Implementing Secure Shared Memory for Side-Channel-Resistant Enclaves

by

Miguel Gomez-Garcia

S.B., Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 2022

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2023

Authored by: Miguel Gomez-Garcia
Department of Electrical Engineering and Computer Science
May 12, 2023

Certified by: Mengjia Yan
Assistant Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by: Srini Devadas
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Implementing Secure Shared Memory for Side-Channel-Resistant Enclaves

by

Miguel Gomez-Garcia

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 2023, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

With the rise in cloud computing, it has become more critical than ever for remote users to get strong security guarantees to secure sensitive computation they run on untrusted machines. Enclaves or Trusted Execution Environments (TEEs) are a powerful trusted computing primitive that can address this problem; through carefully co-designed hardware and software mechanisms, enclaves enforce strong isolation and integrity properties. While many enclave implementations already exist, most do not consider the threat of microarchitectural side channels and transient execution attacks. And although one academic proposal – MI6 – has addressed this stronger threat model, these security guarantees often come at a cost of a more limited capability, as well as performance overheads. As a result, no industrial hardware vendor has made any announcement to include these attacks in their threat model.

This thesis presents research in improving the capabilities of side-channel-resistant enclaves through the addition of secure shared memory, providing a mechanism for enclave applications to communicate with outside processes while maintaining the same strong isolation security guarantees provided by MI6. This allows for the development of a wider range of enclave applications with a significant performance improvement compared to existing enclave communication mechanisms. We hope that this work will demonstrate that enclaves can maintain strong security properties while being able to run a wide range of expressive programs.

Thesis Supervisor: Mengjia Yan
Title: Assistant Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Srini Devadas
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I'd like to thank my co-advisors, Mengjia Yan and Srini Devadas, for guiding me through the chaotic yet captivating world of hardware security. I learned a great amount over the last year and a half with your help.

Additionally, I'd like to thank Jules Drean for being an amazing mentor and putting in an incredible amount of effort to lead us through the project while also making sure I had a clear understanding of everything involving our work throughout the past year and a half. I also want to thank Thomas Bourgeat for being a massive help for the many bumps we ran into along the way, and making himself available at even the most inconvenient times to help.

Finally, I want to thank my friends and family for being supportive of all my academic and career endeavors, as well as my interest in music, as I could not have come this far without them. Their valuable advice on everything ranging from academic matters to challenging life situations has helped carry me throughout my five years at MIT.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cloud computing has become increasingly common in recent years for a variety of reasons. Many programs, such as machine learning model training, need large amounts of compute power that users often don't have. Or for a more simpler reason, a user might simply want to avoid dealing with issues such as machine management and stability. In these cases, it becomes very convenient to use a third party machine not managed by the user, such as Amazon's AWS. However, by using these services rather than running the program on local machines, the user would have to place some amount of trust in the cloud provider to guarantee that the program's sensitive data is protected and that the correct program is executed without any modifications. But when dealing with sensitive data, this trust assumption is often not reasonable. In these cases, a realistic threat model might need to assume that the cloud provider and most of the computing stack, including the operating system, is malicious, threatening the security of the user's program.

In order to deal with such situations, *enclaves* or trusted execution environments (TEEs) were introduced. When a program executes inside of an enclave, it is isolated from the OS or other malicious programs, effectively protected against software attacks. This usually involves isolating a section of memory, a core and other resources for the trusted user program, which the OS and other programs cannot access. Examples of commercial enclaves include Intel's SGX [6][11] and ARM TrustZone [1]. However, these enclaves were found to be insecure against microarchitectural side-

channel attacks. These attacks use shared microarchitectural structures in the processor (such as caches [32][21]) as a medium to leak information from the victim process to the attacker process, even when the two are isolated at an architectural level (for instance, even when they don't share memory). Due to these attacks (and additionally, the difficulty of writing enclave-specific applications), enclaves haven't become the keystone of secure systems.

Nevertheless, one academic proposal, MI6 [2], has introduced hardware mechanisms to enforce strong microarchitectural isolation. To our knowledge, this is the first enclave work to address this strong threat model. Although secure, the enclave's capabilities were somewhat limited, and the set of possible applications was restricted, in particular due to one major limitation: the lack of shared memory with the outside world. Simply allowing shared memory with the OS would break the strong microarchitectural isolation between the trusted and the untrusted environments, opening the door for *transient execution attacks* such as Spectre [14], which exploit speculative execution gadgets in out-of-order processor cores and leak data to the attacker using side channels. However, without memory sharing, enclaves lose the capability to efficiently to communicate with the outside world. With no way for the OS to send data to the enclave or the enclave to send data out, the range of applications is limited to batch computations that can be fully contained within the trusted environment, with no external interactions.

This thesis presents the following contributions. First, we develop the first mechanism for secure shared memory, which is secure against Spectre-like transient execution attacks. We explore an initial (naive) stall-based approach to handling memory accesses under the shadow of a branch, as well as the final rescheduling-based approach. Second, we evaluate its performance compared to an insecure baseline implementing *insecure* naive shared memory, as well as a different primitive for data sharing, nicknamed *secure drawers*. We accomplish these evaluations by developing a series of example workloads of interactive enclave applications. Finally, we also explore additional optimizations to lower the performance impact of frequent enclave-SM context switches, improving the performance of applications where traps

are triggered frequently.

Since this work aims to improve the work presented by MI6, all of the hardware changes are built on RiscyOO, the same open-source RISC-V processor used in MI6. With these developments, we hope to show that enclaves are able to accommodate an even wider variety of programs while maintaining strong security guarantees.

This thesis is organized as follows. Chapter 2 presents a background on RiscyOO, microarchitectural side channel attacks, and enclaves. Chapter 3 provides a description our secure shared memory feature and context switch performance improvements implemented as part of Citadel, a secure processor based on MI6. A detailed performance and security analysis on these modifications is provided in Chapter 4, with a concluding statement in Chapter 5.

Chapter 2

Background

This section introduces concepts relevant to the implementation described in Chapter 3. This includes an introduction of RiscyOO (the processor base for this work), a short overview on microarchitectural timing side channel attacks, and an overview of enclaves including MI6, a previous enclave implementation that this thesis builds upon.

2.1 RiscyOO

The base processor for this work (as well as MI6) is RiscyOO [8], an out-of-order RISC-V processor developed at MIT. The processor is written in Bluespec SystemVerilog (BSV), and uses a framework called Composable Modular Design (CMD) [33]. This framework allows very high flexibility in using the different modules of the design, and is based on the modules performing actions atomically, enforced by the BSV compiler.

In order to achieve high performance, most modern processors — including RiscyOO — are out-of-order, which means that instructions can be executed outside of program order for a more efficient use of resources. This requires some additional microarchitectural structures to ensure proper behavior. Two important structures that are relevant to this work are the *reorder buffer (ROB)*, which tracks when an instruction is ready to commit, and the *reservation station (RS)*, which tracks when an instruc-

tion is ready to execute. Together, the reorder buffer and the reservation stations ensure that instructions are executed when their operands become available, as long as it does not contradict program order. They are also responsible for killing mispredicted instructions and committing right path instructions. After leaving the RS, the instruction enters its corresponding pipeline. In this work, we mainly concern ourselves with the memory execution pipeline, which handles the address translation of memory accesses before handing off the request to the Load-Store Queue.

2.1.1 Reorder Buffer

The *reorder buffer (ROB)* is a circular buffer used to track the status of different instructions throughout the different execution pipelines. After an instruction exits the rename stage, it is then inserted at the tail of the ROB. Every cycle, the different execution pipelines notify the ROB when an instruction has finished executing, and the instructions' statuses are updated in the ROB after they are committed. To guarantee correctness, the effect of an instruction is permanently committed only once it reaches the head of the ROB (guaranteeing that all previous instructions in the program have been committed as well).

