Eliminating Timing Channels in Microkernels

MSc Project Description

Simon Vinding Brodersen(rzn875)

February 17, 2026

1 Description

1.1 Background and Motivation

For decades, the dominant threat model for software security has centered on memory safety vulnerabilities, including buffer overflows, format-string bugs, use after free errors, and integer conversion flaws. These vulnerabilities have historically been the industry's "low hanging fruit" for attackers, because exploitation is straightforward and consequences are severe. Modern cyberattacks still rely heavily on these vulnerabilities, which remain in systems software written in C and C++.

In response, the industry is increasingly adopting strong countermeasures, including memory-safe languages like Rust, sophisticated compiler mitigations (e.g., stack canaries), and dynamic analysis tools. As these defenses mature and memory exploits become harder to execute, attackers are shifting their attention to "higher-hanging fruit" including but not limited to: timing channels, cache-based side channels and interrupt-based leaks all part of the term "side-channel attacks".

Unlike memory corruption, which alters the program's state or control flow, timing attacks exploit the physical execution of the software. Timing channels arise when execution time depends on secret or security-relevant internal states. An attacker can observe execution time, and infer sensitive information such as private keys, memory-access patterns, scheduling decisions, or resource contention. Timing channels violate intended isolation boundaries at the process, VM, or network level, allowing seemingly independent programs to leak information indirectly. Such attacks have already been shown to be detrimental and as such it is not something that can be left untouched [1], [2].

A variety of countermeasures have been proposed to address these issues. In cryptography domain specific languages such as FaCT provide transformations of potentially timing sensitive high level code to low level constant time code [3]. This ensures that in an isolated environment the low level code will not leak information about its secrets through timing side channels. However, practical systems rarely operate in such controlled environments: operating systems provide no guarantees regarding scheduling decisions and do not flush caches or registers on context switches. Consequently, opportunities for microarchitectural channel leaks can still occur.

Previous work has been done to attempt to prevent such leaks. For instance hardware specific extensions have been implemented which provide mechanisms for clearing vulnerable micro architectural state and guaranteeing a history independent context switch latency [4]. Other work has examined the work required for time protection and have provided an implementation for the seL4 microkernel showing it is possible to protect against with little impedance on performance [5].

1.2 Problem statement

Despite their importance, timing side channels remain poorly defended in mainstream systems. There exists little to no tools to stop these attacks and many programs run as if these vulnerabilities do not exist.

The central problem this thesis addresses is: *To what extent can timing side-channels be mitigated through a combination of hardware aware software design and program transformation?*

Specifically, this project investigates whether it is viable to transform standard programs into "timing-secure" variants that are resilient to observation, and how the responsibility for this security should be distributed between the underlying hardware platform and the software running on top of it.

This will be done by surveying existing microkernel implementations - with a particular focus on their context-switching mechanisms - and identifying concrete examples of how timing channels can arise in practice. Building on these observations, the project will evaluate the feasibility of extending constant time program transformations, similar to those seen in cryptography, to general-purpose software on microkernels. The project aims to examine the hardware requirements necessary for such transformations to be efficient and what division of responsibility between hardware and software is required to enable practical timing-secure execution in microkernel based systems.

1.3 Project-Specific Learning Objectives

- Explain and understand the theoretical and practical mechanisms of side-channel leaks across hardware, OS and program layers.
- Survey existing side-channel mitigation techniques including but not limited to: hardware mechanisms and OS-level strategies.
- Design and implement a proof-of-concept program transformer to explore the practical viability of program transformation as a defense strategy.
- Evaluate the security and performance trade-offs of the implementation. In particular on an established microkernel.
- Gather the findings into design recommendations for future operating systems, hardware abstractions or tool chains, showing how side channel-secure execution could be supported in the future.

Contents

2 Introduction

Side-channels occur when a program is able to gather information through channels not meant for communication. One such example is timing channels, which occur when a programs execution time varies based on information throughout the program execution. This way timing channels bypass the operating system's (OS's) security enforcement leaking information through the timing of observable events. Prominent examples of such channels includes side channels in cryptographic libraries [6], [7]. Further, attacks showed that it is possible to do across VM's running in a cloud computing setting [8]. Not only that, side-effects from out-of-order executions on modern processors gave way for the meltdown attacks to read arbitrary kernel-memory locations including personal data and passwords [2]. Mitigations against the Meldown attacks, where kernel memory is hidden from user space are moving forward [9] Mitigations against these attacks have been proposed. Notably in cryptography a push towards more constant-time libraries gave way to implementations such as FaCT [3], which aims to guarantee constant-time execution with respect to information marked secret. However, while this does produce seemingly constant-time program, CPU's speculative execution still leaves doors open for timing attacks such as the ones seen in [1].

