Bridge: A Leak-Free Hardware-Software Architecture for Parallel Embedded Systems

Gongqi Huang Leon Schuermann Amit Levy

Princeton University

Nov 03, 2024

Embedded Devices are Ubiquitous

- Personal medical devices
- Industrial automation
- Hardware root of trust
- …

Embedded Devices are Ubiquitous

- Personal medical devices
- Industrial automation
- Hardware root of trust
- …

As they handle increasingly sensitive data, developers must uphold both *functional safety* and ***security***

Changed Environment, New System Architecture

- Split application into multiple *security domains*
  - E.g., Medical device: sensor + wireless
- In practice, well-established techniques are used
  - Process abstraction
  - Virtual memory
  - Language-level isolation

Changed Environment, New System Architecture

- Split application into multiple *security domains*
  - E.g., Medical device: sensor + wireless
- In practice, well-established techniques are used
  - Process abstraction
  - Virtual memory
  - Language-level isolation

Such approaches are good at enforcing functional safety but less effective in maintaining security due to ***timing channels***

Timing Channels Subverts Security

- Exploits *timing variations* due to *shared state*
  - Fundamentally breaks the security guarantee
  - Mitigations are whack-a-mole
- More and more timing channels are disclosed!

Timing Chan rity

- Exploits *timing* *state*
  - Fundamental arantee
  - Mitigations a
- More and more losed!

Meltdown: Reading Kernel Memory from User Space

Moritz Lipp[1], Michael Schwarz[1], Daniel Gruss[1], Thomas Prescher[2],
Werner Haas[2], Anders Fogh[3], Jann Horn[4], Stefan Mangard[1],
Paul Kocher[5], Daniel Genkin[6,9], Yuval Yarom[7], Mike Hamburg[8]
[1]*Graz University of Technology,* [2]*Cyberus Technology GmbH,*
[3]*G-Data Advanced Analytics,* [4]*Google Project Zero,*
[5]*Independent (www.paulkocher.com),* [6]*University of Michigan,*
[7]*University of Adelaide & Data61,* [8]*Rambus, Cryptography Research Division*

Abstract

The security of computer systems fundamentally relies on memory isolation, e.g., kernel address ranges are marked as non-accessible and are protected from user access. In this paper, we present Meltdown. Meltdown exploits side effects of out-of-order execution on modern processors to read arbitrary kernel-memory locations including personal data and passwords. Out-of-order execution is an indispensable performance feature and present in a wide range of modern processors. The attack is independent of the operating system, and it does not rely on any software vulnerabilities. Meltdown breaks all security guarantees provided by address space isolation as well as paravirtualized environments and, thus, every security mechanism building upon this foundation. On affected systems, Meltdown enables an adversary to read memory of other processes or virtual machines in the cloud without any permissions or privileges, affecting millions of customers and virtually every user of a personal computer. We show that the KAISER defense mechanism for KASLR has the important (but inadvertent) side effect of impeding Meltdown. We stress that KAISER must be deployed immediately to prevent large-scale exploitation of this severe information leakage.

1 Introduction

A central security feature of today's operating systems is memory isolation. Operating systems ensure that user programs cannot access each other's memory or kernel memory. This isolation is a cornerstone of our computing environments and allows running multiple applications at the same time on personal devices or executing processes of multiple users on a single machine in the cloud.

On modern processors, the isolation between the kernel and user processes is typically realized by a supervisor bit of the processor that defines whether a memory page of the kernel can be accessed or not. The basic idea is that this bit can only be set when entering kernel code and it is cleared when switching to user processes. This hardware feature allows operating systems to map the kernel into the address space of every process and to have very efficient transitions from the user process to the kernel, e.g., for interrupt handling. Consequently, in practice, there is no change of the memory mapping when switching from a user process to the kernel.

In this work, we present Meltdown[10]. Meltdown is a novel attack that allows overcoming memory isolation completely by providing a simple way for any user process to read the entire kernel memory of the machine it executes on, including all physical memory mapped in the kernel region. Meltdown does not exploit any software vulnerability, *i.e.*, it works on all major operating systems. Instead, Meltdown exploits side-channel information available on most modern processors, e.g., modern Intel microarchitectures since 2010 and potentially on other CPUs of other vendors.

While side-channel attacks typically require very specific knowledge about the target application and are tailored to only leak information about its secrets, Meltdown allows an adversary who can run code on the vulnerable processor to obtain a dump of the entire kernel address space, including any mapped physical memory. The root cause of the simplicity and strength of Meltdown are side effects caused by *out-of-order execution*.

Out-of-order execution is an important performance feature of today's processors in order to overcome latencies of busy execution units, e.g., a memory fetch unit needs to wait for data arrival from memory. Instead of stalling the execution, modern processors run operations

[9]Work was partially done while the author was affiliated to University of Pennsylvania and University of Maryland.

