School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

A Usable System Model for Time Protection

by

Varun Sethu

Thesis submitted as a requirement for the degree of

Bachelor of Engineering in Computer Engineering

Submitted: November 2025

Supervisor: Prof. Gernot Heiser & Dr Rob Sison

Student ID: z5362311

Abstract

This thesis explores how the current system model for time protection on seL4 can be extended to support cross-domain communication. To achieve this, it proposes and evaluates a design for cross-domain notifications and also proposes a design for cross-domain shared memory. Additionally, this thesis argues that a more efficient implementation of cross-domain shared memory is not presently achievable on the evaluation platform. Doing so would require introducing new hardware features or reworking the current time-protection implementation to adopt alternative LLC-partitioning schemes.

Acknowledgments

I would firstly like to thank *Professor Gernot Heiser* and *Dr Rob Sison* for their invaluable guidance and supervision during this project. It has been an absolute blessing and really helped me stay on focus.

I would also like to thank *Julia Vassiliki* and *Nils Wistoff*. Their support was absolutely critical, as much of the work achieved during this thesis would have been practically impossible without them. I would like to thank Julia for all her work in setting up the remote Cheshire infrastructure and porting time protection to Cheshire, as well as for the help she provided when I was debugging finicky timing channels or dealing with arcane hardware issues I had never encountered before. I am also grateful for her patience in answering my many questions and guiding me through any problems I faced with tooling. I would like to thank Nils for answering all the random questions I had about Cheshire, despite being half-way across the world and busy with his PhD thesis.

I would like to thank *Lesley Rossouw*, *Sai Nair* and *Charran Kethees* for their discussions throughout the year about their honours work. Particularly Lesley, who has been an invaluable source of advice and guidance, acting as a sort of informal buddy at TS.

And finally, I would like to thank my friends and family, namely *Shaji*, *Bindu*, *Tarun* and *Prashansa* for all their support during what has been quite an intense year.

Abbreviations

CPU Central Processing Unit

TLB Translation Lookaside Buffer

MMU Memory Management Unit

LLC Last Level Cache

OS Operating System

DPLLC Dynamically Partitionable Last Level Cache

WCET Worst-Case Execution Time

Contents

Chapter 1

Introduction

As the global demand for computing grew, CPU manufacturers needed to come up with more innovative architectures and designs to improve the performance of their chips. To achieve this, manufacturers decoupled the interface from implementation to produce an *instruction set architecture* (ISA). The ISA acted as an API for the CPU, allowing manufacturers to scale the complexity of their implementation (known as the *microarchitecture*) without affecting existing software targeting the CPU. Over the years, several features have been introduced to improve the performance of CPUs, with two very specific examples being *caches* and *speculative execution*. Caches aim to reduce the cost of accessing main memory, acting as a temporary storage of data that was previously accessed by the CPU. The idea being that if a program had accessed a piece of data before, it was likely to access it again. Speculative execution, on the other hand, allowed the CPU to begin executing instructions for a branch point in a program that had not yet been taken, greatly speeding up many programs. There have been several other advancements in CPU design, all of which enable hardware to present a consistent interface to software while still achieving significant performance gains.

As CPU design progressed, there were warnings that the increasing complexity of the microarchitecture could be exploited by bad actors [Ge et al., 2017]. Many such warnings were ignored until 2018, when the Spectre [Kocher et al., 2019] and Meltdown [Lipp et al., 2018] attacks were introduced. These attacks relied on vulnerabilities around speculative execution's impact on caches, which unknowingly allowed attackers to read data that the *operating system* (OS) would otherwise not permit. At their core, these attacks stem from the increasing complexity of microarchitectures, which forced CPUs to manage more intricate interactions and the lasting effects these interactions had on the internal *microarchitectural state*. Kocher et al. [2019] exploited this complexity and discovered a mishandling of caches during speculative execution, where speculatively executed instructions left erroneous measurable traces in the CPU cache. Since the cache is shared between multiple threads and never reset, attacking processes could observe sensitive information that they should not have access to via these erroneous cache impacts.

1.1 Covert Channels and Side Channels

Secure systems maintain restrictions on which subcomponents can communicate with each other. However, when these rules are broken, and a method of communication opens up that does not use a legitimate or intended communication path, then the system maintains a *covert channel*. In contrast, *side channels* refer to the unintentional leakage of information within a system. Unlike covert channels, which require two parties to actively communicate, side channels allow an attacking process to gain information about a victim process without the victim's knowledge.

Spectre and Meltdown demonstrated how channels can be used to leak information in a manner disallowed by a system's security policy, specifically how *microarchitectural timing channels* can be used to achieve this goal. Microarchitectural timing channels exploit variations in program execution time – caused by microarchitectural state – to encode and transmit information. For example, if a CPU stores a piece of memory within its cache, subsequent accesses to that memory will be significantly faster. Two processes can exploit this behaviour to communicate via a shared, read-only memory buffer: one process accesses the first byte of the buffer to load it into the cache, while the other measures its access time for the same byte. A fast access may indicate a "1", while a slow access may indicate a "0" – successfully sending information over a buffer that neither process has permission to write to.

Covert/side channels can be created through any mismanaged shared resource, and the increasing complexity of microarchitectures has provided fertile ground for the introduction of additional mismanaged resources. To address these issues, Ge et al. [2019] introduces *time protection* – a set of OS-level mechanisms aimed at preventing the creation of timing channels on increasingly complex microarchitectures. The authors demonstrate how these mechanisms can be applied to prevent timing channels within systems built on top of seL4.

1.2 seL4 and Time Protection

Microarchitectural timing channels exploit shared resources to transmit information that would otherwise be restricted by a system's security policy. Therefore, any method aimed at eliminating timing channels must implement mechanisms to partition resources and restrict sharing. In the context of time protection, there are two primary methods for resource partitioning: *spatial partitioning*, which involves dividing state into non-overlapping regions, and *temporal partitioning*, which involves dividing shared resources across time [Ge et al., 2019]. While most microarchitectural state can only be partitioned temporally, certain components, such as the *last-level cache* (LLC), can also be partitioned spatially.

seL4 is a *microkernel* with security being a core pillar of its design, guaranteed by a small *trusted compute base* and thorough *formal verification* [Klein et al., 2009, 2014]. Significant engineering work has already gone into extending the kernel to support time-protection. The current system model for time protection in seL4 revolves around the idea of *domains*,

which are a collection of software components and processes that represent a single unit in a system's security policy. Domains were initially introduced by Murray et al. [2013] to prevent leakage of information through scheduling decisions, but Ge et al. [2019] later extended them to prevent leakage via timing channels. seL4 identifies a few key shared resources that must be partitioned between these domains to achieve time protection:

1. Microarchitectural state that supports spatial partitioning (such as the LLC).
2. Microarchitectural state that does not support spatial partitioning (such as TLBs).
3. Interrupts.
4. The operating system kernel itself.

To spatially partition resources between domains, the model assigns each domain a specific region of the resource and ensures it can access only that region, preventing any observation or interference from others. Temporal partitioning is achieved by assigning domains a fixed *time slice*; at the end of a domain's time slice, all resources that cannot be spatially partitioned are reset and flushed, preparing them for the next domain [Ge et al., 2019]. Since the kernel itself is a shared resource and difficult to partition, seL4 provides each domain with its own dedicated copy of the kernel, where each kernel copy maintains a unique text segment, stack, and global data segment [Ge et al., 2019].

While powerful, the model does not provide any mechanism for safe communication across domains, making it impractical for many real-world use cases. A natural next step in the development of this model is to introduce mechanisms that allow controlled communication between domains.

1.3 Extending the Model of Time Protection

All useful systems require the ability for subcomponents within the system to communicate. For a system built around traditional seL4, communication between threads of execution is achieved through *shared memory* and coordinated via an asynchronous signalling mechanism known as *notifications*. Unfortunately, as outlined earlier, systems built on top of time-protected seL4 have no similar mechanisms for communicating. Using the traditional methods directly would completely violate the time-protection guarantees that the time-protected kernel aims to provide.

As such, it then becomes natural for us to explore extending the existing model of time protection in seL4 to support communication. More specifically, explore how to introduce notifications and shared memory across temporally isolated domains. All communication primitives discussed in the remainder of this thesis involve a writer/sender domain and a reader/receiver domain, and are one-way. Information flows exclusively from the writer to the reader domain, and we take deliberate measures to ensure that no back channel exists from the reader to the writer. We will ensure that our designs prevent leaking information that the writer/sender did not intend on sending through the communication primitive.

1.3.1 Shared memory

A key design pillar of seL4 is the user-level management of memory, which allows user-level threads to share buffers of memory by mapping physical frames into multiple address spaces. This makes shared memory a simple way to achieve communication in seL4, and a technique that many applications will use to communicate large amounts of data quickly. Given this, it makes sense to explore introducing shared memory as a cross-domain communication mechanism within time-protected seL4. Extending the current system model to support shared memory is not straightforward. Simply mapping the buffer into each domain would violate the security guarantees the current model of time protection provides, as it would require domains to share portions of the LLC — something the model strictly prohibits. Additionally, special measures must be taken to prevent the introduction of a back channel from the reader to the writer.

1.3.2 Notifications

Notifications in seL4 act as *semaphores*, enabling threads to wait (block until a signal is received) and also signal (unblock the first thread that waited on the notification). Any model of communication using shared memory must also maintain a mechanism for coordinating access to this shared memory, which is precisely the problem notifications solve. It then becomes natural for us to attempt to generalise notifications such that they can be used across domains. Like shared memory, cross-domain notifications are not an easy feat. They are essentially a problem of coordinating communication across distinct copies of the kernel while still maintaining time protection and not introducing any back channels.

1.4 Thesis Problem Statement and Outline

Well-defined communication mechanisms between domains are essential for any system built on time-protected seL4. As such, this thesis will explore extending time-protection to enable strict one-way cross-domain shared memory and notifications. The proposed communication primitives are designed to prevent the leakage of information beyond what is explicitly conveyed through these primitives.

We will explore these primitives over several chapters. Chapter 5 and Chapter 6 will outline the design and implementation of a new cross-domain notification object, enabling threads in distinct domains to signal each other without leaking any additional information. Chapter 7 will then evaluate the presented design by conducting various *timing channel benchmarks*, demonstrating that no information, except signals, can flow between domains.

Chapter 8 will explore possible designs for cross-domain shared memory. It will argue that there is only one presently viable software-only method for implementing cross-domain

shared memory. The alternative method will require reworking the present implementation of time-protection on our target platform rather substantially. Chapter 8 will also demonstrate how the proposed design for cross-domain shared memory can be implemented without extending the kernel any further, only requiring a working implementation of cross-domain notifications. Chapter 9 will then discuss the implementation details of the presented designs. Finally, Chapter 10 will evaluate the presented design and implementation of cross-domain shared memory. It will demonstrate that no information can leak through shared memory buffers outside what is explicitly written to them. Our exploration of cross-domain shared memory will then conclude with a motivation for future work. Arguing that a more efficient implementation of cross-domain shared memory will require reworking the present implementation of time protection on our target platform slightly – particularly, the LLC partitioning mechanism that it employs.

Chapter 2

Background

Before continuing any further, we must first examine some key ideas and terminology that underpin this thesis.

2.1 Caches and Their Architecture

Caches are the key example of microarchitectural state within essentially all modern CPUs. They exist to mitigate the fact that accesses to main memory are significantly slower than CPU operations, aiming to speed up subsequent memory accesses by storing previously used data. At a high level, caches rely on two key principles: *temporal locality*, the idea that recently accessed data is likely to be accessed again; and *spatial locality*, the tendency for programs to access memory locations close to those previously used. When reading from main memory, a CPU will initially check the cache for the data being requested, and if present, will read it directly from the cache. If the requested address is not present, the CPU will read an entire cache line (typically 32 – 64 bytes) containing the target address from main memory and store the result within the cache. Reading an entire *cache line* instead of individual bytes allows the processor to exploit spatial locality, as subsequent requests for nearby addresses can be immediately serviced by the cache.

2.1.1 Address decomposition

To understand how an address is decomposed and used to index a specific entry in a cache, we will consider a hypothetical 32-bit machine. On this machine, a cache line will consist of 2048 bytes; therefore, we require $\log_2(2048)$, or 11 bits to index an individual byte within this cache line. Alongside this, we will require some bits in the address to index into the actual cache itself, these bits are known as the *cache identifier*. On this hypothetical architecture, the top 21 bits are dedicated to the cache identifier and the

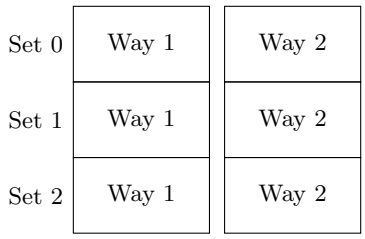
bottom 11 bits are dedicated to the *offset*, which is used to index bytes within a cache line. This decomposition is pictured in Figure 2.1.

Figure 2.1: Hypothetical decomposition.

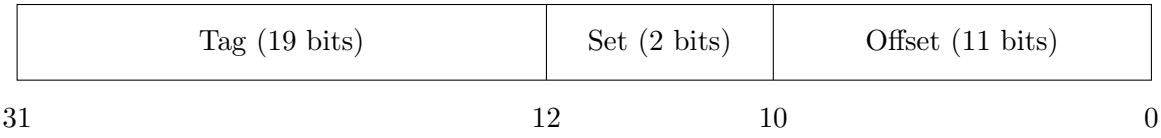
The actual specifics as to how the identifier field is structured depends on the overall cache architecture. Upon a lookup, the cache will use the identifier to perform a search of all entries present within the cache. If the requested identifier is found, the CPU will use the offset bits to index the corresponding byte in the cache line.

2.1.2 Cache architectures

There are many cache architectures, with the one most commonly used in practice known as a *set-associative* architecture. The set-associative architecture decomposes the cache into a set of buckets known as *sets*. These sets are then further broken down into smaller individual slots known as *ways*.



In a set-associative architecture with 4 sets, an address on a 32-bit machine with 2048 byte long cache lines is decomposed as a $\log_2(2048) = 11$ bit offset, $\log_2(4) = 2$ bit set identifier, and a 19 bit tag.

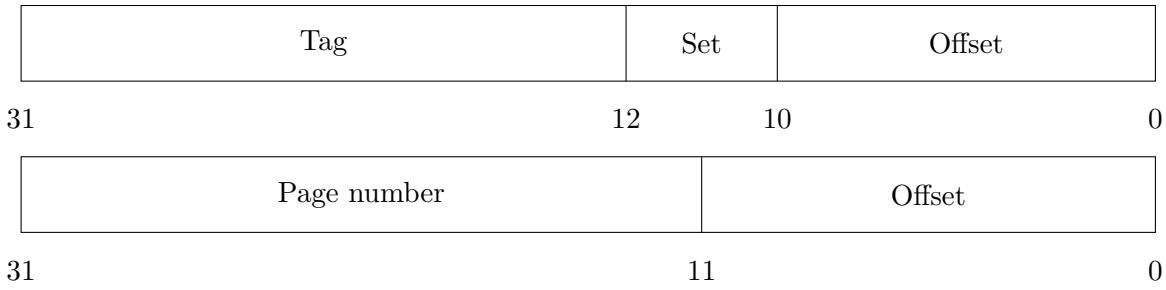


The set a cache line belongs to is determined purely by its address, specifically, the 2 set bits. When a cache line is inserted, the hardware uses these bits to determine the appropriate set for insertion and then places the line in the first free way in the set. If no ways are free, it will evict a way according to some eviction policy and place the cache line there. The story is similar for a lookup. The hardware will first determine what set a

line belongs to and then perform a parallel lookup for the tag within the set. If an entry with the same tag bits is identified, then it has successfully resolved the cache line to an entry in the cache.

2.1.3 Cache colouring

Consider a set-associative cache on a 32-bit machine with 4 sets (2 set bits). In this cache, there are 11 bits dedicated to the offset and 19 bits dedicated to the tag. Alongside this, the machine also supports virtual memory, with the upper 20 bits of an address being used to determine the page number and the lower 12 bits being used to index into the page. This presents us with two differing ways that addresses are decomposed on the machine, with the first being how the cache deals with addresses and the second being how the *memory-management unit* (MMU) decomposes addresses.



The interplay between these two decomposition schemes give rise to an interesting property. Observe that the last bit in the page number coincides with the upper bit of the set identifier. It then follows that the cache lines residing in a page ending in 0 can only map to the sets 0 (00) and 1 (01), whereas a cache line residing in a page ending in a 1 can only map to sets 2 (10) and 3 (11). This situation is depicted more clearly in Figure 2.2.

The distinct regions of the cache that arise from this interplay between virtual memory and set-associative caching structures are known as *cache-colours*. The interesting insight is that addresses that reside in different cache colours will never map to the same cache sets within a cache. This insight will be of great value later when describing the present model of time protection as cache colouring is used to spatially partition specific caches.

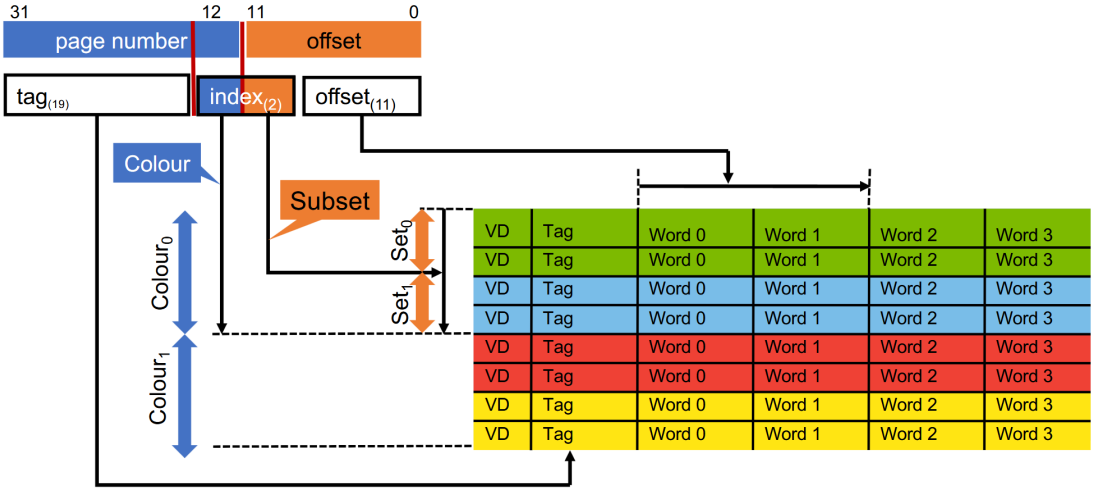


Figure 2.2: Cache colouring [Heiser, 2024].

2.1.4 Cache hierarchy

Caches within a machine are organised into strict hierarchies, typically the L1, L2 and L3 caches, with the last cache in any hierarchy being known as the *last-level cache* (LLC). These caches differ in their size and performance characteristics. The L1 cache, which is closest to the CPU, is optimised for fast reads, but maintains a rather small size (32 KiB on many modern chips). The L2 and L3 caches, on the other hand, lie further away from the CPU and maintain sizes on the order of a few MB (around 1 MB for the L2 and 4–64 MB for the L3). Alongside differing performance and size requirements, the caches differ in what addresses they use to index into the cache. Typically, L1 caches are *virtually indexed*, meaning that they use the virtual address when indexing into a cache. L2 and L3 caches, on the other hand, are *physically indexed*, using the physical address for cache lookups.

On many architectures, the L1 cache is split into two components: the *L1 data cache* (L1-D) and the *L1 instruction cache* (L1-I). The L1-D exclusively holds data, while the L1-I exclusively holds instructions. This split-cache design improves CPU performance by enabling the simultaneous access of both data and instructions.

The existence of multiple cache levels logically leads to the question of how data is duplicated between levels. In an *inclusive cache*, if data is present in the L1 cache, then it must also be present in the lower level L2 and L3 caches. The inverse holds for an *exclusive cache*, where data being present in the L1 cache implies that it is not present in either the L2 or L3 caches. Many ARM chips, such as the ARM Cortex series chips, maintain an exclusive policy between the L1 and L2 caches, whereas many Intel x86 chips maintain inclusive hierarchies.

The RISC-V *system-on-chip* (SoC) used for implementing the ideas discussed in this thesis

maintains a two-level cache hierarchy, one on-core cache (L1) and an LLC shared between multiple cores. The cache hierarchy guarantees that if a cache line is not present in the L1 cache, then a read of that line will bring it into both the L1 cache and LLC. The hierarchy does not maintain this inclusivity relationship forever though. It is possible for the same cache line to be later evicted from the L1 while still being present in the LLC, this presents interesting consequences when attempting to reason about the existence of a cache line within the cache hierarchy.

2.2 Timing Channels

Secure systems maintain restrictions on what subcomponents can communicate with each other. However, when these rules are broken, and a method of communication opens up that does not use a legitimate or intended communication path, then the system maintains a *covert channel*. Covert channels enable the unauthorised flow of information in a manner that the security policy of a system would otherwise disallow [Lampson, 1973]. Tangential to the idea of a covert channel is a *side channel*, which enables the unintentional leakage of data. Side channels consist of some victim and attacker process, with the attacker process using the side channel to spy on the victim process without its knowledge [Kocher, 1996]. The main distinction between a side channel and a covert channel is the covert channel's requirement of cooperation to transmit information, this requirement also implies that covert channels maintain a significantly larger *capacity* and can transmit more information with a higher resolution. An interesting point is that despite requiring cooperation, covert channels can also be used to transmit information "unknowingly", but they require introducing a *Trojan* to the victim process. Trojans are a piece of malicious code that is unknowingly introduced into a victim (usually via shared libraries) and leaks information to the attacking process silently [Anderson, 1972].

Shared microarchitectural state, such as caches discussed previously, enable the construction of *microarchitectural timing channels*. Microarchitectural timing channels exploit variations in program execution time – caused by microarchitectural state – to encode and transmit information. For example, if a CPU stores a piece of memory within its cache, subsequent accesses to that memory will be significantly faster. Two processes can exploit this behaviour to communicate via a shared, read-only memory buffer: one process accesses the first byte of the buffer to load it into the cache, while the other measures its access time for the same byte. A fast access may indicate a "1", while a slow access may indicate a "0" – successfully sending information over a buffer that neither process has permission to write to. We exploit timing channels using *timing attacks*, with the two primary timing attacks used throughout this thesis being the Flush + Reload attack [Yarom and Falkner, 2014] and the Prime + Probe attack [Liu et al., 2015; Osvik et al., 2006; Percival, 2005], both of which can be used as an exfiltration technique in the Spectre and Meltdown attacks [Kocher et al., 2019; Lipp et al., 2018].

2.2.1 Flush + Reload

The Flush + Reload attack consists of a victim and an attacker process (also known as a spy), the attacker aims to learn something about the victim process by observing how the victim interacts with a buffer of memory shared between the two of them. When both processes share memory, they also inadvertently share parts of the last-level cache (LLC). The attacker exploits this by observing cache access patterns, allowing them to extract sensitive information from the victim in a way that violates the system's security policy [Yarom and Falkner, 2014].

The attack consists of three stages, first, the attacker flushes specific cache lines from the LLC using an instruction such as `clflush`, it then yields time to the victim process. The victim process runs for a bit and, with each memory access into the shared buffer, brings more data into the LLC. The spy, when it gets another time slice, will then time the amount of time it takes to access cache lines within the shared buffer. Fast access implies that the victim touched the cache line while running, whereas slow access implies that it did not. Using this information, the spy can infer what cache lines the victim accessed, and consequentially infer information regarding the victim's execution. This attack can be used to target RSA and trace the execution of the *square-and-multiply* routine, tracing this routine is sufficient to allow the attacker to break encryption [Yarom and Falkner, 2014].

```
while (true)
{
    flush_llc();
    yield_timeslice();

    uint32_t start = current_time();
    shared_buffer[0];
    uint32_t elapsed = current_time() - start;

    if (elapsed < threshold)
    {
        // Victim accessed this cache line
    }
}
```

Listing 1: Example spy process.

2.2.2 Prime + Probe

The previous attack relied on the existence of a shared buffer between the spy and victim process. In practice, this is not a requirement, and the Prime + Probe technique extends

cache timing channels to scenarios with no shared buffer. The attack introduces the notion of a *victim set*, which is a set of cache lines in the victim's address space that the spy is interested in observing. The spy process must then determine how the victim set maps into its address space to construct an *eviction set* – a set of cache lines that overlap with the victim's set. If the attack targets the virtually indexed L1 cache, this mapping is a straightforward one-to-one correspondence.

To begin the attack, the spy will first *prime* the eviction set by touching each cache line in the set, bringing them into the cache. Like the previous technique, the spy then yields its time-slice back to the victim. As the victim runs, it may potentially evict cache lines that the spy had brought into the cache. This is problematic, as when the spy is next allocated a time-slice, it can use this information to infer what cache lines the victim touched. In the final stage of the attack, the spy iterates through all the cache lines in its eviction set. If the access to a cache line is fast, it implies that the line was not accessed by the victim. Conversely, if the access is slow, it suggests that the victim evicted the spy's entry by accessing the same cache set [Liu et al., 2015; Osvik et al., 2006; Percival, 2005].

To effectively pull off this attack, the spy must know how cache lines in its address space overlap with cache lines in the victim's address space. This requirement is relatively easy to satisfy when targeting the L1 cache, as it is virtually indexed. However, there are still practical attacks that target the LLC using similar techniques [Liu et al., 2015; Osvik et al., 2006; Percival, 2005].

2.3 seL4 Background

seL4 is a capability-based microkernel with a strong focus on security and correctness. It is formally verified [Klein et al., 2009, 2014] to ensure both functional correctness and enforcement of its security properties [Murray et al., 2013; Sewell et al., 2011]. Being a microkernel, it provides a relatively small set of features and delegates most traditional OS-responsibilities to user-level threads that communicate over *IPC*. Typical seL4 based systems consist of some *initial thread* and upon boot, seL4 will hand information regarding the current system environment to this initial thread, from which it can create new threads and manage resources.

2.3.1 Capabilities

seL4 uses capabilities to control access to kernel resources, where a capability is defined as an unforgeable token representing the rights to access an entity or object [Dennis and Van Horn, 1966]. User-level threads interact with kernel resources via *invocations* on capabilities, with all kernel resources typically exposing their APIs via such invocations. Capabilities are organised into *CSpaces*, with a CSpace representing a structured tree of capabilities. To invoke a capability, user applications use a *CPtr* (capability pointer) to

index into a CSpace and identify the desired capability. In addition to invocations, capabilities support *copying* and *minting*, copying will create a duplicate of the capability with the same access rights whereas minting involves duplicating the capability with reduced rights [seL4 Foundation, 2024].

Memory management in seL4 is handled entirely at user-level, with the kernel having no method for dynamically allocating any data structures that it requires. As such, the kernel must provide an abstraction over physical memory that user-level applications can supply when requesting new kernel objects. This abstraction is known as *untyped memory*, with the kernel providing *untyped capabilities* for the management of this memory to the initial thread. These capabilities can be *retyped* into various data structures such as TCBs, notifications and physical frames of memory [seL4 Foundation, 2024]. Additionally, untyped capabilities have a boolean property `device` which indicates whether the memory is writeable by the kernel or not, as it may be in an area of RAM not addressable by the kernel. Device untyped capabilities can only be retyped into *frame* objects [seL4 Foundation, 2024].

2.3.2 Threads

Threads in seL4 model an execution context, allowing for the management of processor time [seL4 Foundation, 2024]. The key bookkeeping structure for a thread in seL4 is the *thread control block* (TCB) which maintains various metadata regarding a thread; namely the thread's CSpace and virtual address space (*VSpace*). On boot, the kernel constructs an initial thread with a basic CSpace. This thread uses the untyped capabilities provided to it to create additional threads for the system running on top of seL4. Threads are given a priority between 0 and 255. The seL4 scheduler will always schedule the thread with the highest priority if it is runnable. If there are multiple runnable threads with the maximal priority, then they are scheduled in a first-in-first-out, round-robin style fashion [seL4 Foundation, 2024].

2.3.3 Virtual memory management

seL4 provides virtual memory management through thin APIs over the hardware for manipulating hardware paging structures, rather than the higher-level abstractions found in systems like Linux. Common to all hardware architectures is the *Frame*, representing a frame of physical memory. Frames are mapped into VSpaces by manipulating the appropriate paging structures for the VSpace. The separation of frames and VSpaces also enables frames to be mapped into two separate address spaces, enabling shared memory between threads. It should be noted, however, that when attempting to map a frame into two distinct VSpaces, the capability for the frame must be duplicated and each mapping should correspond to a unique copy of the initial frame capability [seL4 Foundation, 2024].

2.3.4 Notifications

Notifications are a synchronisation primitive provided by seL4, they represent a set of binary semaphores and support the usage of small 32-bit badges to differentiate notifiers. They, like everything else in seL4, are allocated by retyping untyped memory, with the returned capability allowing for threads to either: *signal*, *poll* or *wait* on the notification. Signalling alerts any threads currently waiting on the notification, while waiting simply blocks the current thread until the target notification is signalled. The internal structure of a notification is simple, consisting of a 32-bit notification word and a queue of threads currently blocked on it [seL4 Foundation, 2024].

The exact behaviour of signalling or waiting on a notification depends on its internal state, which can be *waiting*, *active*, or *idle*. A notification enters the *waiting* state when a thread blocks on it, and transitions to the *active* state when it is signalled while no threads are waiting.

When signalling on an *idle* or *active* notification, the kernel will bitwise OR the badge value associated with the signal with the data word currently stored in the notification. If the notification is in the *waiting* state, however, the badge value is immediately delivered to the head of the blocked thread queue. When a thread waits on a *waiting* or *idle* notification, the requesting thread is added to the end of the thread queue, and the kernel marks the requesting thread as blocked. If, however, the notification is in the *active* state, the data word stored in the notification is immediately delivered to the thread and the notification's badge is reset to 0 [seL4 Foundation, 2024]. A state diagram outlining the above is illustrated in Figure 2.3 with pseudocode outlining the process in more detail being provided in Listing 2 and Listing 3.

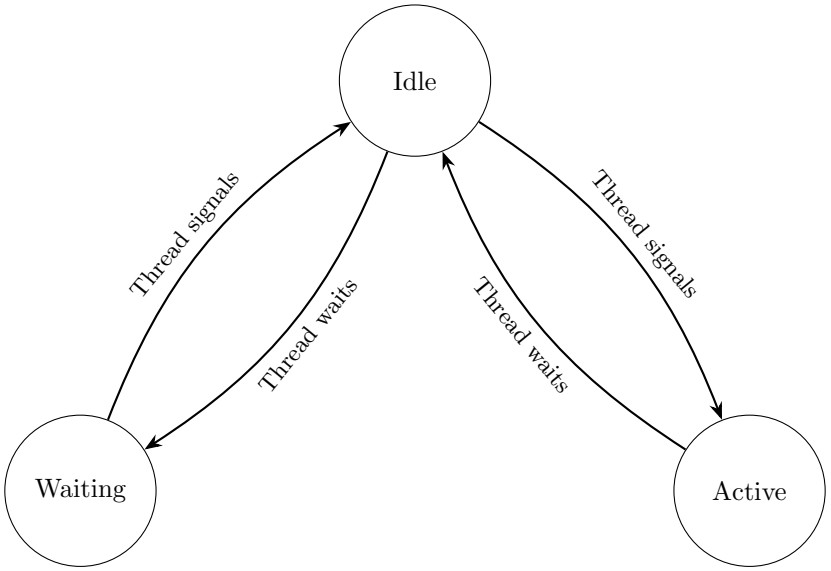


Figure 2.3: Notification state diagram.

```
if (ntfn.state == IDLE || ntfn.state == ACTIVE)
{
    ntfn.badge = bitwise_or(ntfn.badge, badge);
    ntfn.state = ACTIVE;
}
else if (ntfn.state == WAITING)
{
    head = dequeue(ntfn.tcb_queue);
    head.badge = badge;
    head.state = RUNNING;

    if (ntfn.tcb_queue is empty)
    {
        ntfn.state = IDLE;
    }
}
```

Listing 2: Signal pseudocode.

```
if (ntfn.state == IDLE || ntfn.state == WAITING)
{
    enqueue(ntfn.tcb_queue, thread_);
    thread_.state = BLOCKED;
    ntfn.state = WAITING;
}
else if (ntfn.state == ACTIVE)
{
    thread_.badge = ntfn.badge;
    ntfn.badge = 0;
    ntfn.state = IDLE;
}
```

Listing 3: Wait pseudocode.

An important and interesting implementation detail regarding notifications is that since the kernel cannot perform any dynamic allocations, the queue of threads blocked on a notification is stored within the TCBs associated with the threads themselves. The notification maintains a pointer to the TCB for the first thread in its queue, and the *next* and *prev* pointers for the queue are maintained within each TCB.

2.3.5 Endpoints and IPC

seL4 use small kernel objects known as *endpoints* to perform *inter-process communication* (IPC). IPC is the microkernel mechanism for the transmission of small amounts of data and capabilities between threads. This same mechanism is also used for communication with kernel-provided services. Endpoints consist of a queue of threads waiting to send, wait or receive messages and can be optionally badged to distinguish senders on an endpoint [seL4 Foundation, 2024].

2.3.6 IRQs

Interrupts on seL4 are delivered as notifications, with a thread able to configure an interrupt to be delivered on a particular notification. A thread can then wait for the interrupt by blocking on the configured notification. seL4 introduces the `IRQHandler` capability to represent the ability for a thread to configure a notification to receive an interrupt [seL4 Foundation, 2024].

2.3.7 Domains

To maintain confidentiality, seL4 allows for threads to be associated with distinct scheduler partitions known as a *domains* [Murray et al., 2013]. Domains are statically configured at compile-time with a cyclical schedule and are not preemptible, an example domain schedule is provided in Figure 2.4. Threads can be associated with a domain, and the kernel will cycle between domains and only ever schedule the threads associated with the currently active domain. If there are no threads that can currently be scheduled in the domain's time-slice, then the kernel will schedule a dedicated *idle thread*.

Threads are assigned to domains via the `DomainSet_Set` invocation on the `Domain` capability passed to the initial thread. There is only one `Domain` capability, so the `Domain_Set` invocation differentiates domains with the domain number passed to it [seL4 Foundation, 2024]. The length of a domain's time-slice is provided as a multiple of the kernel *tick interval*, which is the duration of time between timer interrupts.

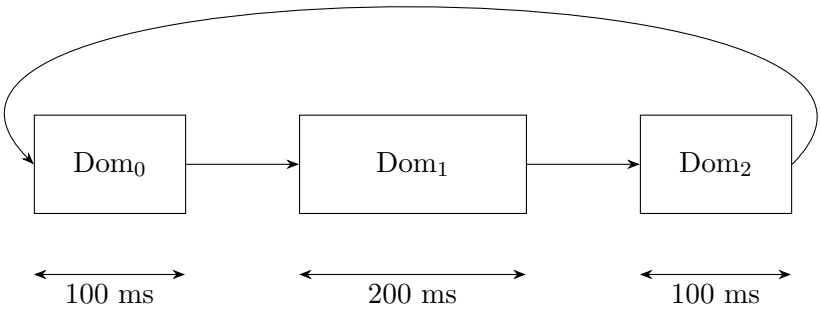


Figure 2.4: A static schedule for a hypothetical three domain system.

2.4 Cheshire

seL4 is capable of running on various platforms and devices. The platform of interest to us during this thesis is *Cheshire*. Cheshire is a minimal SoC built around the RISC-V CVA-6 core [Ottaviano et al., 2023]. Its goal is to provide a lightweight and configurable compute environment that is capable of easily targeting FPGAs with limited resources. Due to its highly configurable nature, different deployments of Cheshire can have different platform specs, the deployment used by this thesis for benchmarking has a core that runs at 50 MHz, maintains a 32 KiB L1 data-cache, a 16 KiB L1 instruction-cache, and a 128 KiB LLC.