The attack front is clear, but the solution is not as simple. While some proposed solutions focus on the user side implementing secure programs, this creates a huge burden on the programmer to implement secure code. Instead implementations of secure microkernels [5], [10] attempt to guarantee security through the OS. The operating system would then be in charge of partitioning domains in regards to the microarchitectural state, such that no channel between domains is possible. This is already done for memory, and examples of separation kernels on the memory level have been formally proven [11], similar work attempts to provide same rigorous proof for the absence of timing channels [12]. However, currently no formal proof for multicore systems exists, and the proof also relies on the secure system to track the touched addresses.

3 Background

3.1 Side-channels

Side-channels are unintended information channels, that leak information about the process. These channels infer secret data often through observable execution time of the process. The general case of such a channel can be seen in the snippet below:

```
1: if secret then
2:     Some long computation
3: else
4:     Do nothing
5: end if
```

This algorithm is a naive example, where we simply branch on some secret, and the execution time of the program. If for example this algorithm was part of some library, then an attacker could infer information about the secret by simply measuring the execution time of the call to the algorithm. This example is quite naive, and is one of the side channels constant time programming can stop, such that we might do an equally long computation in both of the branches of the if statement. However, this still comes with some precautions, because even though the original program might seem sensibly constant time, some compilers might try to optimise the code, which could introduce new side-channel attacks [13].

Further, more complex examples of side channels includes those specified within [1], where speculative execution can be used to leak information about an otherwise safe algorithm. An example from the paper is the following:

```
1: if x < array1_size then
2:     y = array2[array1[x] * 4096]
3: end if
```

This function could again be part of some library of system call, which receives from the user an unsigned integer x. The process, which runs this library code or system call has access to both array1 and array2. It does a bounds check on x, to make sure we have no out of bounds access, which could trigger an exception or access sensitive memory. If this algorithm was called multiple times, with correct input where $x < array1\_size$, then we might train the branch predictor on modern CPUS such that it would speculatively execute the "safe" path, which could bring in the location read from `array2` into the cache, and the attacker could then use

attacks such as Flush+Reload or Prime+Probe to reveal which part of array2 was loaded into the cache. With this, it can gather the index of the loaded item, and can figure out what was read from array1[x], which could be some arbitrary secret that was obtained by the out of bounds indexing. A more detailed explanation along with a proof-of-concept is provided in [1], but this small example shows how seemingly safe code, might still leak secret information.

A multitude of possible attacks making use of these side-channels are presented in [14], [15].

3.2 Covert channels

While side-channels are passive and exploit unintended leakage, covert channels involve two processes working together. The usually revolves around a sender (Trojan) and a receiver (spy), where the sender has access to high security data but is restricted from communicating directly to the receiver. To bypass this, the sender makes use of some shared resource to signal information to the receiver.

Generally two main categories of covert channels exists; **Storage channels:** which modify some stored object, e.g. a file or some metadata; **Timing channels:** which change the time of different observable events for the receiver. For this thesis storage channels are not referenced, as kernels such as sel4 have already proven the absence of them [11].

Covert timing channels work through the same mechanisms as the aforementioned side channel, but pose as a worst case, where the sender can make use of any and all leakage methods to convey the information. This way, we are not limited to the visible gadgets that might exist in a vulnerable program, but the information might also be conveyed through repeated use of non-constant time system calls, the receiver could then measure the cache misses on the cache sets which is know to be touched by the system calls such as done in the evaluation of [5].

3.3 Isolation

Throughout many timing channel mitigations isolation is a common factor. There exists two types of isolation, namely spatial isolation and temporal isolation. Spatial isolation partitions software into distinct memory spaces, which prevents separate domains from interfering with each other. This separation is most commonly used for DRAM memory, where the OS kernel rely on hardware like the memory management units (MMUs) or memory protection units (MPUs). The MMU translates virtual memory addresses into physical addresses in main memory. It also generally provides

memory protection by implementing access control rules and regulations stopping illegal usage of particular memory locations [16]. Another notable examples of spatial isolation is cache coloring, which I will come back to later.