2.1.2 Reservation Station

The reservation stations are used to track when an instruction is ready to be executed. There are multiple reservation stations for the different execution pipelines: ALU, FPU, and the memory instructions. Instructions are inserted into their reservation station after the rename stage (simultaneously with the ROB), with entries including information about the required operands for execution. Every cycle, the entries in the reservation stations are updated to reflect if their operands are ready (that is, if dependent register values have been computed by previous instructions). Following this, the scheduler selects, for each reservation station, the oldest instruction that is ready to execute (has all of its required operands), and issues it for execution to its corresponding execution pipeline.

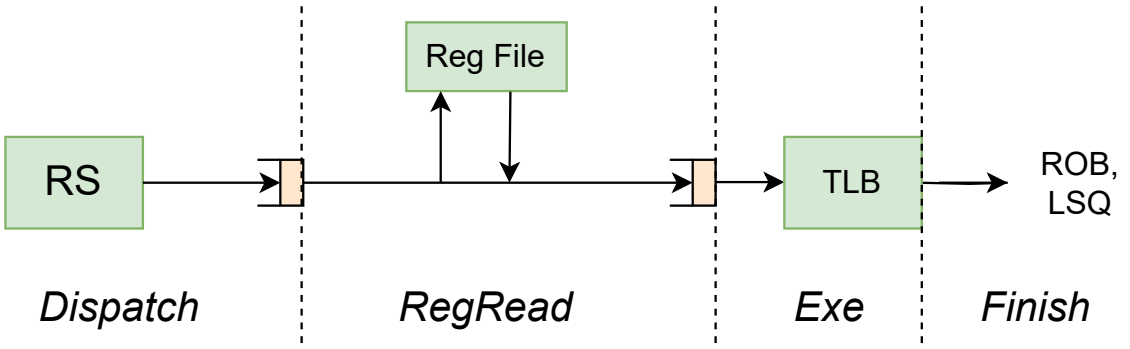

Figure 2-1: The Memory Execution Pipeline

2.1.3 Memory Execution Pipeline

The memory execution pipeline is responsible for the translation of memory requests prior to them being issued from the Load-Store Queue (LSQ), as well as detecting memory-related traps. The pipeline is divided into four stages separated by FIFO queues. The Dispatch stage first reads and dequeues a memory request from the RS as soon as it indicates that one is ready for translation. Since the logic for dequeuing from the RS is complex, this requires its own stage. The data inserted into the pipeline includes its LSQ tag, operand physical register indices, ROB tag, and some extra bits to keep track of speculation data. The Register Read stage then reads the required operands from the register file (memory address and data value in case of a store), reading bypass values if necessary, and inserts these values into the pipeline to replace the physical register indices. The Execute stage issues the request for translation, sending it to the Translation Lookaside Buffer (TLB), a form of cache for address translation. Prior to translating, the TLB checks if the address to be translated lies within the protection domain (further covered in section 2.4.2). The L2 TLB will perform a page table walk if the entry is not found in a TLB entry. Finally, the finish stage will retrieve the result from the TLB, check for any misalignments and faults, update the LSQ with the new physical address, and update the ROB with any relevant status bits needed. From here, the LSQ takes over to issue the load/store to the memory subsystem, and then stores the value in the register file in the case of a load.

2.2 Microarchitectural Timing Side Channel Attacks

In recent years, side channel attacks have become a prominent security concern and have been increasingly studied. Side channel attacks use shared microarchitectural structures to leak a secret from the victim's security domain to the attacker's, rather than attempting to directly access the secret by exploiting a software bug. Within the victim's security domain, a *transmitter* encodes the signal to be leaked and modulates the channel, for instance, by causing the eviction of a cache line. Depending on whether the channel is *stateless* or not, the effect on the channel might only be observable while the modulation is occurring. In the attacker's security domain, a *receiver* observes the effect of the channel modulation and decodes the signal. There is a wide variety of structures that can be used for these attacks, such as the cache [32][21], DRAM [12] [24], and even the power supply [15], which can emit high-pitched noises when the CPU is under heavy load. However, in this work we only focus on microarchitectural side channels. One of the most commonly exploited side channels is the cache subsystem, as it is relatively easy for the attacker to manipulate.

2.2.1 Cache Attacks

There are many different approaches to launching a cache side-channel attack[21][32]. One prominent example of these attacks is Flush + Reload, which can be mounted when the attacker shares memory with the victim. The attacker first *flushes* the cache line(s) that it interested in monitoring, such as by using the `clflush` in x86. This guarantees that the next access to the line will have a long latency, as it has to read from main memory. After waiting for a period of time, the attacker accesses the memory location that it is monitoring. This is the *reload* step. If the victim accessed the cache line during the waiting period, the attacker will observe a short latency as the line has been brought back into the cache. This attack can be used in multiple ways, for example, to determine if/how a victim executes code at certain addresses, or to infer a secret value based on the specific cache set the victim accesses, as shown below.

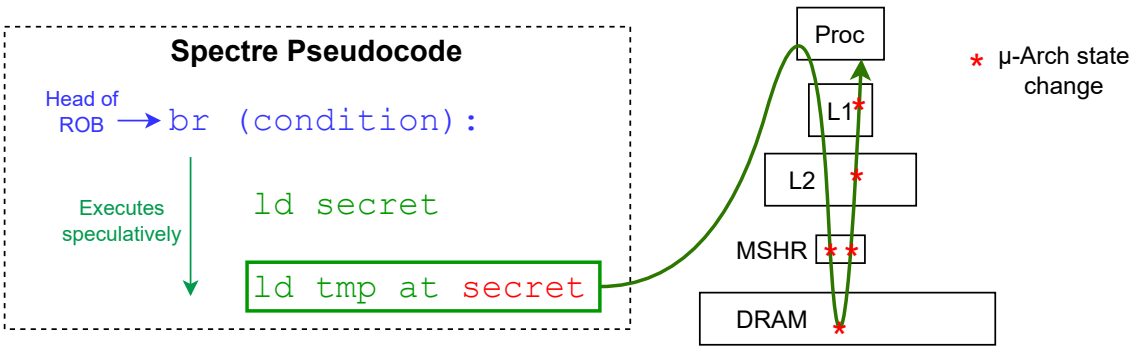

Figure 2-2: Spectre Pseudocode

2.2.2 Spectre and other Transient Execution Attacks

In out-of-order processors, an instruction can be executed speculatively even if it is not part of the program's correct execution flow and will not be committed to the architectural state. For example instructions after a branch, a return instruction or an instruction that will trigger an exception, are able to execute before the correct execution flow is determined, for instance, before the branch condition is resolved. If the branch resolves to opposite path than what was predicted, the architectural effects of speculative instructions can be discarded, and program flow continues as if they had never been executed in the first place. However, even though the architectural state is unchanged, there are usually still measurable effects on some microarchitectural structures, such as the cache.

Transient execution attacks are a form of attack that exploit this speculative execution mechanism in order to leak secret data. By training one of the many predictors in the processor to predict in a particular manner and then intentionally causing it to mispredict and speculate, the attacker code can leak information through a microarchitectural side channel that ultimately is not reflected in the architectural state. These attacks are not easily defended against, due to the fact that speculative execution is present in almost all commercial processors, and completely removing this feature would cause a drastic performance impact.

The first and most widely known form of transient execution attack is Spectre [14]. An example attack is shown in Figure 2-2. For this attack, a victim's secret is

leaked through the use of a *gadget*, which is a section of victim code with three main characteristics:

1. It trains a predictor in a certain way and then causes the processor to speculate and mispredict,
2. It accesses a secret value while speculating, and
3. It leaks the secret value through a side channel.

For example, the branch in the first line could be attempting to protect against illegal memory accesses by checking that the input only attempts to load data within a certain vaild address range. However, if the attacker calls this gadget repeatedly with inputs that satisfy the input check, the branch predictor eventually gets trained such that it predicts that the branch is not taken. After training the predictor, the attacker provides a specially crafted input that will try to instruct the code to access some victim secret. At this point, the processor, predicting that the branch is not taken, continues to speculatively execute the next instructions, which will load the secret from the indicated memory location. Before the branch condition has been resolved, it then transmits the secret through a cache side channel by performing a second access whose address depends on the value of the secret. By the time the branch is resolved and the speculative instructions get squashed, the instructions have already speculatively loaded data into the caches. With this microarchitectural effect, the attacker is able to use Prime+Probe or some other form of cache side-channel attack to retrieve the secret's value, since the attacker can deduce the secret-dependent address at which the new data was stored.

