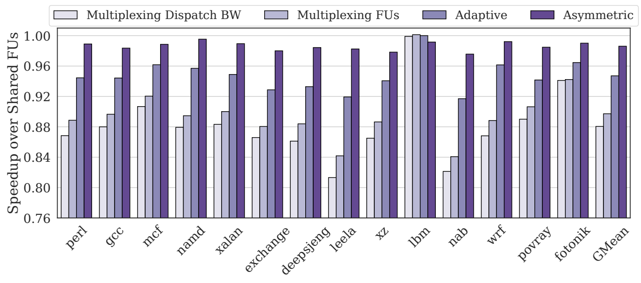


Figure 9: Partitioning Schemes for Functional Units.

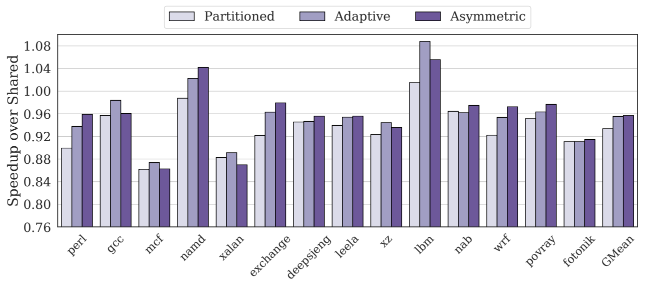


Figure 10: Partitioning Schemes for Fetch Bandwidth.

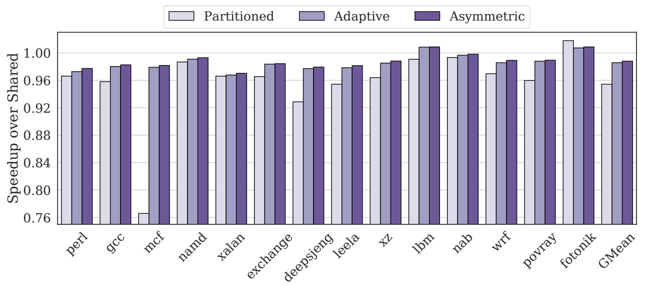


Figure 11: Partitioning Schemes for Caches.

jan thread can borrow resources. Further examination of the experiments shows that the covert channels in the fully shared pipeline are created by contention on mainly two resources: the fetch bandwidth and the functional units. We observe similar results when the spy uses different instructions.

Note that while this study shows that our mitigation strategies completely stop the covert channels that can be found with this approach, this test does not give complete coverage of all shared resources, particularly structures not documented by Intel or AMD. Also, these are the results of simulation (the only way to evaluate most new hardware mitigation techniques), and a real processor may contain other leaks not simulated.

## 7.2 Performance Evaluation

Figure 7 shows the results of our mitigation schemes applied to the pipeline resources mentioned in Section 6. Each bar represents the average results of running a benchmark on one thread with each of the other benchmarks on the other SMT thread. Static partitioning of the pipeline resources, as expected, imposes a significant performance cost. On average, it slows the execution by 10% compared to dynamically-shared resources. The performance overhead goes as high as 24% for some benchmarks (*mcf*). However, for one program (*lbm*), static partitioning significantly improves the performance. That is because *lbm* frequently exhausts the entire store queue on a dynamically shared pipeline, which causes the other thread to stall due to lack of store queue entries. In this case, *lbm* gets no benefit from more queue entries, and only gets extra interference by causing the other thread to get backed up. Therefore, statically partitioning the store queue achieves better performance for *lbm*, and our schemes further accentuate that advantage. Our adaptive partitioning reduces the performance overhead of partitioning to only 2% on average (5% if we ignore *lbm*), and consistently reduces the performance overhead of partitioning across all the benchmarks.

Asymmetric SMT further improves the performance and even provides a 2% speed-up over the shared pipeline (thanks again to *lbm*). But even excluding *lbm*, Asymmetric SMT almost fully restores the performance of a fully shared pipeline. These results show that opportunistically borrowing resources is highly effective at maintaining high utilization of partitioned resources. Note that, as mentioned in Section 6, for each pair of the benchmarks, we run the experiment twice. In each run, a different benchmark is considered as the trusted (borrower) thread in the Asymmetric SMT experiments.

Figure 8 shows the performance of the Asymmetric SMT architecture in a different, more realistic setting. On one thread, the SMT processor runs the SunSpider Javascript benchmark on Duktape engine, and on the other thread, it runs a trusted cryptography benchmark from the WolfSSL suit. This resembles computation that a web browser might perform. Asymmetric SMT allows the trusted threads (AES and RSA in this case) to borrow resources from the untrusted threads. The combination of our adaptive partitioning and Asymmetric SMT, on average, reduces the overhead of partitioning from 24% to 11% for RSA. For the AES benchmark, Asymmetric SMT not only completely restores the performance overhead of static partitioning, but also outperforms the fully shared baseline by 7% on average. The performance gain mostly comes from the pairs of benchmarks for which static partitioning performs better than the fully shared pipeline, such as *regexp-dna*. These benchmarks exhibit frequent resource full events (e.g., high number of physical register full in *regexp-dna*) that stall both threads in a shared pipeline. In such cases, partitioning allows one thread to continue execution and thus improves overall performance.