[10]Using the practice of responsible disclosure, disjoint groups of authors of this paper provided preliminary versions of our results to partially overlapping groups of CPU vendors and other affected companies. In coordination with industry, the authors participated in an embargo of the results. Meltdown is documented under CVE-2017-5754.

Timing Chan ty

- Exploits *timing* ate
  - Fundamental antee
  - Mitigations a
- More and more sed!

Spectre Attacks: Exploiting Speculative Execution

Paul Kocher[1], Jann Horn[2], Anders Fogh[3], Daniel Genkin[4],
Daniel Gruss[5], Werner Haas[6], Mike Hamburg[7], Moritz Lipp[5],
Stefan Mangard[5], Thomas Prescher[6], Michael Schwarz[5], Yuval Yarom[8]
[1] Independent (www.paulkocher.com), [2] Google Project Zero,
[3] G DATA Advanced Analytics, [4] University of Pennsylvania and University of Maryland,
[5] Graz University of Technology, [6] Cyberus Technology,
[7] Rambus, Cryptography Research Division, [8] University of Adelaide and Data61

***Abstract*—Modern processors use branch prediction and speculative execution to maximize performance. For example, if the destination of a branch depends on a memory value that is in the process of being read, CPUs will try to guess the destination and attempt to execute ahead. When the memory value finally arrives, the CPU either discards or commits the speculative computation. Speculative logic is unfaithful in how it executes, can access the victim's memory and registers, and can perform operations with measurable side effects.**

Spectre attacks involve inducing a victim to speculatively perform operations that would not occur during correct program execution and which leak the victim's confidential information via a side channel to the adversary. This paper describes practical attacks that combine methodology from side channel attacks, fault attacks, and return-oriented programming that can read arbitrary memory from the victim's process. More broadly, the paper shows that speculative execution implementations violate the security assumptions underpinning numerous software security mechanisms, including operating system process separation, containerization, just-in-time (JIT) compilation, and countermeasures to cache timing and side-channel attacks. These attacks represent a serious threat to actual systems since vulnerable speculative execution capabilities are found in microprocessors from Intel, AMD, and ARM that are used in billions of devices.

While makeshift processor-specific countermeasures are possible in some cases, sound solutions will require fixes to processor designs as well as updates to instruction set architectures (ISAs) to give hardware architects and software developers a common understanding as to what computation state CPU implementations are (and are not) permitted to leak.

I. Introduction

Computations performed by physical devices often leave observable side effects beyond the computation's nominal outputs. Side-channel attacks focus on exploiting these side effects to extract otherwise-unavailable secret information. Since their introduction in the late 90's [43], many physical effects such as power consumption [41, 42], electromagnetic radiation [58], or acoustic noise [20] have been leveraged to extract cryptographic keys as well as other secrets.

Physical side-channel attacks can also be used to extract secret information from complex devices such as PCs and mobile phones [21, 22]. However, because these devices often execute code from a potentially unknown origin, they face additional threats in the form of software-based attacks, which do not require external measurement equipment. While some attacks exploit software vulnerabilities (such as buffer overflows [5] or double-free errors [12]), other software attacks leverage hardware vulnerabilities to leak sensitive information. Attacks of the latter type include microarchitectural attacks exploiting cache timing [8, 30, 48, 52, 55, 69, 74], branch prediction history [1, 2], branch target buffers [14, 44] or open DRAM rows [56]. Software-based techniques have also been used to mount fault attacks that alter physical memory [39] or internal CPU values [65].

Several microarchitectural design techniques have facilitated the increase in processor speed over the past decades. One such advancement is speculative execution, which is widely used to increase performance and involves having the CPU guess likely future execution directions and prematurely execute instructions on these paths. More specifically, consider an example where the program's control flow depends on an uncached value located in external physical memory. As this memory is much slower than the CPU, it often takes several hundred clock cycles before the value becomes known. Rather than wasting these cycles by idling, the CPU attempts to guess the direction of control flow, saves a checkpoint of its register state, and proceeds to speculatively execute the program on the guessed path. When the value eventually arrives from memory, the CPU checks the correctness of its initial guess. If the guess was wrong, the CPU discards the incorrect speculative execution by reverting the register state back to the stored checkpoint, resulting in performance comparable to idling. However, if the guess was correct, the speculative execution results are committed, yielding a significant performance gain as useful work was accomplished during the delay.