2.4.1 LLC

Cheshire features a highly configurable set-associative LLC [Ottaviano et al., 2023]. This thesis uses a configuration of the LLC that consists of 64-byte cache lines and 256 distinct cache sets, with each cache set consisting of 8 distinct ways. All-in-all, this produces an 128 KiB LLC with 4 distinct cache colours. An interesting feature of Cheshire's LLC is the ability for each of its ways to be individually configured as *scratchpad memory* (SPM) at runtime, providing the host with fast on-chip SRAM [Ottaviano et al., 2023]. It is important to note that if a way is configured as SPM, it cannot be used for regular caching purposes, therefore configuring a single way as SPM will reduce the effective associativity of the cache by one. The LLC maintains a randomised eviction policy, whereby a way is evicted at random when a set is full. This randomised eviction policy maintains interesting consequences for the design of cross-domain shared memory, as will be discussed in Chapter 8.

2.4.2 Microarchitectural flush

Throughout this chapter, we have seen a lot of literature that demonstrate how microarchitectural state within a CPU can be used to exploit timing channels by constructing timing attacks. Existing work highlights the ineffectiveness of modern architectures to protect against such attacks due to the inability to reset microarchitecture state effectively [Ge et al., 2018].

Drawing from these lessons, the CVA-6 core that Cheshire is built around introduces a special `fence.t` instruction that flushes the L1 cache and resets all other microarchitectural state within the CVA-6 core to a known state [Wistoff et al., 2021]. Since the latency of the `fence.t` operation depends on the state of said micro-architectural state, the instruction also features a simple time-padding mechanism that will pad the time between the last *core-local timer interrupt* (CLINT) and the `fence.t` invocation to some configured *worst-case execution time* (WCET) [Wistoff et al., 2023]. The fact that the instruction pads relative to the last timer interrupt allows it to be used to pad any operations that proceed `fence.t` to some WCET.

To enable time-padding, the implementation exposes a custom `cspad` *control-status register* (CSR), the core will read this value on the arrival of a timer-interrupt and use this when performing time-padding. It is also worth pointing out that `fence.t` is not the only instruction configured to flush the L1-D cache, Cheshire configures the CVA-6 core so that the regular `fence` instruction also flushes the L1-D cache.

2.4.3 OpenSBI

The RISC-V architecture maintains three privilege levels: machine-mode (highest privilege with unrestricted system access), supervisor-mode (for OS kernels), and user-mode (for applications). Since supervisor-mode is not the highest privilege level and does not have direct access to the hardware, many operating systems targeting RISC-V are deployed ontop of a thin firmware layer known as *OpenSBI*. Whenever the kernel attempts to access a CSR or any other hardware resource that it is not configured to have permissions for, Cheshire will trap into OpenSBI who will then interact with the hardware on behalf of the kernel [RISC-V International, 2025]. Cheshire maintains its own fork of OpenSBI with some minor platform-specific changes, this is the build of OpenSBI that we will use throughout this thesis [PULP Platform, 2025b].

2.5 Time Protection in seL4

Time protection refers to a collection of operating system mechanisms that jointly prevent interference between separate domains within a system, ensuring that it is impossible to construct a timing channel between what should be isolated domains [Ge et al., 2019]. To achieve time protection, the current model partitions all resources shared between

domains either *spatially*: which involves dividing state into non-overlapping regions, or *temporally*: which divides shared resources across time [Ge et al., 2019]. To spatially partition resources, the model ensures that each domain is assigned a specific region of the resource and guarantees that it can only access its dedicated region. Temporal partitioning is achieved by assigning each domain a fixed *time slice*; at the end of a domain's time slice, all resources that cannot be spatially partitioned are reset and flushed in preparation for the next domain [Ge et al., 2019]. However, resource partitioning alone is insufficient to achieve time protection. To fully realise time protection, seL4 must enforce all the following requirements [Ge et al., 2019]:

R1. When switching domains (known as a *domain switch*), the OS must reset microarchitectural state to a defined state, unless the hardware supports spatially partitioning such state.

R2. Each domain must have its own private copy of the OS text, stack and global data (as much as possible).

R3. Access to any remaining shared OS data must be deterministic to avoid leakage.

R4. State flushing must be padded to its worst-cast latency.

R5. When sharing a core, the OS must disable or partition any interrupts apart from the preemption timer.

2.5.1 Requirement 1 for Time Protection

Some microarchitectural state such as the LLC can be spatially partitioned using techniques such as cache colouring. Cache-colouring based partitioning assigns a unique colour to each domain to prevent interference between domains within the LLC. The implementation of this partitioning scheme in seL4-based systems is relatively straightforward and can be done with virtually no help from the kernel. On boot, the kernel will provide the initial thread with the `seL4_BootInfo` structure, which, alongside much else, contains a description of the physical ranges of memory associated with each untyped capability [seL4 Foundation, 2024]. Knowing the physical addresses associated with untyped capabilities allows the initial thread to divide memory into coloured "pools", which can then be used to allocate kernel structures and memory frames of a specific colour. As long as the initial thread ensures that all data structures and frames for a domain are allocated from a single coloured memory pool, cache colouring guarantees that cache lines associated with different domains will not overlap, thereby effectively spatially partitioning the LLC among domains [Ge et al., 2019].

Not all microarchitectural state can be partitioned, and in such cases the kernel must reset this state upon a domain switch. The unfortunate reality however, is that the kernel is limited here by the extent to which the architecture it is targeting supports flushing microarchitectural state [Ge et al., 2018]. x86 provides limited support for resetting on-core state, while ARM supports flushing the L1 D-cache, TLB and branch-predictor. The story is slightly different on RISC-V with all SoCs built around CVA-6 supporting the `fence.t` microarchitectural flush and padding instruction [Wistoff et al., 2021, 2023].

2.5.2 Requirements 2 & 3 for Time Protection

System calls will leave some residual impact on microarchitectural state, i.e. they may maintain some cache footprint that is potentially observable outside the kernel [Ge et al., 2019]. This issue can be addressed by having each domain maintain a unique copy of the kernel to prevent the kernel from being used as a timing channel [Ge et al., 2019].

Ge et al. [2019] proposed a policy-free *kernel clone* mechanism, whereby the initial thread can construct a copy of the kernel from user-provided frames of memory. Each cloned kernel maintains its own kernel-code and read-only data segments. Realistically, some data needs to be shared between clones and special care is taken to ensure that accessing this shared data is sufficiently deterministic, i.e. by pre-fetching shared data when switching between kernels. The kernel clone mechanism enables the initial thread to construct a fully partitioned system, with each domain allocated its own unique kernel, backed by memory drawn from the domain's coloured memory pool. It is worth pointing out that not all ports of time-protection maintain this kernel-cloning mechanism. Work done by Buckley et al. [2023] in porting the time-protected kernel to RISC-V introduced a static policy, where the kernel clone for each scheduler domain is created on boot. On the RISC-V port, the initial thread cannot create kernel clones and can only associate threads with an existing kernel clone by attaching them to a target domain.

2.5.3 Requirement 4 for Time Protection

As previously outlined, all resources that cannot be spatially partitioned must be temporally partitioned by flushing them during a domain switch. The latency of these flushing instructions typically depends on the contents of the resources [Ge et al., 2019]. For example, the flush latency of the L1 cache varies depending on the number of cache lines currently stored within it. It then becomes a requirement that all these microarchitectural flush operations be padded to their worst-case execution latency. If this is not done, then a channel exists via the flush latency.

In practice, time-padding is achieved by configuring an upper bound on the WCET of the domain switch latency. If there is no direct-hardware support for time-padding, then the kernel will busy-wait until this specified WCET has elapsed. On Cheshire and other CVA-6 based systems, there exists support for WCET padding via the `fence.t` instruction. CVA-6 allows for the time between the core-local timer interrupt and a subsequent `fence.t` invocation to be padded to some WCET [Wistoff et al., 2023]. Without padding, the next domain can observe variations in the domain switch latency, as the domain switch latency directly affects when the first thread in the new domain is scheduled [Ge et al., 2019].

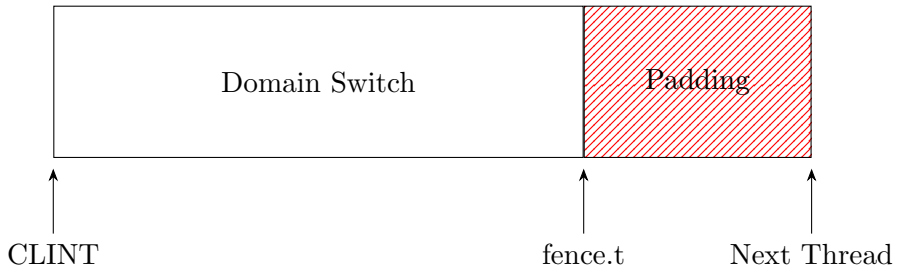


Figure 2.5: Padding the domain switch latency.

2.5.4 Requirement 5 for Time Protection

Interrupts can be used as a channel between domains, a Trojan can program interrupts to fire during the spy's time slice. This is problematic as the system clock at the time the interrupt is fired can be used to encode messages [Ge et al., 2019]. This channel, although maintaining a relatively low *bandwidth*, is still problematic. To mitigate this, the model allows for the partitioning of interrupts, with only some set of interrupts being associated with each domain [Ge et al., 2019].

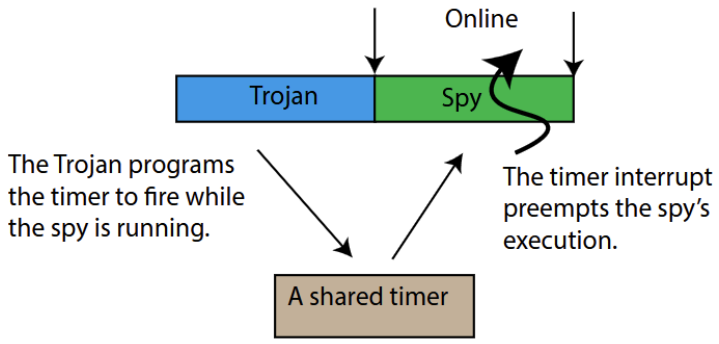


Figure 2.6: Interrupt being used as a channel between domains [Ge, 2019].

2.5.5 Time Protection on RISC-V

The original implementation of time-protection targeted the ARM and x86 architectures; however, due to the lack of architectural support, there was still at least one timing channel on each architecture that could not be closed [Ge, 2019; Ge et al., 2018]. By observing these limitations, it was argued that to fully realise time protection, a new hardware-software contract was required, one that allowed for microarchitectural state to be fully flushed [Ge et al., 2018].

Due to the limitations in x86 and ARM architectures, Wistoff et al. [2021] proposed the `fence.t` instruction as an extension to RISC-V. Further work carried out by Buckley et al. [2023] ported time-protection to RISC-V, more specifically a RISC-V based system using the CVA-6 core. Doing so allowed them to make use of the new `fence.t` instruction to flush on-core microarchitectural state. Additionally, this port made several simplifications to the model of time-protection, namely the fact that the provided implementation now constructs a unique kernel clone for each domain at boot, switching the currently active kernel image whenever there is a domain switch [Buckley et al., 2023].

Ge et al. [2019] used pre-fetching to determinise shared-kernel data in the LLC during domain switches. With time-protection now tied to seL4's domain scheduler, most shared-kernel data can be partitioned, leaving only a small remainder accessed deterministically on domain-switches [Buckley et al., 2023]. Further unpublished work carried out by Julia Vassiliki, Nils Wistoff, Dr Rob Sison and Professor Gernot Heiser has ported the original RISC-V re-implementation to Cheshire. By doing so, all remaining shared kernel data can now be placed within scratchpad memory, thereby eliminating any potential timing channels as SPM is never cached by the LLC. The implication of this, however, is that the effective associativity of Cheshire's LLC is now 7 instead of 8, as one way is marked as SPM. All the designs discussed in this thesis will be implemented on top of the Cheshire port.

2.5.6 The domain switch on RISC-V

Understanding the domain switch operation will be of vital importance for the rest of this thesis so we will take some time now to look at it in more detail on RISC-V. On Cheshire, the domain switch is triggered by the arrival of a CLINT. On a CLINT, the interrupt handling path will reset the timer interrupt and schedule a new one to arrive some number of milliseconds in the future. After resetting the timer interrupt, the kernel will identify that the current domain's time-slice has elapsed and trigger a domain switch.

The domain switch occurs in two distinct phases, during the first phase, the kernel will identify the next domain to schedule and switch to its kernel image. After switching the kernel image, the kernel now operates within the cache colour associated with the kernel of the new domain. The kernel will now perform a `fence.t` invocation to erase any impact the old domain had on the non-partitionable microarchitecture. The implication of performing the `fence.t` after switching the kernel image is that only the operations performed up to and including the `fence.t` are padded to a WCET. These operations are: handling the CLINT, resetting the timer, switching the kernel image, and invoking `fence.t`. The second half of the domain switch, while not strictly part of the domain switch itself, involves determining the first thread in the new domain to schedule and scheduling it. It is important to note that all the domain switch operations happen after resetting the time interrupt. As such, the latency of all these operations are charged to the new domain, and the new domain can observe this latency by observing when its first thread is scheduled.

2.6 Time Protection in seL4 — Taking It Further

The current model of time protection in seL4 requires all kernel data structures and memory frames used within a domain to be allocated from that domain's coloured memory pool. As a result, there is no clear way for threads in two different domains to communicate with each other while still maintaining time protection. Communication in most deployments of seL4 involves using shared memory to transfer large amounts of data between threads, alongside the use of notifications to orchestrate the access to this data; it then becomes natural for us to explore extending these communication primitives such that they can be used across domains. The remainder of this thesis is dedicated to exploring this very problem. We explore cross-domain notifications in Chapter 5 and Chapter 6, and cross-domain shared memory in Chapter 8 and Chapter 9. Our exploration of cross-domain notifications and shared memory will lead to the introduction of new kernel objects that enable domains to signal each other and communicate without leaking information. We will also look at how the proposed design for cross-domain shared memory can be implemented at user-level, completely cutting the kernel out of the picture.

Chapter 3

Related Work

This chapter surveys various timing channel mitigation techniques and explores how they can potentially inform the design of cross-domain shared memory and notifications. We will also examine the existing literature on timing channel benchmarking and explore it may guide the method by which we may evaluate our proposed designs for cross-domain notification and shared memory.

3.1 Constant Time Programming

A common and recommended approach in many cryptographic applications is to use *constant-time programming* to eliminate channels [Bernstein, 2005]. To focus on how these techniques may be applied, we will look at Meier et al. [2021], who builds upon these principles to guide the implementation of the constant-time arithmetic library `saferith`. The authors build upon three key requirements that constant-time programming aims to maintain:

1. No loops that leak the number of iterations taken.
2. No memory accesses that leak the address or index that was accessed.
3. No conditional statements that leak which branch was taken.

The authors propose a simple API that ensures all applications using the library are compliant with the above requirements. While basic benchmarks are presented to demonstrate the constant-time behaviour of operations such as `Exp`, the evaluation lacks substantive evidence showing whether constant-time programming effectively eliminates timing channels in typical cryptographic algorithms. In fact, Schneider et al. [2024] demonstrates that many constant-time programming techniques are undermined during compilation, primarily due to aggressive compiler optimisations and the introduction of secret-dependent

instructions. This highlights the fragility of such techniques and the difficulty of implementing them correctly in practice. Additionally, certain constant time techniques such as avoiding secret-dependent table lookups require direct hardware support [Page, 2003].

Discussion and Takeaways

The literature highlights that many constant time programming techniques are relatively fragile and difficult to implement, on top of this, many techniques offload the responsibility of time protection onto the application. seL4 aims to ensure that time protection is an abstraction provided by the OS, much like memory isolation; confining even untrusted programs from leaking information to the outside world. Relying on every application running on seL4 to use constant time techniques fundamentally breaks this abstraction.

While it is infeasible to expect all applications to use constant-time techniques when interacting with the communication primitives we construct, some of these ideas still provide practical guidance for the design of such primitives. When implementing primitives such as cross-domain notifications, it is crucial to maintain the invariant that operation latency remains independent of notification state. The principles of constant-time programming could find some applications here and act as a guide for maintaining such an invariant, but any design using these principles will require thorough benchmarking and verification.

3.2 Pre-fetching and Forced Determinism

Ge et al. [2019] illustrates the use of *pre-fetching* to ensure that all access to *shared kernel data* (SKD) within time-protected seL4 is sufficiently deterministic. The approach involves identifying all the cache lines associated with shared kernel data and then manually "touching" them on a domain switch to force them into the L1 cache and LLC [Ge et al., 2019]. The authors demonstrate the effectiveness of this approach when combined with separated kernel images in closing the timing channel associated with a shared kernel image. An important omission in the paper, however, is the sensitivity of the proposed method to specific cache implementation details, particularly the inclusiveness or exclusiveness of the cache hierarchy. In the absence of architectural guarantees, it remains unclear how the timing behaviour of subsequent accesses to pre-fetched cache lines are affected if those lines are evicted from any part of the cache hierarchy.

Buckley et al. [2023] extends this analysis and presents some insight into the use of pre-fetching for shared kernel data in time-protected seL4 on RISC-V. They highlight that pre-fetching is completely insufficient for closing timing channels through SKD, and the only principled solution for closing such channels is via targeted flushes of the address associated with shared kernel data from the LLC. They also highlight the verification challenges with pre-fetching, as verifying the effectiveness of pre-fetching requires a detailed model of the LLC's eviction policy to truly prove correctness and safety. It is worth pointing out that these conclusions are drawn in the context of shared kernel data, where due to the lack of partitioning and colouring, the cache lines of which can be arbitrarily affected by the

activities of threads in partitioned domains. There remains the question of whether pre-fetching is effective when we can guarantee that threads cannot interfere with the cache sets associated with the pre-fetched memory block, aside from direct reads and writes, that is, when the memory being pre-fetched resides in a dedicated colour.

Discussion and Takeaways

Buckley et al. [2023] argues that pre-fetching fails to close timing channels through shared kernel data. This implies that pre-fetching is not a suitable solution for implementing cross-domain shared memory if the colour used to back the shared memory is common to either the reader or writer domain. This stems from the fact that when cache lines belong to a common colour, as is the case with SKD and domain partitioned data, it becomes difficult to predict, without a formal model of the eviction policy, how pre-fetching interacts with cache lines that map to the same cache set(s) as SKD within the LLC, as well as how those overlapping lines, in turn, affect the data being pre-fetched.

There is still, however, a remaining question as to if pre-fetching works if the shared buffer is allocated its own dedicated colour. In this situation, we are no longer concerned with how pre-fetching affects overlapping cache lines that are unrelated to the buffer. Instead, we focus on how it affects the cache lines associated with the shared buffer. It is entirely plausible that even without a definitive conclusion about whether a specific cache line in the shared buffer is present, we may still conclude that pre-fetching ensures the *probability* of a cache line being present in the LLC is independent of the buffer's state before pre-fetching. In such a situation, we would not be able to observe or construct a timing channel via the microarchitectural state of the buffer.

As such, pre-fetching *may* be an effective solution for determinising the microarchitectural state associated with a buffer, it is just that the results of Buckley et al. [2023] strongly suggest that it is ineffective if the colour of the buffer is a colour common to any other domain. Pre-fetching may be a viable approach, but any implementation of shared memory that uses it must be benchmarked thoroughly to demonstrate that no timing channel via the microarchitectural state of the buffer exists after pre-fetching.

3.3 Noise Injection

The noise injection technique aims to reduce the signal-to-noise ratio of a timing channel by introducing "noise" into the attacker's measurements. This can be done either by injecting noise directly into timing measurements or by perturbing memory access patterns so that the attacker observes noisy behaviour. Approaches such as virtualised timestamps, as proposed by Vattikonda et al. [2011], attempt the former but remain largely infeasible under the current model of time protection due to lack of hardware support. Consequently, the more promising approach is the techniques that Brickell et al. [2006] employed to produce a secure AES implementation. The implementation outlined in the paper involves compacting, randomising and preloading lookup tables, which were sufficient in removing the timing vulnerabilities in AES outlined by Bernstein [2005].

Discussion and Takeaways

A potential application of this technique would involve touching random cache lines associated with a shared buffer on a domain switch, thereby introducing noise to any information that could be leaked via this buffer. However, as Cock et al. [2014] points out, noise is usually an ineffective means of closing channels. As they state, what is really needed is noise that is *anti-correlated* to the distribution of the timing channel. In many situations, this is impossible or infeasible to generate, or it may incur a significant performance overhead. It should also be noted that noise injection is not a principled solution; it relies heavily on experimentation, as the amount of noise that must be introduced to mitigate a channel depends on the capacity and resolution of the channel. As such, it is unclear if this technique is truly one that will find much application in any design for cross-domain shared memory or notifications.

3.4 Time-Padding

Ge et al. [2019] outlines the use of *time-padding* to mitigate channels that arise due to the latency of operations with non-constant execution time. The authors employ this approach to close the *cache-flush* channel, which results from variable L1 flush latencies. The method involves padding operations with non-constant latencies to their worst-case execution time, and the authors provide empirical evidence to support the efficacy of this approach, as the cache-flush channel outlined in the paper is sufficiently closed. Time-padding has also been explored thoroughly before, Braun et al. [2015] used it to design a set of compiler directives for generating fixed time functions, they do this by employing software-based time-padding and use uniform random noise to handle secret dependent error.

Discussion and Takeaways

This approach was effective in closing the cache-flush channel studied by Ge et al. [2019], as well as various other channels studied by Wistoff et al. [2023]. It is worth pointing that this method does have one fundamental flaw, failure to accurately bound the WCET latency will result in a complete failure of this mechanism to mitigate any timing channel. Additionally, a purely software-based approach to time-padding does present some performance overhead, justifying the introduction of a hardware-based padding primitive in Wistoff et al. [2023].

Regardless, this technique is clearly effective, and due to its integration within the existing system model, is very much a technique worth using when implementing cross-domain shared memory and notifications. It may find particular value if we ever encounter the need to flush or determinise the microarchitectural state associated with a shared memory buffer or pad signals on a notification to a constant time.

3.5 Cache Partitioning

The previous techniques focused on mitigating timing channels when resources are shared between domains. It is also important to explore partitioning schemes – particularly cache partitioning – and how these alternatives might support the construction of cross-domain notifications and shared memory. One possible direction for this thesis is to migrate seL4's current time protection model to a dynamic cache partitioning scheme. This would allow cache partitions to be created on the fly and assigned to shared buffers or data structures between domains. However, unfortunately due to time constraints this avenue of work was not explored.

3.5.1 CATalyst

Liu et al. [2016] outlines a method for dynamically partitioning the LLC using Intel's cache allocation technology (CAT). CAT allows for the creation of up to 4 "classes of service (COS)" and allows for the association of logical processors with a COS. The implication of this interface is that all threads on the same logical processor share a COS.

CATalyst introduces *secure pages* and splits the LLC in two, with one half used for pinning secure pages in the cache and the other for regular caching. This effectively turns the LLC into a "hybrid" cache, with the secure half acting as a software-managed cache and the unsecured half acting as a hardware-managed cache. At a high level, the idea is that a process will interact with the secure portion of the cache via the `map_sp` and `unmap_sp` system calls. The `map_sp` system call will load secure pages into the requesting process and pin them in the secure cache, the process can then use the secure page without fear of the page being evicted by cache contention.

The authors use this idea to close a side channel that they identified in *GnuPG*; using `map_sp` and `unmap_sp` to map/unmap the pages containing the "square routine" onto secure pages. The security evaluation provided in the paper illustrates that the technique is indeed capable of eliminating the identified side channel.

Discussion and Takeaways

One potential application of CATalyst in the context of this thesis is to use secure pages to back shared buffers. However, this approach presents several challenges. The paper does not describe how one would map secure pages into multiple processes, requiring further exploration. Moreover, utilising CATalyst would constrain the approach to x86 architectures. Furthermore, it is unclear if CATalyst is compatible with seL4's current use of cache colouring to partition the LLC, or if a suitable compatible alternative exists. Regardless, CATalyst provides a valuable exploration of an alternative cache partitioning scheme beyond cache colouring and introduces dynamic partitioning approaches that do not depend on static cache partitions.

3.5.2 SecDCP

Wang et al. [2016] presents a dynamic cache partitioning scheme known as SecDCP that allows for the association of security tiers with parts of the LLC. The authors outline that previous attempts to construct dynamically partitioned caches were vulnerable to cache timing attacks, as the hardware would need to infer partition sizes based on the run-time behaviour of applications and design their solution to prevent this. It is important to note that this paper's definition of dynamic partitioning consists of caches with arbitrary partitions, the amount of cache lines associated with these partitions are then updated *dynamically* based on the needs of applications.

SecDCP requires the system to denote several security tiers and a hierarchy among these tiers, from this hierarchy it can resize arbitrary partitions by moving cache ways from higher to lower tiers. The consequence of this design is that SecDCP can only guarantee the *one-way protection* of leakage from HI tiers into LOW tiers.

Discussion and Takeaways

SecDCP's reliance on defined security tiers to define partitions makes it somewhat limiting, as it would require consumers of time protection in seL4 to define a hierarchy among their domains. Additionally, since the hardware has complete control over how partitions are sized, there is limited room for software to control just how much of the cache is allocated to each partition, especially since in general every domain in a time-protected seL4 base system would be allocated to the same security tier (no domain is more important than another). All of these issues make it unclear specifically how much benefit approaches like SecDCP provide over traditional cache partitioning techniques such as cache colouring.

3.5.3 Cheshire's dynamically partitioned LLC

Cheshire's *dynamically partitioned last level cache* (DPLLC) is a recently integrated feature that enables the creation of cache partitions within its LLC. Its interface allows software to specify the number of cache sets allocated to each partition and tag arbitrary address ranges with a chosen partition. Unfortunately, there is not much public documentation available, but much can be learned by inspecting the System Verilog code associated with PULP Platform's LLC [PULP Platform, 2025a]. The DPLLC is meant to be used with PULP Platform's transaction tagger [PULP Platform, 2023], which allows for the association of arbitrary memory ranges with a special tag; in the case of DPLLC, this tag is the cache partition. The main functional feature of DPLLC is the ability to selectively flush arbitrary cache partitions.

Discussion and Takeaways

DPLLC presents a dynamic cache partitioning scheme that grants software fine-grained control over the cache sets assigned to partitions, alongside this, it allows partitions to be created with finer grained granularity than a page, which is something neither CATalyst

nor cache colouring allows for. We can make use of DPLLC by tagging all shared buffers and data structures used between a pair of domains as belonging to a special partition. On a domain switch, we selectively flush this partition, clearing all state related to these structures from the LLC. DPLLC also enables greater control over the latency of this flush operation, as we can directly control how many cache sets are associated with a partition.

Since DPLLC creates partitions on the granularity of sets, the sets used for a partition will always overlap with the sets allocated to some colour. Thus, using DPLLC would require the current implementation of time-protected seL4 to move away from cache colouring for partitioning the LLC and towards DPLLC, as the two approaches are fundamentally incompatible. There is a decent amount of engineering work involved with this task, and is well outside what is achievable during an honours thesis. However, moving the present implementation to DPLLC and re-implementing our proposed communication primitive designs may serve as an excellent direction for further work.

3.5.4 ARM MPAM

ARMv8.4-A introduced the *Memory System Resource Partitioning and Monitoring* (MPAM) architecture to enable the partitioning of memory system components within an SoC [Arm Limited, 2024b]. The architecture allows supervisory systems such as operating systems and hypervisors to physically partition memory system resources among different execution contexts. The core mechanism behind MPAM is the ability to tag every memory system transaction with special identification metadata that follows it throughout the memory hierarchy. The key identifier for MPAM is the *partition ID* (PARTID) which identifies the execution context a request belongs to and the partitions it has access to. Not all memory system components support MPAM, so before using MPAM the supervisory software must identify components that support MPAM as well as their supported capabilities. One component of interest to us is the last-level cache, which on many ARM SoCs can have their ways assigned by supervisory software to specific PARTIDs. ARM MPAM also extends beyond LLCs. Where memory components support it, MPAM can regulate bandwidth, allowing memory controllers to throttle requests from specific PARTIDs to enforce bandwidth limits.

Discussion and Takeaways

Unlike Cheshire's DPLLC, ARM MPAM is a general and flexible architecture, it simply provides mechanisms for supervisory software to create and control partitions and leaves the actual enforcement and implementation of those partitions up to SoC designers. Cheshire's DPLLC on the other hand targets a very specific RISC-V SoC, so anything developed using DPLLC maintains limited generality.

However, the generality of MPAM also maintains a downside, without a clear target platform it is difficult to assess how applicable or useful MPAM is, as its applicability is entirely dependent on what functionality the memory system exposes and how they respect partitions. If we have a memory system such as the one present in the ARM Neoverse core, that

supports way-based partitions instead of set-based partitions like DPLLC, then it becomes possible to use MPAM for LLC partitioning alongside cache colouring. One potential application for this would be to maintain the existing cache colouring implementation to partition memory between domains, however when creating a shared buffer we will pin that buffer to a specific way in the LLC. By doing this we can, in effect, confine accesses to the shared buffer to one specific cache way, preventing reads into the shared buffer from interfering with other entries within the LLC and vice versa.

Additionally, the fact that this is an ARM-only architectural extension means that it provides limited value within this thesis, which will focus on the implementation of shared memory and notifications on RISC-V. We may, in the future, find that when extending these constructs to ARM that MPAM maintains some practical value.

3.6 Speculation Barriers

ARM maintains an `SB` instruction, which acts as a memory barrier that prevents speculative execution until after the barrier has executed [Arm Limited, 2024a]. This in effect eliminates *particular* side channel attacks, such as the bounds check bypass outlined in Kocher et al. [2019]. Concurrently, x86 also features a speculation barrier via the `LFENCE` instruction, that was retroactively patched after the results of Kocher et al. [2019]; Lipp et al. [2018] to prevent speculation [Intel Corporation, 2024]. RISC-V presently has no standardised architectural support for speculation barriers; however, there does exist an existing extension proposal to introduce them in the near future [RISC-V International Security Horizontal Committee, 2025].

Discussion and Takeaways

Speculation barriers do not really solve timing channels, they simply patch the specific exploitation path of speculative execution. In essence, they are simply band-aids to a much deeper problem, as using them still allows for the construction of timing channels through the microarchitecture. As such, it is unlikely they will find much utility when attempting to implement cross-domain notifications and shared memory, as speculation barriers fail to truly mitigate timing channels.

3.7 Cross-Domain Notifications

Buckley et al. [2023] presents some minor theoretical coverage on the problem of cross-domain notifications, specifically, some concerns that may arise with their implementation. They highlight that we cannot use traditional notifications between domains as the act of signalling on a notification will result in the kernel writing to different addresses depending on the notification state, these different addresses will leave differing impacts on the microarchitecture, allowing for a potential timing channel. Additionally, they argue that the potential mitigations required to prevent the construction of timing channels

through notifications, such as the need to pad signals to a WCET, results in cross-domain notifications likely having to be a distinct kernel object from traditional notifications.

Discussion and Takeaways

The problems outlined by Buckley et al. [2023] present some interesting insights into how an implementation of cross-domain notifications may be achieved, many of which will form a baseline level of influence for the design of cross-domain notifications discussed in this thesis. The insight that notification delivery has the potential to be used as a timing channel through microarchitectural state implies the need to defer notification delivery to a domain switch, doing so allows the `fence.t` invocation that occurs before entering the domain again to erase the microarchitectural impact that notification delivery maintains. The previously discussed constraint, alongside the potential performance impact of various potential mitigations, also motivates the requirement to have cross-domain notifications be a distinct kernel object entirely.

3.8 Channel Benchmarking

Cock et al. [2014] uses the *channel matrix* [Shannon, 1948] to evaluate timing channels. A channel can be viewed as consisting of a sender S and a receiver R, the sender places inputs drawn from a set I into the channel, from which the receiver infers some output drawn from a set O [Cock et al., 2014]. The channel matrix then consists of the conditional probability of a reviver observing $o \in O$ given an input $i \in I$. Figure 3.1 illustrates an example channel matrix with a clear timing channel, and Figure 3.2 demonstrates a matrix with no observable channel.

The channel matrix is constructed by obtaining a large histogram from experimental data. For each output o_i, the technique observes the number of times an input i_j resulted in the receiver reading o_i. From this channel matrix, Cock et al. [2014] calculates the *Shannon capacity* [Shannon, 1948] as a measure of the strength of the channel. There already exists tooling for capturing these histograms from experiments within the seL4 ecosystem, namely, *channel-bench* [Ge and Millar, 2019] which will be used extensively over the course of this thesis.

Ge [2019] highlights a flaw in the approach Cock et al. [2014] uses to quantify the strength of observed channels. The Shannon capacity metric treats all pairs (i, o) as statistically independent; hence, it misses any patterns between pairs. Ge [2019] instead, proposes the use of *mutual-information* (MI) [Shannon, 1948] to quantify the strength of a channel. Mutual-information represents the average number of bits of information that a computationally unbounded receiver can learn about the input by observing the output. I.e. a high MI value indicates a large reduction in the uncertainty of the secret given some known output value. Ge [2019] also models time measurements as a continuous probability distribution instead of a discrete one like Cock et al. [2014], as treating output time as purely discrete would indirectly treat values as unordered and equivalent, i.e. a cluster of uniquely

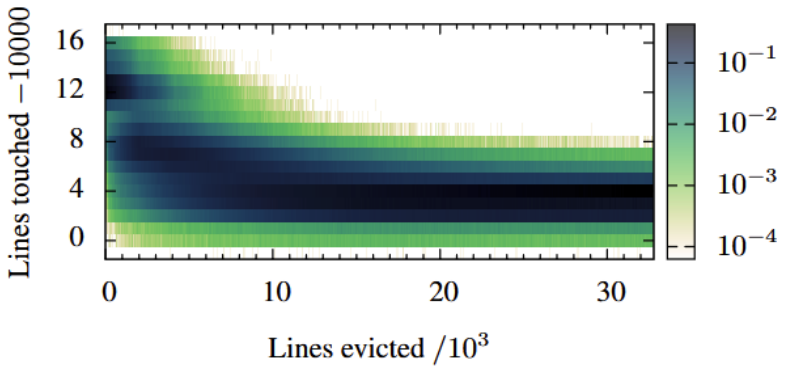


Figure 3.1: An example channel matrix [Cock et al., 2014].

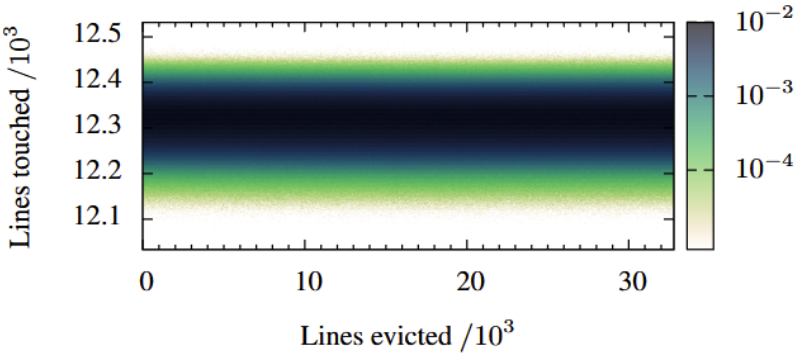


Figure 3.2: Channel matrix with no channel [Cock et al., 2014].

high values would be treated the same as a set of unique uniformly distributed values. After sampling and constructing the corresponding probability distribution for each input i, Ge [2019] then uses the rectangle method to estimate the mutual information (denoted as $\mathscr{M}$) for the collected data.

Since values are sampled from benchmarks, there will always be some level of noise. As such, it is impossible to full determine if a timing channel is closed via empirical measurements alone, instead, we can only determine if the data collected provides any evidence for the existence of a timing channel [Ge, 2019]. To determine this, Chothia et al. [2013]; Ge [2019] suggest first determining the mutual information of a zero leakage channel and comparing the mutual information of the collected benchmark results to that. This zero-leakage mutual information is estimated by shuffling the outputs in the collected benchmark data and associating them with random inputs, the mutual information is then computed and recorded, with this process being repeated 200 times [Chothia et al., 2013]. From the collected data, one can now calculate a mean, standard deviation and the exact 95% confidence interval required for a measurement to correspond with zero leakage (denoted as $\mathscr{M}_0$). If the observed mutual information $\mathscr{M}$ is greater than $\mathscr{M}_0$ then the observed data provides evidence for a leak, if the inverse is true ($\mathscr{M} \leq \mathscr{M}_0$) then the

dataset contains no evidence of a leak [Chothia et al., 2013; Ge, 2019]. There exists tooling for conducting this statistical test, namely `LeakiEst` [Chothia et al., 2013] which will be used throughout this thesis.