Next, we take a closer look at the performance of the proposed schemes by examining their effects on the individual pipeline resources. Among the resources that we partition, we find that the most significant contributor to the performance cost is the execution ports/functional units. Figure 9 shows the results of an isolated experiment where all resources are dynamically shared except the execution ports. We examine four partitioning schemes for the execution ports: two different static partitioning schemes as well as our proposed adaptive and asymmetric SMT.

*Multiplexing the dispatch bandwidth* refers to the method where we only dispatch instructions from one thread each cycle. It severely impacts performance. On average, it reduces the performance by 12%, compared to dynamically shared execution ports, and is as high as 19% for some benchmarks. The main reason for such poor performance is that, at each cycle, there are not enough instructions from only one thread to fully utilize the execution ports. Another scheme is *multiplexing individual functional units* instead of the dispatch bandwidth as a whole (described in Section 5.3.2). This is also the default static partitioning scheme used for the summary results of Figure 7. This improves the utilization of the execution ports over dispatch bandwidth multiplexing. However, the cost of static multiplexing of the functional units is still high (10%, on average) compared to shared execution ports. Adaptive partitioning is able to reduce that overhead to only 6%. Adaptive partitioning, even with extremely long adaptation intervals, is highly effective for the execution ports as different programs naturally exhibit different usage distributions for different functional units. Asymmetric SMT reduces this overhead even further to only 1%.

The next big contributor to the performance cost is the fetch bandwidth. Figure 10 shows the results of another isolated experiment where we only partition the fetch bandwidth and keep other resources dynamically shared. The baseline shared architecture uses ICount [12] to dynamically determine the best thread to fetch from. However, as Section 4.2 shows, this can be used to construct covert channels. One way to mitigate that is to use a strict round-robin scheme (*partitioned* bars in the figure) where the fetch unit alternates between the threads each cycle. A partitioned fetch policy does not have the ability to choose the fetching thread based on dynamic conditions each cycle; therefore, it loses 7% performance overhead, on average. We get small gains from adapting the partition (and so restore some small amount of the dynamic adaptation), then a bit more from asymmetric SMT, which enables us to retrieve some of the unused fetch bandwidth.

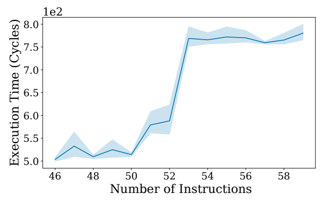
Figure 11 shows the effect of our partitioning scheme applied to the cache hierarchy (L1 instruction and data, and L2 caches). On average, the static partitioning of the cache sets into two equally sized partitions imposes 6% performance overhead. This is greatly influenced by one benchmark (*mcf*) that has a large cache working set. Adaptive partitioning reduces the overhead to only 2%. It allocates more cache ways in each set to the thread that shows more misses during the adaptation period. Asymmetric SMT further reduces the overhead to only 1%.

One interesting aspect of these results, compared to Figure 7, especially considering results not shown for other individual resources, is that the performance costs incurred overall are far less than the sum of the costs for individual mitigations. This is expected on a well-balanced architecture, as these processors are designed to be. In a well-balanced architecture, restricting one resource but not others will always make that resource a bottleneck. But in a (hypothetically) perfectly balanced architecture, restricting all resources may have no more negative impact than restricting one.

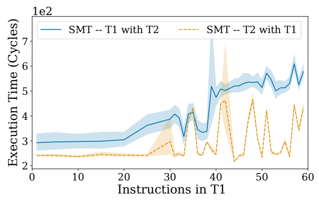
## 8 Conclusion

This paper provides the first comprehensive and exhaustive analysis of sharing-based security vulnerabilities in modern, high-performance SMT processors. This analysis shows that despite the fact that many resources are statically partitioned, there still remain many resources that are dynamically shared and present high bandwidth leakage channels. Among the channels identified are some previously unknown, including fetch bandwidth dynamic sharing and dynamically shared issue bandwidth, each enabling channels of over 500 Kbps.

This work also examines some novel, unified approaches to mitigation that can be applied throughout the pipeline. These provide high isolation between threads (allowing collectively a few bits of leakage over, for example, 100,000 cycles) while retaining most of the performance of a fully dynamically shared, insecure SMT implementation. Adaptive partitioning gets within 5% of shared SMT, and asymmetric SMT, which further enables unfettered performance of a trusted thread in the presence of an untrusted, all but eliminates the loss.



(a) Single Thread



(b) SMT

Figure 12: Reverse Engineering the Reservation Station Sharing Mechanism.

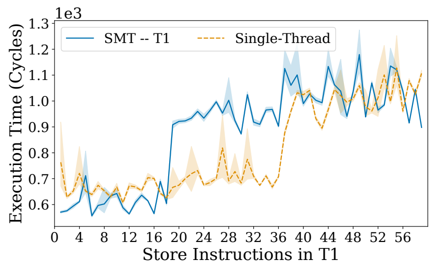


Figure 13: Reverse Engineering the SQ Sharing Mechanism. In single-thread mode, we see a spike in the execution time at 36, exactly the size of our SQ. In SMT mode, T2 only executes NOPs, but still causes T1's SQ to be halved.

It is common for SMT execution to be disabled in security-critical code, or in the presence of frequent untrusted execution streams. This work shows that SMT contention-based vulnerabilities can be reduced below the level of other known vulnerabilities, making SMT execution a viable alternative for secure execution. We do so while still preserving the bulk of the performance benefit of SMT.