The latter isolation type, namely temporal isolation, must ensure that the service received from shared resources by the software in one domain cannot be affected by the software in another domain [17]. An example, which would require such isolation is any shared part of the OS between different domains. Calling shared OS code should not have any measurable timing affect for another domain calling the same OS code.

To achieve temporal isolation one must clean any microarchitectural state affect by shared calls. This is where an instruction such as `fence.t` [4] would clear this state. That is, when spatial isolation is not possible, then one must implement temporal isolation by flushing any dirty state before switching domains to ensure complete isolation.

3.4 Time protection

3.5 Cache coloring

4 Related work

4.1 The missing OS Abstraction: sel4

In the paper Time Protection: The missing OS Abstraction [5] a customized version of the sel4 microkernel is presented, which aims to protect against all possible timing channels via. the OS.

4.1.1 Threat model

In the paper they present the threat model as being a victim program, where a Trojan within the program uses any means necessary to attempt to leak information through timing channels. There are no restrictions on the Trojan outside it simply running in a separate domain than that of the receiver/spy program.

Specifically they differentiate between two types of channels, namely the **covert** and **side** channel. The covert channel requires the collusion between the sender and the receiver and represents the worst-case scenario, where the sender intentionally attempts to transmit information. The side channel is instead when the sender leaks information to the receiver unwittingly. That is, the sender leaks information via

normal program execution. By protecting against the covert channel it immediately follows, that the side channel is protected.

4.1.2 Channels

Two main types of channels are referenced throughout the paper. The on-core state, including the L1, L2 caches, Translation Lookaside Buffers and more, which are unique to each core on the CPU and the shared state which includes higher level caches that are shared between different cores. To protect against all, they provide the following five core requirements:

- *Flush on-core state:* When time-sharing a core, all on-core microarchitectural state must be flushed on domain switch, unless the hardware supports specific partitioning of this state.
- *Partition the OS:* Each security domain must have a private copy of OS text, stack and global data. That is, all the dynamic OS kernel memory must have the same separation as userland execution.
- *Deterministic data sharing:* Access to any remaining shared OS data must be sufficiently deterministic to avoid information leakage.
- *Flush deterministically:* State flushing must be padded to its worst case latency.
- *Partition interrupts:* When sharing a core, the OS must disable or partition any interrupts other than the preemption timer. (This is only a concern intra-core).

4.1.3 Cloning the OS

In the paper they introduce a method for cloning the OS. A userland program, which has the capability of cloning gives up a subset of it's own allocated memory for the clone call, and the kernel then creates a kernel image copy in the user-supplied memory. This memory now gets marked as kernel data, meaning that the userland can not change it, but that any syscalls will run through this kernel copy.

This works via the security domains of sel4. The initial kernel creates a master Kernel_Image which represents the present kernel and includes the clone right. It hands this capability along with the size of the image to the initial user process, which then partitions the memory into security domains, where it clones the OS into each of these new security domains.

This cloning consists of three steps:

1. The user process retypes some Untyped memory into an uninitialized Kernel_Image and Kernel_Memory of sufficient size.

2. it allocates address space identifier to the uninitialised Kernel_Image.

3. it invokes the Kernel_Clone on the new Kernel_Image.

4.1.4 Cache coloring

To protect the caches that are too large to be flushed, cache coloring is implemented. This means, that separate security domains are unable to access the same parts of caches, such that they have no influence on each other. This reduces the total amount of available cache space for each domain, and it means that the process which creates each security domain has to take this factor into account, such that the distribution of the cache space is sufficient for all domains.

4.2 S3K

S3k provides an implementation for a real time operating system, which aims to provide a capability-based multicore partitioning kernel for embedded RISC-V systems. S3K provides spatial and temporal isolation[1], time protection and dynamic resource reconfiguration. Further, S3K's scheduler guarantees deterministic process dispatch, free from microarchitectural interference. [10]

It does via. 3 main attributes

- Capability model;
- Time-driven capability based scheduler with systematic use of `fence.t` with worst time padding, and constant time kernel operations;
- Predictable constant time system calls.