Due to the huge performance benefits of speculative execution, it is present in virtually all desktop and server processors today.

2.3 Enclaves

For programs that handle particularly sensitive data, a major point of concern for developers is how to ensure that the data is not leaked to attackers. For example, if

users want to perform some remote computation (in the cloud) on sensitive data, we would like some guarantee that the data cannot be accessed by any untrusted entity such as the cloud provider.

To address these issues, Trusted Execution Environments (TEEs), also known as *enclaves*, were developed. Enclaves are a set of hardware and software mechanisms that create an environment where code is able to execute securely. The presence of the hardware mechanisms, such as an additional privilege level or cache partitioning, are critical for the security guarantee of strong isolation to remain even when the host machine's OS is compromised by an attacker.

Enclaves are not a new concept, as academic TEE proposals date back twenty years to XOM [19] and AEGIS [28]. Industry implementations also began as early as 2004 with Arm's TrustZone (which although not directly a TEE, provided hardware primitives to develop enclaves)[1], and continued with Intel's Software Guard eXtensions (SGX)[11] and AMD's Secure Encrypted Virtualization (SEV)[26], which each offer different implementations and benefits. However, the actual public adoption of enclaves has been slow for many reasons. First, developing and adapting software for enclaves is often not a trivial task. There is usually a significant amount of work and intermediary steps required in launching, initializing, and communicating with an enclave, which additionally hinders the expressivity of the code. Additionally, there is no standardized runtime or execution environment to allow even some of the most used libraries to run easily. Furthermore, the security of enclaves in recent years has been undermined, especially since the emergence of transient execution attacks, such as Spectre [5]. With new Spectre variants being discovered each year, it is difficult for hardware manufacturers to keep up with these vulnerabilities, and many have decided to not focus on addressing these attacks. Since 2021, Intel has discontinued SGX on their 11th and 12th generation Intel Core processors, but has kept the feature for their server processors.

However, in recent years an accademic proposal, MI6 [2], was developed, showing that TEEs that provides strong microarchitectural isolation are feasible while implementing several ideas from Sanctum [7], including the use of a security monitor.

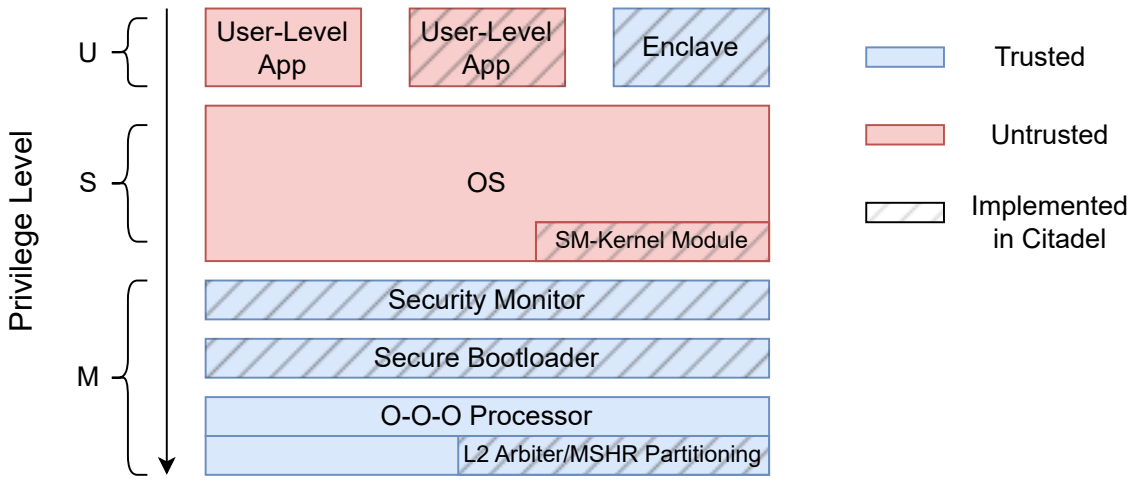

Figure 2-3: Citadel Computation Stack

2.4 MI6

Presented in 2019, MI6 builds off the RiscyOO processor (see Section 2.1), and provides the base for the rest of the modifications for this work. MI6 provides a secure enclave implementation with *strong microarchitectural isolation*, a property that states that the enclave is completely isolated from the rest of the machine at the microarchitectural level. In particular, any attacks that require the attacker and the enclave to be collocated on the same machine are part of the threat model while attacks that can be mounted by an attacker located on a different machine than the enclave (for instance, classical timing attack exploiting termination time) are not covered by the threat model. This guarantees that the processor is protected against side-channel attacks, since a remote attacker cannot interact with the microarchitectural structures that can leak data (apart from using the public-facing enclave API). MI6 achieves this by making sure resources between enclave and non-enclave processes (including the OS) are isolated, either spatially or temporally. Additionally, the OS can only interact with the enclave through the use of the API provided by the *security monitor*, a trusted piece of software that runs at a higher privilege mode.

2.4.1 Security Monitor

The security monitor (SM) [18] is a piece of software that is part of the Trusted Computing Base (TCB) of the enclave platform. The main purpose of the SM is to create a link between the high-level software security policies and the hardware level invariants. This involves acting as a liaison between the untrusted processes and enclaves, providing an API to create, initialize, and manage enclaves. In addition, the SM intercepts interrupts before they reach the OS and implement trap handlers to process interrupts and exceptions in the enclave.

Before loading the security monitor, the processor's bootloader runs a process to create a certificate that identifies the SM binary used. This certificate can then be used in an attestation protocol to prove the SM integrity to the user. Therefore, the SM cannot be compromised by the attacker, and is given a higher set of permissions than the rest of the software running on the processor. The SM runs at the highest privilege level (machine mode), which is higher than the OS. This allows the SM to enforce isolation, for example, by setting up the hardware mechanisms to prevent OS-accesses to enclave memory. In addition, in order to avoid leaking secrets by sharing the SM with untrusted processes, each enclave is also provided with its own private copy of the SM. Strong isolation between the OS and the SM is also enforced on context switches through the flushing of the microarchitecture. When this occurs, all of the different microarchitectural structures (caches, branch predictors, TLBs, etc.) are purged by the hardware prior to transferring control in order to avoid information leakage.

2.4.2 Enclave Page Table

Another important mechanism implemented by MI6, and borrowed from Sanctum [7], is the usage of enclave-private page tables. In processors with virtual memory, page tables are sections in memory that describe the mappings from virtual to physical memory. Whenever a process tries to access/write data at a certain address, the address needs to be translated from virtual to physical memory, which is accomplished

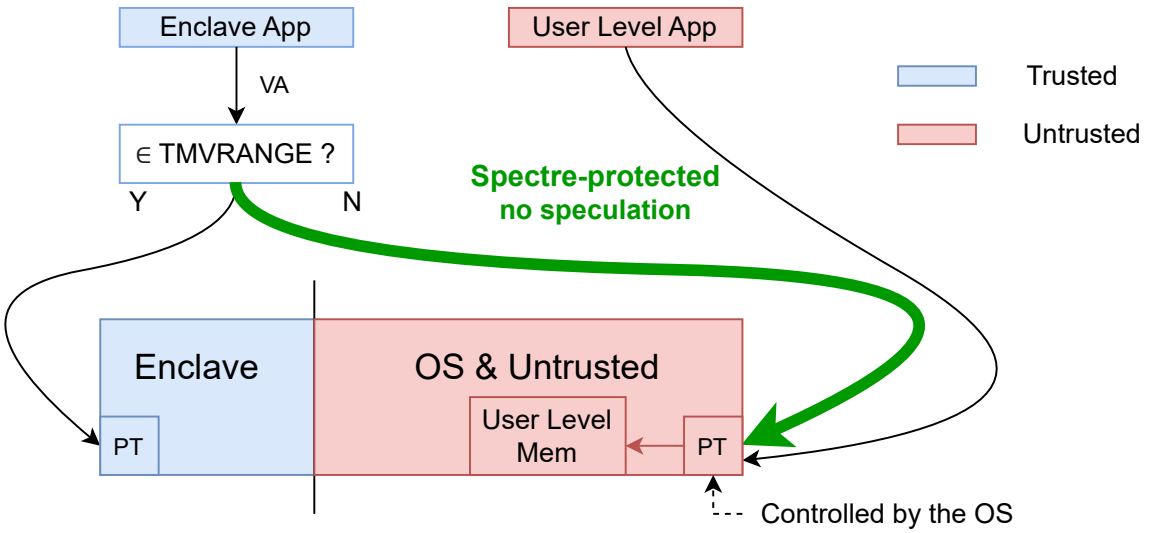

Figure 2-4: Dual Page Tables

through a *page table walk*. The output from this process is a physical address that can be used directly by the memory controller to perform the memory operation.