## Acknowledgment

The authors would like to thank the anonymous reviewers for their helpful comments. This research was supported by NSF Grant CNS-1850436, NSF/Intel Foundational Microarchitecture Research Grants CCF-1823444 and CCF1912608, and DARPA contract HR0011-18-C-0020.

## A Appendix

### A.1 Extra Details on Covert Channels

Listing 4 and Listing 5 show the microbenchmarks that we use for isolating contention on the Reservation Station and the Store Queue.

Figure 12 shows the results of running Listing 4 (described in Section 4.5) in single-thread mode and in SMT mode on Ivy Bridge. In single-threaded mode, we observe a spike in the execution time if the code consumes more than 54 RS entries. In SMT mode, on one thread we run the same microbenchmark, while the other thread executes NOP instructions. We observe that Ivy Bridge limits the maximum number of allocated RS entries to 40 entries, and does not let one thread use all of the 54 RS entries.

Similarly, Figure 13 shows the results of running Listing 5 in both single-thread and SMT modes. This microbenchmark

Table 3: Delay, Area, and Power Results for Different Implementation of the Dispatch Unit.

| Module | Delay (ns) | Area ($\mu m^2$) | Static Power (mW) | Dyn. Power (mW) |
|---|---|---|---|---|
| Shared Dispatch | 0.955 | 7567 | 0.168 | 4.118 |
| Partitioned Dispatch | 0.951 | 7660 | 0.170 | 4.900 |
| Asymmetric Dispatch | 1.037 | 12147 | 0.284 | 6.508 |
| Mult 32×32 | 1.318 | 7597 | 0.163 | 6.835 |

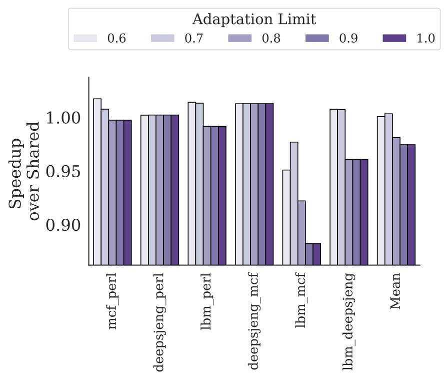
includes a series of stores which are dependent on the first long-latency load. When the number of store instructions exceeds the size of the store queue, the processor cannot issue the store instruction for which we do not have an available SQ entry, nor any of the following instructions. That is because even in an out-of-order Intel processor, the rename and allocation stages happen in order. If you run out of out-of-order resources such as physical registers, IQ, ROB, or SQ entries, the rename and/or allocation will be stalled. It is only *after* these stages that the processor can identify the dependencies between the instructions and dispatch the instructions out of order to the execution units. Thus, if a store instruction cannot proceed due to lack of SQ entries, the store and all younger instructions–independent of their type–will be stalled. As a result, in single-thread mode we see a spike in the execution time at 36 stores, which is precisely the size of the store queue in our Ivy Bridge processor. We also run the same code in SMT mode, along with another thread that does not consume any SQ entries, i.e., it only executes NOPs. In that experiment, we observe that the number of SQ entries available for T1 is exactly half of the SQ, suggesting a static partitioning scheme for the SQ.

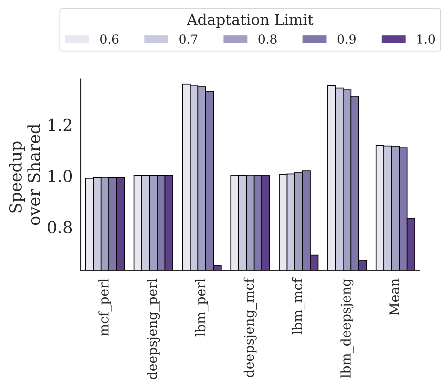
```
CLFLUSH [RDI]
CLFLUSH [RSI]
MFENCE
MOV RAX, [RSI] #Long-Latency Load
CMP RBX, RAX   #Waits in RS
...            #Consume N RS entries
CMP RBX, RAX
MOV RAX, [RDI] #Long-Latency Load
```

Listing 4: Microbenchmark for Making Contention on Reservation Station. When N is larger than available reservation station entries the two loads cannot be issued in parallel.

```
CLFLUSH [RDI]
CLFLUSH [RSI]
MFENCE
MOV RAX, [RSI] #Long-Latency Load
MOV [RBX], RAX #Consumes a SQ entry
...            #Consume N SQ entries
MOV [RBX], RAX #Consumes a SQ entry
MOV RAX, [RDI] #Long-Latency Load
```

Listing 5: Microbenchmark for Making Contention on Store Queue. When N is larger than available store queue entries the two loads cannot be issued in parallel.


(a) Physical Registers


(b) Store Queue.