3.9 Summary

In this chapter, we looked at various techniques presented by the literature for mitigating timing channels. Unfortunately, some techniques such as noise injection often prove too complex and fragile for practical use; however, there remains much practical value in constant time techniques and time-padding. Ideas such as pre-fetching may work, but they will require rigorous benchmarking and evaluation when used. All these techniques can greatly guide the development of cross-domain notifications and shared memory, and we will see exactly how in Chapter 5 and Chapter 8.

We also explored the variety of techniques for cache partitioning outside of cache colouring, with approaches such as CATalyst enabling for dynamically locking pages of memory within the cache and DPLLC allowing the OS to assign arbitrary memory ranges to different partitions. Unfortunately, however, due to the scope of what can be achieved during a thesis, the usefulness of these techniques will remain unexplored, as many of them require using an alternative architecture or completely reworking the current implementation of time protection. These techniques do, however, serve as great potential options when attempting to port any constructs we introduce to alternative architectures.

The chapter concluded with a discussion of channel benchmarking and the techniques presently employed to quantify the strength of timing channels. The literature strongly suggests using mutual information to compute the strength of timing channels and provides plenty of existing tooling for doing so, with one such example being `LeakiEst` [Chothia et al., 2013].

Chapter 4

Benchmarking Methodology

Benchmarking timing channels is a key focus of this thesis, and as such it is worth looking at how exactly channel benchmarks will be conducted throughout this thesis. All benchmarks that we will develop will run on top of the *channel bench* toolchain [Ge and Millar, 2019], with the RISC-V specific modifications made by Buckley et al. [2023]. The toolchain features many methods for conducting various attacks over different microarchitectural resources such as the LLC, L1-I cache, L1-D cache, etc. We will be using these methods to develop the variety of timing channels that this thesis will explore.

4.1 Quantifying Leakage

We will quantify leakage using mutual information and compute the mutual information of a benchmark result using `LeakiEst` [Chothia et al., 2013]. Since all systems have a degree of noise, the mutual information measurement alone does not reveal much, and, a judgement about leakage can only be made relative to the mutual information corresponding to zero information leakage. The mutual information of a zero leakage channel is estimated by shuffling the observed outputs and associating them with random inputs, this process is repeated 200 times to attain a mean, standard deviation and 95% confidence interval for the strength of the zero information leakage channel, the 95% confidence interval is denoted as $\mathscr{M}_0$. We say that if the observed mutual information of our channel is greater than the 95% confidence interval for zero information leakage, then the benchmark result in question provides evidence for the existence of a timing channel. I.e. if $\mathscr{M} > \mathscr{M}_0$ then there exists a leak.

4.2 Channel Matrices

A channel matrix represents the conditional probability of an observed output (i.e. Spy probing time, y-axis) given some discrete input (i.e. number of cache sets primed by a

Trojan, x-axis). All channel benchmark results in this thesis will be presented with a channel matrix to visually represent a channel. Since channel matrices represent a conditional probability distribution, they must be presented as a heatmap, where colours in the heatmap indicate the probability as per the scale on the right. Figure 4.1 provides an example of a channel matrix with a clear timing channel. Since the conditional distributions of the cycle count observed by the Spy when the secret is 0 or 1 differ, we can conclude from the channel matrix that a timing channel exists.

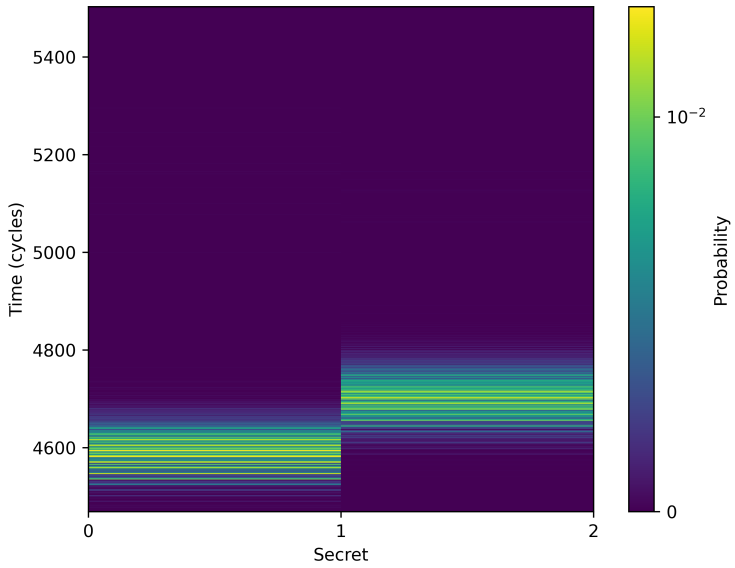


Figure 4.1: An example channel matrix.

4.3 Benchmarking Environment

All benchmarks are conducted on Cheshire, running on the Genesys2 board [Digilent, Inc.]. We will implement and benchmark all proposed designs on top of the port of time protection to Cheshire carried out by Julia Vassiliki. We will run all benchmarks with off-core peripherals such as the VGA controller disabled, as contention from these devices introduces timing channels where one should not exist, more information as to how this occurs is presented in Appendix B. All benchmarks will consist of a Trojan and a Spy, each confined to their domains D_t and D_s respectively. The benchmarks we conduct will assign each domain a time slice of one timer tick, with the amount of time between timer ticks varying based on the benchmark. Our benchmarks will also maintain the static domain schedule of, D_t followed by D_s. Unless stated otherwise, each domain will receive two cache colours each, equating to 64 KiB of space within the LLC allocated to each domain. When conducting a benchmark, we will collect 20,000 samples and construct a channel matrix from those 20,000 samples. As a reminder of relevant platform specifications, Cheshire runs at 50 MHz, maintains a 32 KiB L1 data-cache, a 16 KiB L1 instruction-cache, and a 128 KiB LLC.

Chapter 5

Notification Design

This chapter, and its corresponding implementation chapter, is dedicated to exploring the introduction of a new notification object that can be used between domains in time-protected seL4. The new object should allow for the transmission of signals between domains, and disallow any information that is not a signal from leaking between the two domains. We will make the previous requirement more concrete by first specifying a model of cross-domain notifications.

A cross-domain notification is an object that connects a signalling/sender domain A with a receiving domain B. Information can flow from $A \to B$ via the notification. Our design will maintain the following information flow requirements:

R1. Information can only flow from $A \to B$ by A explicitly signalling on the notification. Additionally, the impact that the execution of threads in A may have on the microarchitectural state associated with the notification should not enable information flow from $A \to B$.

R2. Only information *related* to signal delivery can flow from $A \to B$, for example, through changes induced in the microarchitectural state associated with the unblocked thread's TCB when delivering a signal to a thread in B. All other information flow from $A \to B$ is prohibited.

R3. No flow of information from $B \to A$ is permitted.

R4. Information about the communication between A and B via the notification should not leak to any external domain C.

Before continuing on with the rest of this chapter, it is worth revisiting the existing seL4 notification semantics as outlined in Chapter 2. All discussed details are also summarised in the state machine illustrated in Figure 2.3.

1. All notifications start off in the `IDLE` state. This state models a notification that presently has no waiters or threads that have actively signalled on it.

2. If a signal is made on a notification in the `IDLE` state, it transitions to the `ACTIVE` state and the badge value associated with the caller's notification capability is stored in the notification word.

3. If a wait is made on a notification in the `IDLE` state, it transitions to the `WAITING` state and the thread that requested the wait is immediately blocked.

4. If a notification is in the `WAITING` state, i.e. there is a thread currently in its wait queue, then a signal on that notification immediately delivers the provided badge to the head of the wait queue.

5. If a notification is in the `ACTIVE` state, i.e. has been signalled on previously, then a signal on that notification just performs a bitwise OR of the notification's current badge with the badge provided by the signalling thread.

6. If a notification is in the `ACTIVE` state, i.e. has been signalled on previously, then a wait on the notification will resolve immediately and not block the calling thread.

Now that the present semantics for notifications have been outlined, it is useful to discuss the constraints imposed by the information flow requirements when attempting to generalise notifications so that they can be used across domains. First, consider the `signal` operation. Presently, when performing a signal on a notification, seL4 performs different actions depending on the notification's current state. I.e. if the notification is in the `WAITING` state, then the signal will unblock the first queued thread. Alternatively, if the notification is in the `ACTIVE` state, the signal will simply bitwise-OR the badge value against the current notification word. As Buckley et al. [2023] points out, these differing operations will leave different impacts on non-partitioned microarchitecture. Namely, the L1 data and L1 instruction caches. This cannot be mitigated easily, and allows for information to flow from $B \rightarrow A$ via B's readiness to receive a signal; completely violating the information flow constraints. The only reasonable solution to this problem is to defer the actual delivery of notifications to the domain-switch, specifically, at the start of domain B's time-slice. By deferring delivery to a domain switch, we make it impossible for A to observe the microarchitectural impact a signal has, as it is erased by the `fence.t` instruction that is executed before entering domain A again. The issue pointed out by Buckley et al. [2023] also elicits a secondary constraint, namely the constraint that the instruction-cache lines and memory addresses that the `signal` operation touches be independent of the notification state. We can satisfy the aforementioned constraint by drawing upon the constant-time programming techniques discussed in Chapter 3.

5.1 The Problem of Multiple Signals

Since the information flow requirements mandate that signal delivery take place at domain switch time, there is a question as to how multiple signals on the same notification are handled. To see this, assume that we have a notification with two present waiters. Two signals are then performed on this notification, the first with badge σ_1 and the second with

badge σ_2. The existing notification semantics would deliver σ_1 to the first waiter and σ_2 to the second waiter. This behaviour is difficult to replicate when notifications are delivered at the domain switch, as we lose information regarding what thread should receive what badge value. Introducing a separate structure for storing these badges for delivery does not work as it would imply the need for `signal` to touch different addresses depending on the notification state, violating the $B \rightarrow A$ information flow constraint. The only reasonable resolution to this problem is to accumulate the badge values by bitwise or'ing them and delivering the accumulated value $\sigma_1 \mid \sigma_2$ to all unblocked threads. The problem of multiple signals also introduces a secondary question. Since all unblocked threads must see the same badge value, if n signals to a notification are made, should we awaken n threads on the next domain-switch or just one? We will now look at each option in detail and examine its implications.

5.1.1 N signals awakens N threads

With these semantics, if a thread in A makes n signals to a notification with n waiters, then all n waiters are unblocked. The primary challenge with this design lies in actually delivering the notifications at domain switch time without introducing a channel. In the present implementation of time-protection on RISC-V the time taken to perform a domain switch from $A \rightarrow B$ is deducted from the time allotted to the first thread in B, with the part of the domain switch whose latency is dependent on A being padded to a worst-case execution time. Implicit in this operation is the assumption that the latency of a domain switch is less than the time between timer interrupts. If this assumption is violated, then the latency of the domain switch operation becomes observable by an external domain C as it affects when C is scheduled.

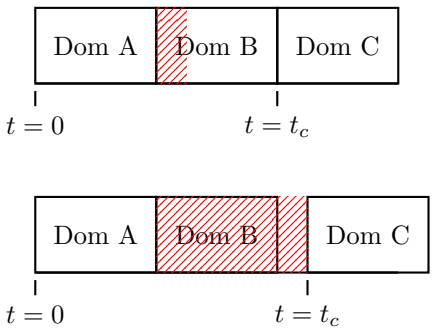


Figure 5.1: Domain switch latency (red) affects scheduled time of C.

Allowing an unbounded number of notification deliveries to occur during a domain switch complicates the design, as it removes any guarantee on the upper bound of domain switch latency. This could, in extreme cases, lead to the catastrophic scenario illustrated in Figure 5.1, where there is a channel through the fact that C is scheduled later in time than it should be. In principle, we could resolve this issue by determining the maximum number of notifications that may ever be delivered and then configure the timer interval to accommodate the delivery of that many notifications safely. However, such an approach

would expose an extremely brittle API to users, where even a minor misconfiguration of the bound could completely compromise the time-protection guarantees of cross-domain notifications. Another solution would be to deliver as many notifications as possible within B's time slice and defer the remainder to the next time-slice. However, this introduces a degree of implementation complexity, as it is difficult to guarantee that the system will never exceed B's allocated budget.

5.1.2 N signals awakens 1 thread

The primary shortcoming of the previous design was that domain A could awaken an unbounded number of threads in domain B. Unfortunately, accommodating for this behaviour introduced some implementation complexity. All this complexity can be mitigated by redefining the semantics of notifications so that all the signals issued by A within its time-slice are coalesced together, awakening only a single thread in B. Under this model, we only need to ensure the timer interval is long enough to support the delivery of a single signal, a significantly easier and less error-prone task than what was suggested earlier. Due to the simplicity of this design, this is the approach that this thesis will focus on; however, the n signals awakes n threads model provides an excellent source of further work.

5.2 Notification Design and API

The high-level design for cross-domain notifications that this thesis will explore can now be summarised rather simply. The design involves introducing a new cross-domain notification object. The delivery of signals on that notification is deferred until the next domain switch that enters B. We will perform the notification delivery after switching to B's kernel image, i.e. after the `fence.t` invocation. If A performs multiple signals on the notification while a thread in B is blocked on it, we will simply accumulate all these signals and only awake one blocked thread in B.

We will now look at the API exposed by cross-domain notifications in more detail. The API will consist of three primary operations `Signal`, `Wait`, and finally, `Poll`. Like traditional notifications, these operations are primarily wrappers around the `Send`, `Recv` and `NBRecv` system calls. Additionally, since the proposed design for cross-domain notifications requires a mechanism to associate notifications with their receiver domain B, the design extends the `Domain` capability with a new `SetNotifications` invocation. The new invocation allows for this relationship between notification and domain to be constructed.

5.2.1 Signalling

A thread can signal on a notification via the `Signal` operation. Upon doing so, the badge associated with the cross-domain notification capability is bitwise-OR'd against the present notification word stored in the notification. A signal will only ever unblock a single thread,

with multiple repeated signals within the same domain time-slice only accumulating badge values and never unblocking more than one thread. At the next domain switch entering the receiver domain, if the notification was marked as ready for delivery, the head of the blocked queue is unblocked and the notification word associated with the notification is reset to zero.

5.2.2 Waiting

A thread can block on a notification via the `Wait` operation. Upon doing so, the thread is placed into a queue of threads that are presently blocked on the target notification. If a `Wait` is performed on a notification that had previously been signalled on but had no waiters, then the `Wait` operation resolves immediately and the notification word is written to the caller's badge register. The chosen semantics for `Wait` preserve the existing behaviour of notification waiting, while also supporting a clear definition of `Poll`. If a `Wait` on an active notification were to block the calling thread until domain B's next time-slice, `Poll` would no longer be well-defined, as it would allow a thread to observe the notification state earlier than what `Wait` would allow for.

5.2.3 Polling

A `Poll` is much like a traditional `Poll`. If the notification is presently active, the call will succeed and the notification word associated with the notification will be returned. If the notification is not active then the call will return immediately but nothing will be written to the calling thread's badge register.

5.2.4 Domain association

Notifications need to be associated with the receiver domain so that signals can be delivered at domain switch time. This is done via the `SetNotifications` invocation attached to the domain capability. The invocation takes the receiver domain's number and the capability for the notification and attaches it to a per-domain queue of notifications that require transmission upon entering a new domain.

5.2.5 API as a state machine

Much of the API for cross-domain notifications can be summarised into a simple state machine consisting of four states: `Idle`, `Active`, `Waiting` and finally `Deferred`. The `Idle`, `Active` and `Waiting` states mean the same thing as traditional notifications, they indicate if a notification presently has any waiters or if the notification has previously been signalled upon. The new state `Deferred` indicates if a notification requires delivery at domain-switch time. A notification will only transition into this state upon a `Signal`

and will immediately transition out after a domain switch into the receiver domain. We can summarise all these details into the state diagram illustrated in Figure 5.2. For brevity, the diagram refers to t_n as the numbers of remaining threads that are presently blocked on the notification after delivery.

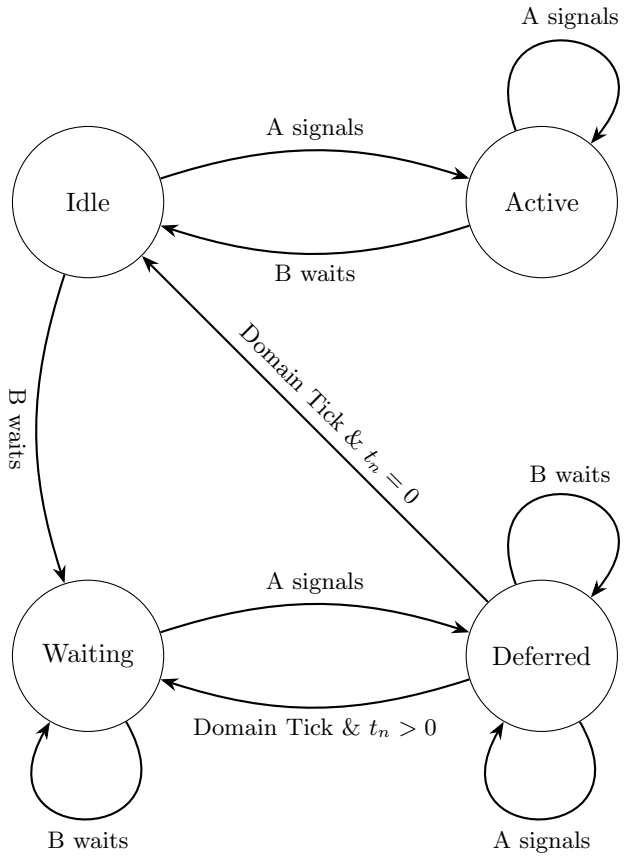


Figure 5.2: Cross-domain notification state diagram.

5.3 Information Flow Requirements

We now revisit the information-flow requirements outlined at the start of this chapter and discuss any modifications that may need to be made to our design to meet them. To maintain the information flow requirements discussed in the initial cursory design, we must take special care to ensure that the act of signalling or receiving on a notification cannot be used as a timing channel.

We first consider the act of signalling on a notification. Since signal delivery is now deferred to the domain switch, it is impossible for the latency of a signal to be dependent on the number of threads that are presently waiting on the notification. Additionally, as long as the same actions are performed regardless of the notification state, the act of signalling will not leak any information about the notification state through the microarchitecture.

As such, the only channel via the `signal` operation that now needs mitigation is the cache side channel, where the latency of the `signal` operation depends on the notification's presence in the LLC. This channel is easily mitigated by padding the `signal` operation to its WCET. The requirement to pad `signal` also provides a justification as to which colour the notification should be allocated out of. If we are to allocate the notification from the same colour as the sender, then the cache lines associated with the notification may overlap with the cache lines associated with the sender domain's kernel clone. The implication of this is that when signalling on a notification, the act of touching the notification may inadvertently evict cache lines associated with the kernel's system call exit path, affecting the latency of the system call as a whole. This implies that we must pad the *entire* system call to some WCET.

There is, however, an even simpler solution. If we do-away with the cache line overlap and allocate the notification out of the receiver's colour, then accessing the notification will never evict cache lines associated with the sender's kernel. With this change, only the signal-handling function within the system call needs to be padded to its WCET, leading to a significantly simpler implementation. If we allocate the notification from the receiver's colour, pad the signal operation to its WCET, and ensure that the signal performs no operations that depend on the notification state, then we can be assured that there will be no information backflow from $B \to A$.

The information flow requirements also mandate that there should be no flow of information from $A \to B$ outside to what is conveyed by a signal. Since the notification is allocated from the receiver (B's) colour, the only way for A to impact the microarchitecture associated with the notification and convey any information is by A explicitly signalling on the notification. As such, we require no further mitigations.

The final information-flow requirement to address is the need to prevent leaks to an external domain C. Under the proposed semantics for cross-domain notifications, the only potential source of leakage arises from variability in domain switch latency. However, as discussed earlier, if the timer interval is sufficiently long enough to support the delivery of a single notification, this requirement remains satisfied.

Regarding domain switch latency, another question is whether the domain switch latency overhead introduced by notification delivery could act as a channel to leak information from $A \rightarrow B$. Since the variability of this operation can only depend on the activities of B, it is impossible for A to influence this latency without actually performing a signal and triggering a delivery. In such a case, the information conveyed by this timing variation is strictly equivalent to the information conveyed by the signal.

So in summary, to ensure that we are completely maintaining the information flow requirements outlined earlier, we must ensure that the notification object is allocated from the receiver B's colour and the `signal` operation is padded to its WCET. No further design modifications are required.

5.4 Summary

To summarise, this thesis will focus on a simplified implementation of cross-domain notifications that maintain the API semantics outlined in Section 5.2. To maintain the information-flow requirements outlined initially, the notification will be allocated from the receiver's colour, and the `signal` operation will be padded to its WCET and never perform any state-dependent actions. All these considerations lead to a relatively simple design for cross-domain notifications, which should maintain a relatively straightforward implementation.

Chapter 6

Notification Implementation

Chapter 5 focused on providing a rough illustration of the chosen design for cross-domain notifications. This chapter will thus inspect some implementation concerns with cross-domain notifications.

6.1 General Implementation Details

The implementation of cross-domain notifications presented in this thesis involves extending the kernel to introduce a relatively simple structure consisting of the cross-domain notification's present state, present value for the notification word, a linked list of TCB's presently blocked on the cross-domain notification, and finally, the `next` pointer associated with a linked list of cross-domain notifications attached to a domain. The linked list of blocked TCBs is maintained by hijacking the intrusive linked list that is presently used for normal notifications. This is safe, as it is impossible for a thread to be blocked by a regular notification and cross-domain notification at the same time. When attached to a domain, notifications are placed in a per-domain linked-list of notifications that require transmission when entering the associated domain. The head of this linked-list is placed in scratchpad memory, and the next pointers are placed within the actual notification object itself. It is worth noting that the head of the linked-list could also reasonably be placed in partitioned kernel memory, as our notification semantics dictate that notification delivery only occurs after switching the kernel image.

6.2 Signal Implementation

We now turn our attention towards the implementation of the `signal`/`send` operation. As outlined in Chapter 5, the `signal` operation will simply just mark the notification as requiring delivery at a domain switch. Additionally, the actions performed by `signal` should be independent of the notification state. The second constraint mandates that the

`signal` routine must be branchless, or specifically, should never branch on a secret such as the notification state. Listing 4 provides some pseudocode for the implementation of `signal`.

```
#define b(x) -(!!(x))
#define branchless_assign(condition, if_value, else_value) \
    (((b(condition)) & (if_value)) | (~(b(condition)) & (else_value)))

void sendSignal(cross_dom_notification_t* ntfnPtr, word_t badge) {
    word_t old_badge = get_ntfn_badge(ntfnPtr);
    word_t new_badge = old_badge | badge;

    notif_state_t notif_state = get_notif_state(ntfnPtr);
    notif_state_t new_state = branchless_assign(
        (notif_state == Waiting) || (notif_state == Deferred),
            /* if = */ Deferred,
            /* else = */ Active
    );

    set_ntfn_badge(ntfnPtr, new_badge);
    set_ntfn_state(ntfnPtr, new_state);
}
```

Listing 4: Illustration of the send routine.

The provided code-snippet makes reference to the `branchless_assign` macro, this is a simple macro that will perform a conditional assignment using arithmetic and bit-wise operations. The expanded macro will convert any boolean value to a string of 1's, allowing it to be chained together with other operations to perform an assignment without any branches. What has been left out from the code snippet above is the WCET padding operation applied to the `signal` routine, this is largely because implementing software-based time-padding in a manner that eliminates all timing channels deserves a dedicated section.

6.3 Time-Padding

Time-padding involves padding some variable operation to a constant cycle count t_c. One potential way to pad an operation to a target number of cycles t_c is to sample the number of cycles the operation takes and then busy-wait in a tight loop until t_c total cycles have elapsed. Some rough pseudocode of this operation is provided in Listing 5. However, when we apply this directly to the `signal` operation, there still exists a minimal timing channel. The channel-matrix for this channel when padding to 700 cycles is presented in Figure 6.1.

```
register uint64_t start = riscv_read_cycle();
// notification signal code
register uint64_t end = riscv_read_cycle();
register uint64_t elapsed = end - start;

// Busy wait until elapsed cycles
while (elapsed < target) {
    end = riscv_read_cycle();
    elapsed = end - start;
}
```

Listing 5: Time-padding loop.

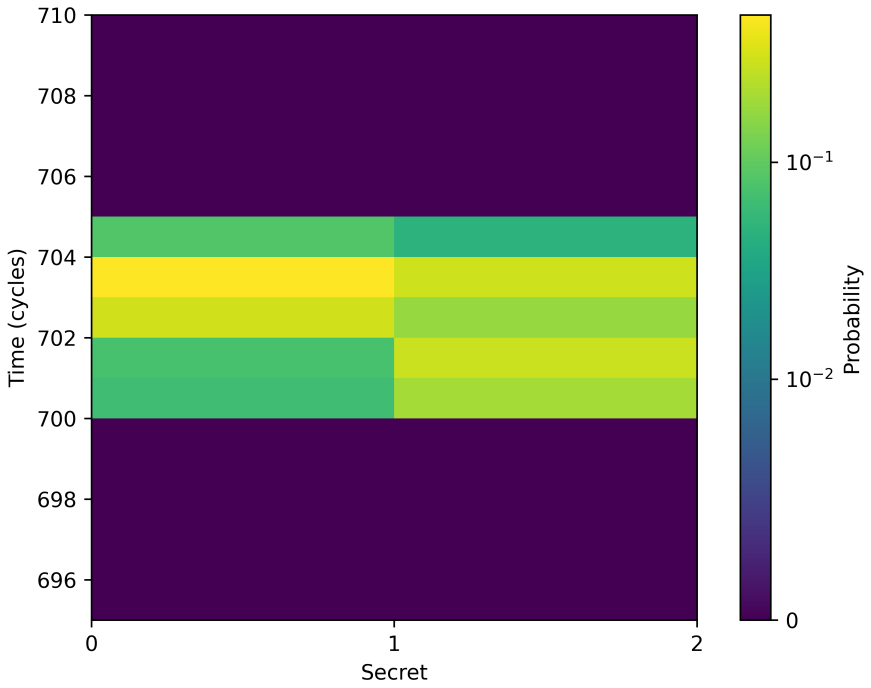


Figure 6.1: Benchmark result with target of 700. $\mathscr{M} = 0.0854$. $\mathscr{M}_0 = 0.0005$.

In the experiment we conducted to attain this channel-matrix, the Trojan is the domain B, and maintains a priming buffer in its own domain. To send a secret of 1, it will prime the entire buffer in order to evict the notification from the LLC. To send a secret of 0, it will do nothing. The primary feature of the channel matrix illustrated in Figure 6.1 is the fact that `elapsed` varies between 700 and 704. This variation in `elapsed` is explained by the fact that each iteration of the busy-wait loop has a latency greater than one cycle, so there will always be a margin of error when attempting to pad to some constant cycle

count. In fact, it is this very error that gives rise to the observed timing channel, as the two possible distributions of `elapsed` diverge quite greatly. In general, if there are points between the two distributions where the probabilities diverge greatly, then there will always be a timing channel. This is because an adversary can be more confident that the observed value came from the secret distribution that has a higher probability for that value.

The natural question, then, is how the naive padding loop gives rise to these two probability distributions, and why they differ. To answer this, we examine the mechanics of the time-padding loop. In the steady state, each iteration of the padding loop will take the same amount of time, say n cycles. So in the steady state, the padding loop can essentially be viewed as a function that will repeatedly add n to some cycle count x until it reaches t_c. As such, we can characterise the padding loop as a simple mathematical function f that takes some input cycle x and spits out some padded cycle count $f(x)$ as an output, where

$$f(x) = x + n \cdot \left\lceil \frac{t_c - x}{n} \right\rceil .$$

We can further simplify this function through some algebra, providing the following expression for the time-pad function:

$$\begin{aligned} f(x) &= x + n \cdot \left\lceil \frac{t_c - x}{n} \right\rceil \\ &= x - n \cdot \left\lfloor \frac{x - t_c}{n} \right\rfloor \\ &= t_c + (x - t_c) - n \cdot \left\lfloor \frac{x - t_c}{n} \right\rfloor \\ &= t_c + (x - t_c \mod n). \end{aligned}$$

This characterisation lets us view the padding loop as a transformation on a probability distribution: it takes some initial latency distribution and produces a new one by applying f. Thus, to see why the naive padding loop fails, we must understand how f alters key features of the `elapsed` distribution and how these alterations create timing channels.

As an initial example, consider two initial distributions for `elapsed`, one sampled from when the Trojan primes, and the other from when the Trojan does not prime. Assume that the primed distribution has mode x_p and the unprimed distribution has mode x_u. If $x_p - t_c$ and $x_u - t_c$ are not congruent modulo n, it is obvious that this padding function will map them to distinct values; this is particularly problematic if the distributions are very "spiky" with not a lot of variance, as it will result in the final output distributions having values concentrated around a particular mode, where the modes of the two distributions differ. Another problematic case for the padding operation is if one of the distributions is uniform while the other is "spiky". The padding function will map the latencies from the uniform distribution uniformly among the values: $\{t_c, t_c + 1, t_c + 2, \cdots, t_c + n - 1\}$, while it will cluster the spiky distribution around $t_c + (m - t \mod n)$, where m is the spiky distribution's mode. We can see that because the padding operation is sensitive

to characteristics of the initial input distributions, there are several ways it can fail and leave a residual channel. Therefore, a simple or naive padding strategy is insufficient for eliminating timing channels through execution latency.

6.3.1 Solution 1 — Hardware support

All the padding loop is doing is mapping values from the input distribution to distinct buckets. Now, if there is only one such bucket, i.e. if $n = 1$, then it is clear that the distribution after time-padding will not retain any qualities of the input distribution as all modes or "features" will map to the same bucket. Constructing a padding-loop in software where $n = 1$ on Cheshire is infeasible as there will always be some cycle overhead associated with sampling the current cycle count, comparing it to a target value, and potentially breaking out of the loop upon completion. To implement a loop where $n = 1$ we require direct hardware support, similar to what is provided by `fence.t`. Presently, there exists no such support on Cheshire, and `fence.t`'s padding capabilities are unusable as they cannot be used in the system call path where there is no timer interrupt to pad relative to.

6.3.2 Solution 2 — Noise injection

Braun et al. [2015] addresses the error problem by adding noise: they sample from a uniform distribution and add the resulting number of extra cycles to the final value after padding. To see why this works it is useful to first concentrate on the padding function itself

$$f(x) = t_c + (x - t_c \mod n).$$

Due to the mod operation in the function, we can essentially view the padding transformation as taking some x and then mapping it to some point on a circle with n distinct boxes. The key issue with the original transformation function was the fact that f would not spread the input distribution uniformly along this circle, and would potentially concentrate the distribution at certain boxes along the circle. What Braun et al. [2015] achieve by introducing noise is the ability to "force" uniformity onto f. To see how, assume we are an ant sitting in some box along the circle, we are not concerned about how we arrived at that box, just the fact that we are there. Now say we roll a die with the numbers $\{0, 1, \cdots, n-1\}$ written onto it, and after doing so, we move that many boxes along the circle. Since the number of boxes we move past is random and has a uniform distribution, we note that regardless of where we started off, we can end up essentially anywhere else along this circle with equal probability. This fact allows us to "force" uniformity onto f, as adding a uniform random variable $U \sim \mathcal{U}(0, n-1)$ to the output of f "washes" away the distribution and forces the final distribution to be uniform.

The actual implementation of this approach is a little more tricky though, as we need to find a way to burn a random number of cycles on top of the padding operation. The

approach we will take in this thesis involves subtracting a random number sampled from $\mathcal{U}(0, n-1)$ from the `start` timestamp before padding, and then using this adjusted value in the time-padding loop. Doing this is essentially equivalent to turning the time-padding loop into the transformation

$$f(x) = t_c + ((x - U - t_c) \mod n).$$

Where $U \sim \mathcal{U}(0, n-1)$. It is not obvious why this operation has the same "washing" away properties as we previously discussed, so we will prove the effectiveness of this technique with a rough mathematical argument presented in Appendix A.

Experimental evidence
To verify the effectiveness of this approach, we will construct a similar experiment to what was constructed to demonstrate the initial timing channel with naive padding. Some rough pseudocode for the experiment is provided below, the important detail worth pointing out here is the fact that instead of padding `end` − `start`, we are instead padding `end` − `start` − `random()` mod 5.

```
register uint64_t start = riscv_read_cycle();
register uint64_t __start = start + random() % 5;

// notification signal code
// ...
// ...

register uint64_t end = riscv_read_cycle();
register uint64_t elapsed = end - __start;

// Busy wait until elapsed cycles
while (elapsed < target) {
    end = riscv_read_cycle();
    elapsed = end - __start;
}
```

This experiment was run on Cheshire and 20,000 samples were collected. When padding to a target value of 700 and sampling `elapsed` after padding, we attain the channel matrix in Figure 6.2. The two distributions are clearly uniform, demonstrating the effectiveness of this approach. It is worth pointing out that an adversary attempting to use the execution latency as a timing channel does not observe `elapsed`, they actually observe `end` − `start`. To demonstrate that this value is not secret dependent either, we construct a channel matrix where `end` − `start` is the "output" value, giving us the result in Figure 6.3. The distributions in this channel matrix are not uniform, they are actually the distributions attained by summing two uniform random variables. The reason for this is evident via

some algebra.

$$\begin{aligned}\texttt{end} - \texttt{start} &\sim \texttt{end} - (\texttt{\_\_start} - \mathcal{U}(0,4)) \\ &\sim (\texttt{end} - \texttt{\_\_start}) + \mathcal{U}(0,4) \\ &\sim \mathcal{U}(700,704) + \mathcal{U}(0,4).\end{aligned}$$

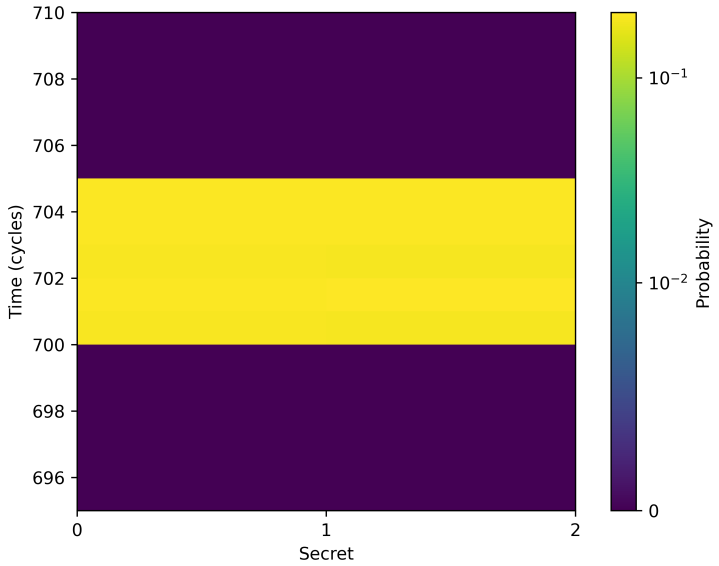


Figure 6.2: Benchmark result with target of 700. $\mathscr{M} = 0.0000$. $\mathscr{M}_0 = 0.0004$.