However, page tables can also leak information through their own memory access patterns. By observing which addresses get translated during the execution of the victim program, the attacker can deduce information about a secret if it causes memory accesses at specific addresses as a result [31]. In order to prevent this, each enclave has full and exclusive control of its own private page table located within enclave private memory, while the OS and other user-level processes use the OS page table, which is under control of the OS.

Additionally, there are two special CSRs, `evbase` (enclave virtual base) and `evmask` (enclave virtual mask), which are used to determine if a memory access should be permitted. If the request originates from enclave code, these two CSRs can be used to calculate a range of virtual addresses that are deemed trusted for the enclave to access (i.e., lie within enclave memory). If the memory access is trusted, the translation continues as expected, otherwise a fault is triggered and the address translation is not allowed to continue. Together, this security check and the isolated page tables guarantee that a process external to the enclave cannot deduce any access patterns for enclave code/data.

Chapter 3

Secure Shared Memory

3.1 Introduction to Secure Shared Memory

MI6 (Section 2.4) is the first proposal to address microarchitectural side channels and speculative execution attacks for enclaves. However, while MI6 provides a solid baseline for a secure processor, its capabilities are relatively limited when it comes to executing real-world programs. One important aspect of the design was that processes outside of the enclave, including the operating system, can only interact with the enclave via the API provided by the security monitor and a restricted message passing mechanism. However, while this API provides the ability to create, initialize, and manage enclaves, it does not include a way to provide data or receive output efficiently from the enclave while it is running.

This can be a major limitation for common, interactive programs that could benefit from running in an enclave, for example, the decryption of a stream of data such as TLS packets received over a high-speed network or a service that signs or verifies a series of incoming messages.

There are multiple ways to allow untrusted processes and enclave processes to share data with each other, of which two are relevant to this work. The first one is to use a *transactional or secure drawer* (SD) approach. This procedure begins with the sender process copying the data it wants to transmit into the memory region that will be used as a drawer. After the copy is done, the memory region ownership is

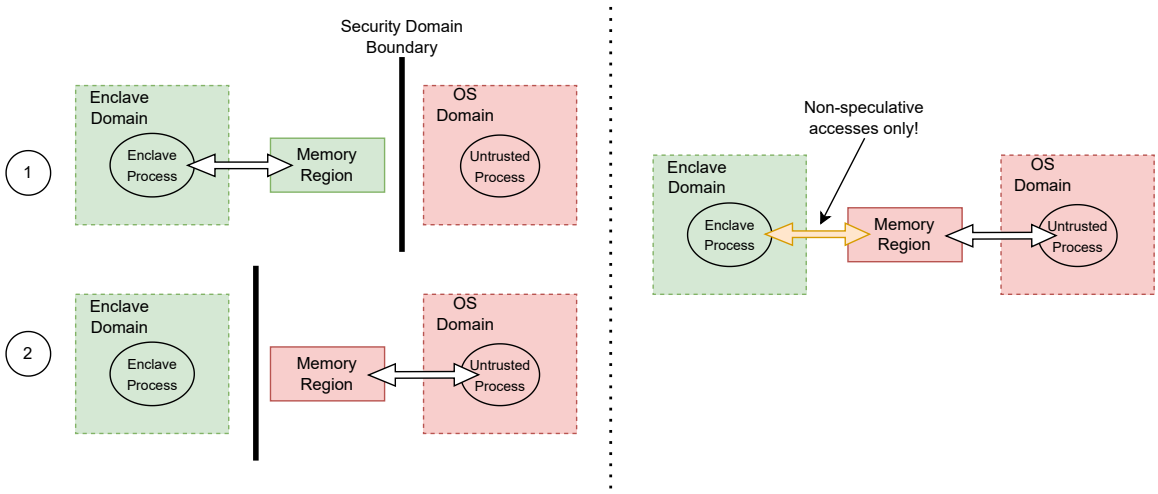

Figure 3-1: Secure Drawers (left) vs. Secure Shared Memory (right): SD's undergo a 2-stage process due to memory region ownership switching, while SSM does not.

transferred to the receiver's security domain using the relevant SM API calls (i.e., "pushing the drawer"). Finally, now that the receiver has access to this region of memory, it can read/write whatever it needs into the drawer, and corresponding data is then pulled back into the original security domain.

The second approach is to use *secure shared memory* (SSM). This involves implementing a mechanism that allows the enclave to access memory outside of its trusted domain under certain conditions. This allows both the enclave process and an untrusted process to write or read from the same locations with minimal overhead. However, in order to prevent leaking information, the processor pipeline needs to be modified to prevent speculative accesses from shared memory. In other words, the processor should only issue a memory access to shared memory if it is certain that the access will be executed given the current program flow. In order to guarantee this, several changes in the memory execution pipeline of the processor are required.

While SD only requires the addition of L2 flushing to MI6, there are several drawbacks to this implementation. First, the memory can only be pushed and pulled in chunks of 32MB, as this is the region granularity that is used to assign memory to different security domains. While this might be acceptable for large chunks of data that need to be processed, this poses a disadvantage for more interactive applications

that deal with light but frequent data streams. Since SSM does not require switching security domains, data can be directly accessed at any granularity. A second disadvantage is that data needs to be copied into the drawer and context switching is a lengthy process, including flushing the drawer's L2 region from the cache. Finally, a third disadvantage of SD is that it impacts the source code's expressivity, making the development of interactive enclave applications more difficult than it already is. With SD the whole process of pushing and pulling the drawer, with all the relevant SM API calls, needs to be outlined in the code, making it much more convoluted and verbose. With SSM, the mechanism is handled by the hardware and the software can simply perform loads and stores, allowing for simple, easy to write code.

3.2 Threat Model

The setup is as follows: a (remote) user wants to run a computation over some sensitive data inside of a (local) secure enclave. We trust the processor and hardware to be correctly implemented and bug free. We assume the presence of a hardware Root of Trust (RoT) that consists of a secure Read Only Memory (ROM) which contains the Secure Bootloader code that, we assume, cannot be altered. We also assume the presence of a Physical Unclonable Function (PUF) [10] to generate a chip-specific root key or secure non-volatile memory that stores the key. The rest of the trusted code base is minimal and includes the Secure Bootloader, the Security Monitor (together 15K lines of code (LOC)) and the enclave code. This means we neither consider attacks that exploit software bugs in the enclave's code nor protect enclaves from outputting their own secrets. Nevertheless, most of the software stack is considered compromised by the attacker, who can potentially run arbitrary malicious code. This includes user-level and other enclave programs collocated on the same machine, but also the OS and/or a potential hypervisor.

We are concerned with an attacker mounting software attacks from code collocated on the same machine as our enclave. This means we ignore any attacks requiring physical access to the machine such as physical (e.g., noise, electromagnetic field,

etc.) side channels. We also ignore side channels that can be mounted in software, but exploit the physical layer like power side channels. We do not consider Rowhammer attacks as part of our threat model. We consider denial-of service attacks *against* the OS and consider a specific threat model where user-level applications and malicious enclaves are allowed to attempt any kind of software-mounted attacks against the OS. However, we do *not* consider denial-of-service attacks *by* the OS.