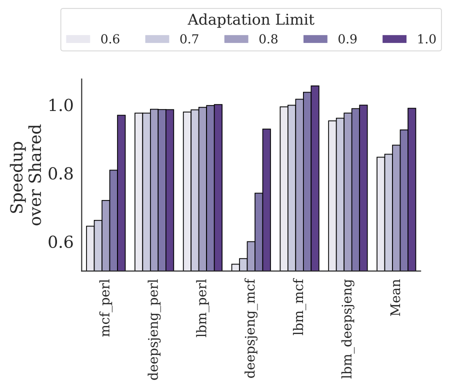
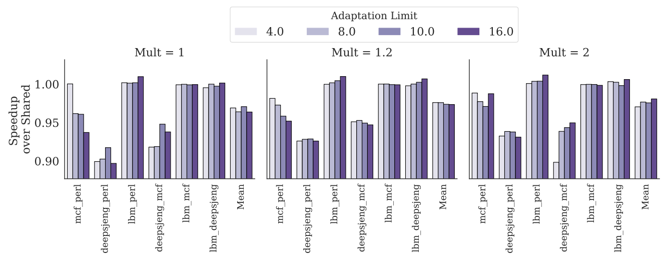

(c) Caches.

Figure 14: Parameter Search for Adaptive Partitioning of Three Example Stateful Resources.


(a) Functional Units.

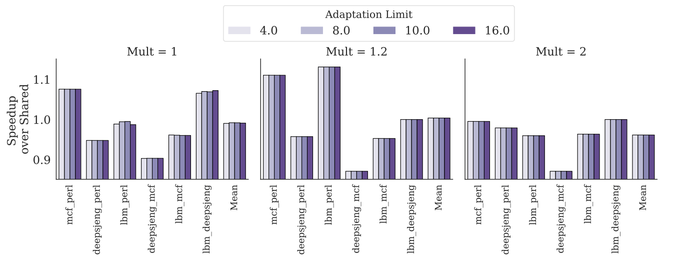

(b) Fetch Bandwidth.

Figure 15: Parameter Search for Adaptive Partitioning of Two Example Stateless Resources. We only increase the share of a thread, if the full counter of that thread is larger than *Mult* times of the counter of the other thread.

## A.2 RTL Model of Asymmetric SMT

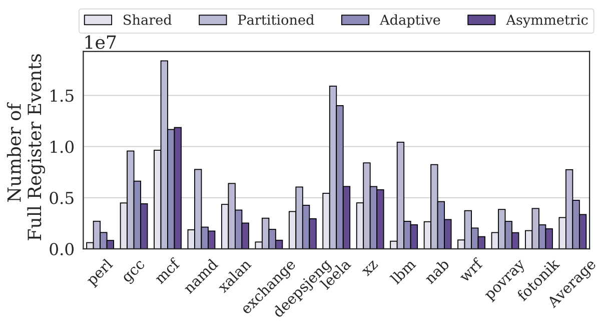
To fully evaluate the effects of resource borrowing on cycle time, power, and area (and to supplement our simulation-based performance results), we implement different partitioning schemes on an example resource in Verilog HDL. To that end, we choose the dispatch unit, the biggest contributor to the performance loss of partitioning and likely the most latency-critical, for which we implement three different sharing schemes: fully shared dispatch, partitioned dispatch, and Asymmetric dispatch. Fully shared dispatch assigns the functional units to the instructions marked "ready" in the queue, in a simple first-in-first-out fashion. The partitioned dispatch is similar to the shared, but it only dispatches instructions from a single thread each cycle. The Asymmetric dispatch is similar to the partitioned dispatch, but it assigns any unused functional units to the trusted thread. We then use the Synopsis Design Compiler Q-2019.12-SP5-3 with the 45 *nm* NanGate standard cell library [104] to synthesize and obtain timing, area, and power information.

Table 3 shows the post-synthesize analysis of different implementations of the dispatch unit. Our Asymmetric scheme increases the delay of the dispatch unit from 0.955 *ns* to 1.037 *ns*. However, this 8.6% extra overhead does not affect the processor's cycle time, as it is not enough to put the dispatch unit on the critical path of the whole processor core. As an example, we show that the (pipelined) integer multiplication unit has a longer delay. To see this, we implement a three-cycle multiplication module [107], imitating Skylake's three-cycle integer multiplication design. Our results show that the delay of our Asymmetric dispatch is still significantly smaller than the cycle time determined by the multiplier (the longest of the three stages); thus, our Asymmetric dispatch will not affect the cycle time.

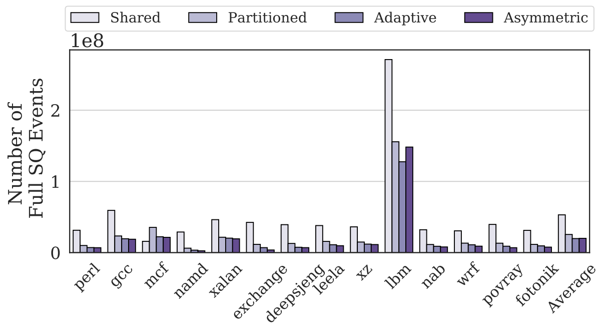
Asymmetric dispatch covers a 16% larger area compared to a shared dispatch unit. However, this is also not a matter of concern as the dispatch unit constitutes only a tiny fraction of a modern processor's die area. The Skylake core, for example, has an area of 8.73 $mm^2$ [108]. The asymmetric dispatch overhead, thus, will be only 0.051% of the core area (calculated conservatively, not accounting for the technology node differences). Similarly, the power overhead is also not considerable compared to the total power consumption of an out-of-order core, which could be in the order of tens of Watts.