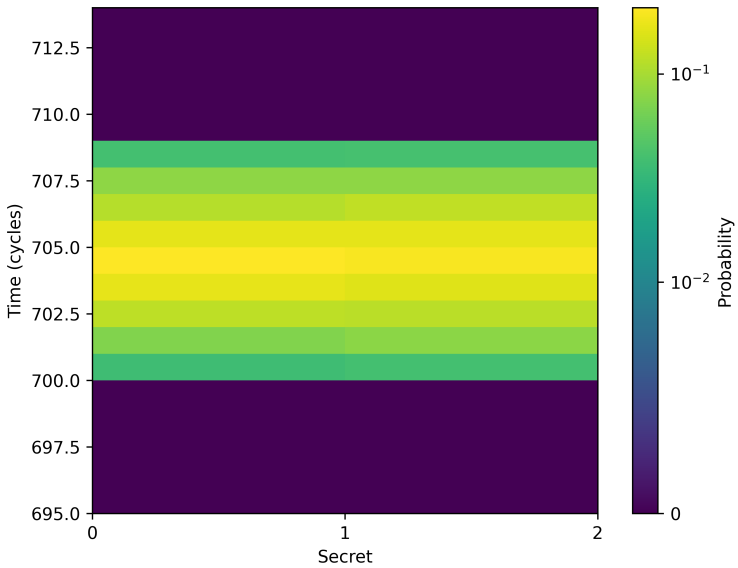


Figure 6.3: Benchmark result with target of 700. $\mathscr{M} = 0.0002$. $\mathscr{M}_0 = 0.0003$.

6.3.3 Solution 3 — Error correction

The fundamental issue with naive time-padding is that padding preserves qualities of the original distributions. The previous solution attempted to resolve this by inserting noise, however we can also attempt to resolve this issue by determining how far from some pre-defined upper bound `elapsed` is after padding. We then execute `noop`'s until we reach that upper bound. The software-routine for this operation is relatively straightforward. The RISC-V assembly for this operation is illustrated in Listing 6. A key implementation detail that is not immediately obvious is the need to align the array of no-ops to a cache line. This requirement follows from the information backflow constraints discussed in Chapter 5. Without cache-line alignment, different values of `elapsed` would access different L1-I cache lines, resulting in a potential channel through the L1-I cache.

```
li   t0, 700
sub  t0, (elapsed), t0
la   t1, 1f             # jump to the address that will result in the
slli t0, t0, 1          # correct offset
add  t1, t1, t0         #
li   t2, 706            # don't jump if elapsed > 706 (never happens)
bge  (elapsed), t2, 2f  #
jr   t1                 # finally jump to correct noop
.align 4                # align so all noops are on the same
1:                      # L1-I cache line
    addi x0, x0, 0
    addi x0, x0, 0
    addi x0, x0, 0
    addi x0, x0, 0
    addi x0, x0, 0
    addi x0, x0, 0
2:
```

Listing 6: RISC-V assembler for error correction.

Experimental evidence
Like the previous approach, we require experimental evidence to determine the validity of this approach. We construct an experiment similar to what was previously conducted, except this time we construct a channel matrix using `end` − `start` where `end` is sampled after error correction. The pseudocode for the experimental setup is provided below.

```
register uint64_t start = riscv_read_cycle();
// notification signal code
register uint64_t end = riscv_read_cycle();
register uint64_t elapsed = end - start;

// Busy wait until elapsed cycles
while (elapsed < target) {
    end = riscv_read_cycle();
    elapsed = end - start;
}

perform_error_correction(); // inlined
end = riscv_read_cycle();
log_tuple(notification_state, end - start)
```

The channel matrix we attain when padding to 700 cycles and collecting 20,000 samples is provided in Figure 6.4. The design clearly works, as evidenced by the channel matrix.

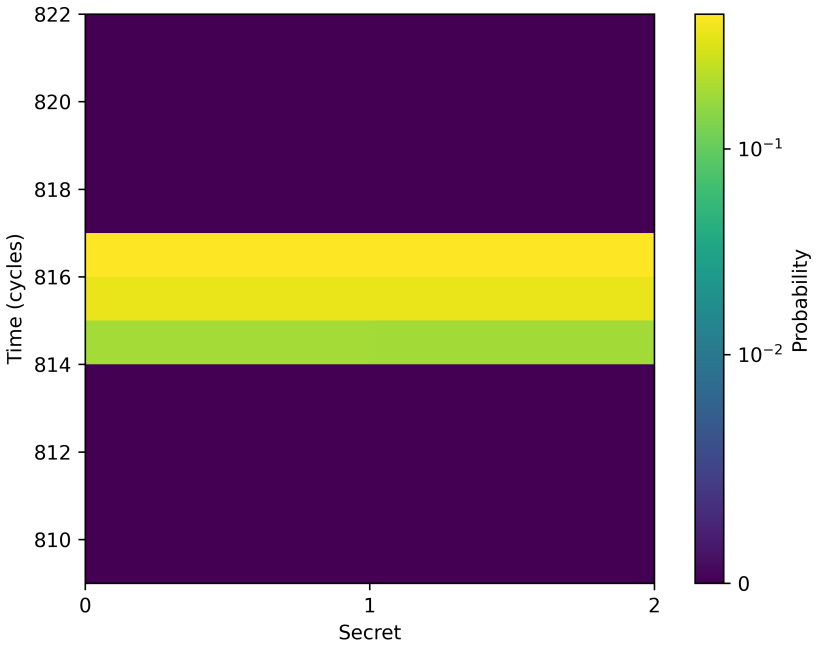


Figure 6.4: Benchmark result with target of 700. $\mathscr{M} = 0.0001$. $\mathscr{M}_0 = 0.0004$.

6.3.4 Bounding the WCET

Before performing an evaluation of the techniques presented earlier, we must first bound the WCET of the signal routine. The WCET can be found experimentally by saturating the LLC and L1-D with dirty cache lines and populating the L1-I with unrelated instructions. After saturating the caches, we perform a `signal` operation and log how long the `sendSignal` function takes in particular. Saturating the LLC and L1 will ensure that no part of the kernel is present in any part of the cache hierarchy, and bringing in cache-lines associated with the kernel will force the eviction of dirty cache lines that must be written back to main memory. Repeating this experiment 10,000 times on Cheshire provides a WCET of 230 cycles for the `sendSignal` function. The implementation of `send` presented in this thesis will thus pad `sendSignal` to 250 cycles to retain a margin of error.

6.3.5 Evaluation and discussion

To decide between the two approaches, we will conduct a benchmark to determine the overhead each respective approach introduces to the `signal` operation. The benchmark we carry out will time how long the *entire* `send` operation takes, rather than just sampling the value of $\texttt{end} - \texttt{start}$ after time-padding. By doing this, we incorporate effects such as a latency overhead imposed by the branch mis-predict on the final iteration of the padding loop.

To benchmark the noise-based approach, we use a *linear shift feedback register* (LFSR) seeded with the current cycle time. This choice has clear security drawbacks, as LFSRs are susceptible to known-plaintext attacks and seeding with the current cycle makes the output predictable. The RISC-V platform, and Cheshire in particular, does not provide a hardware-based random number generator or any entropy sources that could be used to seed a PRNG. Despite these limitations, evaluating performance with an LFSR remains useful, as it provides a reasonable lower bound on how the noise-based method compares to the error-correction approach. Benchmarking the noise approach also requires rebounding the WCET. When carrying out the same methodology as previously discussed, we arrive at a new WCET bound of 350 cycles when including the cost of generating a random number using the LSFR. The results of the micro-benchmarks are presented in the table below.

Method	**Mean (Cycles)**	**Standard Deviation**
Error Correction	350	5
Noise Introduction	398	13

We observe that even with a relatively insecure and lightweight method of generating random numbers, the error-correction approach maintains slightly superior performance. Consequently, we adopt the error-correction approach for time-padding in the remainder of this thesis, as it offers lower latency overhead and introduces no security-related complexity. The situation may differ on architectures equipped with a hardware-based random number generator, but no such feature currently exists on Cheshire.

6.4 Wait Implementation

The implementation of `wait` just involves progressing the notification state machine based on its present state. As was outlined in Chapter 5, if the notification is `idle` then a `wait` progresses it to the `waiting` state and if a notification is `active` then `wait` progresses it to the `idle` state. There are also no special mitigations that need to be performed by the wait routine, as the notification is allocated from the receiver's colour. As such, we are presented with the rough implementation of `wait` presented in Listing 7.

```
void receiveSignal(
    cross_dom_notification_t* ntfnPtr,
    tcb_t* thread,
    bool_t isBlocking
) {
    word_t        badge       = get_ntfn_badge(ntfnPtr);
    notif_state_t notif_state = get_notif_state(ntfnPtr);
    tcb_queue_t   wait_queue  = get_ntfn_wait_queue(ntfnPtr);

    if (notif_state == Active) {
        set_ntfn_badge(ntfnPtr, 0);
        set_ntfn_state(ntfnPtr, Idle);
        set_thread_badge(thread, old_badge);
        return;
    }

    if (!isBlocking) {
        set_thread_badge(thread, 0);
        return;
    }

    tcb_queue_t updated_queue = tcbAppend(wait_queue, thread);
    notif_state_t new_state = notif_state == Idle
            ? Waiting,
            : notif_state;

    set_ntfn_wait_queue(ntfnPtr, updated_queue);
    set_ntfn_state(ntfnPtr, new_state);

    blockTCB(thread);
}
```

Listing 7: Illustration of the wait routine.

6.5 Domain Switch Delivery

Notifications are placed in a per-domain linked-list, with the head of the linked-list stored in scratchpad memory. On a switch into a new domain, the kernel iterates over all notifications for which the domain is the receiver and advances their state according to the state machine in Chapter 5. If a notification is in the `deferred` state, then the head of its blocked queue is unblocked and the notification's current word is written to the threads badge register. With all these considerations, we arrive at the rough pseudocode for the domain switch operation presented in Listing 8.

```
void domainSwitchDelivery(cross_dom_notification_t* ntfnPtr) {
    notif_state_t notif_state = get_notif_state(ntfnPtr);
    tcb_queue_t   wait_queue  = get_ntfn_wait_queue(ntfnPtr);
    word_t        badge       = get_ntfn_badge(ntfnPtr);

    if (notif_state == Deferred) {
        tcb_t* thread = wait_queue.head;
        tcb_queue_t updated_queue = tcbDequeue(wait_queue);
        notif_state next_state = updated_queue.head == NULL
            ? Idle
            : Waiting;

        set_ntfn_wait_queue(ntfnPtr, updated_queue);
        set_ntfn_state(ntfnPtr, next_state);
        set_ntfn_badge(ntfnPtr, 0);
        unblock_thread(thread);
    }
}
```

Listing 8: Illustration of the notification delivery routine.

Chapter 7

Notification Evaluation & Discussion

We will now turn our attention towards evaluating the presented design and implementation for cross-domain notifications. The design will be evaluated by conducting various channel benchmarks and rudimentary performance benchmarks. As a note on notation, $A \to B$ denotes a cross-domain notification from a sender domain A to a receiver domain B. Additionally, all benchmarks will consist of a Trojan and a Spy. The benchmarks may differ in whether the Trojan or the Spy is on the sending or receiving end of the notification. In such situations, the threads will be distinguished using parentheses. The suffix (A) will indicate that a thread is the sender of the notification and (B) will indicate that a thread is the receiver of the notification.

7.1 Timing Channel Benchmarks

Recall the information flow constraints that cross-domain notifications must satisfy:

R1. Information can only flow from $A \to B$ by A explicitly signalling on the notification. Additionally, the impact that the execution of threads in A may have on the microarchitectural state associated with the notification should not enable information flow from $A \to B$.

R2. Only information *related* to signal delivery can flow from $A \to B$, for example, through changes induced in the microarchitectural state associated with the unblocked thread's TCB when delivering a signal to a thread in B. All other information flow from $A \to B$ is prohibited.

R3. No flow of information from $B \to A$ is permitted.

R4. Information about the communication between A and B via the notification should not leak to any external domain C.

We will now conduct a series of channel benchmarks to validate that these constraints are satisfied. Where appropriate, each result for cross-domain notifications is accompanied by a corresponding result obtained using a normal notification. This allows us to demonstrate that the benchmarks are sensitive to the feature under evaluation. The benchmarking environment is identical to that outlined in Chapter 4 and all channels are quantified using `LeakiEst`.

7.1.1 Unrelated information flow

The information flow requirements R1 and R2 mandate that no data can flow from $A \to B$ except through explicit signals. While this should be mitigated by the fact that the notification is allocated from the receiver's colour, there is much value in benchmarking and confirming this fact experimentally. In effect, we must benchmark that the impact A's activity may have on the LLC cannot leak through the microarchitecture associated with the notification. To evaluate this, we use a benchmark consisting of a priming buffer in A and a notification $A \sim B$ with A acting as the Trojan and B as the Spy. The Trojan encodes a single-bit secret by either touching the entire priming buffer to send a secret value of 1, or doing nothing to send a secret value of 0. The Spy infers said secret by timing how long a `poll`/`wait` (`recv`) on the notification takes. We expect the current design to mitigate this channel as the notification is allocated from the receiver's colour (domain B), ensuring that none of the cache lines for the priming buffer overlap with those of the notification.

Performing this benchmark with the `poll` operation is relatively straightforward, as B can simply time how long a `poll` takes. The `recv` variant of this benchmark is slightly more tricky, as the thread that performed the `recv` will be immediately blocked if the notification is in the `idle` state. As such, the latency of the `recv` operation is not directly observable by the thread performing the `recv` but instead by the thread that is scheduled after the `recv`, as `recv`'s latency will influence the time at which it is scheduled. Thus, to carry out this benchmark the Spy B must consist of two threads, one performing the `recv` and the other observing how long the operation takes.

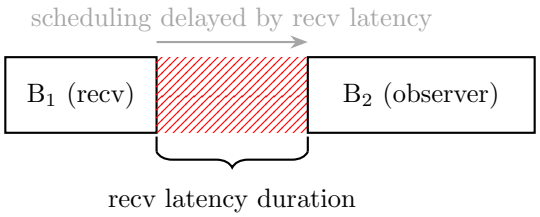


Figure 7.1: Effect of `recv` latency on scheduling of B_2.

The observer thread in B can observe the `recv` latency using a small modification of the system tick detection logic described by Ge [2019]. The method Ge [2019] presents consists of a tight loop where the Spy repeatedly polls the current cycle count, placing it in

a variable. If there is a large jump in this cycle count, then a system tick has occurred. We can modify this method to sample the difference in the time variable before and after the jump, this difference will incorporate the latency of `recv`. The code snippet for achieving this is provided in Listing 9. It is worth pointing out that this measurement will also capture the domain switch latency, but this should be appropriately padded to a WCET using `fence.t`'s padding capabilities.

```
uint64_t volatile current_time;
uint64_t volatile last_recorded_time;
SEL4BENCH_READ_CCNT(last_recorded_time);
for (;;) {
    SEL4BENCH_READ_CCNT(current_time);
    if (current_time - last_recorded_time > TS_THRESHOLD) {
        break;
    }

    last_recorded_time = current_time;
}

uint64_t elapsed_time = current_time - last_recorded_time;
```

Listing 9: Spy can detect differences between times-slices.

We will conduct this experiment with a kernel timer tick interval of 70ms and a domain switch WCET bound of 150,000 cycles. The results for the `recv` variant of this channel are presented in Figure 7.2. Due to the lack of a logically equivalent benchmark that can be carried out on notifications, no equivalent benchmark is provided.

The results highlight a small timing channel. This timing channel arises due to an existing timing channel related to scheduler data and the domain switch latency. More information regarding this channel is provided in Appendix B, specifically in Section B.3. Unfortunately, this benchmark while, not directly measuring the domain switch latency, will be influenced by timing channels through it, so we must apply the temporary work-around for this channel of pre-fetching relevant scheduler data. When we do so, we arrive at the channel matrix depicted in Figure 7.3, highlighting a clear lack of a timing channel. Note that further work is required to eliminate the base timing channel outlined in Appendix B, after which this benchmark should be rerun with those fixes applied. The workaround of pre-fetching is relatively fragile and may break, so it is not a real solution to the base timing channel.

When conducting the `poll` variant of this channel, we attain the channel matrix in Figure 7.4 indicating a clear lack of a timing channel. All in all, based on the collected results, there is strong evidence to suggest that the design prevents unrelated information flow from $A \rightarrow B$. Note that this benchmark only evaluated how information may flow

from $A \to B$ via the impact they have on the notification within the LLC. Benchmarking this is sufficient, as all other microarchitectural state is reset by `fence.t` when switching from domain A to domain B.

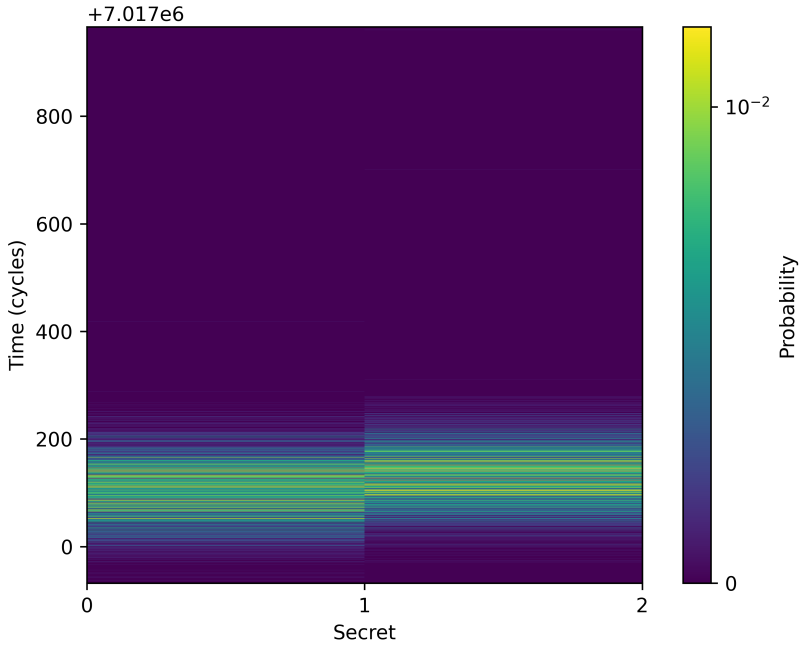


Figure 7.2: Unrelated flow via recv with scheduler channel. $\mathscr{M} = 0.031$. $\mathscr{M}_0 = 0.0022$.

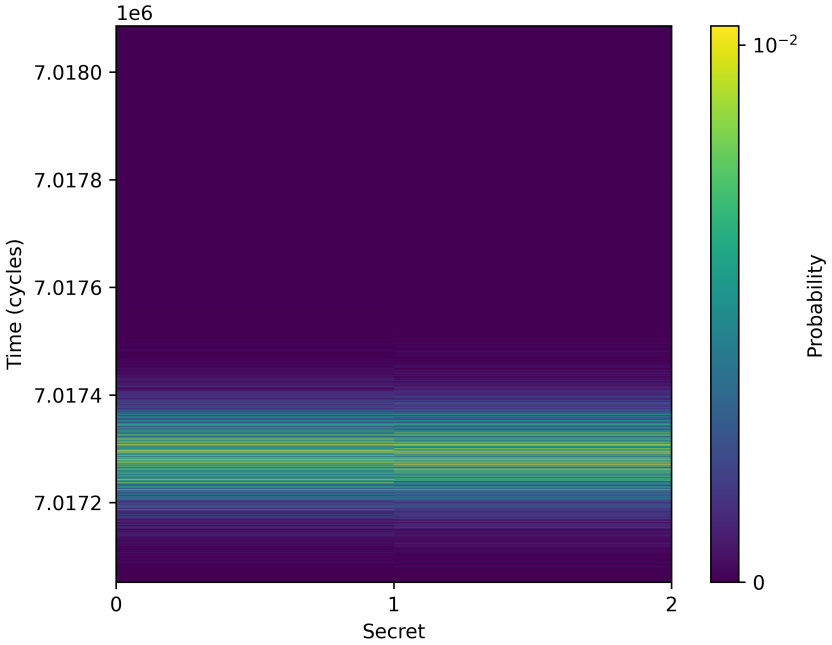


Figure 7.3: Unrelated flow via recv. $\mathscr{M} = 0.0002$. $\mathscr{M}_0 = 0.0006$.

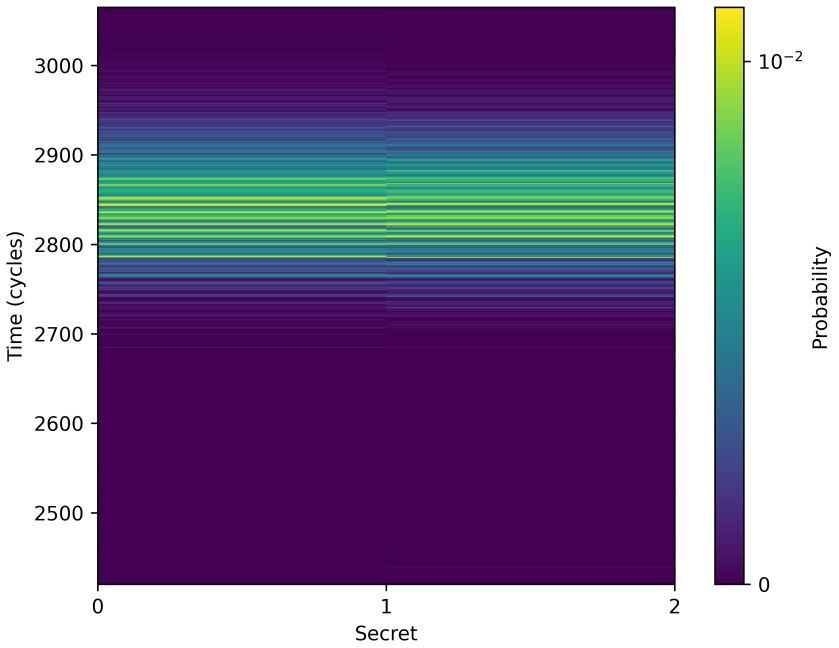


Figure 7.4: Unrelated flow via poll. $\mathscr{M} = 0.0003$. $\mathscr{M}_0 = 0.0008$.

7.1.2 Backflow channel

The information flow requirement R3 prohibits any data flow from $B \to A$, so we must ensure that B's activity within its address space cannot leak through the notification. Since the notification is allocated from B's colour, A can only observe information regarding B's activities by interacting with the `signal` operation. Specifically, by observing how the latency of `signal` varies depending on the notification's presence within the LLC.

To evaluate that this constraint is maintained, we use a benchmark with a priming buffer in B and a cross-domain notification $A \sim B$. In this benchmark A is the Spy while B acts as the Trojan. The Trojan encodes a binary secret by either touching the entire priming buffer to send a secret of 1, or doing nothing to send a secret of 0. The Spy will infer the value of said secret by timing how long a `signal` on the notification takes. The act of touching the priming buffer serves to evict the cache lines for the notification from the LLC. We expect the current design to mitigate this channel, as `signal` is padded to its worst-case execution time.

When running the benchmark with a regular notification, we attain the results in Figure 7.5. When re-running the benchmark with cross-domain notifications, we attain the results in Figure 7.6. Note that in this benchmark, both the traditional and cross-domain notification are in the `idle` state, and no thread in B is presently blocked on the notification. We can see that the presented design for cross-domain notifications maintains no timing channel via the notification's presence within the LLC and, as such, effectively mitigates this form of leakage from $B \to A$.

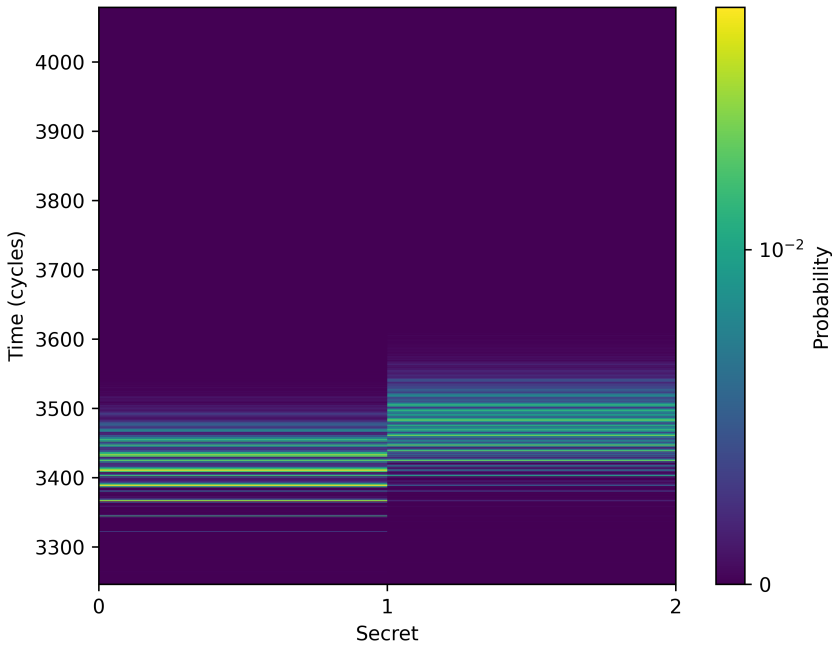


Figure 7.5: Traditional notification backflow channel. $\mathscr{M} = 0.2317$. $\mathscr{M}_0 = 0.0002$.

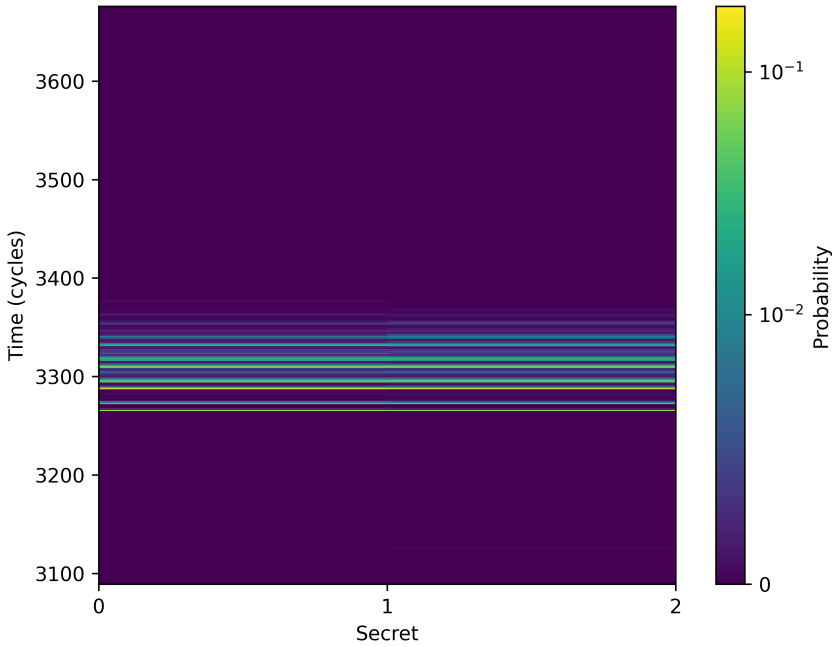


Figure 7.6: Cross-domain notification backflow channel. $\mathscr{M} = 0.0009$. $\mathscr{M}_0 = 0.0011$.

7.1.3 Readiness channels

Another potential source of leakage from $B \to A$ arises from B's readiness to receive a signal, as is the case with traditional notifications. In the case of traditional notifications, if a thread in B is blocked on the notification, then the `signal` operation must perform additional work to unblock the blocked thread. This difference results in variations in signal latency and in the set of L1-I and L1-D cache lines accessed during the operation, both of which can serve as timing channels. As such, we must construct channel benchmarks to demonstrate that these channels are not possible with cross-domain notifications.

The first channel consists of a Trojan (B) and Spy (A) with a cross-domain notification $A \sim B$ between them. To send a binary secret, the Trojan will perform a `wait` on the notification to send a secret value of 1, or do nothing to send a secret value of 0. The Spy will receive said secret by timing how long a `signal` takes. We expect the latency of `signal` to be independent of whether the Trojan is blocked on the notification or not, as delivery is deferred to the domain switch. When performing this benchmark with regular notifications, we attain the results in Figure 7.7. However, when we repeat the experiment with cross-domain notifications, we attain the results in Figure 7.8. It is clear from these results that the cross-domain notification maintains no timing channel via the latency of the signal operation and the impact B's readiness may have on it.

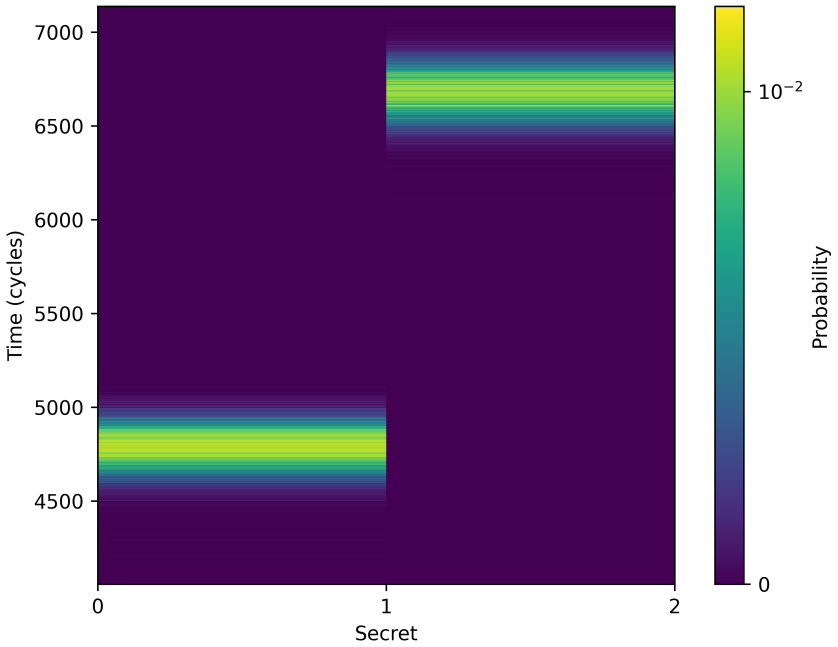


Figure 7.7: Signal latency for a traditional notification. $\mathscr{M} = 1.0$. $\mathscr{M}_0 = 0.0002$.

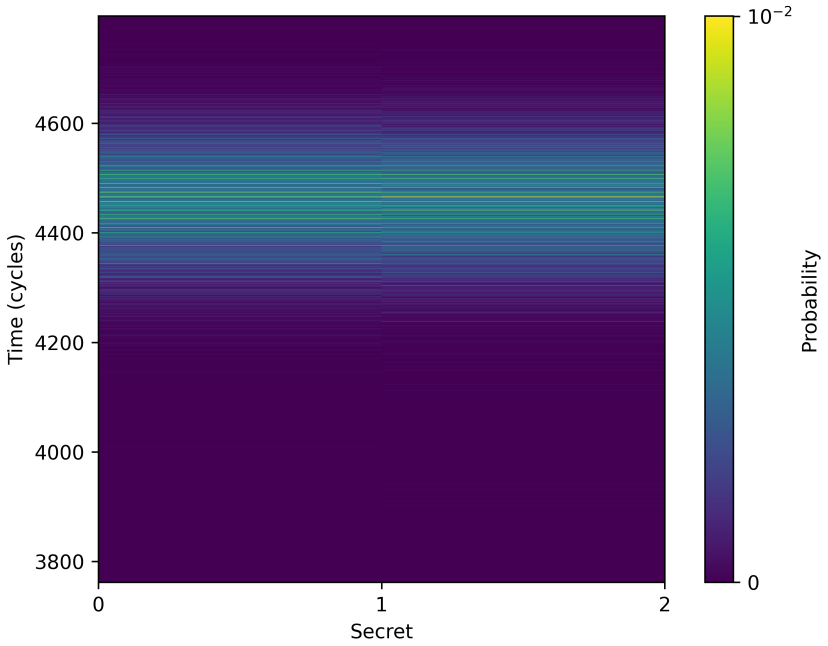


Figure 7.8: Signal latency for a cross-domain notification. $\mathscr{M} = 0.0001$. $\mathscr{M}_0 = 0.0002$.

We will benchmark the L1-I and L1-D channels using a similar set of techniques to what

Ge et al. [2019] employed to benchmark timing channels through a shared kernel image. The benchmark consists of a Trojan (B) and a Spy (A), connected by a cross-domain notification $A \sim B$. It proceeds in three phases: in the first two, the Spy and Trojan cooperate to help the Spy identify which cache lines are accessed by the kernel during a `signal` operation for differing values of the notification state. The final phase then performs the actual benchmark.

During the first two phases, the Spy attempts to infer what L1-I/D cache lines are accessed during a `signal` by using the Prime + Probe technique. The Spy will prime some L1-I/D priming buffer, perform a signal, and then compare probing costs to infer what cache lines were accessed. This process is repeated a few times until the Spy eventually constructs a complete picture of which cache lines are accessed by the kernel. In the first phase, the Trojan will repeatedly block on the notification to force it into the `waiting` state. This means that the Spy can be assured that all cache lines accessed during the first phase are used when performing a signal on a `waiting` notification. In the second phase, the Trojan will do nothing and will simply clear the notification word by performing a `poll`. This means that the Spy can be assured that all the cache lines accessed during this second phase are used when performing a `signal` on an `idle` notification. Upon the conclusion of these phases, the Spy has two sets of cache lines, one set represents the set used when performing a signal on an `idle` notification, and the other represents the set used when performing a signal on a `waiting` notification.

During the actual benchmarking phase, i.ėthe final phase, the Trojan will attempt to leak a binary secret to the Spy by waiting on the notification, with a secret value of 0 indicating no wait and a secret value of 1 indicating a wait. The Spy will then attempt to infer this secret by priming the cache lines derived earlier, performing a `signal` and then timing how long it takes to probe those same cache lines again. For ease of analysis, we will conduct this benchmark in two separate runs. During the first run, the Spy will only Prime+Probe the cache lines associated with a notification in the `waiting`, doing this will reveal if the Spy can infer the secret via the cache lines accessed on a `waiting` notification. The second run is identical to the first, except we use the `idle` cache lines instead. If both these runs reveal no channel, then we can conclude that the design for cross-domain notifications contains no channel through the L1-I/D cache.

The results for the L1-I benchmark when conducted with a regular notification are presented in Figure 7.9 and Figure 7.10 with Figure 7.9 illustrating the results from conducting the benchmark with the idle sets and Figure 7.10 being the results from conducting the benchmarks with the waiting sets. When repeating this experiment using a cross-domain notification, we attain the result illustrated in Figure 7.11 for the idle sets and Figure 7.12 for the waiting sets. These results demonstrate that unlike the traditional notifications, cross-domain notifications do not maintain a channel through the impact the `signal` operation has on the L1-I cache.

In a similar vein to the L1-I results, the results for the L1-D benchmark when performed with a regular notification are provided in Figure 7.13 when using the idle sets and Figure 7.14 when using the waiting sets. When repeating this experiment with a cross-domain notification, we attain the result illustrated in Figure 7.15 when using the idle sets and

Figure 7.16 when using the waiting sets. It is clear from the presented results that unlike traditional notifications, cross-domain notifications do not maintain a channel through the impact the `signal` operation has on the L1-D cache. From all these results, it is clear that cross-domain notifications do not maintain a timing channel via B's readiness to receive a signal.

7.1.4 Summary

The timing channel benchmarks that we have conducted demonstrate that cross-domain notifications maintain the information flow constraints that we sought to satisfy initially. The design does not allow information that is not related to a signal to flow from $A \to B$. Additionally, it disallows flow from $B \to A$, whether that be through the impact B might have on the notification's presence within the LLC, or B's readiness to receive a signal.