We consider the usual threat model for secure enclaves and extend it to include collocated transient execution attacks and microarchitectural side channels. More precisely, we consider *software attacks where the attacker program needs to be running on the same machine as the victim.* This means we do not consider attacks that could still be mounted even if the attacker and the victim program were not collocated on the same machine, such as attacks exploiting the timing of public architectural events explicitly instructed by the programmer.

As a result, our platform aims at protecting enclaves against traditional side channel attacks such as cache attacks, but also (almost) all transient execution attacks as described in [4] (including Spectre [14, 29, 13, 5, 16, 22] and Meltdown [20, 27, 30] variants) However, we do not protect against timing completion attacks or attacks using the timing of IO like NETSpectre [25], as those could be mounted even by an attacker that is not collocated on the platform. We leave the mitigation of this kind of attacks to the enclave programmer.

3.3 Implementation Details

3.3.1 Identifying Insecure Accesses

In order to implement SSM, the processor must be modified in order to prevent speculative loads from being emitted. The challenge is that this also includes memory accesses that are part of the address translation, since accesses to the OS page tables are observable from outside the enclave. This mainly comprises of changes in the memory execution pipeline such that:

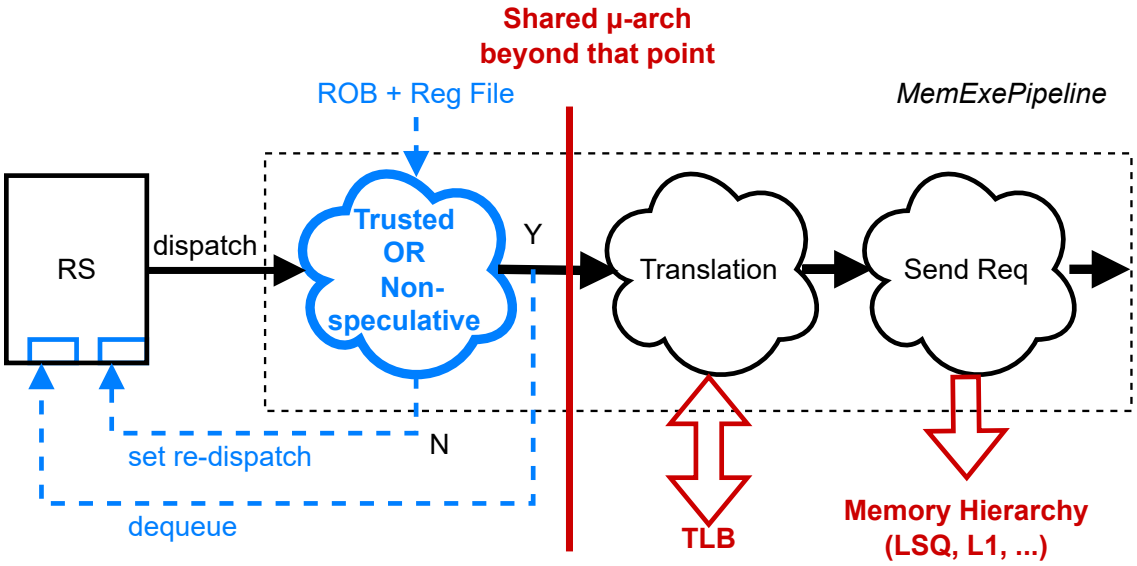

Figure 3-2: Modified MemExePipeline

1. When an enclave initializes a memory access, the memory execution pipeline now checks if the memory access corresponds to private enclave memory or insecure shared memory (using the `TMVBASE` and `TMVBASE` CSRs). If the instruction accesses shared memory, it will then check if the instruction is speculative by checking if it has reached the head of the ROB.

2. If it detects an insecure (shared) memory access that is *also* speculative, it should ensure that the instruction does not even trigger an address translation until it is made non-speculative.

There were two different approaches to this that were explored: a naive stall-based approach and a rescheduling-based approach.

3.3.2 Naive Stall-Based Approach

Logic to perform security checks and send back shared-memory speculative requests is added to MemExePipeline (cf. Figure 3-2). First, we add logic in the translation execution stage to determine if the request is accessing the enclave's trusted memory or untrusted shared memory using `TMVBASE` and `TMVMASK`. If the request is accessing untrusted memory, an additional check is performed to see whether the access is

speculative or not. The top entry from the ROB is compared to the tag of the request in the pipeline. If the tag matches, this indicates a *non-speculative* access, since all other previous instructions in the program have been committed. In this case, the request is dispatched to the next stage of the pipeline; otherwise, the insecure access must be processed differently in order to ensure it becomes non-speculative before translation.

The first approach to handling insecure (i.e. non-speculative and shared) memory accesses is to stall the memory instruction at the point where the insecure access is detected (and before memory translation) until the instruction is at the top of the ROB, indicating that it is no longer speculative. This occurs after reading the register file for the arguments of the access, and before translation is executed.

However, while this approach seems reasonable at first, it leads to cases of deadlock due to the instructions being issued out of order, as demonstrated by the following example. Consider the following memory instructions in program order:

1. **I1**: a load from the address at `r2` into `r1`,
2. **I2**: a store from `r1` to the address at `r3`, and
3. **I3**: a load from the address at `r4` into `r5`.

In this case, the instructions are first enqueued in order into the Reservation Station(RS). **I1** is issued first into the pipeline. Following this, the RS attempts to issue **I2**. However, **I2** has a dependency on **I1**, so it must wait until it completes. As a result, **I3** is issued instead. However, if `r4` contains the address of a shared memory location, **I3** will stall until it is determined non-speculative. But since **I2** comes before **I3** in program order, there is no way for **I2** to commit and allow the execution of **I3** because it is stuck behind it in the pipeline. This causes deadlock, and so a different approach must be taken to handle speculative memory accesses.

3.3.3 Rescheduling-Based Approach

MemExePipeline

The second approach to handling insecure memory accesses is through implementing a rescheduling mechanism in the MemExePipeline. Whenever an access to shared memory is found to be speculative, the request is squashed, and a signal is sent to the MemRS indicating that it should re-dispatch the request once it reaches the head of the ROB. The request may never reach the head of the ROB if there is rollback due to mis-speculation.

Decorrelating Request Dispatching and Request Dequeuing

Our mechanism requires the MemRS to retain the accesses to shared memory in order to re-dispatch them once they become non-speculative. The original design performs a MemRS data dispatch and dequeue simultaneously once the request enters the MemExePipeline. These two operations now need to occur at different stages in the pipeline: data is first dispatched from the MemRS and is dequeued in the translation execution stage if and only if the request is determined to be secure. We modify the interface between MemRS and MemExePipeline to provide the correct interfaces and logic. This includes augmenting the request data with the corresponding MemRS entry index.

MemRS

The MemRS logic is modified to implement re-dispatching of requests once they become non-speculative. First, several new status bits are added for each MemRS entry, specifically to track if a) a given request is still to be dispatched for the first time, and b) if a given request needs to be re-dispatched. The first bit (*to_dispatch*) is necessary since entries can now still reside in the MemRS in the time between being dispatched and dequeued (or being signaled for redispatch), and should not be selected for dispatch again. The second bit (*to_redispatch*) is set by the MemExePipeline when it squashes a speculative insecure request. We additionally add logic

to compare each MemRS entry to the entry at the head of the ROB, and raise the entry's *dispatch_ready* signal on a match. Finally, the dispatch logic is modified to use the new status bits. Rather than simply checking for the oldest valid entry whose arguments are ready, an additional condition is added to check if either *to_dispatch* is set for the entry or *to_redispatch* and *redispatch_ready* are set for the given entry.

Disabling the Mechanism for the OS

Blocking speculation on memory access can have a significant performance impact. We want to make sure that the OS is still allowed to speculate on its own memory accesses when not using the trusted memory mechanism. As a result, we disable our mechanism when `TMVRANGE` is set to a specific "empty range". Note that the OS could always be modified to take advantage of the trusted memory mechanism and protect itself from speculative side channel attacks.