4.2.1 Implementation

The implementation provides a minimal trusted computing base (TCB) with strong isolation and security guarantees. The basic execution unit for S3K is a *process*, a self contained entity with resources defined by its capabilities. These processes are designed to host bare-metal real-time operating systems or applications, with

[1]Spatial means separation of memory, where as temporal indicates that the failure of one component should not affect the schedulability of another.

the kernel acting as a bare-metal hypervisor. The number of processes is statically defined at compile time. Processes use *monitor capabilities* and capability revocation to manage other processes, reclaim resourceses, and repurpose existing processes for new tasks. [10]

Capabilities: Every S3K capability operation has a bounded worst-case execution time (WCET) and is free of timing side-channels. Capabilities are implemented in software, and are stored in a capability derivation tree (CDT), which are stored in memory in a statically allocated capability table (ctable).

Scheduling: S3K's scheduler is a partition scheduler, which divides time into *major frames*, which are further partitioned into *minor frames*. S3K treats minor frames as first-class capabilities, which enables dynamic reconfiguration of computing resources while maintaining process isolation. [10]

A major frame is divided into a fixed number of time slots, each with equal duration. A single time slot represents the smallest scheduling unit. A time slice capability grants control over a range of time slots on a hardware thread(hart). The process that owns the time slice capability receives a minor frame, which is less than or equal to the major frame. Each time slice capability has an *enable* field, which specifies if the minor frame is active. If set, the owner of the time slice is scheduled during the corresponding minor frame; otherwise the minor frame represents idle time.

The kernel is in principle non-preemptable, it uses preemption points, to ensure the WCET is bounded by constant. The kernel checks for preemption at the start of every system call, and aborts the system call if preemption is required.

By the definition of domains in S3K processes from different domains are never scheduled in parallel. Thus, during a domain switch, all running processes of the current domain are preempted, invoke the scheduler, and are subsequently delayed by the temporal fence.

4.2.2 Time protection

S3K uses the temporal fence instruction at context switches, which flushes the core-local microarchitectural state.

S3K is designed to reside entirely within the small scratchpad memory (SPM) backed by the core-local L1 cache.This enables the kernel's microarchitectural footprint to be efficiently flushed and avoids reliance on larger caches.

S3K only protects a process's in-kernel state from side-channels. It does not protect a process's user-level state. Each process is responsible for protecting their own user-level execution from side-channels in higher level caches. This is different than what is provided within [5], where cache-coloring guarantees that processes in different domains do not access the same parts of the cache, and thus prevents side channels even in larger cases.

All system calls are designed to be non-interfering and constant-time with respect to the invoking process's authorized observations; that is, all control flow decisions and memory accesses depend only on data the process is permitted to observe.

4.2.3 Evaluation

They evaluate the implementation by looking at a variety of different elements, most significant for ours is: scheduling jitter and side-channels. Initially they find that the Inter Process Communication (IPC) operations have the highest recorded non-preemptable cost of 1,195 cpu cycles, together with an estimated 16,000 cycles for data cache write-back for temporal fence, they set the CSPAD to 18,000 cycles for the fence.t instruction.

Scheduling Jitter: A measurer process, shceduled once per major frame, records the *mcycle* performance counter value upon dispatch. To simulate interference they introduce interfering processes that execute random system calls and pollute the cache. They find, that when the measurer exclusively occupies the SPM, there is no impact from the other processes on the measurements. However, when the measurer occupies the dram, then there is noticeable jitter, which they note is in some cases due to the small part of code in the measurer that records the dispatch time and not the scheduler itself. When more than one process is scheduled at a time, then there is jitter in the scheduler as well.

Side channels: They implement a Trojan process and a spy process. The Trojans signal a "0" by remaining idle and a "1" by behaving like the interferes from before. The spy then measures jitter, system call execution times, and cache access times to infer the Trojans' signal.

They find that when the measurer and Trojan both reside in the dram, then there is a near perfect information flow, whereas when both reside in the SPM, there is no measurable amount of information flow.