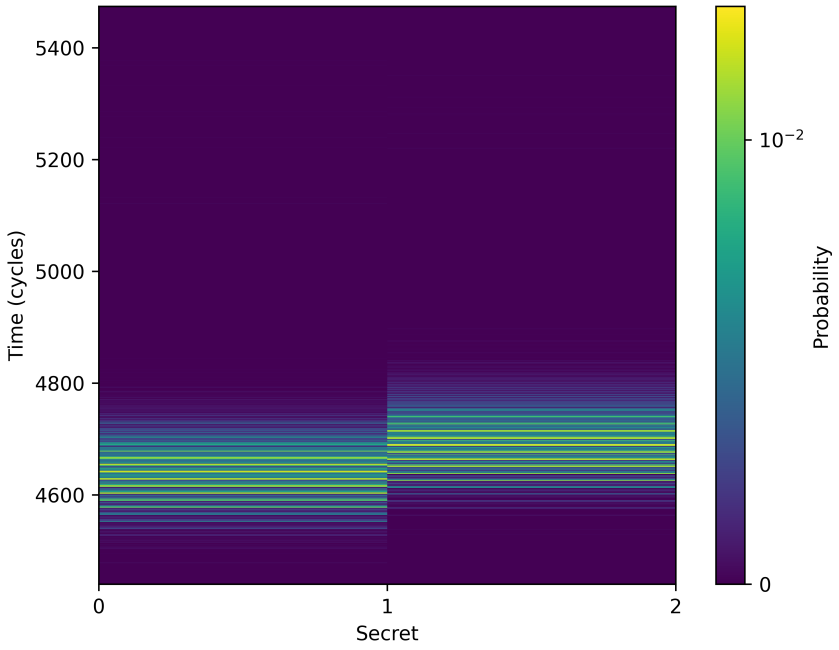


Figure 7.9: L1-I benchmark for a traditional notification using idle sets. $\mathscr{M} = 0.1442$. $\mathscr{M}_0 = 0.0011$.

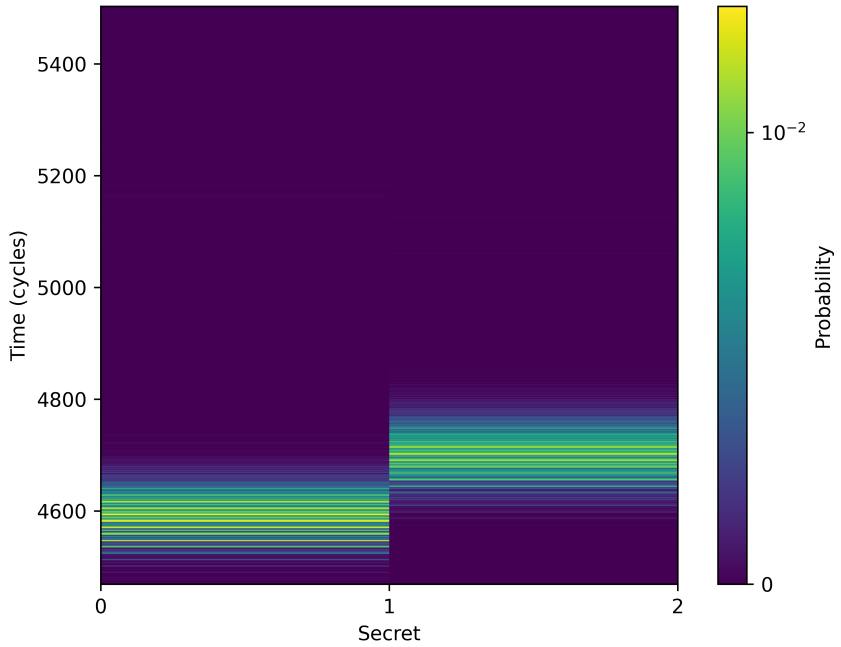


Figure 7.10: L1-I benchmark for a traditional notification using waiting sets. $\mathscr{M} = 0.5849$. $\mathscr{M}_0 = 0.001$.

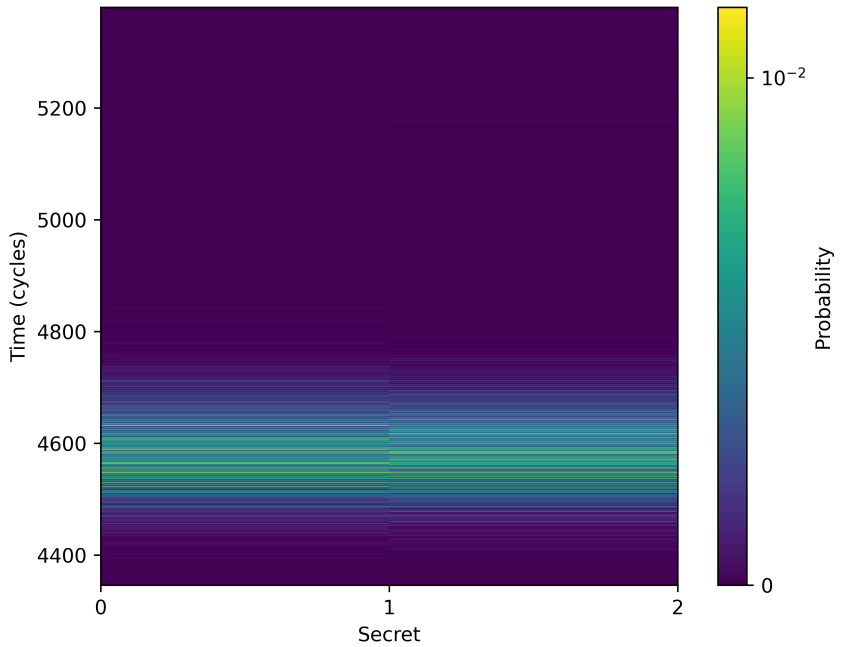


Figure 7.11: L1-I benchmark for a cross-domain notification using idle sets. $\mathscr{M} = 0.0003$. $\mathscr{M}_0 = 0.0011$.

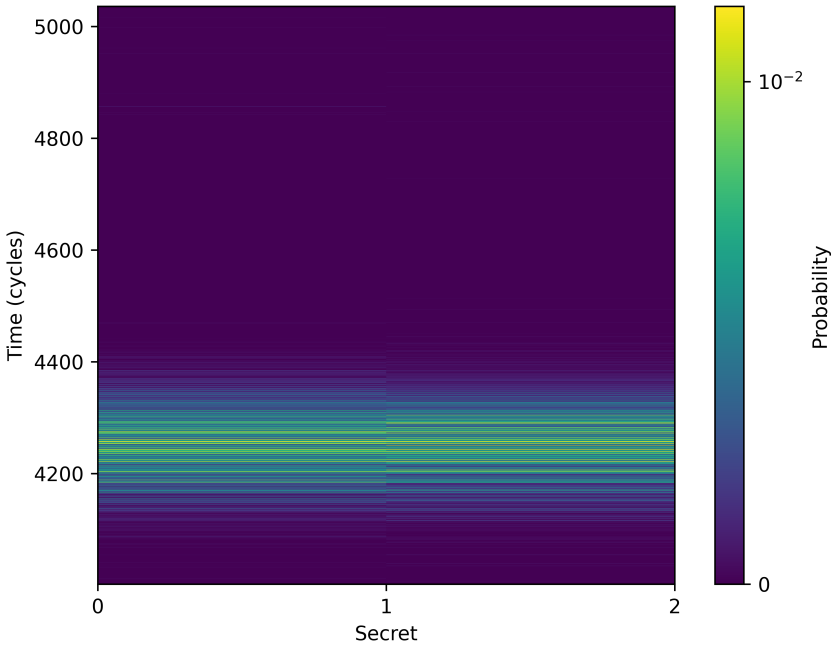


Figure 7.12: L1-I benchmark for a cross-notification using waiting sets. $\mathscr{M} = 0.0006$. $\mathscr{M}_0 = 0.0011$.

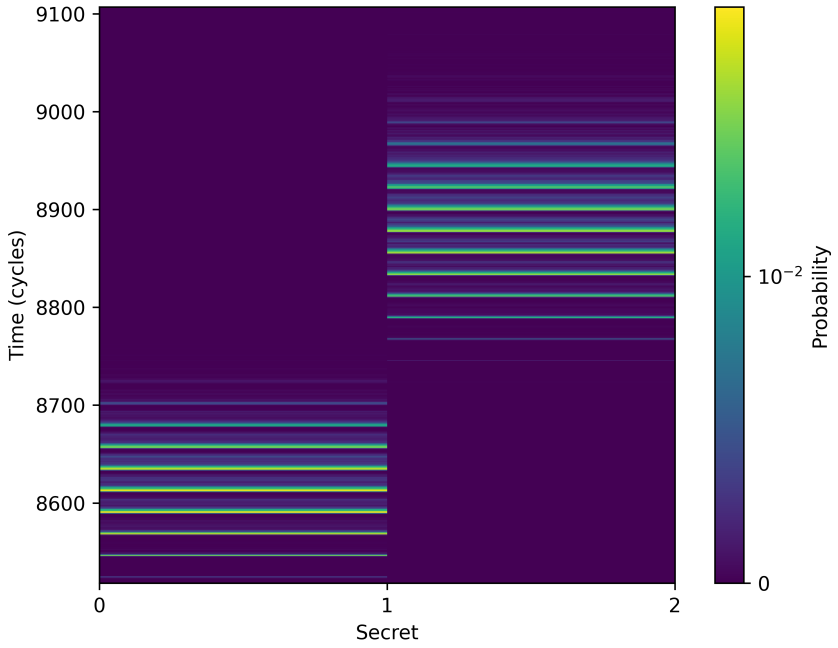


Figure 7.13: L1-D benchmark for a traditional notification using idle sets. $\mathscr{M} = 0.991$. $\mathscr{M}_0 = 0.0002$.

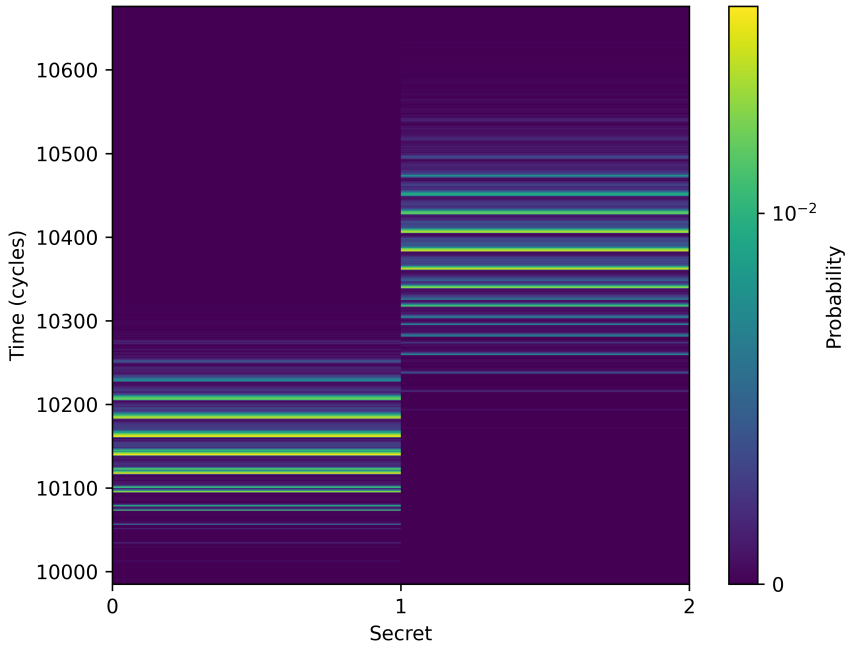


Figure 7.14: L1-D benchmark for a traditional notification using waiting sets. $\mathscr{M} = 0.8677$. $\mathscr{M}_0 = 0.0002$.

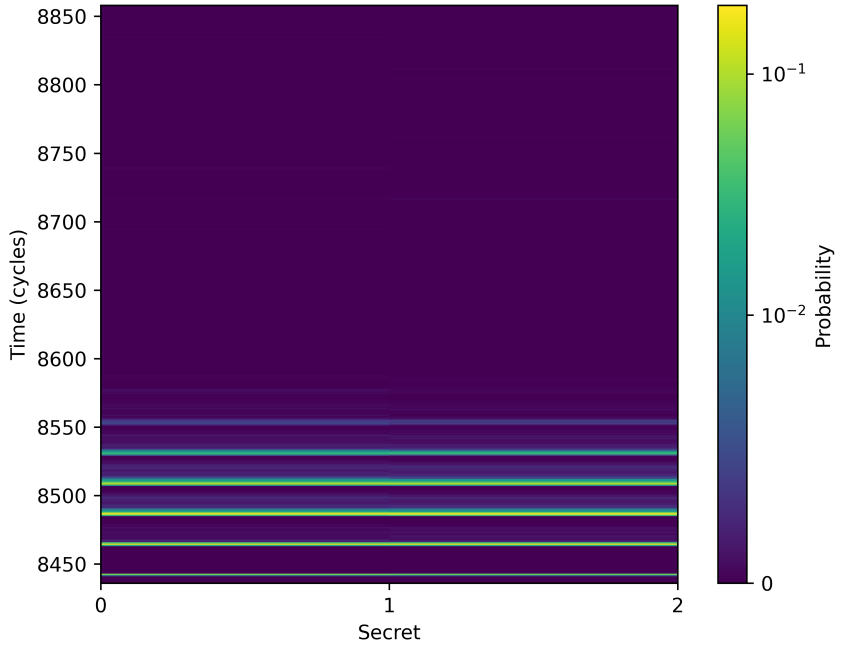


Figure 7.15: L1-D benchmark for a cross-domain notification using idle sets. $\mathscr{M} = 0.0003$. $\mathscr{M}_0 = 0.0007$.

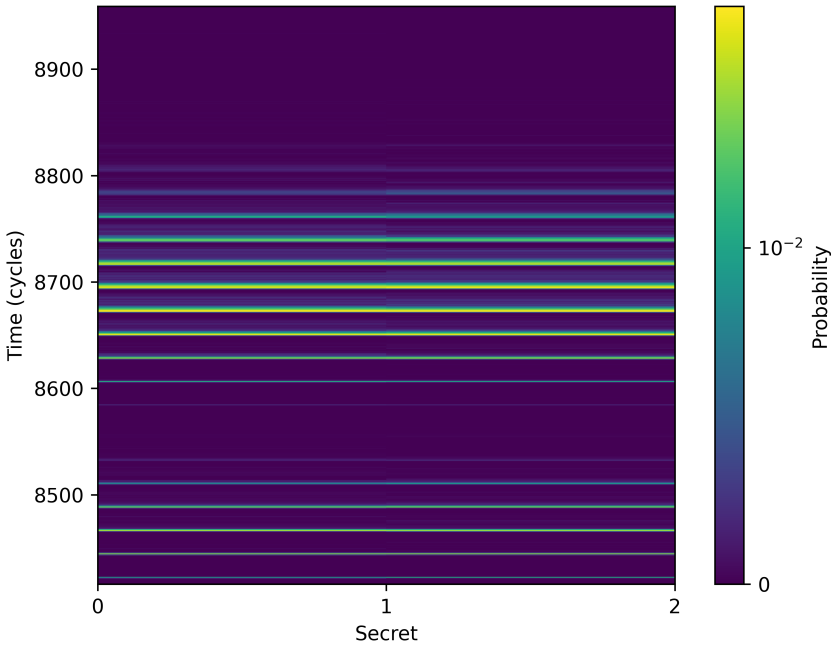


Figure 7.16: L1-D benchmark for a cross-domain notification using waiting sets. $\mathscr{M} = 0.0$. $\mathscr{M}_0 = 0.0003$.

7.2 Performance

All the benchmarks presented in this section are micro-benchmarks, benchmarking only key details associated with cross-domain notifications. These key details provide some insight on the performance implications of using cross-domain notifications, but further work must be done to conduct more holistic benchmarks that use cross-domain notifications under practical and realistic use cases. All results presented in this section are presented as warm/cold pairs. The cold results are the latency of the operation of interest when none of the cache lines for the notification or kernel are present in the cache hierarchy. The warm results correspond to the latency of the operation of interest when the cache lines associated with the notification and kernel are present in the cache hierarchy. We evict the cache lines associated with the notification and kernel by using Cheshire's LLC flush operation, which performs a complete flush of the last level cache. The L1 cache is cleared by using the `fence` instruction (note that this is not `fence.t`).

7.2.1 Signal latency

We first study the latency of signals on cross-domain notifications and compare them to the baseline of signals performed on regular notifications in the `idle` state. All measurements are attained directly from user-level by measuring the latency of the `signal` operation.

Benchmark	Mean (Cycles)	Standard Deviation
Regular Notifications (Cold)	3591	10 (0.28%)
Regular Notifications (Warm)	1211	6 (0.50%)
Cross-Domain Notifications (Cold)	3147	7 (0.22%)
Cross-Domain Notifications (Warm)	1347	11 (0.82%)

In the cold-case, cross-domain notifications maintain comparable performance to traditional notifications. In the warm case, however, there is a slight degradation in performance, this is attributed to the need to pad the signal handling function for cross-domain notifications to a WCET.

7.2.2 Poll latency

We will now study the latency of the poll operation and present two sets of results. All measurements are made directly from user-level by measuring the latency of the `poll` operation.

Benchmark	Mean (Cycles)	Standard Deviation
Regular Notifications (Cold)	2961	8 (0.27%)
Regular Notifications (Warm)	1020	2 (0.21%)
Cross-Domain Notifications (Cold)	2913	9 (0.31%)
Cross-Domain Notifications (Warm)	988	2 (0.20%)

Evidently, cross-domain notifications does not maintain a degradation in the latency of the poll operation, with both the cold and warm results being similar to that of regular notifications.

7.2.3 Domain switch overhead

The final metric we are concerned with is the domain switch overhead. Cross-domain notifications shift the cost of delivery out of the signal operation and into the domain switch, and charges the receiver of the notification for this latency. As such, to establish baseline results for the domain switch overhead, we require two sets of measurements. First, the latency of the domain switch without cross-domain notifications, and second, the latency of the signal operation with traditional notifications in the `waiting` state, i.e. when they are delivering a notification. We observe the domain switch latency by probing the cycle count immediately after `fence.t` within the kernel, and then again once we have entered the first thread. By taking the difference between these two values, we in-effect measure how long it takes to schedule the first thread immediately after the `fence.t`. Since cross-domain notifications are delivered after `fence.t` this will measurement will also incorporate the latency of the delivery operation. We do not include the operations

before `fence.t` because they are padded to a WCET and will merely present as a constant overhead on all our measurements.

The first result we look at is the latency of the domain switch operation without cross-domain notifications. This will serve as a baseline. We only study the cold benchmark result here because the warm criteria will be difficult to replicate in the domain switch path. This difficulty stems from the challenge in guaranteeing that all cache lines associated with the notification and next domain's kernel image are present within the cache hierarchy.

Benchmark	**Mean (Cycles)**	**Standard Deviation**
Domain Switch (Cold)	4806	17 (0.35%)

The next measurement we look at is the latency of a `signal` with a traditional notification in the `waiting` state. The results of this benchmark suggest that the latency of the actual delivery part of this operation, that is, the part of the operation associated with unblocking the blocked thread and writing to its badge register, is approximately 2,219 cycles. This latency was derived by subtracting the (cold) signal latency on an idle traditional notification from the (cold) signal latency of a traditional notification in the waiting state. In an efficient implementation of cross-domain notifications, the overhead the design introduces to the domain switch should be inline with this result.

Benchmark	**Mean (Cycles)**	**Standard Deviation**
Signal on Waiting Notification (Cold)	5810	15 (0.26%)

We now look at the domain switch latency overhead imposed by cross-domain notifications in more detail. We are interested in three distinct cases:

1. When there is *no notification* to deliver at the domain switch;
2. When there is a notification to deliver, and the target thread is the *highest priority thread* in the domain; and
3. When there is a delivery to be made, but *not to the highest priority thread.*

Looking at these three cases will provide a relatively complete picture of the domain switch overhead imposed by this design.

Benchmark	**Mean (Cycles)**	**Standard Deviation**
No Delivery (Cold)	5016	18 (0.36%)
Delivery Max Priority (Cold)	7392	16 (0.22%)
Delivery Not Max Priority (Cold)	7619	20 (0.26%)

We are particularly interested in the "not-max priority" case as in the "max-priority" case, notification delivery serves to bring the cache lines associated with the next thread's TCB into the cache hierarchy, resulting in a lower overhead. In the not-max priority case, notification delivery imposes an overhead of 2,813 cycles on the domain switch, representing a 27% overhead compared to the cost of unblocking a thread with traditional notifications (2,219 cycles), which is quite substantial. The overhead was calculated by subtracting the cold domain switch latency from the not-max priority delivery latency.

Much of the overhead introduced by our design is attributable to the overhead imposed by reading from the notification object and checking if there is a notification to deliver (i.e. the no delivery case). The remainder of the overhead is likely attributable to the implicit cost associated with performing this operation after a `fence.t`, which would have completely flushed all microarchitectural resources such as the TLB.

7.3 Discussion

The results of this chapter demonstrate that our presented design for cross-domain notifications satisfies the information flow constraints that we had sought to satisfy initially. Our design achieves this with minimal latency overhead except on the domain switch path, where it imposes a 27% overhead above what would be naively expected when one moves signal delivery to the domain switch path.

7.4 Further Work

There is still some further work to be done. The channel highlighted in Appendix B, Section B.3 needs to be properly mitigated and the benchmarks for the unrelated flow channel need to be re-conducted after doing so. The results we have attained with the temporary work-around suggest no timing channel, but our work-around is fragile and not a principled solution to the initial channel.

Additionally, we need to conduct macro-benchmarks where cross-domain notifications are used under a more realistic workload, this allows for a deeper understanding of their performance implications.

Furthermore, the implementation we studied placed the queue of notifications associated with a domain in scratchpad memory. In principle, we could easily move this out of SPM and place it within the data segment associated with the receiver domain's kernel image. This is rather simple to do, but unfortunately was not done during this thesis.

Finally, we presented the N-signals-awaken-N-threads model in Chapter 5, whereby N signals made by threads in A will awaken N threads in B blocked on a notification. We argued against this design and chose our present design due to the simpler implementation story. However, we could have very reasonably chose the N signals awakes N threads

model too. This presents a potential avenue for further work on cross-domain notifications, implementing this alternative model and benchmarking its implications.

Chapter 8

Shared Memory Design

This chapter and its corresponding implementation chapter explores the introduction of cross-domain shared memory. We will begin by first looking at the information flow constraints that our cross-domain shared memory design must satisfy. To motivate the information flow constraints, consider a cross-domain memory buffer shared between two domains, A and B. Let A be the *writer* domain and B the *reader* domain. We require that only information explicitly written by A to the buffer may flow to B, and that no information may flow from $B \rightarrow A$. We can specify this more concretely with the following constraints:

R1. Information can only flow from $A \rightarrow B$ by A writing into the buffer. Additionally, the impact that reads and writes performed by A have on the microarchitectural state of the buffer should not introduce a usable timing channel.

R2. Any influence that threads in A have on the buffer's microarchitectural state must not form a usable timing channel. This means that if the execution of threads within A incidentally evicts cache lines associated with the buffer, then the impact left by this eviction must not allow for information leakage.

R3. Information flow from $B \rightarrow A$ is prohibited. The execution of threads in B must not affect the buffer's microarchitectural state in a way that forms a timing channel. For example, the side effects of B reading from the shared buffer must not become a usable timing channel.

To implement a practical shared-memory model, the kernel should avoid arbitrating buffer access through system calls, as this would introduce unnecessary overhead and a cumbersome API. As a result, the operating system has no mechanism for controlling how each domain interacts with the buffer or how they impact the microarchitectural state associated with the buffer. It then follows that any implementation of cross-domain shared-memory must either: partition the microarchitecture associated with the buffer, or reset it to a defined state. This observation suggests two possible designs for shared memory: either the kernel determinises the buffer's microarchitectural state on a domain switch, or

the buffer is split in two, with each domain maintaining its own copy of the buffer, and the kernel synchronising them after the writer's time slice. We will attempt to address both these possible designs in this thesis.

8.1 Determinisation on Domain Switch

The first shared-memory design involves determinising the buffer's microarchitectural state on a domain switch. This can be achieved in two ways, either by pre-fetching the cache lines for the buffer in software, or by employing hardware support. Presently, Cheshire supports creating arbitrary cache partitions and associating physical memory ranges with them, as well as selectively flushing these partitions from the LLC. However, this conflicts with the present, static, colour-based LLC partitioning used in time protection. As such, using this dynamic partitioning mechanism would require reworking the existing RISC-V implementation of time protection, something which is quite infeasible over the course of a thesis. There is no other mechanism for flushing ranges of memory from the LLC within Cheshire outside DPLLC.

The next possible option for determinisation is pre-fetching. In Chapter 3 we saw that Buckley et al. [2023] argued that pre-fetching cannot fully close timing channels through shared kernel data, as without a formal model of the LLC it is difficult to make any claims regarding the presence of SKD cache lines or cache lines that overlap with SKD within the LLC. It is important to note that this limitation mainly stems from the fact that SKD cache lines can overlap with cache lines accessible to arbitrary domains. As such, there is a possibility that pre-fetching may still be effective if the shared buffer is allocated from a unique colour. This would prevent domains from influencing the buffer's state within the LLC without directly accessing the buffer. In such a scenario, we may find that a Trojan is unable to exert any influence over the probability that a cache line be present in the LLC after determinisation, effectively closing any timing channel, even with a randomised LLC eviction policy.

The remainder of this section will explore pre-fetching and ultimately demonstrate that even allocating a buffer from a unique colour fails to close all timing channels, leaving temporary channels that decay in strength. This leads to the eventual conclusion that determinisation on Cheshire can only be achieved with hardware support, motivating a direction of future work for this thesis.

8.1.1 Cache inclusivity

Before conducting any experiments or analysis, it is important to note that pre-fetching relies on the cache hierarchy being either inclusive or exclusive. Cheshire guarantees inclusivity only on the initial read; i.e. if a cache line is absent from the L1-D, accessing it brings the line into both the L1-D and LLC. Afterwards, inclusivity is not maintained, and a line may reside in the L1-D without being present in the LLC. The implication of this

is that we must flush the L1-D cache before pre-fetching. Fortunately, Cheshire supports this via a simple `fence` invocation, which flushes the entire L1-D before continuing with the next instruction. This instruction is distinct from `fence.t`.

8.1.2 Pre-fetching and a common colour

Buckley et al. [2023] presents the conclusion that pre-fetching is a flawed approach to determinisation, but were unfortunately unable to provide any experimental evidence. As such, we will first attempt to recreate their conclusion in the context of a shared buffer allocated from either A's or B's colour.

We will recreate their conclusion by benchmarking a Prime + Probe attack on a bare-metal Cheshire environment with no operating system or firmware. This setup allows us to observe the behaviour of pre-fetching without interference from OS-induced noise. Our benchmark will consist of a Trojan T, a Spy S, with a shared buffer allocated from the Trojan's colour between them. We will *simulate* a kernel domain switch by pre-fetching all cache lines associated with the buffer and then performing a `fence.t` invocation. During a round of the benchmark, the Trojan will encode some secret n by priming n cache lines. The Spy will observe said secret by timing how long it takes to probe the entire shared buffer. Each benchmark we conduct will consist of 20,000 rounds, after which we will plot the channel matrix from the experiment samples. Pseudocode for the benchmark is provided in Listing 10. We will first conduct this benchmark with both the Trojan and Spy allocated a single colour each, and a two-page-long shared buffer between them. In this benchmark, the Trojan can only encode a secret from 0 to 64 as being allocated a single colour implies that it only has access to 64 unique cache lines. The channel matrix for this experiment is provided in Figure 8.1, demonstrating a clear timing channel.

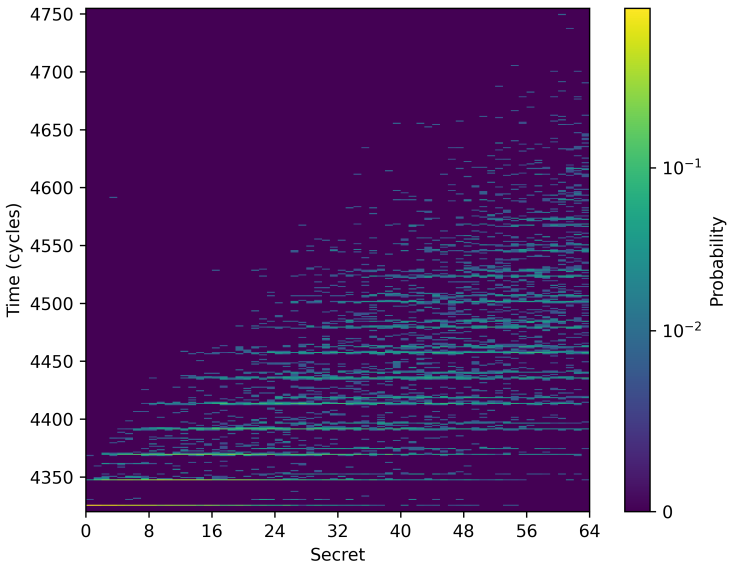


Figure 8.1: Benchmark result for a two-page buffer.

```
void domain_switch(shared_buffer) {
    fence();
    for (int line = 0; line < NUM_LINES(shared_buffer); line++) {
        touch_cache_line(shared_buffer, line);
    }
    fence.t();
}


void benchmark(shared_buffer) {
    domain_switch(shared_buffer);
    priming_buffer = allocate_priming_buffer(trojan_colour);

    for (int round = 0; round < NUM_ROUNDS; round++) {
        // TROJAN
        int secret = random() % 65;
        prime(priming_buffer, secret);

        // DOMAIN SWITCH
        domain_swtich(shared_buffer);

        // SPY
        int latency = probe(shared_buffer);
        record(secret, latency);

        // DOMAIN SWITCH
        domain_swtich(shared_buffer);
    }
}
```

Listing 10: Prime+Probe benchmark pseudocode.

To understand why a timing channel arises, it is useful to first recall that Cheshire's LLC maintains a randomised eviction policy [Ottaviano et al., 2023]. Additionally, recall that on Cheshire, a single cache colour spans 64 cache lines. Thus, the first cache lines from both pages in a two-page-long buffer will map to the same cache set within the LLC. With this in mind, we focus on the first cache line within the buffer, and examine its presence in the LLC during a round of the benchmark.

For ease of notation, let $p_0, p_1, \cdots, p_7$ denote the cache lines from the priming buffer that map to the same cache set as the cache line under consideration. During a prime, the Trojan touches all of these lines in order to saturate the corresponding set in the LLC and evict everything within it. Next, let s_0 and s_1 denote the first cache line from the first and second page in the shared buffer respectively. All of $p_0, p_1, \cdots, p_7, s_0$ and s_1 map to

the same cache set within the LLC.

We will first consider the scenario where the Trojan had not primed $p_0, p_1, p_2, \cdots, p_7$. We can conclude that in this scenario, the cache set associated with these cache lines within the LLC will be empty. As such, after the kernel performs the pre-fetching routine on a domain switch, the set associated with the line of interest will only contain s_0 and s_1. This scenario is depicted in Figure 8.2.

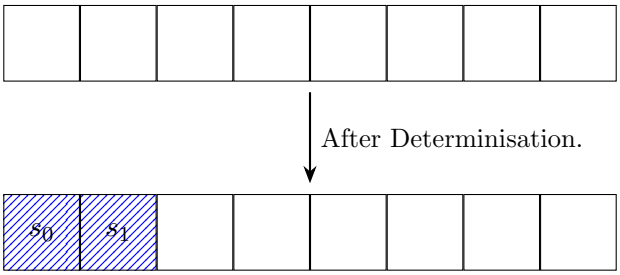


Figure 8.2: Unprimed scenario.

Now consider the second situation, where the Trojan had indeed primed $p_0, p_1, \ldots, p_7$. Before determinisation, the cache set we have been studying is completely saturated with these priming cache lines. When the kernel pre-fetches the buffer, it first touches s_0. With no free space in the set, the LLC must evict a line at random to accommodate the request for s_0. Next, when s_1 is pre-fetched, the same situation occurs: the lack of space forces a random line eviction. Since the cache set is saturated with priming lines, s_0 now has a 1/8 chance of being evicted. As such, after determinisation there is a 1/8 chance of s_0 not being present within the LLC. A diagram of this situation is presented in Figure 8.3. This variation in the probability of s_0's presence in the LLC depending on whether the Trojan primed or not is what gives rise to the timing channel in Figure 8.1.

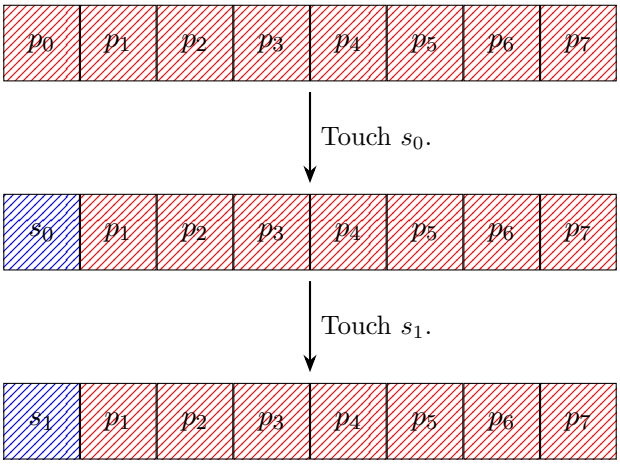


Figure 8.3: Primed scenario.

This situation, in which the Trojan can influence the probability that pre-fetching evicts earlier buffer cache lines during determinisation, is difficult to mitigate because it stems from a fundamental property of the LLC's eviction policy. This experiment confirms the conclusion of Buckley et al. [2023] when applied to Cheshire, specifically validating it within the context of cross-domain shared memory. We will now turn our attention towards the scenario where the buffer is allocated from a dedicated colour and demonstrate why pre-fetching fails in this scenario too.

8.1.3 Pre-fetching and a distinct colour

To motivate why pre-fetching fails with a distinct colour too, we will consider the same Prime + Probe attack that we were previously studying, except this time with a 9-page-long shared buffer. Additionally, we will "partition" this 9-page-long buffer in two. The first page will act as the probing page used by the Spy, and the last 8 pages will act as the priming pages used by the Trojan. The key detail here is that since this buffer is shared between both the Trojan and the Spy, all pages in the buffer, i.e. the probe and prime pages will be pre-fetched on a domain switch. Additionally, as a brief reminder, it is important to remember that this buffer lies in a *distinct* colour from the Trojan and Spy.

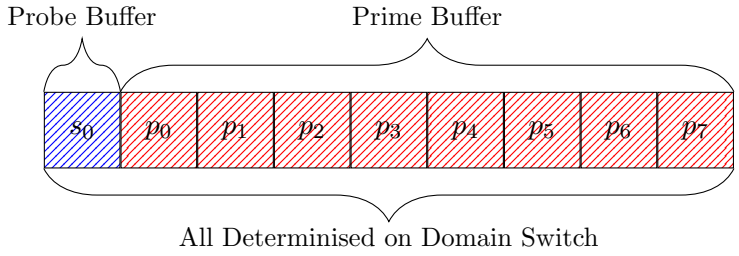


Figure 8.4: Decomposition of the shared buffer.

When we re-run the benchmark outlined in Listing 10 we attain the channel matrix depicted in Figure 8.5.

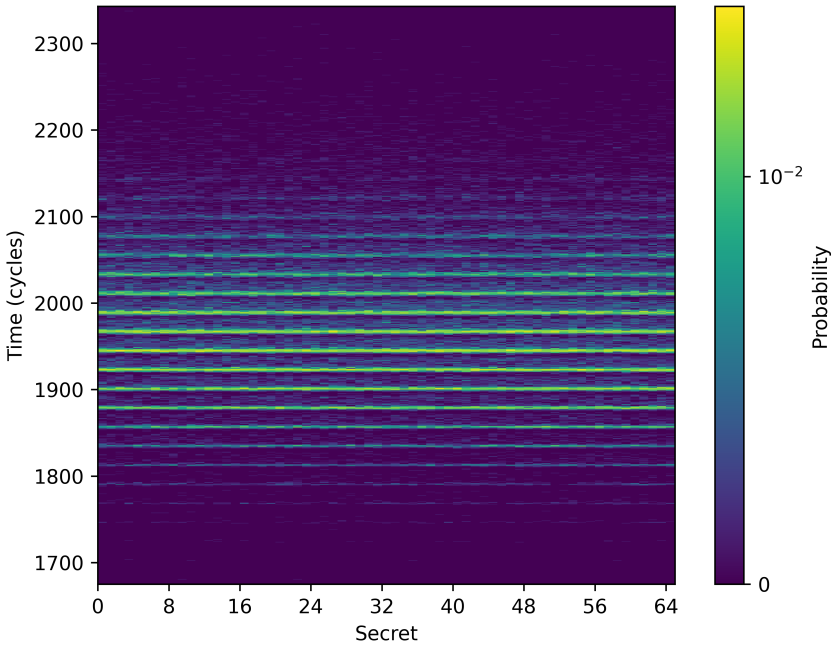


Figure 8.5: Benchmark result for a nine-page buffer.