## A.3 Parameter Search for Adaptive

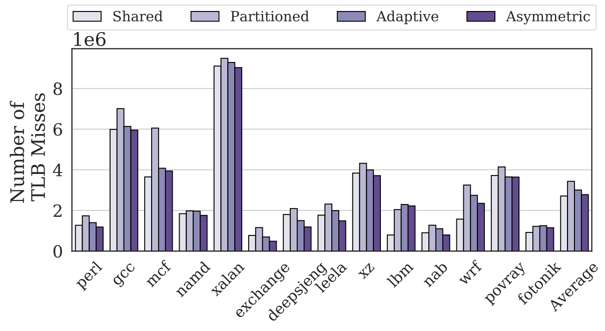
To select the best parameters for our adaptive partitioning and also to analyze the effects of each parameter on the performance, we conduct an exhaustive parameter search. Due to the combinatorial explosion problem, we cannot perform our parameter search study on all of the SPEC17 benchmarks. Therefore, we select four representative benchmarks: (1) a low ILP program that exhibits frequent memory accesses (*mcf*), (2),(3) two integer-heavy workloads (*deepsjeng*, *perl*), and (4) an FP-heavy workload (*lbm*). We then use all combinations of these benchmarks to conduct our parameter search. For each



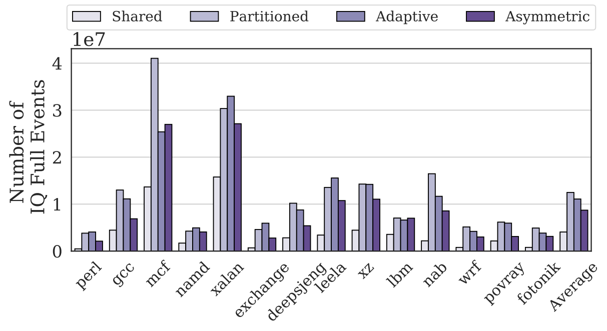
(a) Full Physical Register Events.



(b) SQ Full Events.



(c) Data TLB Misses.



(d) Instruction Queue Full Events.

Figure 16: Impact of Partitioning Schemes on Individual Resource

stateful pipeline resource, we examine five different adaptive limits (thresholds). For the stateless resources, we introduce another parameter, *Mult*. We increase the share of a thread, only if the full counters of that thread is larger than *Mult* times that of the full counters of the other thread – i.e., if *Mult* is 3, one counter must be more than 3X the other to cause a repartition. A higher *Mult* makes it more difficult to change the share of each thread. It particularly helps the resources where the cost of a wrong adaptation decision is high. In these experiments, we set the adaptation interval to $100,000$ cycles.

Figure 14a shows the results of our parameter search for the physical register files. One benchmark pair that is sensitive to physical register file partitioning is *mcf-lbm*. For this combination, a limit higher than 0.7 causes a significant performance cost. That is because when the limit is high, one thread can potentially take up most of the physical registers. On the other hand, a limit of 0.6 hinders our ability to fully adapt physical register file allocation to different execution phases. Therefore, we use a limit of 0.7 for the physical register file. Similarly, Figure 14b and Figure 14c show the results of our parameter search for two other example stateful structures–Store Queue and Caches. An adaptation limit of 1.0 for the SQ severely impacts performance, as it allows one thread to potentially take up all the SQ. This performance impact is reduced as we reduce the limit. Therefore, we choose the smallest adaptation limit (0.6) for Store Queue. For caches, however, the threads can fully take advantage of a larger adaptation limit, and the performance would improve as we allow the partitions to grow larger. Thus, we choose the adaptation limit of 1.0 for caches.

Figure 15 shows the results of the parameter search for two example stateless resources–fetch bandwidth and functional units. The adaptation limit for stateless resources determines the maximum number of consecutive cycles that one thread can hold the resource before it is assigned to the other thread. The results suggest that, for both of these resources, the best performance is achieved when Mult is set to 1.2, and adaptation limit to 8.

## A.4 Partitioning Effects on Other Resources

This section discusses the effects of our schemes on different stateful pipeline resources. Figure 16a shows the number of full register events for static and adaptive partitioning of physical register files (both integer and vector). That is the number of times the rename unit could not rename an instruction due to lack of physical registers. Partitioning, in general, significantly increases the number of full register events. However, the results show that adaptive partitioning can be highly effective at reducing the number of full register events. On average, it reduces the number of full register events by 63%.

Figure 16b shows the number of SQ full events. Dynamically sharing SQ results in a significant number of full events for all benchmarks. *lbm*, in particular, exhibits an exceptionally high pressure on SQ, resulting in $5\times$ more full SQ events than the average. Static partitioning of SQ reduces the number of SQ full events to almost half compared to shared, while our adaptive partitioning reduces that even further to 37%, on average.

Figure 16c shows the effects of adaptive partitioning on the data TLB miss rates. On average, our adaptive partitioning scheme reduces the number of data TLB misses by 14% compared to a static partitioning baseline.

Similarly, Figure 16d compares the number of full IQ events for different partitioning schemes. Static partitioning, in general, significantly increases the number of the IQ full events (by 3×). However, our adaptive partitioning exhibits 11% fewer IQ full events than static partitioning.

## References

[1] L. Bowen and C. Lupo, "The performance cost of software-based security mitigations," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2020.