3.4 Improving Context Switch Performance

3.4.1 A Conservative Flushing Policy

In the original MI6 design, context switches between an enclave process and the SM are always preceded by a flush of the microarchitectural structures, such as the cache and branch predictors. However, the Citadel software implementation is designed such that the enclave and certain parts of the enclave's private SM do not contain or touch any secrets that need to be hidden from each others. This allows for a performance improvement on context switches, as we can eliminate microarchitectural flushes when they are not necessary. In order to implement this, we introduce an extra CSR that enables/disables flushing on context switches. This gives the SM more flexibility to enable flushes only on context switches where this is necessary, improving the performance of most traps and SM-calls from the enclave.

3.4.2 Disabling Predictors

Another approach to improve context switch performance consists of disabling the predictor structures during context switches into the SM. These changes eliminate the need to flush these structures and additionally remove the penalty of having to warm up the predictors after returning to enclave code. This consists of disabling training and usage for three different structures in the fetch stage. The first is the Next Address Predictor (or BTB), which predicts the next PC based on the current PC. By disabling the BTB, we predict every future PC as PC+4 instead. The second structure is the tournament branch predictor. This structure is trained by observing the results of previous branches in the execute stage, allowing a prediction to be made in the fetch stage based on the current branch, local branch history, and global branch history. Disabling the branch predictor pauses the collection of local/global histories and branch result training. The last structure that is disabled is the *return address stack (RAS)*, which keeps track of source addresses whenever a jump with link register is taken, in order to accurately predict the destination of a later return. When disabled, pushing/popping from the RAS. Together, disabling all 3 of these structures prevents enclave code from being influenced to branch/jump to particular locations by actions taken by the SM.

Chapter 4

Security Analysis

4.1 Spectre Attack on Naive Shared Memory

In order to prove the security threat that Spectre-style attacks create on our platfrom, we attempt to mount a Spectre-style attack on both our secure shared memory implementation and a naive and insecure version of shared memory, where speculative loads are permitted. The victim consists of an enclave application running on core 0 that contains a Spectre gadget, and the attacker runs on core 1, exploiting this gadget and using Flush + Reload to observe the effects of the microarchitectural state.

The attack works as follows. The enclave contains a section of code vulnerable to a Spectre attack (also called a Spectre *gadget*), named `enclave_function` in the code above. This function first accesses data in an array located in enclave memory, and then accesses a dependent location in shared memory.

First, the attacker sends many valid requests to the enclave to execute `enclave_function` such that `idx` meets the conditions in the `if` statement. This will effectively train the branch predictor into entering the if statement and speculatively executing the following instructions without waiting for the branch condition to be resolved The attacker prepares the side channel that will be used to transmit the secret by flushing the L2 cache, the first step in a Flush + Reload attack. Following this, the attacker sends a request with an `idx` that is outside the bounds the enclave expects ($0 \leq idx < 6$).

```
// *** shared memory *** //
char *trans_array[256 * 0x1000];
```

```
// *** enclave private memory *** //
public_a = ['p', 'u', 'b', 'l', 'i', 'c'];

// *** enclave code *** //
void enclave_function(idx):
    if((0 <= idx) && (idx < 6)):
        tmp = public_a[idx]; // accesses secret
        res = trans_array[tmp * 0x1000]; // transmits secret
    return res;
```

Figure 4-1: Enclave's Pseudo-Code Snippet

Since the enclave's branch predictor has been trained to not take the branch, this request causes the enclave to mispredict the branch direction and execute the code inside the `if` block. At this point, the enclave speculatively accesses the public array with this `idx`, which will result in loading data within enclave memory that should not be accessed by the program as the index is out of the array bounds. By carefully crafting `idx`, the attacker is able to force the enclave to access and speculatively load a secret as the `tmp` variable. Once the value has beenn loaded, the processor speculatively executes the next instruction, loading a value from `trans_array` with the index determined by the `tmp` value (which potentially corresponds to an enclave secret). This load causes a line from `trans_array` to be loaded into the L2 cache. Finally, the attacker accesses `trans_array` one cache line at a time, and times these accesses using the `rdcycle` instruction. It will observe that one cache line loads faster than the others, indicating that this line was recently loaded into the L2 cache by the enclave. By computing which cache set this corresponds to, the attacker can deduce the value of `tmp`, an infer the potential enclave secret that was accessed speculatively and should have been inaccessible to the attacker.

We successfully mount the attack against the insecure shared memory implementation on the AWS F1 FPGA. The attacker code was able to discern which of the sets was recently accessed and leak the content of enclave memory and its secrets (indicated by around a 45-cycle access time for the accessed set, compared to about

170 cycles for the other loads from DRAM). However, we were not able to successfully mount the attack on our SSM implementation, which is expected given our security analysis.

4.2 Security Analysis

4.2.1 Strong Microarchitectural Isolation

MI6 already enforces strong microarchitectural isolation through partitioning and flushing (See Section 2.4). Therefore, we only need to update the security analysis to cover the new mechanisms we introduced.

We detail strong isolation arguments between each security domains, namely the enclave, the enclave mini-SM, the OS (untrusted software) and the main SM. We also consider the boundary between the OS and the bootloader.

Note that we are not susceptible to Meltdown-style attacks, as the processor does not speculate across privilege levels. For instance, we always perform security checks even on speculative memory accesses and do not forward illegal results in the pipeline. Because we do not consider multithreading across security domains, any state above the L1 cache cannot be shared concurrently between two different security domains. Finally, our processor does not include a prefetcher so we are not susceptible to Spectre variants exploiting them.

Enclave/OS Boundary Ignoring shared memory for a minute, our enclave benefits from the strong isolation property provided by MI6. Thanks to physical memory isolation and the partitioning of the memory hierarchy, the enclave and the OS do not share any microarchitectural state. Because memory and the LLC are still partitioned across context switches, their state is never shared. When freeing a memory region and the associated cache partition, the SM overwrites the entire region with 0's, effectively purging the LLC partition simultaneously. For the L1 and the core state, the SM purges every shared structure on context switch.

Nevertheless, our secure shared memory mechanism introduces shared state be-

tween the two security domains. The important point here is to remember that changes in the newly-shared microarchitecture cannot be performed speculatively: microarchitectural changes are tied to an observable architectural state change. Observations possible through the shared microarchitecture are thus strict equivalents to the observations possible through the accesses to shared memory. Because in our threat model we consider these accesses to be public, no extra information can be leaked through the shared microarchitecture. Secrets can still be leaked through the timing of these public interactions, but as we explain in Section 3.2 such attacks could be mounted even after placing the attacker and the victim on two different machines connected through a network. As a result, we effectively mitigate any transient execution attacks that exploit microarchitectural side channels to leak secrets, which include all Spectre variants besides NETSpectre [25].

Enclave/mini-SM Boundary In Citadel, we relax the conservative flushing policy of MI6. We consider the mini-SM as an extension of the security domain of the enclave. This means we will need to take extra care when interacting with other security domains, but for simple system calls that can be directly served by the mini-SM, we do not need to perform any flushing of the shared microarchitecture.

mini-SM/Enclave Boundary Here again, we consider the mini-SM as an extension of the security domain of the enclave. First, like in MI6, speculation is deactivated in the mini-SM, which means that even if it has access to the entire physical address space, it will not access any data speculatively. Second, we observe that the mini-SM does not perform any sensitive operations (such as keyed cryptography) and does not access any piece of information that should be kept secret from the enclave. As a result, we do not worry about side channels leaking information from the mini-SM to the enclave as no secret information is accessed. In particular, we do not need to flush shared state (L1, etc.) on these specific context switches.

mini-SM/OS Boundary Like in MI6, the mini-SM is hosted within the enclave memory region and its associated LLC partition. Its code is trusted, run non-speculatively, and only performs explicit memory accesses to that region with the exception of accesses to a small shared SM state. These accesses are public and can-

not be used to mount microarchitectural side channels. Because the mini-SM now shares microarchitectural sate with the enclave, we do not relax the flushing policy when context switching from the mini-SM to the OS and we benefit from MI6 strong isolation.