At first glance, the Prime + Probe attack in this configuration does not appear to produce a timing channel, as there is effectively no correlation between the number of primed cache lines and the observed probing cost. This conclusion, however, may be premature. There is a very real possibility that a timing channel could exist, but pre-fetching and the repeated use of the channel may cause its strength to decay over time. As a result, the channel may only be detectable during the first few initial transitions from the Trojan to Spy; however, when benchmarking several thousand rounds, this initial channel may be drowned out by the subsequent absence of a channel in the remaining rounds.

To benchmark this situation more concretely, we will modify the bare-metal benchmark to flush the entire LLC between benchmarking rounds. This new benchmark will, in effect, measure the strength of any timing channel that may exist the first time the Trojan attempts to use it. Flushing the entire LLC will introduce some noise to benchmarking results, so where possible, we will have to pre-fetch code and data related to the benchmark before conducting the next round of the benchmark. When we conduct this modified benchmark on bare-metal, we attain the channel matrix depicted in Figure 8.6. This result depicts a relatively weak timing channel. It is, however, still a timing channel and a method for sending information from the Trojan to the Spy in a manner that would violate the information flow constraints that we had previously outlined.

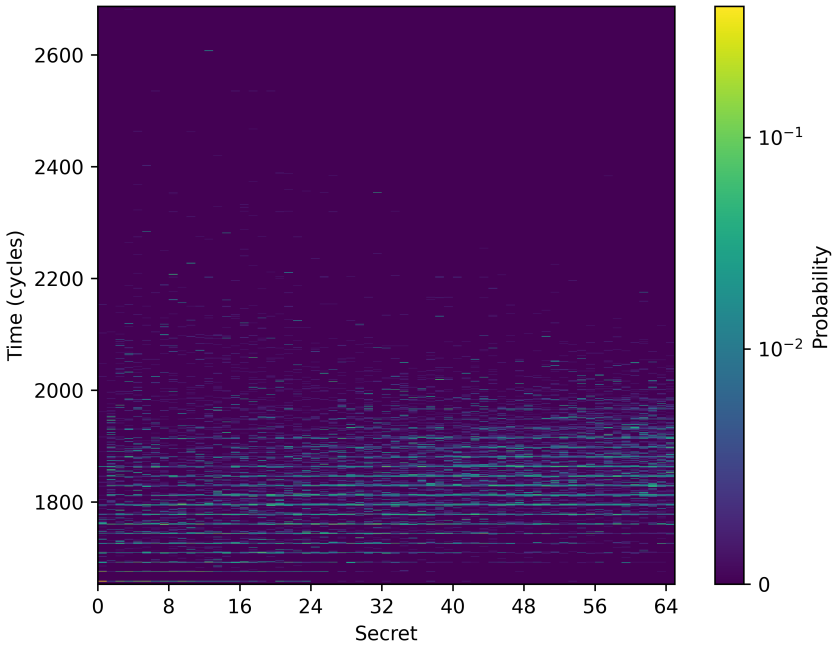


Figure 8.6: First-time use benchmark result for the nine-page buffer.

To understand what is causing this channel, we study a specific cache line s_0 in the probe buffer. This line maps to the same cache set as the lines $p_0, p_1, \ldots, p_7$ in the priming buffer. First, consider the case where the Trojan had not primed $p_0, p_1, \ldots, p_7$. In this situation, the corresponding cache set is empty, and determinisation touches the lines $s_0, p_0, \ldots, p_7$ sequentially. After touching s_0 through p_6, all these lines are expected to reside in the LLC. When p_7 is finally touched, there is a $1/8$ probability of evicting s_0 from the LLC. This $1/8$ probability represents the baseline likelihood that s_0 is not present after determinisation when the Trojan has not primed $p_0, \ldots, p_7$. This scenario is illustrated in Figure 8.7.

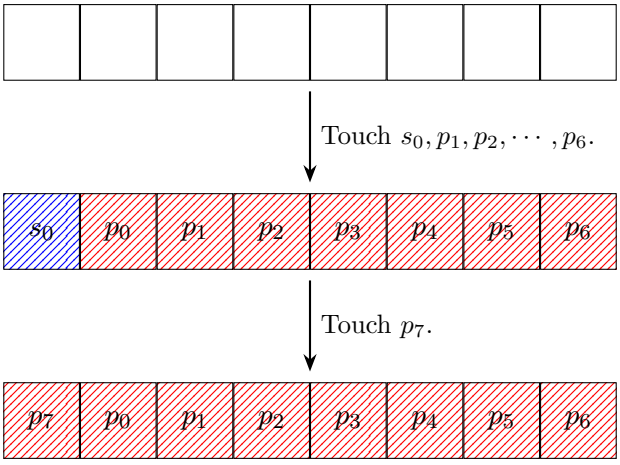


Figure 8.7: Unprimed scenario for 9-page buffer.

Now consider the scenario where the Trojan had primed $p_0, p_1, \ldots, p_7$. In this case, the corresponding set within LLC is fully saturated with these cache lines. When determinisation first touches s_0, it randomly evicts one of these priming lines. Suppose p_1 is evicted. When the routine then touches p_1, it must bring it back into the LLC, and this touch has a $1/8$ chance of evicting s_0. In this example, assume that touching p_1 evicts p_3 instead. When the pre-fetching routine touches p_3, there is again a $1/8$ chance of evicting s_0, and in this case, s_0 is evicted. This possible eviction scenario is depicted in Figure 8.8. The key insight is that pre-fetching in the primed case can create *eviction chains*, where each step carries a small but cumulative probability of evicting s_0. This results in a slightly higher chance that s_0 is not present in the LLC after determinisation compared to the unprimed case. In the example provided in Figure 8.8 the eviction chain is $s_0 \to p_1 \to p_3 \to s_0$. The net result is that this variation in eviction probabilities for s_0 results in the timing channel depicted in Figure 8.6. It is worth noting that the concept of an eviction-chain is something that doesn't appear within the literature and, as far as I can tell, is a new idea.

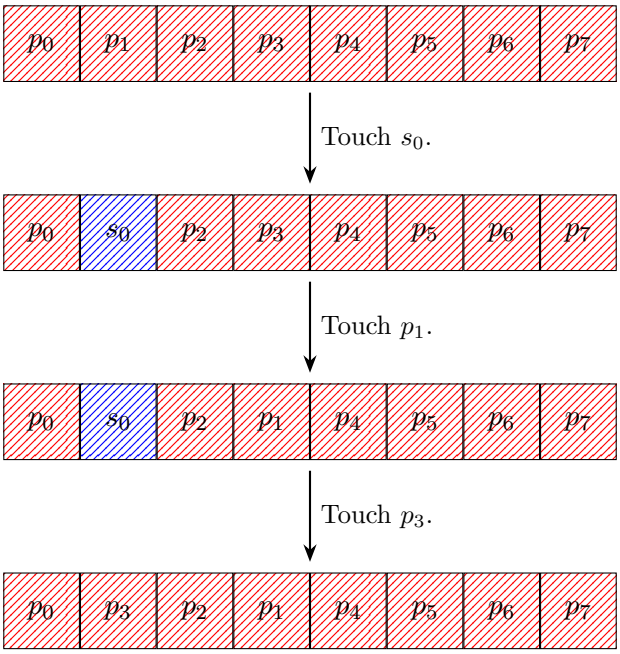


Figure 8.8: Primed scenario for 9-page buffer.

So, we observe that even with a distinct colour, it is possible to construct a timing channel, it is just that the channel can only be used once, and after the first use, pre-fetching works to diminish the strength of the channel. One may then imagine that a potential way to then mitigate this issue is to enforce the constraint that the initial thread setting up the system must perform an initial determinisation of the shared buffer. This would enforce the constraint that at least some part of the shared buffer is present in the LLC before the Trojan and the Spy begin executing. When we make this modification and re-run the benchmark, we end up with the channel matrix depicted in Figure 8.9. While not evident from the channel-matrix directly, `LeakiEst` still reports a minimal timing channel, so it appears as if this strategy does not entirely work.

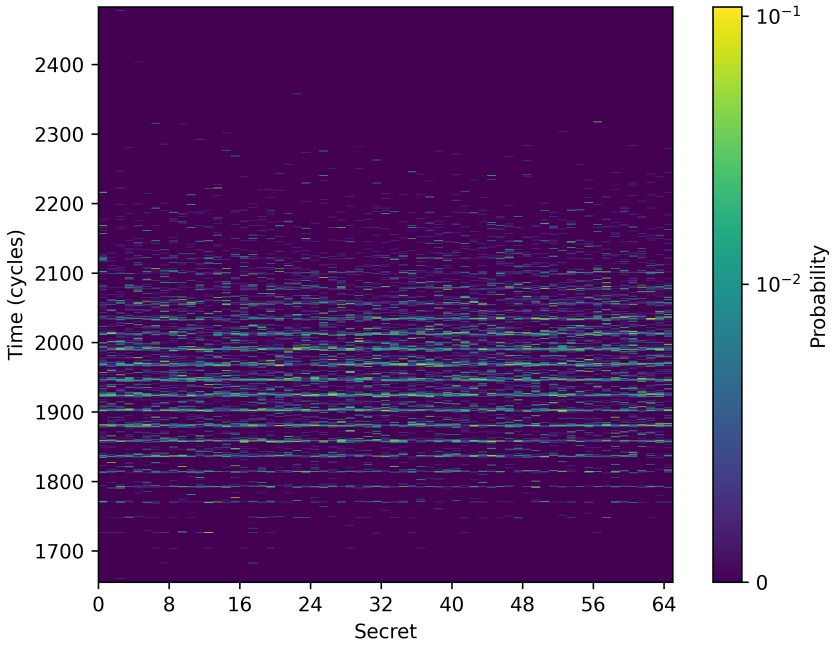
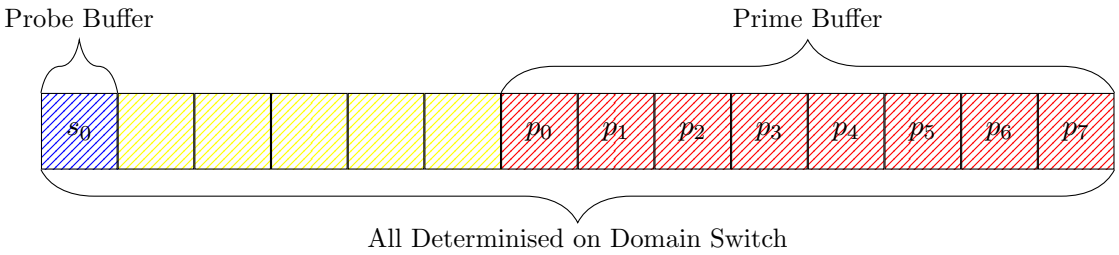


Figure 8.9: First-time use with initial determinisation. $\mathscr{M} = 0.0422$. $\mathscr{M}_0 = 0.008$.

Instead of focusing on the mechanics of the residual timing channel shown in Figure 8.9, we will shift our attention to an alternative buffer configuration designed to produce an even stronger channel. The timing channel illustrated in Figure 8.6 arises from the eviction chains formed during determinisation. It then stands to reason that if we can construct longer or more numerous eviction chains, then we could potentially construct an even stronger channel. One way to encourage this is to introduce pages between the priming and probing buffers that neither the Trojan nor the Spy access. These intermediate pages exist solely to promote the formation of longer eviction chains during pre-fetching. With this in mind, we design a 14-page-long buffer for the next Prime + Probe benchmark. The first page serves as the probe buffer, the final eight pages serve as priming buffers, and the five intermediate pages exist to promote the construction of longer eviction chains.



When we re-run the bare-metal benchmark with this new configuration and initial thread determinisation we obtain the channel matrix depicted in Figure 8.10. Which illustrates a much stronger and significantly more obvious timing channel.

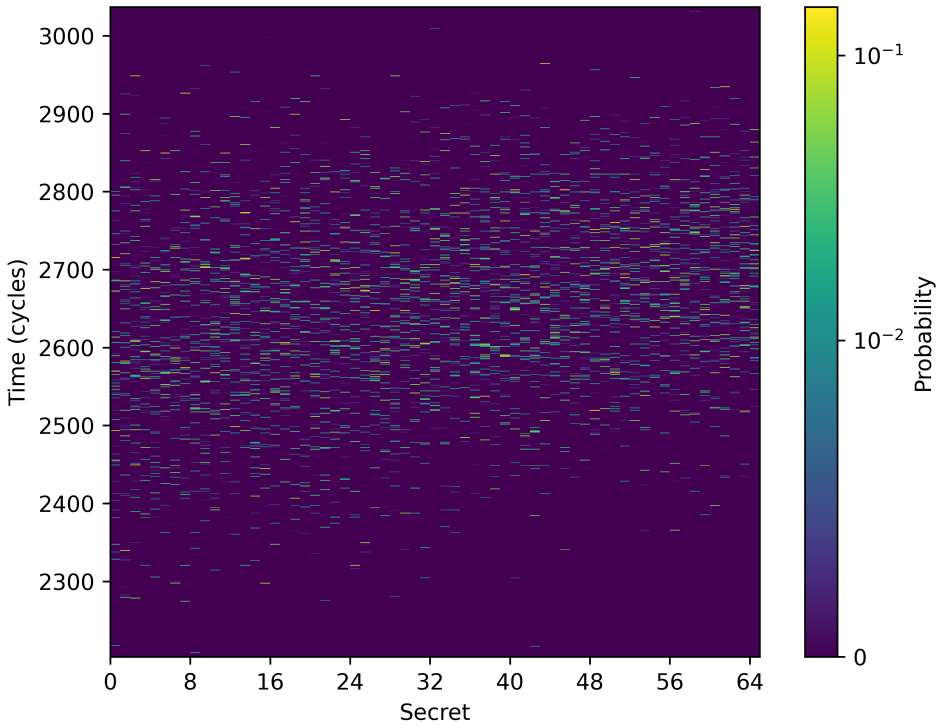


Figure 8.10: First-time use with longer eviction chains. $\mathscr{M} = 0.1179$. $\mathscr{M}_0 = 0.0023$.

Thus, it is clear that even with initial thread determinisation, it is possible to construct a reasonably strong timing channel that allows for the transmission of information in a manner that violates the information flow constraints for shared memory. We can, in fact, use this new buffer to measure how the strength of the timing channel decays with time and use. To measure this, we will construct a simple binary channel where the Trojan will prime all the cache lines in the priming buffer to send a secret of 1, and do nothing to send a secret of 0. The Spy, like before, will simply measure how long it takes to access the entire probing buffer. We will also modify the benchmark so that we take our measurement after the nth round, this will give us a picture of what the strength of the channel looks like after the channel has been used n times. The pseudocode for the modified benchmark is provided in Listing 11.

```
void benchmark(shared_buffer) {
    domain_switch(shared_buffer);

    for (int i = 0; i < NUM_SAMPLES; i++) {
        clear_llc();
        determinise(shared_buffer);

        int secret = 0;
        int latency = 0;

        for (int round = 0; round < NUM_ROUNDS; round++) {
            // TROJAN
            secret = random() % 2;
            if (secret) {
                prime(shared_buffer_priming_part);
            }

            // DOMAIN SWITCH
            domain_swtich(shared_buffer);

            // SPY
            latency = probe(shared_buffer_probe_part);

            // DOMAIN SWITCH
            domain_swtich(shared_buffer);
        }

        record(secret, latency);
    }
}
```

Listing 11: Alternative Prime+Probe benchmark pseudocode.

To measure how the strength of the channel decays, we will vary `NUM_ROUNDS` from 1 to 10 and record how many standard deviations the mutual information of the resulting channel matrix is from the zero-information leakage threshold. The standard deviation used is that of the zero-information leakage distribution computed by `LeakiEst` to determine the zero-leakage threshold. When we conduct this experiment and plot the z-scores, we obtain the plot in Figure 8.11. This result demonstrates how the channel strength decays with time and use, initially starting off strong and slowly decaying towards zero.

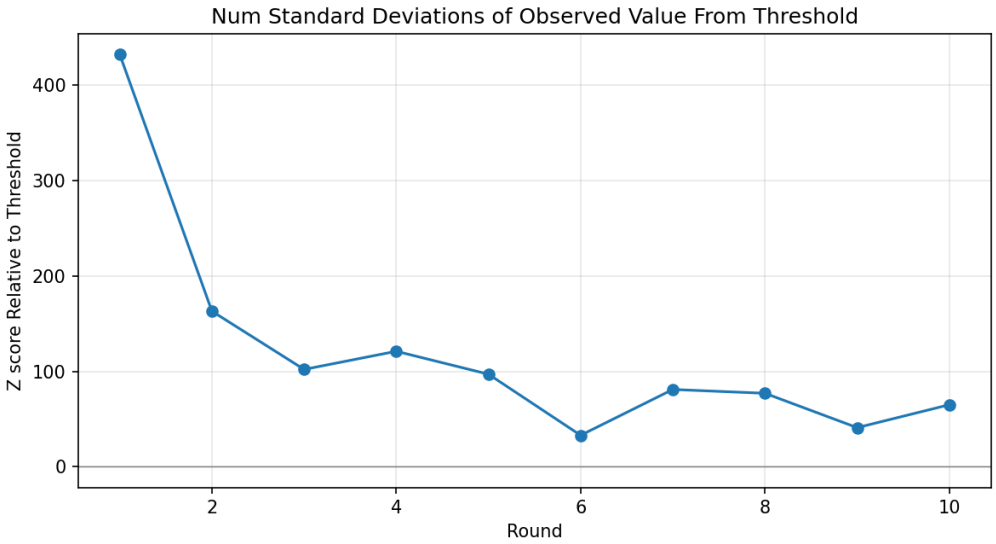


Figure 8.11: Illustration of how channel strength decays with time.

Thus, in the case of the 14-page buffer, we observe that there exists an obvious timing channel, and the strength of said timing channel decays with use and time. This timing channel, decaying or not, is a clear violation of the information flow constraints that we set out for shared memory. It allows for information to flow between the domains via the micro-architectural state associated with the buffer.

It is worth noting that a potential criticism of our benchmarking methodology is the assumption that the cache sets in the LLC to which the shared buffer's cache lines map are initially empty. In practice, this assumption may not hold, as the kernel and boot thread can access arbitrary addresses during boot and initialisation. To partially address this issue, we re-run the experiment outlined in Listing 11 with a minor modification. The modification involves touching a number of random cache lines that reside in the same colour as the shared buffer immediately after flushing the LLC. This simulates the potential impact the boot process and boot thread may have on the timing channel we have been observing.

Unfortunately, in this modified experiment, `LeakiEst` fails to report the standard deviation of the zero-information leakage distribution because it is effectively zero. As a result, to report the results of this experiment, we must compute the difference between the mutual

information and the zero-information leakage threshold as computed by `LeakiEst`. Doing so yields the graph shown in Figure 8.12. The results demonstrate a similar decaying pattern, it is just that the channel decays more dramatically initially and maintains a more gentle degradation after the first round. This result is consistent regardless of however many unrelated cache lines are accessed initially.

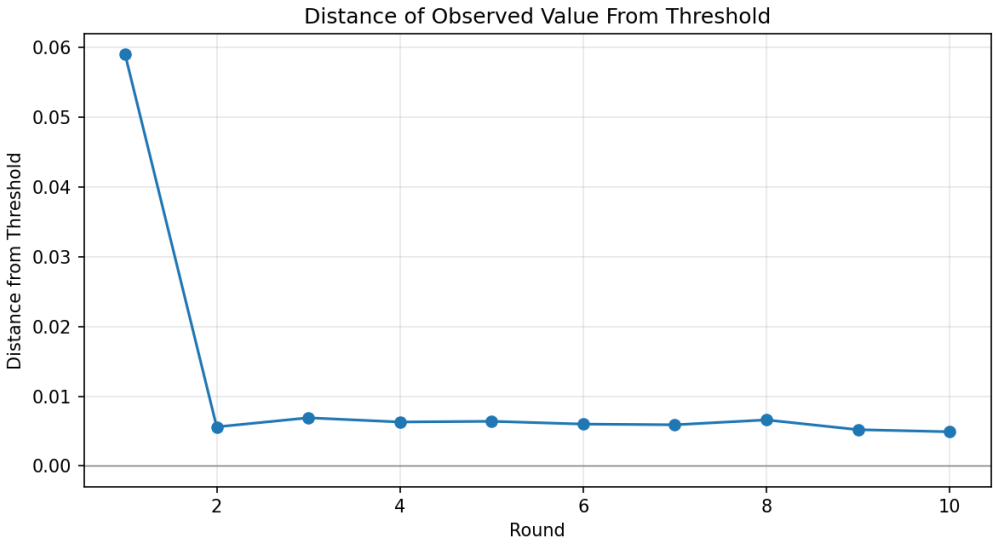


Figure 8.12: Illustration of how channel strength decays with time.

8.1.4 Summary

The results of this discussion highlight that pre-fetching as a method for determinisation is flawed. This was initially argued by Buckley et al. [2023] in the context of shared kernel data, where pre-fetching failed due to the lack of guarantees it gave regarding the state of overlapping cache lines within the LLC after pre-fetching. However, the situation clearly generalises to shared memory between domains too, even when using a dedicated colour. The issues with pre-fetching fundamentally stem from the randomised eviction policy maintained by Cheshire's LLC. The lack of guarantees as to what cache lines are evicted during pre-fetching allows for an adversary to construct timing channels that decay in strength with time and use.

The results of this discussion imply that for determinism to be a viable method for implementing shared memory, we must either have a deterministic LLC eviction policy or mechanisms to completely flush the cache lines associated with a shared buffer from the LLC. While such mechanisms exist on architectures like x86 and ARM, Cheshire provides no support outside DPLLC. Unfortunately, without completely reworking the present implementation of time-protection to utilise DPLLC, or without further guarantees from the hardware, it appears as if determinisation is a dead end.

8.2 Copy-on-Domain-Switch

Since determinisation using pre-fetching is ineffective, we now turn our attention to the idea of copying on a domain switch. This approach requires each domain to maintain a copy of the shared buffer, allocated from its unique cache colour. The kernel or a trusted third-party process then synchronises these copies whenever a domain switch occurs, specifically when leaving the writer domain.

8.2.1 Copy performed by the kernel

This approach involves both A (the writer domain) and B (the reader domain) maintaining a local copy of the buffer allocated from their respective colours. On a domain switch from $A \to B$, the kernel copies the contents of A's buffer into B's copy. Since the buffers are isolated in the LLC, the only way for information to leak between A and B in a manner that violates the information flow constraints is through the latency of the copy operation, which depends on the microarchitectural state of the buffer. In the current implementation of time protection on RISC-V, the time taken to perform the domain switch from $A \to B$ is deducted from B and affects when the first thread in B is scheduled. As a result, the first thread in B can observe the latency of the copy operation by observing the time at which it is scheduled. The timing channel introduced through the copy operation can be mitigated relatively easily by padding the operation to its WCET. This requirement to pad the copy also dictates when the kernel should perform the operation. Since `fence.t` already provides a hardware mechanism to pad all prior operations to a WCET, it is logical to perform the copy before `fence.t`, so as to include the copy latency in `fence.t`'s padding time. It is worth pointing out that this design assumes that the time between timer interrupts is sufficiently long enough to allow for the full buffer to be copied.

Since this design requires modifications to the kernel, it will necessitate the introduction of new kernel APIs and objects; specifically, an object for associating the writer copy of a frame with the reader copy, and an API for linking this object to the writer domain. As with notifications, domain association will be achieved by extending the domain capability and introducing a new `SetTimeProtectedFrames` invocation. This invocation will take a capability pointer to the newly introduced `TimeProtectedFrames` object, which associates the writer and reader frames, and attach it to a per-domain linked list of frames that require copying on a domain switch. Because the kernel must be able to access the reader and writer frames, the frames used in this design cannot be `device` memory. This requirement limits the design, as not all memory available for frames is accessible to the kernel.

8.2.2 Copy performed by a trusted thread

The synchronisation of the buffer copies between A and B could, in principle, be performed by an external thread C_t. However, to maintain correctness, this thread must always satisfy one of two constraints: it must be the last thread in domain A to run before the domain

time-slice elapses, or it must run before any other thread in B is scheduled. Ensuring that C_t is always the last thread to run in A is difficult, since the regular thread-level scheduler does not maintain a static schedule like the domain-level scheduler. A simpler alternative is to attach C_t to domain B and attempt to make it the first thread in B that runs, this can be achieved by assigning C_t the maximum priority in that domain.

This approach, however, raises the concern that C_t could starve other threads in B. To avoid this, we introduce a pair of cross-domain notifications. The first notification, N_1, connects A to B with B as the receiver, while the second, N_2, connects B to A with A as the receiver. In addition, we introduce a new thread D_t in domain A with the maximum priority in A, thereby ensuring it is always the first thread to run when A is scheduled. The coordination between C_t and D_t forms a simple handshake. When B is scheduled, C_t runs first, performs the buffer copy, signals N_2, and then blocks on N_1, allowing other threads in B to execute. When A next becomes active, D_t runs first, signals N_1 to unblock C_t, and then blocks on N_2 to wait until the next domain switch. It is worth noting that this scheme requires an initial invocation of `poll` on each of the notifications to clear their state before entering the copy loops. The alternating signal–wait sequence that this design utilises ensures that buffer synchronisation occurs at each switch without causing starvation. Some pseudocode for these routines is provided in Listing 12. Like with the kernel approach, the copy operation with this approach must be padded to a WCET. This is because the latency of the copy operation affects when the next thread in B is scheduled, as such it is potentially a usable timing channel.

```
void c_t() {
    poll(N_1);

    while (true) {
        copy_buffers_with_wcet();

        signal(N_2);
        wait(N_1);
    }
}

void d_t() {
    poll(N_2);

    while (true) {
        signal(N_1);
        wait(N_2);
    }
}
```

Listing 12: Illustration of the D_t and C_t threads.

The key limitation with this design is the requirement for both the D_t and C_t threads to be the maximum priority threads in their respective domains. However, since there are 256 distinct priorities that a thread can have, this is largely a non-issue. Additionally, it is worth pointing out that implicit in this design is the assumption that the domain tick interval is long enough so that a full copy of the buffer can occur without switching to the next domain. For a sufficiently long buffer this will imply the need for B's domain time-slice to support copying the full buffer.

8.2.3 Which to implement?

There is value in implementing and benchmarking both designs for copy-on-domain-switch. The in-kernel approach adds some complexity to the kernel; however, the constraint that cache lines unrelated to the current domain's colour are only accessed on a domain switch may simplify the eventual possible verification story of this design. In contrast, the user-level approach represents a more principled design, as it does not introduce new kernel features and can be implemented relatively simply.

Chapter 9

Shared Memory Implementation

Chapter 8 focused on providing a rough illustration of the chosen designs for shared memory, namely, the copy-on-domain-switch approach at both user level and with kernel support. This chapter will focus on inspecting some implementation concerns with the respective designs.

9.1 Kernel Approach

Implementing shared memory with the assistance of the kernel requires introducing a new structure known as the `time_protected_frame` for tracking information regarding the reader and writer frames that require synchronisation. The `time_protected_frame` object maintains pointers in the kernel's address space to the reader/writer frames and a `next` pointer associated with a linked list of frames attached to a domain. When attached to a domain, frames are placed in a per-domain linked list of frames that require synchronisation when leaving the associated domain. The head of the linked list is placed in scratchpad memory, although could very reasonably be placed in kernel-specific memory while still maintaining correctness.

9.1.1 Domain and frame association

The `time_protected_frame` object exposes a `SetFrames` invocation, which takes CPtrs to the reader and writer frames and attaches them to the `time_protected_frame`. Since frames also need to be associated with a domain, the implementation will extend the `Domain` capability with a `TimeProtectFrame` invocation, which takes the CPtr of the `time_protected_frame` along with the writer domain, and attaches the frame to the writer domain's linked list of frames to synchronise.

9.1.2 Domain switch operations

On a domain switch leaving some domain A, the kernel traverses A's linked list of frames that require copying and, for each frame, copies the contents of the writer frame into the reader frame. Because the latency of this operation depends on the microarchitectural state of the reader and writer frames, it must be padded to a WCET. Fortunately, since this copy occurs in the domain switch path, we can use `fence.t`'s PAD CSR to pad the operation.

As a reminder, on the CVA-6 core on Cheshire, when a timer interrupt arrives, the core will read the present value for the PAD CSR and use this when performing the subsequent `fence.t` pad. It follows, then, that in order to incorporate the copy latency into `fence.t` pad, we must configure the CSR once we enter the new domain and know how many frames will be copied when *leaving* this new domain. This ensures that the domain switch and copy operation that occurs when *leaving* this new domain is appropriately padded to the correct WCET. Some pseudocode illustrating the effect on the domain switch path is provided in Listing 13.

```
// ...
// ... Old domain
switch_to_new_domain();
// ...
// ... New domain
uint32_t num_frames = count_shared_frames_in_current_dom();
uint32_t total_pad_time = domain_switch_wcet + num_frames * frame_wcet;
configure_pad(total_pad_time);

// This fence.t does not use the new configured padding, it uses the
// old padding. The next fence.t that occurs when leaving this new domain
// will use this updated pad time.
fence.t();
```

Listing 13: Fence.t padding configuration.

Bounding the WCET

We will bound the WCET of the copy operation in a similar manner to what we performed for notifications. We will saturate the LLC and L1-D with dirty cache lines and saturate the L1-I with unrelated instructions, then on a domain switch we will directly sample how long the copy operation takes. After collecting 20,000 samples and taking the maximum, we arrive at a WCET bound of 55,000 cycles for to copy a single frame of memory.

9.2 User-Level Approach

The implementation of this approach closely follows the design that was outlined in Chapter 8, specifically Section 8.2.2. The implementation is a direct mapping of the discussed design, so there are not that many details to elaborate on here.

9.2.1 Constant time-padding and bounding the WCET

The latency of the copy operation affects the scheduling of the next thread in B as the longer the copy operation takes, the further in time non-copy threads in B attain a time slice. We can prevent this from forming a timing channel by padding the copy operation to a worst-case execution time. However, unlike the in-kernel approach, where there exists hardware support for WCET padding, no such support exists for this approach. We can, however, employ the software-based time-padding approach of error correction discussed in Chapter 6. We will bound the WCET with an identical method to what was described for the kernel-based approach. After collecting 20,000 samples and taking the maximum, we attain a WCET of 30,000 cycles to copy a single frame of memory. It is worth pointing out that the WCET of the user-level copy operation is significantly smaller than the WCET of the in-kernel approach, this is not intuitively obvious at all and further work needs to be done in order to figure out why. It may very well be that this is a quirk of Cheshire, and this story may not be true on other RISC-V based platforms.

Chapter 10

Shared Memory Evaluation & Discussion

We will now turn our attention towards evaluating the presented design and implementation for cross-domain shared memory. The design will be evaluated by conducting various channel and performance benchmarks.

10.1 Timing Channel Benchmarks

Recall the information flow constraints that a cross-domain memory buffer between a writer domain A and a reader domain B must satisfy:

R1. Information can only flow from $A \to B$ by A writing into the buffer. Additionally, the impact that reads and writes performed by A have on the microarchitectural state of the buffer should not introduce a usable timing channel.

R2. Any influence that threads in A have on the buffer's microarchitectural state must not form a usable timing channel. This means that if the execution of threads within A incidentally evicts cache lines associated with the buffer, then the impact left by this eviction must not allow for information leakage.

R3. Information flow from $B \to A$ is prohibited. The execution of threads in B must not affect the buffer's microarchitectural state in a way that forms a timing channel. For example, the side effects of B reading from the shared buffer must not become a usable timing channel.

We will now benchmark whether the presented designs for cross-domain shared memory satisfy these constraints. All benchmarks will consist of a Trojan and a Spy, the benchmarks may differ in whether the Trojan or the Spy is the designated writer or reader

for the shared buffer, this will be indicated with parentheses, with (A) indicating that a thread is the writer and (B) indicating that a thread is the reader. Where appropriate, each benchmark result will be accompanied by a corresponding result obtained using a normal shared buffer that does not respect time protection. This allows us to demonstrate that the benchmarks are sensitive to the feature under evaluation. The benchmarking environment is identical to that presented in Chapter 4 and all channels are quantified using `LeakiEst`.

10.1.1 Direct reads/writes benchmark

The information flow requirements R1 and R3 mandate that the impact reading and writing has on the microarchitectural state of the buffer must not form a usable timing channel. As such, our first benchmark will benchmark whether the presented designs for cross-domain shared memory enable this kind of information flow. This benchmark will consist of a Trojan and a Spy domain connected by a cross-domain buffer. The Trojan encodes a secret by reading or writing to n cache lines in the buffer. The Spy observes said secret by timing how long it takes to probe the entire buffer. A correlation between the encoded secret and the Spy's observed probing latency would indicate a timing channel. We will conduct three separate variants of this benchmark. In the first two, the Trojan (A) is the *writer* domain, and we test whether the Spy (B) can infer how many cache lines the Trojan had read from or written to. In the third, the Trojan (B) is the *reader* domain, and we test whether the Spy (A) can infer how many cache lines the Trojan had read from. All benchmarks will use a two-page buffer. This buffer is sufficiently long to span all cache lines allocated to a domain, as two pages map to 128 distinct cache sets (i.e. two colours).

We will first look at the results in the situation where the Trojan is the writer domain A. To demonstrate the sensitivity of this benchmark, we will first conduct the benchmark with a regular non-time-protected buffer, giving us the result in Figure 10.1 when the Trojan reads from the buffer to transmit a secret. When conducting the read variant of this benchmark with a protected cross-domain shared buffer we arrive at the channel matrix in Figure 10.2 for the in-kernel approach and Figure 10.3 for the user-level approach. For the write variant, we arrive at the result in Figure 10.4 for the in-kernel approach and Figure 10.5 for the user-level approach. In the second scenario, where the Trojan is the reader domain B, we arrive at results in Figure 10.6 for the in-kernel approach and Figure 10.7 for the user-level approach. All of these results indicate a clear lack of a timing channel. This is to be expected as the buffer copies lie in distinct colours. Regardless, the benchmark results demonstrate that the presented designs clearly satisfy the first information flow constraint. They prevent information leakage through the impact reads and writes have on the microarchitecture of the buffer.

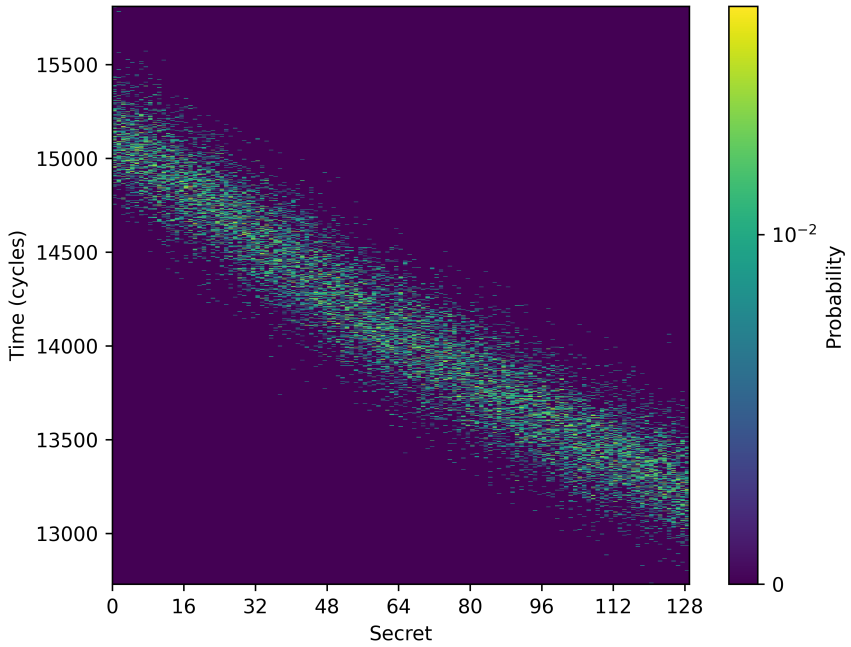


Figure 10.1: Unmitigated A read channel.