[2] N. Jones, "How to stop data centres from gobbling up the world's electricity," *Nature*, vol. 561, no. 7722, pp. 163–167, 2018.

[3] L. Cheng, F. Liu, and D. D. Yao, "Enterprise data breach: causes, challenges, prevention, and future directions," *WIREs Data Mining and Knowledge Discovery*, vol. 7, no. 5, p. e1211, 2017.

[4] T. Armerding, "The 18 biggest data breaches of the 21st century," 2018.

[5] F. Mireshghallah, M. Taram, P. Vepakomma, A. Singh, R. Raskar, and H. Esmaeilzadeh, "Privacy in deep learning: A survey," 2020.

[6] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P)*, 2019.

[7] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack," in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2016.

[8] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *International Symposium on Microarchitecture (MICRO)*, 2016.

[9] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida García, and N. Tuveri, "Port contention for fun and profit," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[10] M. Taram, A. Venkat, and D. Tullsen, "Packet chasing: Spying on network packets over a cache side-channel," in *International Symposium on Computer Architecture (ISCA)*, 2020.

[11] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," *International Symposium on Computer Architecture (ISCA)*, 1995.

[12] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," in *International Symposium on Computer Architecture (ISCA)*, 1996.

[13] Google, "Product status: Microarchitectural data sampling (mds)," 2019. [Online]. Available: https://support.google.com/faqs/answer/9330250?hl=en

[14] M. Larabel, "Openbsd disabling smt / hyper threading due to security concerns," 2018. [Online]. Available: https://www.phoronix.com/scan.php?page=news_item&px=OpenBSD-Disabling-SMT

[15] Red Hat, "Simultaneous multithreading in red hat enterprise linux," 2019. [Online]. Available: https://access.redhat.com/solutions/rhel-smt

[16] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel Corporation, August 2011.

[17] D. M. Tullsen and J. A. Brown, "Handling long-latency loads in a simultaneous multithreading processor," in *International Symposium on Microarchitecture (MICRO)*, 2001.

[18] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher, "Microscope: Enabling microarchitectural replay attacks," in *International Symposium on Computer Architecture (ISCA)*, 2020.

[19] R. Spreitzer, V. Moonsamy, T. Korak, and S. Mangard, "Systematic classification of side-channel attacks: A case study for mobile devices," *IEEE Communications Surveys & Tutorials*, vol. 20, p. 465–488, 2018.

[20] I. Puddu, M. Schneider, M. Haller, and S. Čapkun, "Frontal attack: Leaking control-flow in sgx via the cpu frontend," in *USENIX Security Symposium (USENIX Security)*, 2021.

[21] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse engineering intel last-level cache complex addressing using performance counters," in *Research in Attacks, Intrusions, and Defenses (RAID)*, 2015.

[22] C. Maurice, C. Neumann, O. Heen, and A. Francillon, "C5: Cross-cores cache covert channel," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2015.

[23] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: a timing attack on openssl constant-time rsa," *Journal of Cryptographic Engineering*, vol. 7, no. 2, pp. 99–112, 2017.

[24] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: Practical cache attacks in javascript and their implications," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[25] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[26] P. Vila, A. Abel, M. Guarnieri, B. Köpf, and J. Reineke, "Flushgeist: Cache leaks from beyond the flush," 2020, [arXiv preprint arXiv:1409.0876].

[27] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[28] D. Cock, Q. Ge, T. Murray, and G. Heiser, "The last mile: An empirical study of timing channels on sel4," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[29] O. Aciiçmez, S. Gueron, and J.-P. Seifert, "New branch prediction vulnerabilities in openssl and necessary software countermeasures," in *IMA International Conference on Cryptography and Coding*, 2007.

[30] Y. Bulygin, "Cpu side-channels vs. virtualization malware: the good, the bad or the ugly," *ToorCon: Seattle, Seattle, WA, US*, 2008.

[31] D. Sullivan, O. Arias, T. Meade, and Y. Jin, "Microarchitectural minefields: 4k-aliasing covert channel and multi-tenant detection in iaas clouds." in *Network and Distributed System Security Symposium (NDSS)*, 2018.

[32] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks," in *USENIX Security Symposium (USENIX Security)*, 2018.

[33] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[34] M. Li, Y. Zhang, Z. Lin, and Y. Solihin, "Exploiting unprotected i/o operations in amd's secure encrypted virtualization," in *USENIX Security Symposium (USENIX Security)*, 2019.

[35] M. Kurth, B. Gras, D. Andriesse, C. Giuffrida, H. Bos, and K. Razavi, "NetCAT: Practical Cache Attacks from the Network," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[36] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: Exploiting speculative execution through port contention," in *ACM SIGSAC Con-*

*ference on Computer and Communications Security (CCS)*, 2019.

[37] O. Aciicmez and J. Seifert, "Cheap hardware parallelism implies cheap security," in *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2007.

[38] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

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

[40] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *USENIX Security Symposium (USENIX Security)*, 2018.

[41] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[42] H. Wong, "Measuring reorder buffer capacity," may 2013. [Online]. Available: http://blog.stuffedcow.net/2013/05/measuring-rob-capacity/

[43] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan, "Side-channel vulnerability factor: A metric for measuring information leakage," in *International Symposium on Computer Architecture (ISCA)*, 2012.