SM/OS Bootloader/OS Boundary Similarly to the mini-SM, the SM and the bootloader are trusted and do not speculate on memory accesses but unlike the mini-SM, they might touch sensitive information such as cryptographic attestation keys or enclave secrets. Therefore, like in MI6, the SM and the bootloader use the same dedicated private memory region and use their own private section of the LLC. For the bootloader, we flush all microarchitectural state after initialization. Nonetheless, we can relax the MI6 flushing policy on context switches after SM calls that do not touch sensitive information as no secret can be accessed covertly. Moreover, instead of flushing all data structures, we also make it possible for the SM to neither use nor train a specific structure to save the OS from a cold start. In this case, strong isolation is still enforced as no microarchitectural change is performed before re-entering the OS security domain.

Finally, note that in the case of the SM and the bootloader, even if this consideration is outside of our general threat model, we can also make sure their sensitive operations are constant time. Because their code is executed non-speculatively, and these sensitive functions are written in a constant-time fashion and microarchitectural state is flushed before their execution, the SM and bootloader are protected against timing attacks such as NETSpectre [25] that use attacker-provided inputs to carry out transient execution attacks and leak secrets through timing of public interaction (like the completion time of a system call).

OS/Enclave, OS/SM-mini and OS/SM Boundary We do not protect the OS against microarchitectural side channels.

4.2.2 Other security properties

Runtime Integrity Enclave integrity is guaranteed by our physical memory partitioning scheme that ensures that no adversary can modify any code or data for the

protected security domains (security monitor and enclaves).

OS Availability We make sure that at any time, the OS can send an inter-process interrupt to an enclave and reclaim its resources. Finally, a user-level application cannot directly access the security monitor, but can only interact with it through the SM kernel module. As a result, the security monitor is always available to the OS.

Authenticity Enclave authenticity is achieved by our remote attestation protocol. We use a simpler variant of the remote attestation protocol described in Sanctum [7] whose security is analyzed in [17].

4.2.3 Coherent Memory and L1 Side Channel

During our security analysis of the enclave, we uncovered another type of cache side channel originating from the fact that shared memory shares the same L1 cache as enclave memory . This involves exploiting the cache coherence mechanisms to allow the attacker to deduce if a shared memory lines are located in the enclave's private L1 cache, which can be used to mount a side channel attack as shown in the following attack example.

An attacker first makes use of some enclave API to have the enclave load enough shared memory data to fill the victim L1 cache (for instance by making it access a large array in shared memory). Then, the attacker will let the enclave perform some speculative memory accesses to some enclave memory addresses, which might depend on an enclave secret (see Transient Execution Attacks in Section 2.2.2). This will evict one of the shared memory lines from the L1 cache. The attacker then accesses the same array in shared memory that was loaded by the enclave, timing the accesses of the different cache lines. Here, the attacker will mostly see variations in access times, as the L1 cache of the victim will need to either invalidate the copy if the line in the L1 (in case of a write) or write-back the line to the L2 (in the case of a read). However, if the line has been previously evicted by the access to secret data, the writeback or invalidation is not required anymore and the access time will be significantly shorter which indicates to the attacker the address of the speculative secret-dependent access.

There are various ways to defend against this attack. The first is to partition the L1 cache such that shared and private enclave memory do not share the same ways. This way, enclave memory accesses will never be able to evict shared memory accesses, and vice versa. The second form of defense is to prevent L1 caching when accessing shared memory, which can be accomplished through Riscy-OOO's Direct Memory Access (DMA) mechanism. We defer the implementation of a defense mechanism to future work.

Chapter 5

Performance Evaluation

5.1 Testbench

Citadel's processor is a modified out-of-order 2-core Riscy-OOO machine building on top of modifications made for MI6 machine [2]. The parameters used for the cores and the memory subsystem are given in Figure 5.1. Citadel runs on an AWS EC2 F1 instance, equipped with a Virtex UltraScale+ FPGA. On this FPGA board, Citadel can be clocked at 30 MHz, which is similar to the baseline design. We use Connectal [3] to allow communication between the FPGA and the host computer; this is accomplished by a variety of BSV and C++ wrappers to make use of the Host-Target Interface (HTIF) debugging bus. We use the RISC-V tool chain with GCC 12.2.0, and compile all software with the -O3 flag. Unless mentioned differently, all performance numbers are collected by running the benchmark on the FPGA using performance counters and an average of ten runs.

5.1.1 Patching Connectal

Due to the Connectal repository being out of date, there were a couple of fixes to be made before the design was able to be tested on FPGAs. This included changes to the Connectal infrastructure such as updating Makefiles and scripts to make them compatible with newer GCC versions, updating DRAM wrappers to use the Bluestuff

Frontend	2-way superscalar, 256 entries BTB, 8 entries RAS, tournament predictor
Reorder Buffer	64 entries, 2-way insert/commit
Reservation Stations	2 stations for ALU (16 entries), 1 for FPU (16 entry), 1 for memory (16 entries)
L1 (I/D)	32KB, 8 ways, 8 outstanding requests
Load-Store Unit	24 LdQ entries, 14 StQ entries, 4 SB entries
LLC (L2)	10 cycles latency, 16 ways, max 16 outstanding requests
Memory	120 cycles latency (24 outstanding requests)

Table 5.1: Citadel microarchitectural configuration

AXI instead of Connectal AXI, and configuring the project to use the AWS F1 derived clock due to MMCM constraints being broken. Additionally, some of the drivers for the F1 instance were updated, as some of the new kernel headers were incompatible with the files meant for the old version, requiring source code edits.

5.2 Secure Shared Memory

In order to highlight the performance advantages of secure shared memory over secure drawers for data exchange with non-enclave processes, we would like to test on a workload that is highly interactive and simulates a real-world enclave application. Therefore, isolating a cryptographic library on an enclave is an ideal situation to evaluate our changes on.

We first write a wrapper around an ED2556 RISC-V library [23], which makes it possible to keep the keys used by the library inside of the enclave and for the library functions to be accessed in a Remote Procedure Call style (RPC). We add a queue to shared memory so untrusted applications can send requests to the library to generate keys (that will stay in enclave memory), or to sign messages using previously generated keys.

For each test, a non-enclave process sends 1000 RPC requests to the enclave to either sign or hash a randomly generated piece of data. In the case of the SSM design, the queue and the data are both located in shared memory, and the queue passes data pointers to the enclave. For SD, the data is first copied into the queue located in the

Normalized Runtime

	Q	Signature	Hash
Secure Shared Memory	1	1.024	1.320
	64	1.024	1.319
Secure Drawers	1	1.559	21.628
	64	1.018	1.746

Communication Overhead

Secure Shared Memory	1	1.5%	4.4%
	64	1.5%	4.5%
Secure Drawers	1	51.6%	1925%
	64	0.8%	86.7%

Table 5.2: Normalized runtime and communication overheads for the Secure Shared Memory and Secure Drawer mechanisms for signing or hashing a thousand random messages sent by the OS. We vary the size of the communication queue between the enclave and the OS.

drawer region of memory. After the queue fills up, the region is pushed into enclave memory to be processed and returned.

We ran the above test varying the queue size (1 individual request or 64 batched requests) as well as the cryptographic operation performed on the data, and averaged the results across 10 runs as the standard deviation was extremely low. We found that SSM provides a significant performance increase when compared to SD in situations where the communication mechanism makes up a significant portion of the total cycles. For example, when performing a hash operation, the enclave demonstrates a 1.32× normalized runtime compared to an insecure shared memory baseline, compared to SD coming in at 1.75×. This difference is only amplified further when reducing queue sizes; when the queue size is reduced to 1 (that is, the drawer is pushed/pulled for every element), the SD runtime increases to 21× of the baseline, compared to SSM which remains at 1.32×.