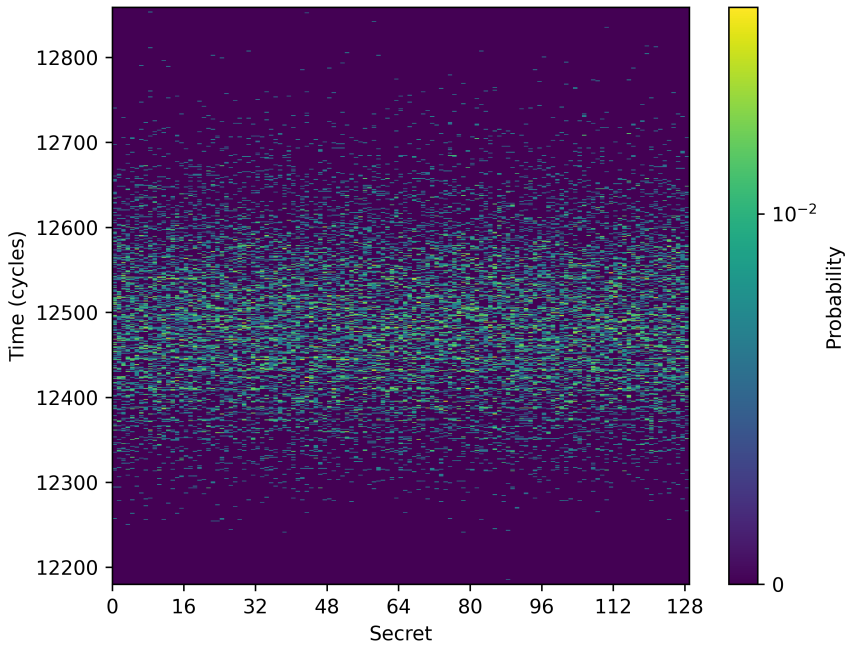


Figure 10.2: Mitigated A read channel (Kernel). $\mathscr{M} = 0.0162$. $\mathscr{M}_0 = 0.0182$.

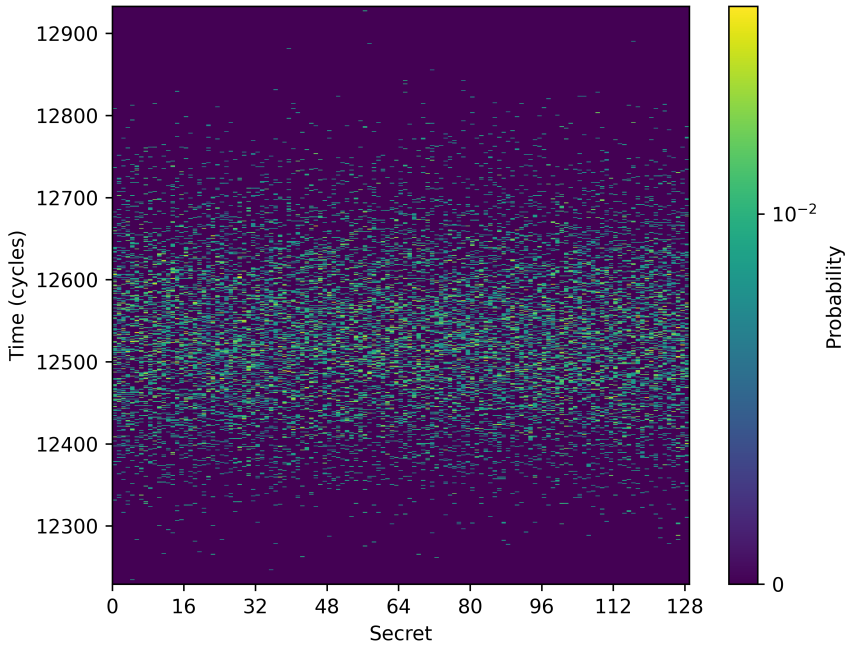


Figure 10.3: Mitigated A read channel (User-Level). $\mathscr{M} = 0.0102$. $\mathscr{M}_0 = 0.0120$.

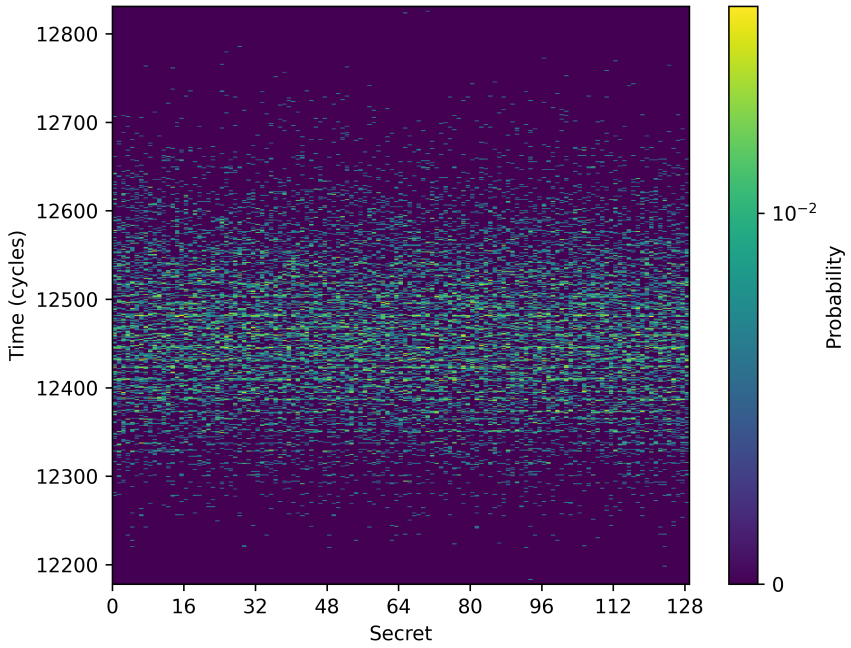


Figure 10.4: Mitigated A write channel (Kernel). $\mathscr{M} = 0.0173$. $\mathscr{M}_0 = 0.0184$.

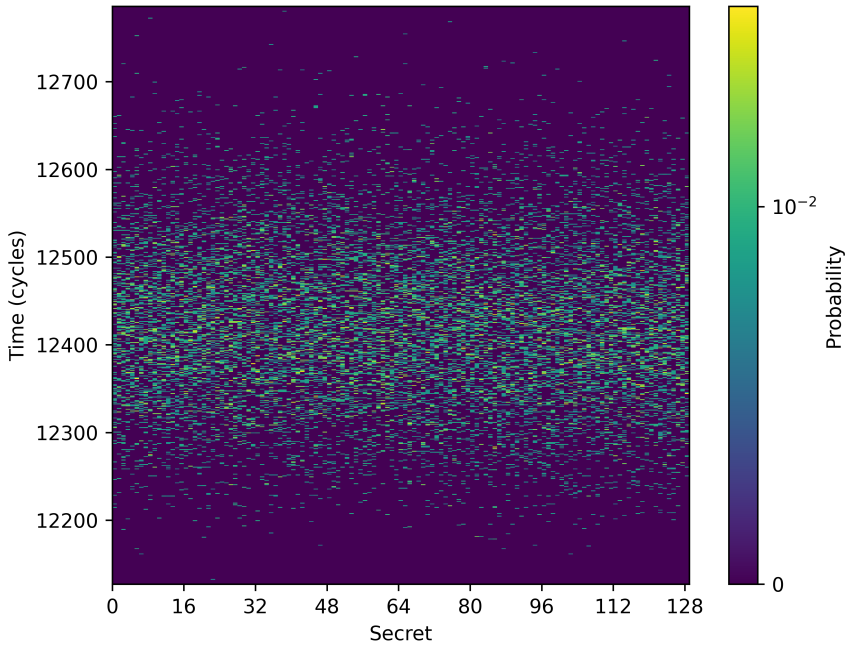


Figure 10.5: Mitigated A write channel (User-Level). $\mathscr{M} = 0.0207$. $\mathscr{M}_0 = 0.0238$.

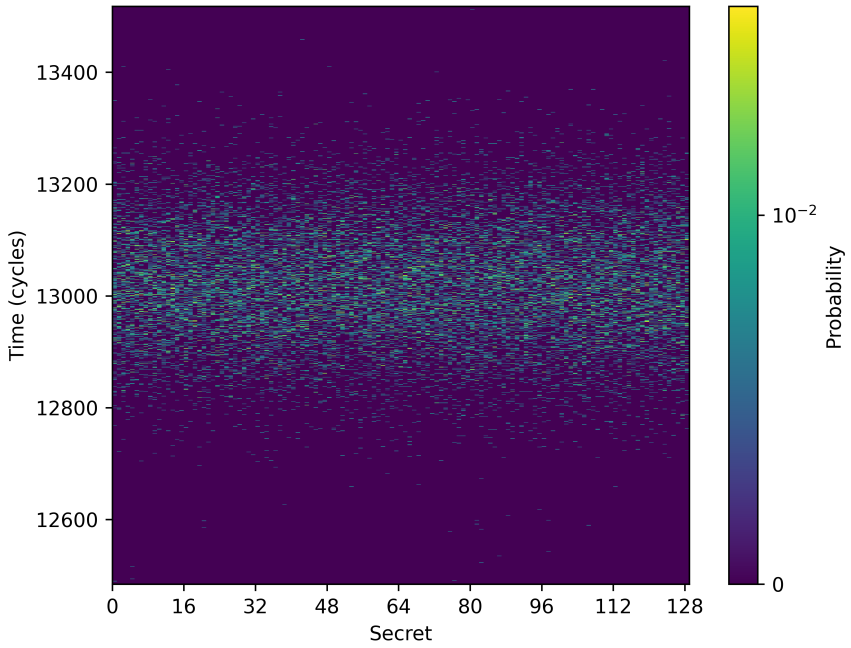


Figure 10.6: Mitigated B read channel (Kernel). $\mathscr{M} = 0.0204$. $\mathscr{M}_0 = 0.0213$.

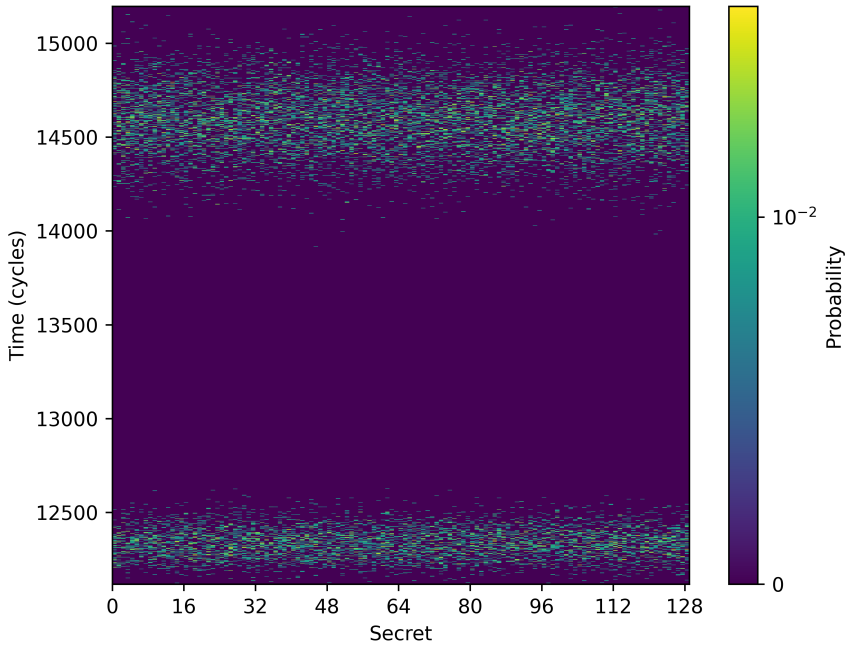


Figure 10.7: Mitigated B read channel (User-Level). $\mathscr{M} = 0.0102$. $\mathscr{M}_0 = 0.0120$.

10.1.2 Copy latency benchmark

Another way for information to flow from $A \to B$ in a manner that would violate the information flow constraints R1 and R3 is via the latency of the copy operation performed when leaving domain A. To benchmark this potential timing channel, we will construct a two-variant test involving a Trojan (A) and a Spy (B). In the first variant, the Trojan encodes a secret by reading from n cache lines in the shared buffer. The Spy then attempts to infer said secret by measuring the time elapsed since it last had a time-slice. The difference in when the Spy last had a time-slice and the present time captures the copy latency for both the kernel and user-level approaches. The Spy can determine this difference using a small modification of the system tick detection logic described by Ge [2019], in a manner similar to what we used to conduct the notification benchmark outlined in Section 7.1.1.

In the second variant of the benchmark, the Trojan primes n cache lines using a priming buffer allocated from its own colour. This will evict n cache lines associated with the Trojan's copy of the shared buffer from the LLC. We expect both the user-level and in-kernel designs to mitigate these channels, as the copy operations are padded to their WCET. In both benchmarks, the Trojan is the writer domain for the shared buffer. It is sufficient to test only this configuration, as the copy operation only occurs after leaving the writer domain, and as such, the latency will only be observable by the reader domain.

All copy latency benchmarks will be conducted with a two-page-long shared buffer, a base domain switch WCET bound of 150,000 cycles, and a kernel timer tick of 40 ms. We will

first examine two baseline results of the benchmark. Both results are from the priming variant of this benchmark, but are executed on a system where the copy operation is not padded to a WCET. The result of this benchmark for the in-kernel approach is presented in Figure 10.8 and Figure 10.9 for the user-level approach. The strange shape of the channel matrix for the in-kernel case is explained by the fact that for most secrets, the copy latency is largely absorbed by the existing WCET for the domain switch operation. Regardless, these baseline results demonstrate that our benchmark is sensitive to variations in the copy-latency.

When we conduct the first variant of this benchmark that involves reading from the shared buffer, we attain the channel matrix in Figure 10.10 for the in-kernel approach and the channel matrix in Figure 10.11 for the user-level approach. For the priming variant, we attain the channel matrix in Figure 10.12 for the in-kernel approach and the channel matrix in Figure 10.13 for the user-level approach. It is worth pointing out that the priming variant of this benchmark is susceptible to interference from the scheduler data channel discussed in Chapter 6 and Appendix B. To mitigate the impact that this channel has on our measurements, we will apply the temporary work-around of pre-fetching scheduler data on a domain switch. It should be noted that pre-fetching is rather fragile and does not fix this initial channel. Additionally, scheduler data is supposedly partitioned between domains, so it is surprising that there is even a channel through it in the first case. It may well be possible that the cause of the channel is much more nefarious, and during this thesis it materialised as a channel through scheduling data. Further work needs to be done to resolve this pre-existing channel and re-run this channel benchmark once a more principled fix has been found. Regardless, the collected results with the applied work-around demonstrate that there is no timing channel through the copy latency, as no channel matrix indicates a timing channel. The broader implication of this benchmark result is that there is no way for information to flow from $A \rightarrow B$ via the copy latency.

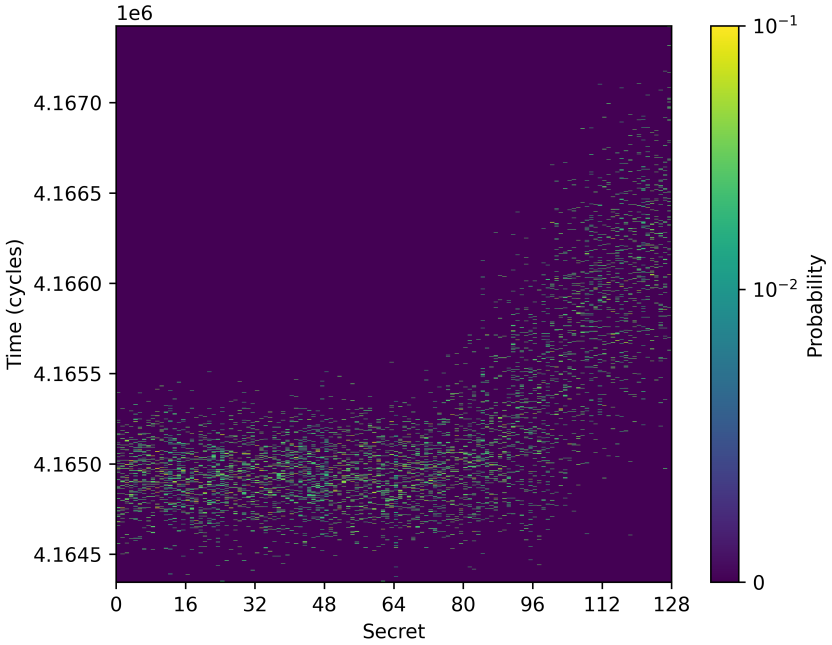


Figure 10.8: Unmitigated domain-switch priming channel (Kernel).

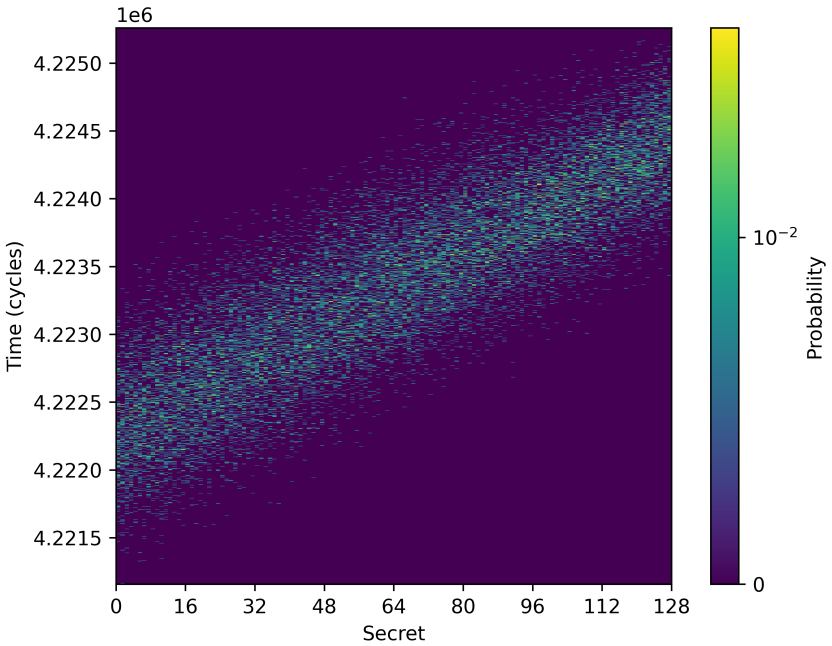


Figure 10.9: Unmitigated domain-switch priming channel (User-Level).

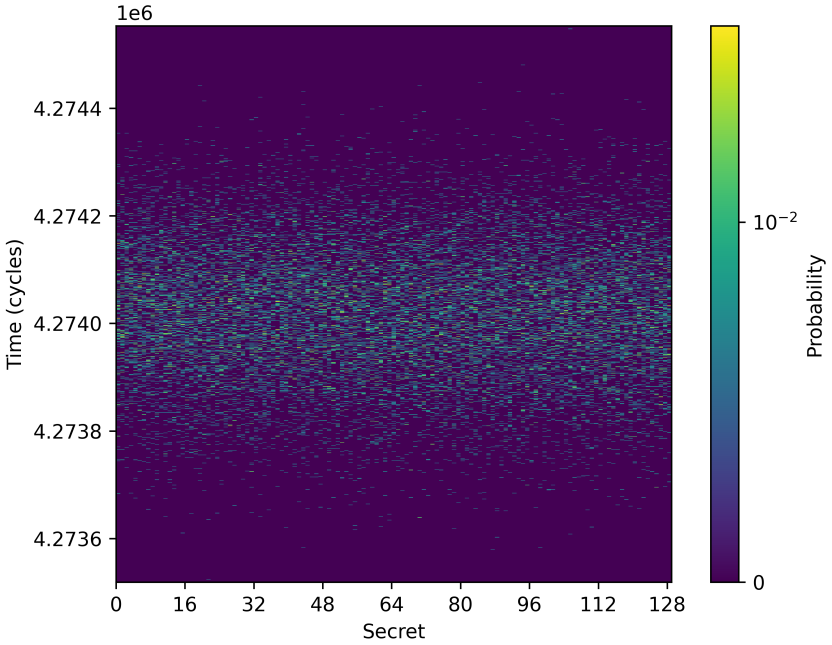


Figure 10.10: Mitigated domain-switch reads channel (Kernel). $\mathscr{M} = 0.0166$. $\mathscr{M}_0 = 0.0184$.

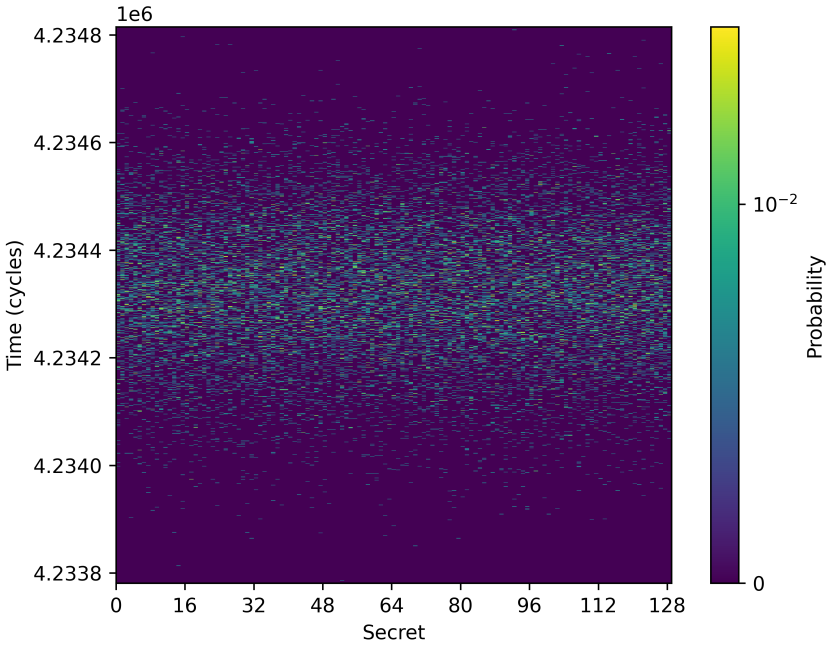


Figure 10.11: Mitigated domain-switch reads channel (User-Level). $\mathscr{M} = 0.0175$. $\mathscr{M}_0 = 0.0193$.

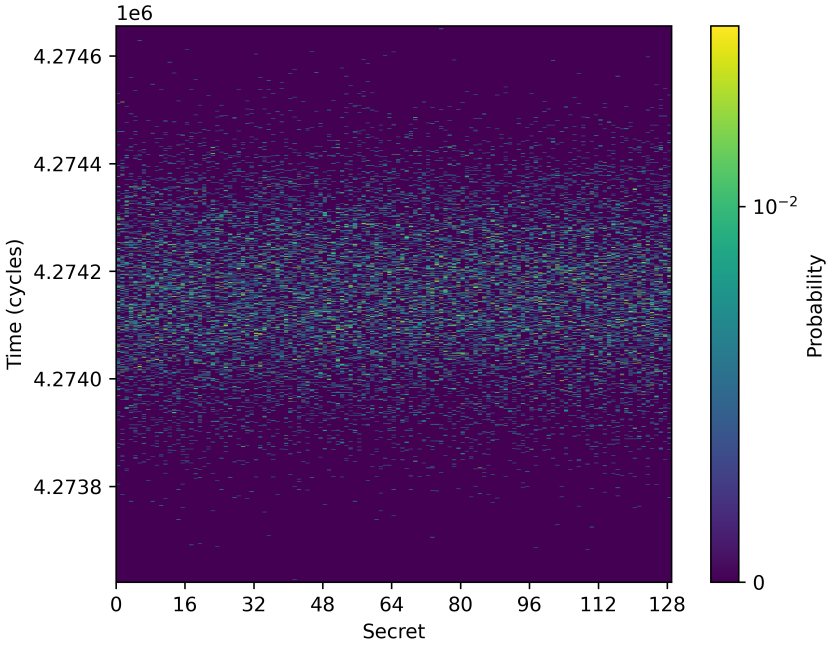


Figure 10.12: Mitigated domain-switch priming channel (Kernel). $\mathscr{M} = 0.0159$. $\mathscr{M}_0 = 0.0183$.

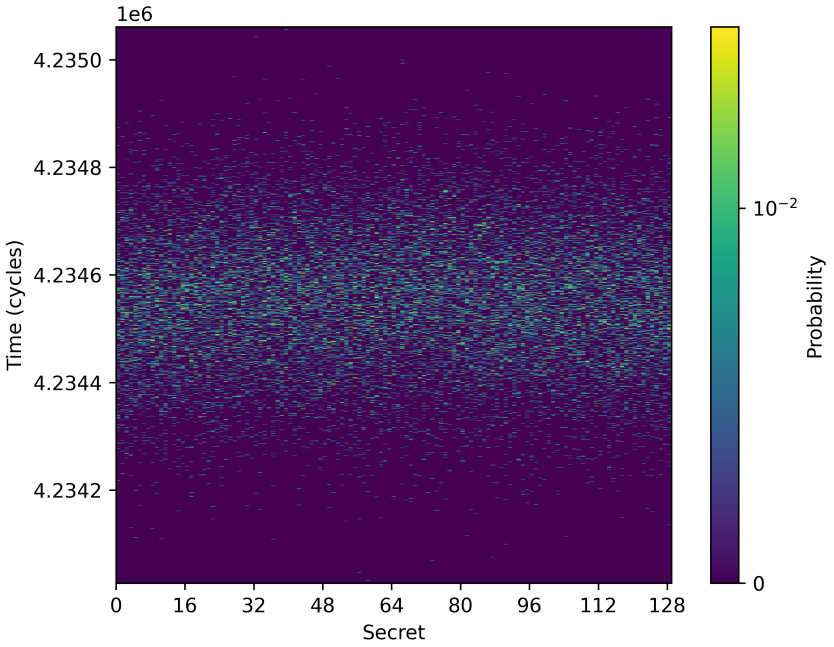


Figure 10.13: Mitigated domain-switch priming channel (User-Level). $\mathscr{M} = 0.0193$. $\mathscr{M}_0 = 0.0193$.

10.1.3 Unrelated activity benchmark

The final way information not written to the buffer can flow from $A \to B$ and $B \to A$ is through the impact that the execution of threads in each domain may have on the microarchitecture associated with the shared buffer, this is outlined in Requirements R2 and R3. The execution of threads may result in the eviction of cache lines associated with the shared buffer, which in the naive case, where a shared buffer is simply mapped between two domains with no time protection, would result in a timing channel. As such, the final set of benchmarks will test whether activities that each domain performs unrelated to reading/writing from the buffer can leak through the microarchitecture associated with the buffer. We expect this to be mitigated by the fact that the copies are allocated from distinct colours, but it is worth confirming this fact experimentally. We will benchmark this with a channel benchmark consisting of a Trojan domain and a Spy domain. The Trojan allocates a priming buffer within its own colour and, to send a secret n, primes n cache lines in that buffer. The Spy then probes the entire shared buffer and measures the total access time. This benchmark is conducted twice: once where the Trojan is the *writer* domain, and once where the Trojan is the *reader* domain of the shared buffer. Conducting both variants allows us to determine whether information unrelated to the buffer's contents can flow from $A \to B$ or from $B \to A$.

To demonstrate the sensitivity of this benchmark, the results obtained when conducting this benchmark with a shared buffer allocated from the Trojan's colour with no mitigation is provided in Figure 10.14, demonstrating a clear timing channel. When we conduct this benchmark with the writer domain A being the Trojan domain, we obtain the channel matrix in Figure 10.15 for the in-kernel approach and Figure 10.16 for the user-level approach. When repeating this experiment but with the reader domain B being the Trojan, we obtain the channel matrix in Figure 10.17 for the kernel approach and Figure 10.18 for the user-level approach. All these results demonstrate a clear lack of a timing channel, highlighting that there is no timing channel through the impact unrelated activity might have on the microarchitectural state on the buffer. Either from $A \to B$ or from $B \to A$.

10.1.4 Summary

All the conducted benchmarks and their corresponding results demonstrate that the presented designs successfully satisfy the information flow constraints for shared memory. The benchmarks demonstrate that information flow from $A \to B$ or from $B \to A$ is impossible through the microarchitectural state associated with the shared buffer. As such, the only way information can flow through the buffer is by A explicitly writing to it.

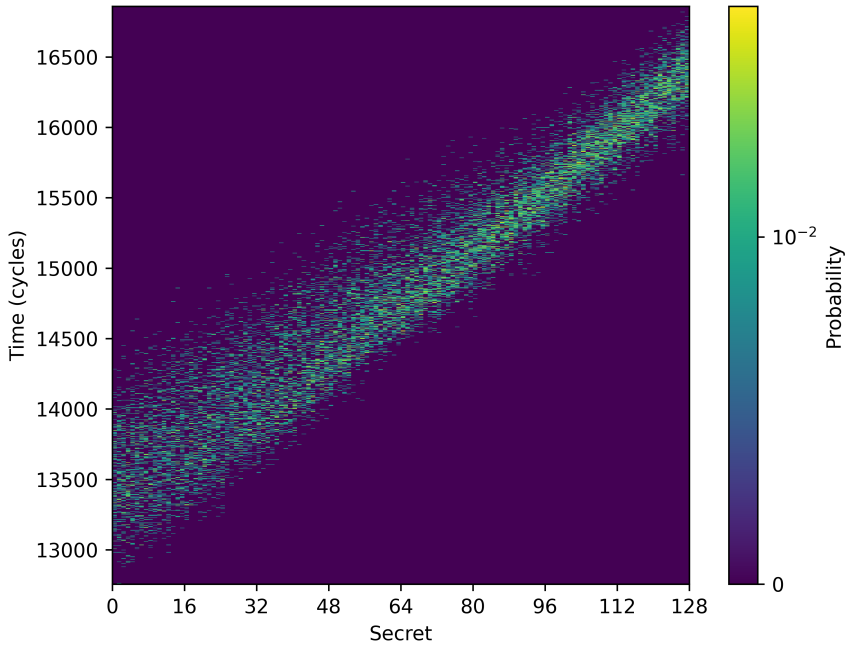


Figure 10.14: Unmitigated unrelated flow channel.

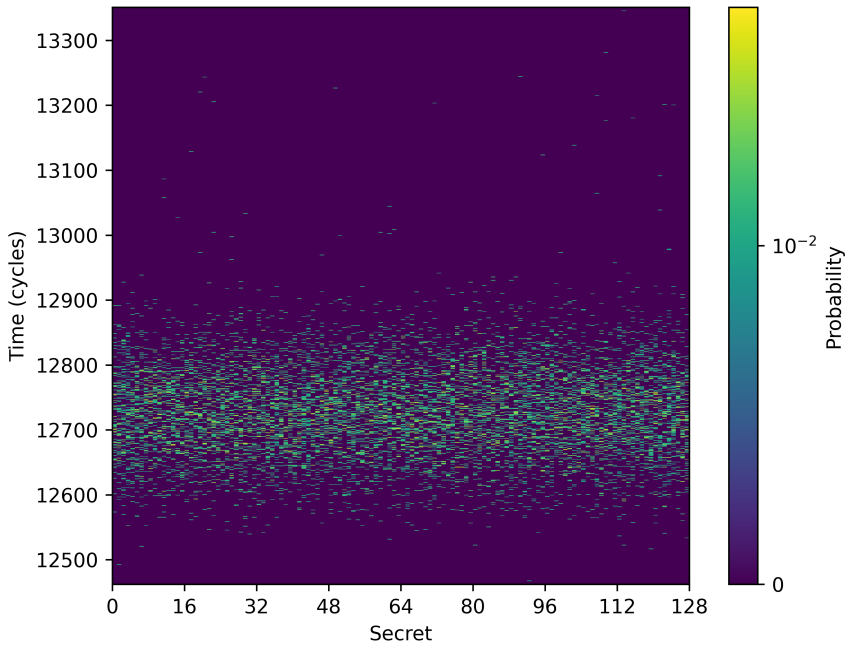


Figure 10.15: Mitigated unrelated flow channel from A (Kernel). $\mathscr{M} = 0.0424$. $\mathscr{M}_0 = 0.0442$.

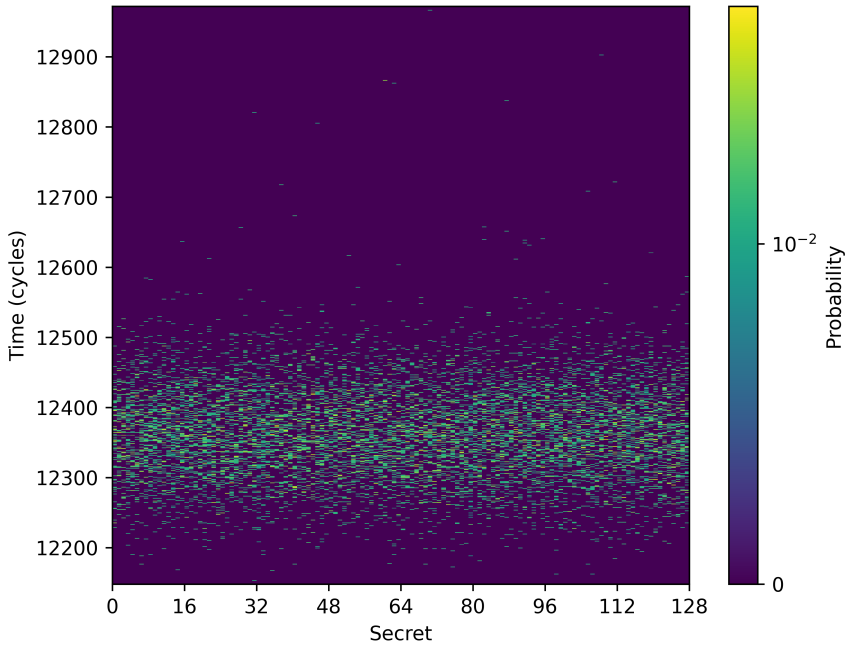


Figure 10.16: Mitigated unrelated flow channel from A (User-Level). $\mathscr{M} = 0.0357$. $\mathscr{M}_0 = 0.0427$.

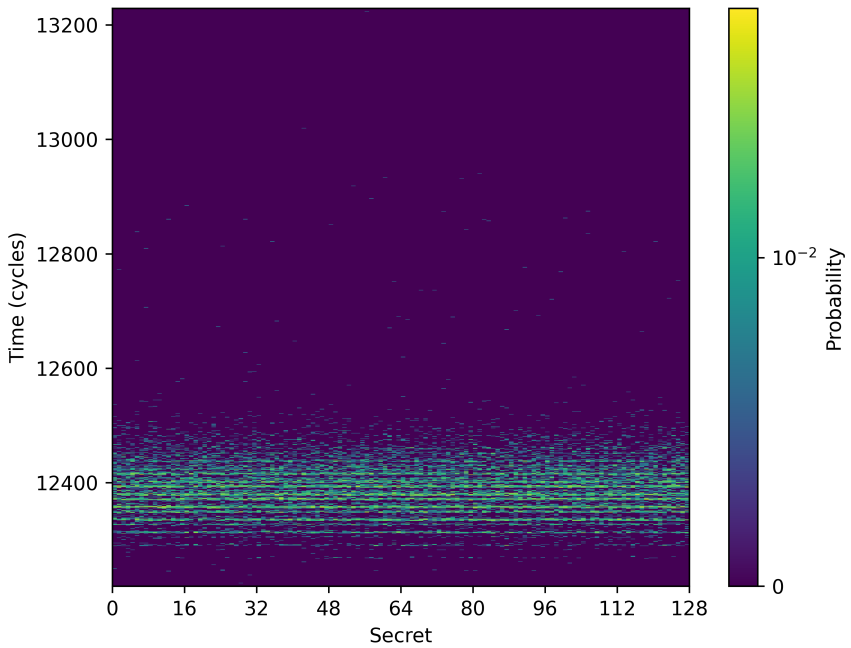


Figure 10.17: Mitigated unrelated flow channel from B (Kernel). $\mathscr{M} = 0.0269$. $\mathscr{M}_0 = 0.0287$.