[44] C. Hunger, M. Kazdagli, A. Rawat, A. Dimakis, S. Vishwanath, and M. Tiwari, "Understanding contention-based channels and using them for defense," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[45] S. B. Dutta, H. Naghibijouybari, N. Abu-Ghazaleh, A. Marquez, and K. Barker, "Leaky buddies: Cross-component covert channels on integrated cpu-gpu systems," in *International Symposium on Computer Architecture (ISCA)*, 2021.

[46] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh, "Rendered insecure: Gpu side channel attacks are practical," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[47] H. Naghibijouybari, K. N. Khasawneh, and N. Abu-Ghazaleh, "Constructing and characterizing covert channels on gpgpus," in *International Symposium on Microarchitecture (MICRO)*, 2017.

[48] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on aes, and countermeasures," *Journal of Cryptology*, vol. 23, no. 1, p. 37–71, Jan. 2010.

[49] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of aes," in *the RSA Conference on Topics in Cryptology*, 2006.

[50] G. Saileshwar, C. W. Fletcher, and M. K. Qureshi, "Streamline: A fast, flushless cache covert-channel attack by enabling asynchronous collusion," in *Proceedings of the Twenty-sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[51] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *Computer Security Applications Conference (ACSAC)*, 2006.

[52] A. Moghimi, J. Wichelmann, T. Eisenbarth, and B. Sunar, "Memjam: A false dependency attack against constant-time crypto implementations," *Int. J. Parallel Program.*, vol. 47, no. 4, p. 538–570, Aug. 2019.

[53] X. Ren, L. Moody, M. Taram, M. Jordan, D. M. Tullsen, and A. Venkat, "I see dead μops: Leaking secrets via intel/amd micro-op caches," in *International Symposium on Computer Architecture (ISCA)*, 2021.

[54] O. Acıiçmez, c. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)*, 2007.

[55] O. Aciiçmez, c. K. Koç, and J.-P. Seifert, "On the power of simple branch prediction analysis," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2007.

[56] A. Fogh, "Covert shotgun," 2016. [Online]. Available: https://cyber.wtf/2016/09/27/covert-shotgun/

[57] B. Gras, C. Giuffrida, M. Kurth, H. Bos, and K. Razavi, "Absynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures," in *Network and Distributed Systems Security Symposium (NDSS)*, 2020.

[58] G. Saileshwar and M. K. Qureshi, "Lookout for zombies: Mitigating flush+reload attack on shared caches by monitoring invalidated lines," [arXiv preprint 1906.02362].

[59] M. Yan, J. Wen, C. W. Fletcher, and J. Torrellas, "Secdir: A secure directory to defeat directory side-channel attacks," in *International Symposium on Computer Architecture (ISCA)*, 2019.

[60] G. Dessouky, T. Frassetto, and A.-R. Sadeghi, "Hybcache: Hybrid side-channel-resilient caches for trusted execution environments," in *USENIX Security Symposium (USENIX Security)*, 2020.

[61] M. Taram, A. Venkat, and D. M. Tullsen, "Context-sensitive decoding: On-demand microcode customization for security and energy management," *IEEE Micro*, vol. 39, no. 3, pp. 75–83, 2019.

[62] M. Taram, A. Venkat, and D. Tullsen, "Mobilizing the micro-ops: Exploiting context sensitive decoding for security and energy efficiency," in *International Symposium on Computer Architecture (ISCA)*, 2018.

[63] M. Taram, D. Tullsen, A. Venkat, H. Sayadi, H. Wang, S. Manoj, and H. Homayoun, "Fast and efficient deployment of security defenses via context sensitive decoding," in *Government Microcircuit Applications and Critical Technology Conference (GOMACTech)*, 2019.

[64] I. Vougioukas, N. Nikoleris, A. Sandberg, S. Diestelhorst, B. M. Al-Hashimi, and G. V. Merrett, "Brb: Mitigating branch predictor side-channels." in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[65] S. Deng, W. Xiong, and J. Szefer, "Secure tlbs," in *International Symposium on Computer Architecture (ISCA)*, 2019.

[66] H. Cho, J. Park, D. Kim, Z. Zhao, Y. Shoshitaishvili, A. Doupé, and G.-J. Ahn, "Smokebomb: Effective mitigation against cache side-channel attacks on the arm architecture," in *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2020.

[67] Q. Xu, H. Naghibijouybari, S. Wang, N. Abu-Ghazaleh, and M. Annavaram, "Gpuguard: Mitigating contention based side and covert channel attacks on gpus," in *Proceedings of the ACM International Conference on Supercomputing (ICS)*, 2019.

[68] A. Purnal, L. Giner, D. Gruss, and I. Verbauwhede, "Systematic analysis of randomization-based protected cache architectures," in *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[69] L. Zhao, P. Li, R. Hou, M. C. Huang, J. Li, L. Zhang, X. Qian, and D. Meng, "A lightweight isolation mechanism for secure branch predictors," 2020, [arXiv preprint arXiv:2005.08183].

[70] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *International Symposium on Computer Architecture (ISCA)*, 2007.

[71] T. Zhang, Y. Zhang, and R. B. Lee, "Cloudradar: A real-time side-channel attack detection system in clouds," in *International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, 2016.

[72] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "Cached: Identifying cache-based timing channels in production software," in *USENIX Security Symposium (USENIX Security)*, 2017.