This performance difference, however, was less apparent when the enclave computation takes much longer to run compared to the communication mechanism. When performing a sign operation in our test, the enclave spends an overwhelming(99%) amount of cycles on this computation, and most of the performance improvement

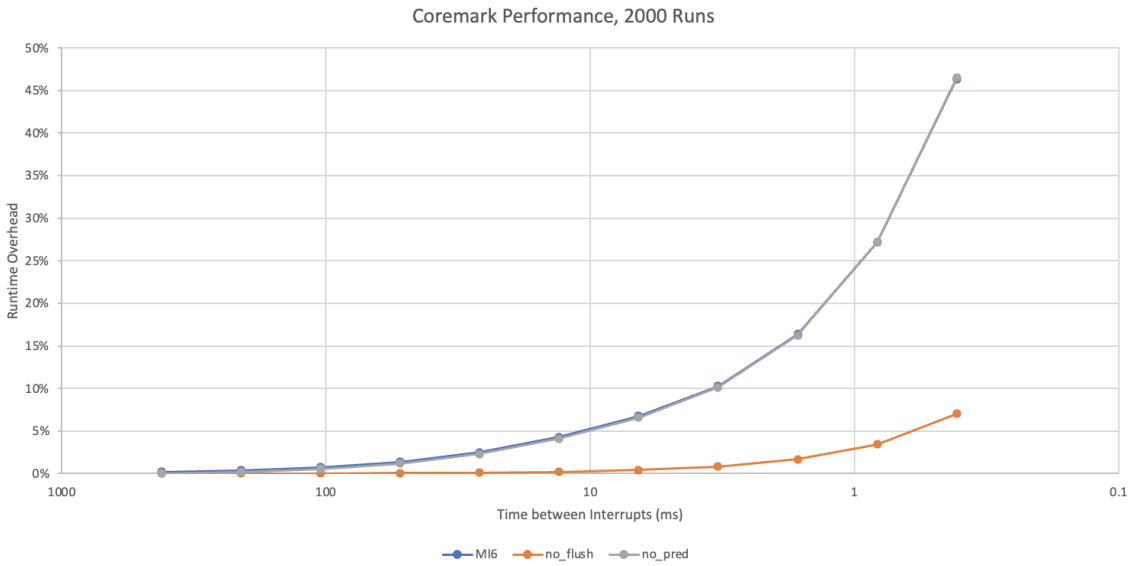

Figure 5-1: Modified MemExePipeline

that SSM provides is amortized. However, even in the most unfavorable situation, SSM still provided a comparable runtime to SD, and at best (with queue size 1) it still significantly outperformed SD.

5.3 Relaxed Context Switch Policy

In order to analyze the performance improvements for programs that observe frequent SM context switches, the processor needs to be evaluated on a test that incurs many traps and/or exceptions. We approach this by running a regular benchmark and enabling a timer interrupt on a specific time interval, triggering a trap and a context switch into the SM before returning to the main benchmark. The benchmark we selected for the test is Coremark [9], a small benchmark that mainly tests raw processor pipeline performance.

The results of the test are shown in Figure 5-1 normalizing the runtime to the lowest value, where context switches have essentially zero overhead due to being so infrequent. As the graph shows, loosening the flush requirements on SM context switches as described in Section 3.4 has a significant effect on decreasing the runtime overhead, showing up to a 25% performance improvement on 1ms interrupts.

However, the approach of disabling prediction structures in the SM did not have as great of an effect as expected. We believe this is because the cost of warming up the cache is not be as big as we originally predicted, either because 1) the branch patterns in Coremark are not particularly complicated, so branch prediction training is very quick, or 2) the branch predictor is not as accurate as we imagined, so little training is needed to get it back to its regular accuracy level.

Chapter 6

Conclusion

As the demand for remote data processing continues to increase over the near future, security concerns regarding private data will only become more common. And while enclaves provide a good solution to securing sensitive data on untrusted environments, there are still some issues that need to be addressed before they become the standard for secure remote computation. Chip makers will want to be able to offer provable isolation in enclaves to the widest range of applications possible, even under transient execution attacks, and developers will additionally be reluctant to adapt programs to enclaves if expressivity is severely impacted.

This thesis makes progress in addressing these issues by exploring methods to further improve security and performance in secure enclave systems. Through careful analysis, we have compared two feasible methods in approaching I/O between the enclave and the outside world: Secure Shared Memory (SSM) and Secure Drawers (SD). The analysis of SD and SSM has shown that SSM is a more flexible and efficient solution for handling data exchange. However, its implementation requires careful changes to the processor design to maintain data security, mainly by ensuring that speculative memory accesses from the enclave to shared memory are never emitted. Moreover, the thesis has highlighted the potential for performance optimization through adjustments to context switch procedures within the Security Monitor (SM), particularly through the elimination of unnecessary flushing.

In conclusion, this thesis contributes research to improve the usability of enclaves

as well as the range of feasible applications, providing valuable insights and practical strategies for both security and performance enhancements. We hope that the findings presented here will be used in the further development of enclave technology and will bring enclaves one step closer to being a widespread solution to secure remote computation.

Bibliography

[1] Tiago Alves and Don Felton. Trustzone: Integrated hardware and software security. ARM white paper, July 2004.

[2] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, and Srinivas Devadas. Mi6: Secure enclaves in a speculative out-of-order processor. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 42–56, 2019.

[3] cambridgehackers. Connectal github. `https://github.com/cambridgehackers/connectal`, 2023. Accessed on 05.10.2023.

[4] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security Symposium*, pages 249–266, 2019.

[5] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 142–157. IEEE, 2019.

[6] Victor Costan and Srinivas Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016.

[7] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 857–874, 2016.

[8] csail csg. Riscy-ooo github repository. `https://github.com/csail-csg/riscy-OOO`, 2020. Accessed on 05.10.2023.

[9] EEMBC. Coremark website. `https://www.eembc.org/coremark/`, 2009. Accessed on 05.10.2023.

[10] Charles Herder, Meng-Day Yu, Farinaz Koushanfar, and Srinivas Devadas. Physical unclonable functions and applications: A tutorial. *Proceedings of the IEEE*, 102(8):1126–1141, 2014.

[11] Intel. Intel® software guard extensions. `https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html`. Accessed on 05.10.2023.

[12] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 361–372, 2014.

[13] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[14] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[15] Paul Kocher, Joshua Jaffe, and Benjamin Jun. https://paulkocher.com/doc/differentialpoweranalysis.pdf. `https://paulkocher.com/doc/DifferentialPowerAnalysis.pdf`. Accessed on 05.10.2023.

[16] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael B Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *WOOT@ USENIX Security Symposium*, 2018.

[17] Ilia Lebedev, Kyle Hogan, and Srinivas Devadas. Secure boot and remote attestation in the sanctum processor. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 46–60. IEEE, 2018.

[18] Ilia Lebedev, Kyle Hogan, Jules Drean, David Kohlbrenner, Dayeol Lee, Krste Asanović, Dawn Song, and Srinivas Devadas. Sanctorum: A lightweight security monitor for secure enclaves. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1142–1147. IEEE, 2019.

[19] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the* 9^{th} *Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 168–177, November 2000.

[20] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[21] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, 2015.

[22] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2109–2122, 2018.

[23] orlp. Ed25519 github repository. `https://github.com/orlp/ed25519`, 2017. Accessed on 05.10.2023.

[24] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for Cross-CPU attacks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 565–581, Austin, TX, August 2016. USENIX Association.

[25] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In *Computer Security–ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part I 24*, pages 279–299. Springer, 2019.

[26] AMD SEV-SNP. Strengthening vm isolation with integrity protection and more. *White Paper, January*, 2020.

[27] Julian Stecklina and Thomas Prescher. Lazyfp: Leaking fpu register state using microarchitectural side-channels. *arXiv preprint arXiv:1806.07480*, 2018.

[28] G.E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 160–171. ACM Press New York, NY, USA, 2003.

[29] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 991–1008, 2018.

[30] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution. 2018.

[31] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656, 2015.

[32] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 719–732, 2014.

[33] Sizhuo Zhang, Andrew Wright, Thomas Bourgeat, and Arvind Arvind. Composable building blocks to open up processor design. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 68–81, 2018.