References

[1] P. Kocher, J. Horn, A. Fogh, *et al.*, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[2] M. Lipp, M. Schwarz, D. Gruss, *et al.*, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[3] S. Cauligi, G. Soeller, B. Johannesmeyer, *et al.*, "FaCT: A DSL for timing-sensitive computation," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019, New York, NY, USA: Association for Computing Machinery, Jun. 2019, pp. 174–189, ISBN: 978-1-4503-6712-7. DOI: 10.1145/3314221.3314605. (visited on 02/03/2026).

[4] N. Wistoff, M. Schneider, F. K. Gürkaynak, G. Heiser, and L. Benini, "Systematic prevention of on-core timing channels by full temporal partitioning," *IEEE Transactions on Computers*, vol. 72, no. 5, pp. 1420–1430, May 2023, ISSN: 2326-3814. DOI: 10.1109/tc.2022.3212636.

[5] Q. Ge, Y. Yarom, T. Chothia, and G. Heiser, "Time Protection: The Missing OS Abstraction," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19, New York, NY, USA: Association for Computing Machinery, Mar. 2019, pp. 1–17, ISBN: 978-1-4503-6281-8. DOI: 10.1145/3302424.3303976. (visited on 02/03/2026).

[6] "A Practical Implementation of the Timing Attack," ResearchGate. (), [Online]. Available: https://www.researchgate.net/publication/2357530_A_Practical_Implementation_of_the_Timing_Attack (visited on 02/17/2026).

[7] O. Aciiçmez, W. Schindler, and Ç. K. Koç, "Improving Brumley and Boneh timing attack on unprotected SSL implementations," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05, New York, NY, USA: Association for Computing Machinery, Nov. 2005, pp. 139–146, ISBN: 978-1-59593-226-6. DOI: 10.1145/1102120.1102140. (visited on 02/17/2026).

[8] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM Conference on Computer and Communications*

Security, Chicago Illinois USA: ACM, Nov. 2009, pp. 199–212, ISBN: 978-1-60558-894-0. DOI: 10.1145/1653662.1653687. (visited on 02/04/2026).

[9] J. Corbet, "KAISER: Hiding the kernel from user space," *LWN.net*, Nov. 2017. (visited on 02/17/2026).

[10] H. Karlsson and R. Guanciale, "Partitioning Kernel With Capability Controlled Temporal and Spatial Partitioning," in *2025 IEEE Real-Time Systems Symposium (RTSS)*, Dec. 2025, pp. 68–81. DOI: 10.1109/RTSS66672.2025.00015. (visited on 02/04/2026).

[11] T. Murray, D. Matichuk, M. Brassil, *et al.*, "seL4: From General Purpose to a Proof of Information Flow Enforcement," in *2013 IEEE Symposium on Security and Privacy*, Berkeley, CA: IEEE, May 2013, pp. 415–429, ISBN: 978-0-7695-4977-4 978-1-4673-6166-8. DOI: 10.1109/SP.2013.35. (visited on 02/17/2026).

[12] S. Buckley, R. Sison, N. Wistoff, *et al.*, *Proving the Absence of Microarchitectural Timing Channels*, Oct. 2023. DOI: 10.48550/arXiv.2310.17046. arXiv: 2310.17046 [cs]. (visited on 02/10/2026).

[13] M. Schneider, D. Lain, I. Puddu, N. Dutly, and S. Capkun, "Breaking Bad: How Compilers Break Constant-Time Implementations," in *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security*, Aug. 2025, pp. 1690–1706. DOI: 10.1145/3708821.3733909. arXiv: 2410.13489 [cs]. (visited on 02/17/2026).

[14] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: The Case of AES," in *Topics in Cryptology – CT-RSA 2006*, D. Hutchison, T. Kanade, J. Kittler, *et al.*, Eds., vol. 3860, Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–20, ISBN: 978-3-540-31033-4 978-3-540-32648-9. DOI: 10.1007/11605805_1. (visited on 02/10/2026).

[15] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 605–622. DOI: 10.1109/SP.2015.43. (visited on 02/04/2026).

[16] "What is Memory Management Unit(MMU)?" GeeksforGeeks. (), [Online]. Available: https://www.geeksforgeeks.org/computer-organization-architecture/what-is-memory-management-unit/ (visited on 02/17/2026).

[17] J. Rushby, "Avionics Architectures: Mechanisms, and Assurance," 1999.