[73] M. Yan, Y. Shalabi, and J. Torrellas, "Replayconfusion: Detecting cache-based covert channel attacks using record and replay," in *International Symposium on Microarchitecture (MICRO)*, 2016.

[74] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, and A.-R. Sadeghi, "Dr.sgx: Automated and adjustable side-channel protection for sgx using data location randomization," in *Annual Computer Security Applications Conference (ACSAC)*, 2019.

[75] M. K. Qureshi, "New attacks and defense for encrypted-address cache," in *International Symposium on Computer Architecture (ISCA)*, 2019.

[76] A. Mambretti, M. Neugschwandtner, A. Sorniotti, E. Kirda, W. Robertson, and A. Kurmus, "Speculator: A tool to analyze speculative execution attacks and mitigations," in *Annual Computer Security Applications Conference (ACSAC)*, 2019.

[77] G. Saileshwar and M. K. Qureshi, "Cleanupspec: An "undo" approach to safe speculation," in *International Symposium on Microarchitecture (MICRO)*, 2019.

[78] S. Ainsworth and T. M. Jones, "Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state," in *International Symposium on Computer Architecture (ISCA)*, 2020.

[79] M. Taram, A. Venkat, and D. Tullsen, "Context-sensitive fencing: Securing speculative execution via microcode customization," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[80] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "Nda: Preventing speculative execution attacks at their source," in *International Symposium on Microarchitecture (MICRO)*, 2019.

[81] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative taint tracking (stt): A comprehensive protection for speculatively accessed data," in *International Symposium on Microarchitecture (MICRO)*, 2019.

[82] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *International Symposium on Microarchitecture (MICRO)*, 2018.

[83] D. Townley and D. Ponomarev, "Smt-cop: Defeating side-channel attacks on execution units in smt processors," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2019.

[84] G. Chen, W. Wang, T. Chen, S. Chen, Y. Zhang, X. Wang, T.-H. Lai, and D. Lin, "Racing in hyperspace: Closing hyper-threading side channels on sgx with contrived data races," in *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[85] P. P. Lai, E. Schuchman, D. Keppel, D. M. Khartikov, P. Xekalakis, J. B. Fryman, A. D. Knies, N. Neelakantam, G. Stellpflug, J. H. Kelm *et al.*, "Apparatus and method for efficiently implementing a processor pipeline," US Patent 10 409 763, Sep. 10, 2019.

[86] C. Percival, "Cache missing for fun and profit," 2005.

[87] G. V. team, "How v8 measures real-world performance," 2016. [Online]. Available: https://v8.dev/blog/real-world-performance

[88] B. J. Smith, "Architecture and applications of the hep multiprocessor computer system," *Real-Time signal processing IV*, vol. 298, pp. 241–248, 1982.

[89] M. Nemirovsky and D. M. Tullsen, "Multithreading architecture," *Synthesis Lectures on Computer Architecture*, vol. 8, no. 1, pp. 1–109, 2013.

[90] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency," in *International Symposium on Microarchitecture (MICRO)*, 2008.

[91] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving cache management policies using dynamic reuse distances," in *International Symposium on Microarchitecture (MICRO)*, 2012.

[92] Z. Hu, S. Kaxiras, and M. Martonosi, "Timekeeping in the memory system: Predicting and optimizing memory behavior," in *International Symposium on Computer Architecture (ISCA)*, 2002.

[93] H. H. Hum, "Dirty line cache," US Patent 6 078 992, Jun. 20, 2000.

[94] J. Lowe-Power *et al.*, "The gem5 simulator: Version 20.0+," 2020, [arXiv preprint arXiv:2007.03152].

[95] R. Sharifi and A. Venkat, "Chex86: Context-sensitive enforcement of memory safety via microcode-enabled capabilities," in *International Symposium on Computer Architecture (ISCA)*, 2020.

[96] G. H. Loh, "3d-stacked memory architectures for multi-core processors," in *International Symposium on Computer Architecture (ISCA)*, 2008.

[97] A. Akram and L. Sawalha, "A survey of computer architecture simulation techniques and tools," *IEEE Access*, vol. 7, pp. 78 120–78 145, 2019.

[98] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs," in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2010.

[99] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program phase analysis," *Journal of Instruction Level Parallelism*, vol. 7, no. 4, pp. 1–28, 2005.

[100] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[101] SunSpider, "Sunspider javascript benchmarks," 2020. [Online]. Available: https://webkit.org/perf/sunspider/sunspider.html

[102] Duktape, "Duktape javascript engine," 2020. [Online]. Available: https://duktape.org

[103] wolfSSL, "wolfssl cryptography librar," 2020. [Online]. Available: https://www.wolfssl.com/docs/benchmarks/

[104] Nangate open cell library. [Online]. Available: https://si2.org/open-cell-library/

[105] A. Fog, "Test programs for measuring clock cycles and performance monitoring," 2020. [Online]. Available: https://www.agner.org/optimize/

[106] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[107] R. Aneesh and S. K. Mohan, "Design and analysis of high speed, area optimized 32x32-bit multiply accumulate unit based on vedic mathematics," *International Journal of Engineering Research and Technology*, vol. 3, no. 4, 2014.

[108] WikiChip, "Skylake (client)," 2020. [Online]. Available: https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)#Core_2