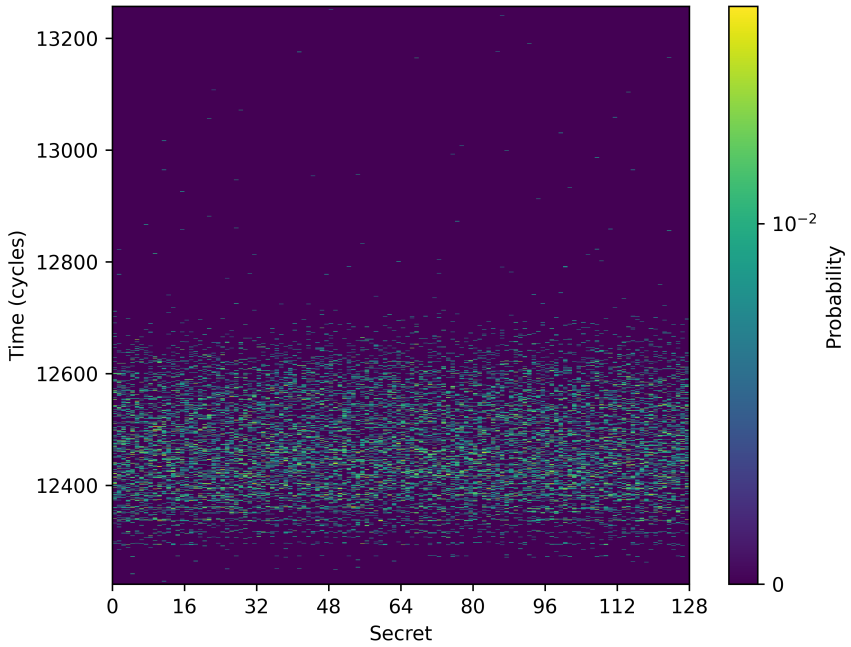


Figure 10.18: Mitigated unrelated flow channel from B (User-Level). $\mathscr{M} = 0.0258$. $\mathscr{M}_0 = 0.0280$.

10.2 Performance

The primary cost associated with the presented cross-domain shared memory designs is the overhead imposed at the domain switch. As such, this will be our primary benchmark focus. It is worth pointing out that this section only performs micro-benchmarks and, as such, is a poor representation of the true costs of cross-domain shared memory. There exists an indirect cost associated with our design of cross-domain shared memory that is difficult to capture without appropriate macro-benchmarks: the act of copying can saturate the portion of the LLC associated with the reader domain (B) with dirty cache lines. For a sufficiently large buffer, this evicts all the reader domain's cache lines from the LLC and saturates the corresponding cache sets, significantly increasing the cost of subsequent cache misses. The current domain switch latency micro-benchmark does not capture this cost; as such, future work must conduct a more holistic macro-benchmark of this approach.

To benchmark the domain switch overhead we will adapt the copy latency benchmark and have it flush the entire LLC and L1 cache between runs, this will in effect measure the overhead imposed by the designs in the cold case, i.e. when none of the cache lines associated with the kernel or shared buffer are present in the LLC. We will conduct the benchmark with various buffer sizes, specifically buffers of size 1 page, 2 pages, 4 pages, 6 pages, 8 pages, $\cdots$, 18 pages, and 20 pages. Our benchmark environment is identical to that of the timing channel benchmarks, the domain switch WCET is configured to be

150,000 cycles and the WCET of copying a single frame is configured as 55,000 cycles for the in-kernel approach and 30,000 cycles for the user level approach.

To serve as a baseline, we first measure the cost associated with a domain switch and selecting + scheduling the first thread when there are no frames to copy. When doing so, we arrive at the result below. It is important to note that this measurement includes the part of the domain switch that `fence.t` pads to a WCET. Additionally, due to errors in the clock reset logic mentioned briefly in Appendix B this benchmark will also include the amount of time it takes to trap into the kernel upon a CLINT.

Benchmark	**Mean (Cycles)**	**Standard Deviation**
Domain Switch Latency	161965	54 (0.03%)

The latency results for variable buffer sizes with the in-kernel approach is provided below.

Benchmark	**Mean (Cycles)**	**Standard Deviation**	**Overhead**
1 Page	216937	55 (0.03%)	34%
2 Pages	272007	63 (0.02%)	68%
4 Pages	382109	76 (0.02%)	136%
6 Pages	492300	83 (0.02%)	204%
8 Pages	602384	92 (0.02%)	272%
10 Pages	712659	101 (0.01%)	340%
12 Pages	822918	113 (0.01%)	408%
14 Pages	932930	121 (0.01%)	476%
16 Pages	1043054	123 (0.01%)	544%
18 Pages	1153043	125 (0.01%)	612%
20 Pages	1263082	122 (0.01%)	680%

And the results of the user-level approach are provided below.

Benchmark	**Mean (Cycles)**	**Standard Deviation**	**Overhead**
1 Page	203423	73 (0.04%)	26%
2 Pages	233835	87 (0.04%)	44%
4 Pages	295159	105 (0.04%)	82%
6 Pages	356057	118 (0.03%)	120%
8 Pages	417190	139 (0.03%)	158%
10 Pages	477877	151 (0.03%)	195%
12 Pages	539024	163 (0.03%)	232%
14 Pages	599834	181 (0.03%)	270%
16 Pages	660675	181 (0.03%)	308%
18 Pages	721304	191 (0.03%)	345%
20 Pages	781821	188 (0.02%)	383%

These results demonstrate that cross-domain shared memory imposes a substantial overhead on the domain switch latency, an overhead that scales linearly with the buffer size. This is unsurprising though given how expensive the worst-case latency of copying a single frame of memory appeared to be. It should be noted that this overhead is largely a function of the memory hierarchy's performance. By minimising the latency cost of L1 and LLC cache misses on Cheshire, we can lower the WCET bound for copying a single memory frame.

10.3 Discussion

In this chapter, we evaluated the copy on domain switch approach and demonstrated that it successfully mitigated all studied timing channels. We also demonstrated that this approach can be implemented at user-level without the assistance of the kernel, and doing so only required a working implementation of cross-domain notifications. Unfortunately, however, the studied design is costly, imposing a significant overhead on the domain switch path. Specifically, in the kernel case, where a two-page buffer alone imposes a 68% overhead on the domain switch, primarily due to the need to pad the copy operation to its WCET. There is also an indirect cost to the copy on domain switch approach: a sufficiently long buffer will cause the copying operation to saturate the LLC with dirty cache lines. Assessing the resulting performance implications of the indirect cost requires macro-benchmarks under realistic conditions, a study that was unfortunately not conducted in this thesis.

Despite the limitations discussed, Chapter 8 suggests that copy-on-domain-switch is the only viable software-only approach to cross-domain shared memory when using colour-based partitioning on Cheshire. As such, if we wish to improve the performance story for cross-domain shared memory then we will require hardware mechanisms to flush specific addresses from the LLC, enabling a determinisation-based approach that does not rely on software pre-fetching.

10.4 Further Work

There are still some threads that have been left unresolved and need to be completed as further work. The primary one being the need to conduct a more thorough holistic benchmark of how copy-on-domain-switch performs under real-world circumstances. Additionally, it was surprising that the WCET for the in-kernel approach was so much worse than that of the user-level approach. This could be a quirk of Cheshire, but is very much something that needs to be investigated a bit more.

The next piece of further work concerns the scheduling data channel outlined in Appendix B. In our channel benchmarks, we applied a temporary workaround to demonstrate that our approaches were effective. However, further work is needed to fully resolve this channel

and re-run the domain switch latency benchmarks once the underlying cause has been identified and addressed.

Finally, the results of this thesis demonstrate that copy-on-domain-switch is the only viable software-only approach to cross-domain shared memory. This approach, however, introduces substantial overhead. As a result, further work should focus on reworking the current implementation of time-protection on Cheshire to leverage its dynamically partitionable last-level cache. Julia Vassiliki is already exploring this approach, but the results of this thesis highlight that it is necessary to achieve a more efficient implementation of cross-domain shared memory. Moving to DPLLC would enable for a determinisation-based approach that does not involve software pre-fetching, significantly reducing the overhead imposed at the domain-switch. A DPLLC based approach to shared memory may involve the introduction of new kernel APIs that allow for frames to be associated with arbitrary LLC partitions. We may also have to extend the `Domain` capability to indicate to the kernel that a partition requires flushing upon a domain switch.

Chapter 11

Conclusion

This thesis explored extending the present system model for time-protection to incorporate cross-domain notifications and shared memory. It achieved this successfully by introducing and benchmarking a design for each communication primitive, demonstrating that no information beyond what is explicitly sent could flow through the primitive. Additionally, this thesis identified a few issues with the existing implementation of time-protection on Cheshire, as outlined in Appendix B. This was not discussed in much detail however due to not being relevant to the overall aim of the thesis. Two of these channels interfered with the results of this thesis, and as such, work was done to try to resolve those channels. One of the identified channels was resolved properly, while a temporary workaround designed to produce clean data for the sake of this thesis was suggested for the other.

11.1 Notifications

The design for notifications that we explored involved introducing a new notification construct similar to the traditional notification. This construct defers the actual delivery of the notification to the domain switch, and, to yield a simpler implementation, coalesces multiple signals on a notification together to only awake one thread in the receiver domain. The benchmarks we conducted demonstrated that this new design disallows information backflow from the receiver domain B to the sender domain A and also prevents information unrelated to a signal from flowing from $A \to B$.

The design is not without its drawbacks. It imposes some overhead above what one would naively expect from moving notification delivery to the domain switch. Furthermore, the design can be generalised a bit further to allow for multiple signals to awake multiple threads in the receiver domain.

11.2 Shared Memory

This thesis also explored the problem of cross-domain shared memory. It demonstrated that pre-fetching as a means of determinising microarchitectural state associated with a shared buffer fails on Cheshire. It also introduced the concept of an *eviction-chain* to explain how this failure occurs. While not explicitly studied, the conclusions of this section should also generalise to any other architecture that maintains a randomised LLC eviction policy, and lacks a mechanism for hardware-assisted determinisation. To the best of my knowledge, the analysis methodology used to explain why pre-fetching fails is not a methodology that has been previously discussed in the literature. The broader implication of this result was that the only remaining feasible mechanism for shared memory that allowed us to retain a colour-based LLC partitioning scheme was copy-on-domain-switch.

We implemented the copy-on-domain-switch approach and demonstrated how the approach could be moved to user-level, requiring no support from the kernel outside of a working implementation of cross-domain notifications. Our benchmark results highlighted that the approach disallows information backflow from the reader domain B to the sender domain A, and prevents information that is not explicitly written to the shared buffer from flowing from $A \rightarrow B$. Copy-on-domain-switch is not without its drawbacks though.

The design introduced a significantly large overhead to the domain switch. The design is also likely to degrade the performance of a system using shared memory because the copy operation evicts cache lines associated with both the sender and receiver from the LLC. However, this last claim requires thorough benchmarking and evidence for verification. Regardless, the results of this thesis demonstrate that this is unfortunately the best that can be achieved without utilising DPLLC.

The results of this thesis also motivate a direction of future work, namely moving the present implementation of time-protection on Cheshire to make use of Cheshire's dynamically partitionable last level cache. DPLLC allows for specific partitions to be flushed from the LLC, opening up a mechanism for determinisation that does not rely on pre-fetching, enabling a more efficient implementation of cross-domain shared memory.

Appendix A — Proof of Noise Uniformity

This chapter provides a more concrete proof of the random-noise technique presented in Chapter 6. Assume that the input value X is some random variable with an unknown distribution. Additionally, we denote U to be a random variable with distribution $\mathcal{U}(0, n-1)$. Let F be the random variable attained by applying the transformation

$$f(x) = t_c + ((x - U - t_c) \mod n)$$

to the random variable X. We will prove that F is uniformly distributed. Substituting everything and performing some algebra, we find that

$$\begin{aligned} F = f(X) &= t_c + ((X - U - t_c) \mod n) \\ &= t_c + (((X - U) \mod n - t_c \mod n) \mod n). \end{aligned}$$

Since t_c and $t_c \mod n$ are constants, the only impact they have on the distribution of F is to make F a shift of the distribution of $(X - U) \mod n$. Thus, for the sake of analysis we will focus on the distribution of $(X - U) \mod n$, as if it is uniform, then F will also be uniform. For ease of notation, we denote $Y = (X - U) \mod n$. It then follows that

$$\begin{aligned} Y &= (X - U) \mod n \\ &= (X \mod n - U \mod n) \mod n \\ &= (X \mod n - U) \mod n. \end{aligned}$$

The important detail here is that the distribution of $X \mod n$ is unknown, what is known, however, is that it is a distribution over $\{0, 1, \cdots n-1\}$. And as such, if X_n denotes the random variable $X \mod n$ then

$$\sum_{x=0}^{n-1} \mathrm{P}(X_n = x) = 1.$$

Now consider $\mathrm{P}(Y = y)$.

$$\begin{aligned}
\mathrm{P}(Y = y) &= \sum_{i=0}^{n-1} \mathrm{P}(U = i) \cdot \mathrm{P}(Y = y \mid U = i) \\
\mathrm{P}(Y = y) &= \sum_{i=0}^{n-1} \frac{1}{n} \cdot \mathrm{P}(X = (y + i) \mod n) \\
&= \frac{1}{n} \sum_{i=0}^{n-1} \mathrm{P}(X_m = (y \mod n + i) \mod n).
\end{aligned}$$

Since we are summing over $i \in \{0, 1, \cdots n-1\}$, we will encounter every possible value of X_m exactly once, regardless of what y is. Thus,

$$\begin{aligned}
\mathrm{P}(Y = y) &= \frac{1}{n} \sum_{x=0}^{n-1} \mathrm{P}(X_m = x) \\
&= \frac{1}{n}.
\end{aligned}$$

Therefore, the probability of encountering a specific y is $\frac{1}{n}$, this result holds regardless of what y is. Thus, Y is uniformly distributed and by extension, so is F.

Appendix B — Existing Timing Channels

This appendix illustrates some timing channels that were found on the RISC-V port of time protection to Cheshire, and where applicable their mitigation. I would like to acknowledge Julia Vassiliki, Nils Wistoff, Dr Rob Sison and Prof Gernot Heiser, all of whom provided a great deal of help and support when working towards investigating some of these timing channels, particularly Julia, who spent a great deal of time and effort investigating the VGA controller channel with me.

B.1 VGA Controller Contention

Cheshire's VGA controller is a special piece of off-core hardware that sits in a tight loop and repeatedly reads from a frame buffer stored in memory. As such, it becomes possible for an attacker to use the contention caused by this controller to leak secrets between domains, even with the mitigation of cache colouring. This is most evident when we benchmark a user-level LLC channel. In this benchmark, the Trojan domain will allocate a priming buffer in its own colour and the Spy domain will allocate a probing buffer in its own colour. The Trojan will then prime cache lines in the priming buffer to encode a secret, and the Spy will observe said secret by timing how long it takes to probe its entire probing buffer. We expect cache colouring to mitigate this issue as the prime and probe buffers should never overlap within the LLC, however when running this experiment on Cheshire we observe the channel matrix in Figure B.1.

This channel is caused by contention induced by the VGA controller. Since the VGA controller is spinning in a tight loop and constantly reading from memory, the more memory requests issued by a Trojan, the more VGA controller requests that are queued and backed up. This directly affects the latency of the Spy's subsequent reads. We can observe that if we change the frequency at which the VGA controller scans the frame buffer, then the strength of the channel reduces. Disabling the VGA controller via OpenSBI is the only true fix for this issue, and doing so results in the timing channel in Figure B.2.

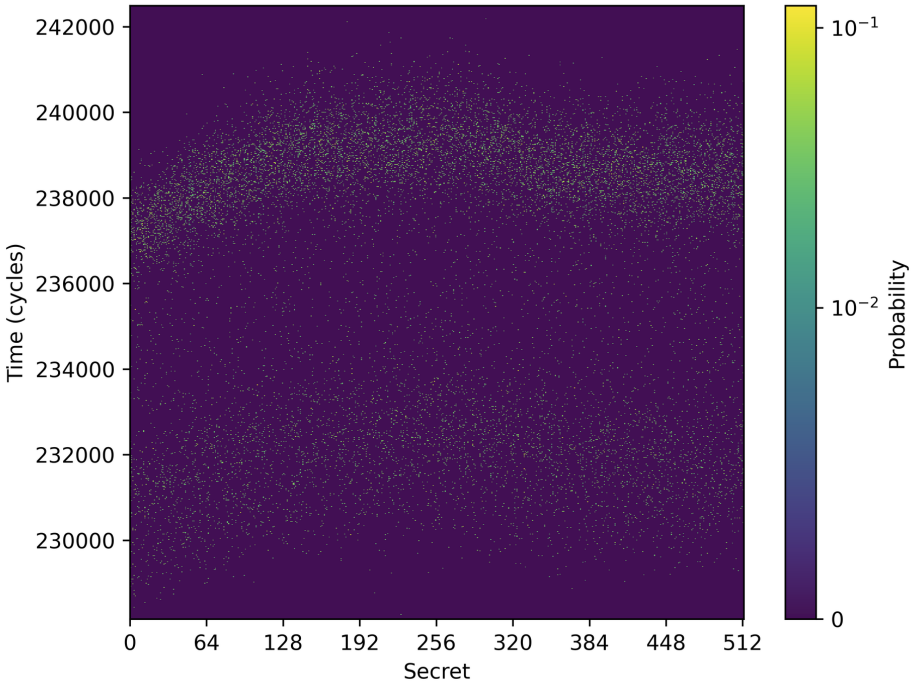


Figure B.1: Unmitigated VGA channel.

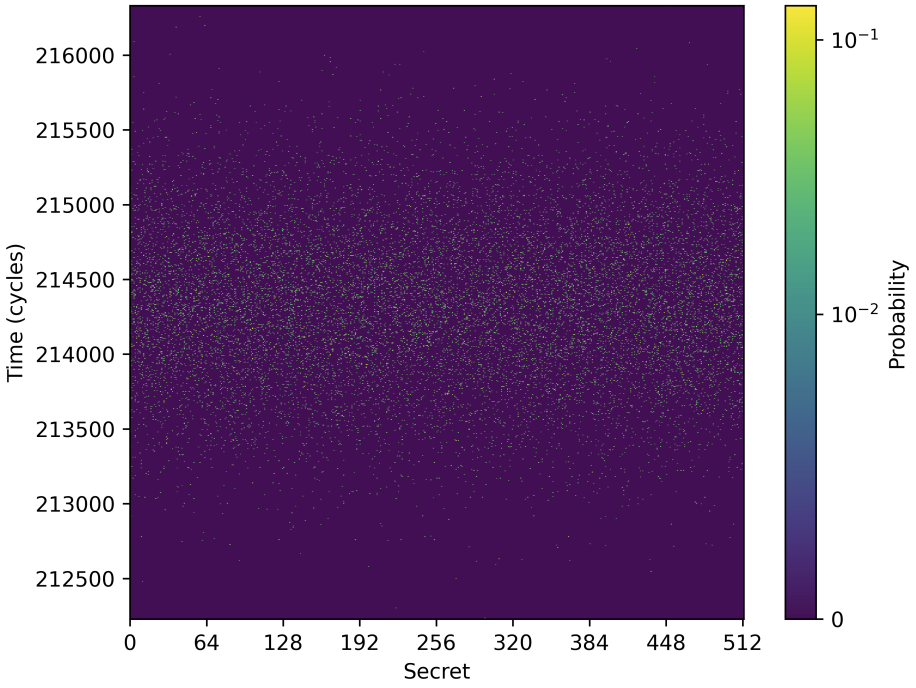


Figure B.2: Mitigated VGA channel.

B.2 Timer Drift Channel

Presently, there does not appear to be hardware support on Cheshire for a core-local timer interrupt that is raised on a periodic basis. As such, the present implementation of time-protection on RISC-V will reset the timer interrupt by taking the current time within the interrupt routine and then scheduling a new interrupt to be raised at time `now() + reset_cycles`. This approach is problematic, as the time it takes the hardware to trap into the kernel and run the interrupt routine is directly affected by the activities of the domain that was running before the CLINT was raised. As a consequence, the timestamp that is read by `now()` is controllable by an attacking process. This can be used as a timing channel.

To demonstrate this channel, we will construct a benchmark similar to what was employed to benchmark the latency of the `recv` function in Section 7.1.1. The Trojan will allocate a priming buffer within its own colour. To send a secret value of 1 the Trojan will prime the entire priming buffer. To send a secret value of 0 it will do nothing. The Spy will attempt to infer this secret by measuring the domain switch latency. It should be noted that the effects of the drift in `now()` are not directly observable in the Spy's next time-slice, but actually in the time-slice that follows that. This is because the drift in `now()` affects when the Spy's time-slice elapses, not when it is scheduled. As such, to demonstrate this channel, we will construct a channel matrix using the secret the Trojan had intended on sending in its last time-slice. The broader implication of this is that the secrets that the Trojan sends through this timer-drift channel are "delayed" by one time-slice. The timing channel for this experiment is provided in Figure B.3.

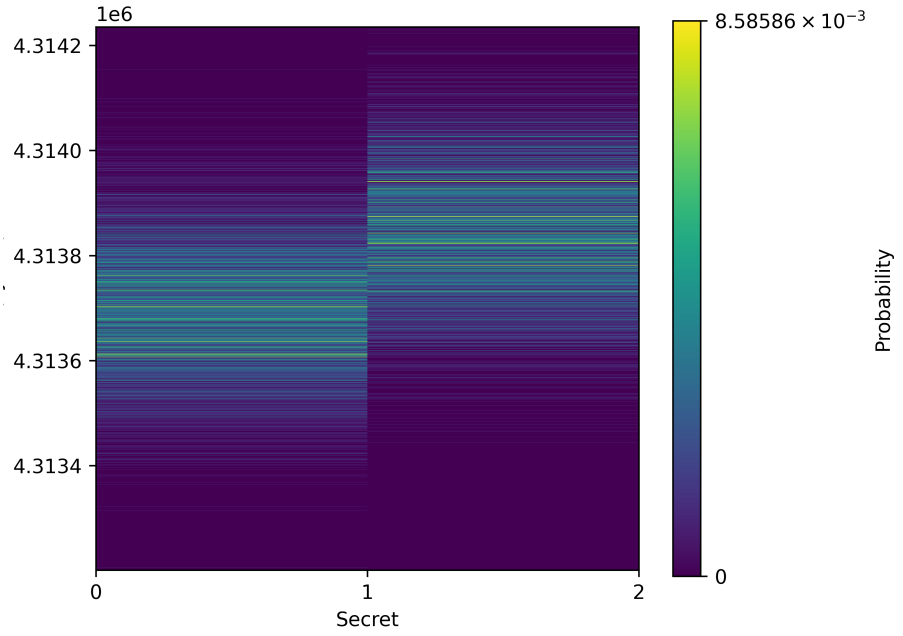


Figure B.3: Timer drift channel.

B.3 Scheduler Data Channel

On Cheshire, there appears to be another timing channel through the domain switch latency. To replicate this channel, we will conduct a benchmark consisting of a Trojan and a Spy. The Trojan will maintain a priming buffer allocated from its own colour. It will send a secret of 1 by priming all the cache lines within that priming buffer, and send a secret of 0 by doing nothing. The Spy will attempt to infer this secret by measuring the domain switch latency. Note that this channel is distinct from the previous channel, the previous channel was about drawing a correlation between the secret the Trojan sends and the drift in the timer. This channel is about observing the correlation between the Trojan's secret and the immediate domain switch latency associated with switching from the Trojan to the Spy. It is worth pointing out that in this benchmark, `fence.t` pad is appropriately configured to pad everything up to the kernel image switch to a WCET. When running the described benchmark, we acquire the channel matrix in Figure B.4.

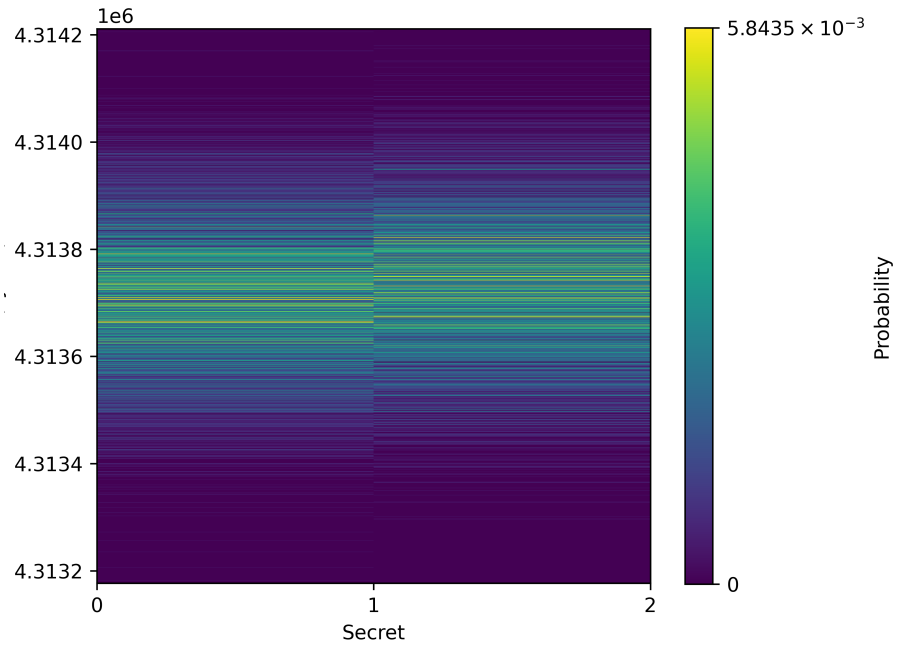


Figure B.4: Domain switch channel. $\mathscr{M} = 0.0019$. $\mathscr{M}_0 = 0.0007$.

By timing the latency of individual functions in the domain switch path the source of this channel was traced back to accesses made to scheduler data, specifically `ksReadyQueuesL1Bitmap`, `ksReadyQueues` and `ksReadyQueuesL2Bitmap`. These structures are accessed when determining the first thread in the new domain to scheduled. Surprisingly, this data is supposed to be correctly partitioned. Each kernel image is supposed to maintain its own copy of these structures that lie within its allocated colour. Another interesting observation is that the strength of the timing channel is correlated with the priority of the Spy thread, the channel matrix we see in Figure B.4 was generated

with a Spy of priority 200.

We will employ a rather hacky work-around to reduce the impact this pre-existing channel has on measurements made in this thesis. The workaround involves pre-fetching appropriate scheduler data on a domain switch. This is to be done before the `fence.t` invocation, so the latency of the operation can be incorporated within the WCET padding. As stated before, scheduler data is supposed to be correctly partitioned between domains. So, the activities of one domain should not influence the state of another domain's scheduler data within the LLC. As such, it is highly likely that the cause of the channel is much deeper and more nefarious, and it has simply just materialised as impacts on scheduler data for now. Much more work needs to be done to identify the true cause of this channel and mitigate it. Unfortunately, due to finding this channel rather late in my thesis, I was unable to properly resolve this existing channel.

Bibliography

[1] James P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, Vol. II, United States Air Force Electronic Systems Division, October 1972. URL `https://apps.dtic.mil/sti/pdfs/AD0772806.pdf`.

[2] Arm Limited. *Arm Architecture Reference Manual for A-profile architecture*. Arm Limited, November 2024a. URL `https://developer.arm.com/documentation/ddi0487/latest`.

[3] Arm Limited. *Arm Memory System Resource Partitioning and Monitoring (MPAM) Architecture Specification*. Arm Limited, November 2024b. URL `https://developer.arm.com/documentation/ihi0099/latest`.

[4] Daniel J. Bernstein. Cache-timing attacks on AES, April 2005. URL `https://cr.yp.to/papers.html#cachetiming`.

[5] Benjamin A. Braun, Suman Jana, and Dan Boneh. Robust and efficient elimination of cache and timing side channels. *arXiv preprint arXiv:1506.00189*, May 2015. URL `http://arxiv.org/pdf/1506.00189v1.pdf`.

[6] Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. *IACR Cryptology ePrint Archive*, 2006:52, February 2006. URL `https://eprint.iacr.org/2006/052.pdf`.

[7] Scott Buckley, Robert Sison, Nils Wistoff, Curtis Millar, Toby Murray, Gerwin Klein, and Gernot Heiser. Proving the absence of microarchitectural timing channels. *arXiv preprint arXiv:2310.17046*, October 2023. URL `https://trustworthy.systems/publications/papers/Buckley_SWMMKH_23.pdf`.

[8] Tom Chothia, Yusuke Kawamoto, and Chris Novakovic. A tool for estimating information leakage. In *International Conference on Computer Aided Verification*, pages 690–695, Saint Petersburg, RU, July 2013. ACM.

[9] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. The last mile: An empirical study of some timing channels on seL4. In *ACM Conference on Computer and Communications Security*, pages 570–581, Scottsdale, AZ, USA, November 2014. ACM.

[10] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9:143–155, 1966. doi: 10.1145/365230.365252.

[11] Digilent, Inc. *Genesys 2 Reference Manual*. Digilent, Inc. URL `https://digilent.com/reference/programmable-logic/genesys-2/start`. Visited 2025-24-11.

[12] Qian Ge. *Principled Elimination of Microarchitectural Timing Channels through Operating-System Enforced Time Protection*. PhD thesis, UNSW, Sydney, Australia, October 2019. URL `https://trustworthy.systems/publications/papers/Ge%3Aphd.pdf`.

[13] Qian Ge and Curtis Millar. seL4 Project channel-bench. `https://github.com/SEL4PROJ/channel-bench`, 2019. Visited 2025-24-11.

[14] Qian Ge, Yuval Yarom, Frank Li, and Gernot Heiser. Your processor leaks information — and there's nothing you can do about it. *arXiv preprint arXiv:1612.04474*, September 2017. URL `https://trustworthy.systems/publications/papers/Ge_YLH_17.pdf`.

[15] Qian Ge, Yuval Yarom, and Gernot Heiser. No security without time protection: We need a new hardware-software contract. In *Asia-Pacific Workshop on Systems (APSys)*, Korea, August 2018. ACM SIGOPS. doi: https://doi.org/10.1145/3265723.3265724.

[16] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time protection: the missing OS abstraction. In *EuroSys Conference*, Dresden, Germany, March 2019. ACM. URL `https://trustworthy.systems/publications/csiro_full_text/Ge_YCH_19.pdf`.

[17] Gernot Heiser. Hardware Considerations: What Every OS Designer Must Know. `https://cgi.cse.unsw.edu.au/~cs9242/24/lectures/02a-hw.pdf`, September 2024.

[18] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A–L*, December 2024. URL `https://cdrdv2-public.intel.com/843847/253666-sdm-vol-2a-dec-24.pdf`.

[19] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, October 2009. ACM.

[20] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, February 2014. doi: 10.1145/2560537.

[21] 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 *IEEE Symposium on Security and Privacy*, June 2019.

[22] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *16th Annual International Cryptology Conference on Advances in Cryptology*, pages 104–113. Springer, 1996.

[23] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16:613–615, 1973. doi: 10.1145/362375.362389.

[24] 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 *USENIX Security Symposium*, May 2018.

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

[26] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *IEEE Symposium on High-Performance Computer Architecture*, pages 406–418, Barcelona, Spain, March 2016. IEEE.

[27] Lúcás Críostóir Meier, Simone Colombo, Marin Thiercelin, and Bryan Ford. Constant-Time Arithmetic for Safer Cryptography. Cryptology ePrint Archive, Paper 2021/1121, September 2021. URL `https://eprint.iacr.org/2021/1121`.

[28] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: from general purpose to a proof of information flow enforcement. In *IEEE Symposium on Security and Privacy*, pages 415–429, San Francisco, CA, May 2013. IEEE. doi: 10.1109/SP.2013.35.

[29] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Proceedings of the 2006 Crytographers' track at the RSA Conference on Topics in Cryptology*, pages 1–20, San Jose, CA, US, 2006. Springer.

[30] Alessandro Ottaviano, Thomas Benz, Paul Scheffler, and Luca Benini. Cheshire: A Lightweight, Linux-Capable RISC-V Host Platform for Domain-Specific Accelerator Plug-In. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 70(10):3777–3781, June 2023. doi: 10.1109/TCSII.2023.3289186.

[31] Daniel Page. Defending against cache-based side-channel attacks. *Information Security Technical Report*, 8(1):30–44, 2003.

[32] Colin Percival. Cache missing for fun and profit. In *BSDCan 2005*, Ottawa, CA, 2005. URL `http://css.csail.mit.edu/6.858/2014/readings/ht-cache.pdf`.

[33] PULP Platform. PULP-Platform's transaction-tagger, 2023. URL `https://github.com/pulp-platform/transaction-tagger`. Visited 2025-24-11.

[34] PULP Platform. PULP-Platform's AXI LLC flamingo, 2025a. URL `https://github.com/pulp-platform/axi_llc/tree/flamingo`. Visited 2025-24-11.

[35] PULP Platform. OpenSBI for Cheshire, 2025b. URL `https://github.com/pulp-platform/opensbi/tree/cheshire`. Fork of OpenSBI with Cheshire platform support. Visited 2025-24-11.

[36] RISC-V International. OpenSBI: RISC-V Open-Source Supervisor Binary Interface, 2025. URL `https://github.com/riscv-software-src/opensbi`. Visited 2025-24-11.

[37] RISC-V International Security Horizontal Committee. Speculation Barriers — Proposal of Work. Technical report, RISC-V International, February 2025. URL `https://riscv.atlassian.net/wiki/spaces/SXXX/pages/272629762/Speculation%2BBarriers%2B-%2BProposal%2Bof%2BWork`.

[38] Moritz Schneider, Daniele Lain, Ivan Puddu, Nicolas Dutly, and Srdjan Capkun. Breaking Bad: How Compilers Break Constant-Time Implementations, October 2024. URL `https://arxiv.org/abs/2410.13489`.

[39] seL4 Foundation. seL4 manual v13.0.0, Jul 2024. URL `https://sel4.systems/Info/Docs/seL4-manual-latest.pdf`.

[40] Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. seL4 enforces integrity. In Marko van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editor, *International Conference on Interactive Theorem Proving*, pages 325–340, Nijmegen, The Netherlands, August 2011. Springer. doi: 10.1007/978-3-642-22863-6_24.

[41] Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 1948. doi: 10.1145/584091.584093. Reprinted in SIGMOBILE Mobile Computing and Communications Review, 5(1):3–55, 2001.

[42] Bhanu C. Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in Xen. In *ACM Workshop on Cloud Computing Security*, pages 41–46, Chicago, IL, October 2011. ACM.

[43] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. SecDCP: Secure dynamic cache partitioning for efficient timing channel protection. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2016. doi: 10.1145/2897937.2898086.

[44] Nils Wistoff, Moritz Schneider, Frank Gürkaynak, Luca Benini, and Gernot Heiser. Microarchitectural timing channels and their prevention on an open-source 64-bit RISC-V core. In *Design, Automation and Test in Europe (DATE)*, virtual, February 2021. IEEE. URL `https://trustworthy.systems/publications/csiro_full_text/Wistoff_SGBH_21.pdf`.

[45] Nils Wistoff, Moritz Schneider, Frank Gürkaynak, Gernot Heiser, and Luca Benini. Systematic prevention of on-core timing channels by full temporal partitioning. *IEEE Transactions on Computers*, 72(5):1420–1430, 2023. doi: 10.1109/TC.2022.3212636.

[46] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium*, pages 719–732, San Diego, California, USA, August 2014.