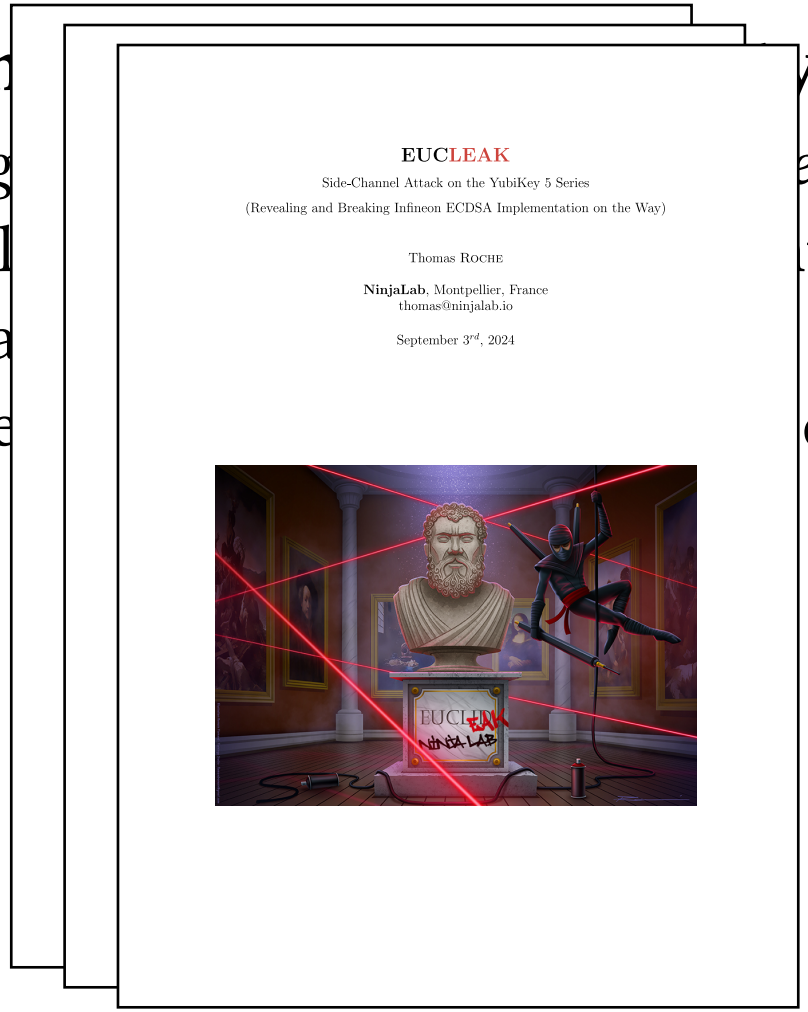
From a security perspective, speculative execution involves executing a program in possibly incorrect ways. However, because CPUs are designed to maintain functional correctness by reverting the results of incorrect speculative executions to their prior states, these errors were previously assumed to be safe.

A. Our Results

In this paper, we analyze the security implications of such incorrect speculative execution. We present a class of microarchitectural attacks which we call *Spectre attacks*. At a high level, Spectre attacks trick the processor into speculatively executing instruction sequences that should not have been executed under correct program execution. As the effects of these instructions on the nominal CPU state are eventually

Timing Chan y

- Exploits *timing* e
  - Fundamental tee
  - Mitigations a
- More and more d!



EUCLEAK

Side-Channel Attack on the YubiKey 5 Series

(Revealing and Breaking Infineon ECDSA Implementation on the Way)

Thomas Roche

NinjaLab, Montpellier, France
thomas@ninjalab.io

September 3rd, 2024

Timing Chan

- Exploits *timing*
  - Fundamental
  - Mitigations a
- More and more

ARTIFACT EVALUATED USENIX SECURITY '24 AVAILABLE FUNCTIONAL REPRODUCED

GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers

Boru Chen *UIUC* · Yingchen Wang *UT Austin* · Pradyumna Shome *Georgia Tech* · Christopher W. Fletcher *UC Berkeley*

David Kohlbrenner *University of Washington* · Riccardo Paccagnella *Carnegie Mellon University* · Daniel Genkin *Georgia Tech*

Abstract

Microarchitectural side-channel attacks have shaken the foundations of modern processor design. The cornerstone defense against these attacks has been to ensure that security-critical programs do not use secret-dependent data as addresses. Put simply: do not pass secrets as addresses to, e.g., data memory instructions. Yet, the discovery of data memory-dependent prefetchers (DMPs)—which turn program data into addresses directly from within the memory system—calls into question whether this approach will continue to remain secure.

This paper shows that the security threat from DMPs is significantly worse than previously thought and demonstrates the first end-to-end attacks on security-critical software using the Apple m-series DMP. Undergirding our attacks is a new understanding of how DMPs behave which shows, among other things, that the Apple DMP will activate on behalf of any victim program and attempt to "leak" *any* cached data that resembles a pointer. From this understanding, we design a new type of chosen-input attack that uses the DMP to perform end-to-end key extraction on popular constant-time implementations of classical (OpenSSL Diffie-Hellman Key Exchange, Go RSA decryption) and post-quantum cryptography (CRYSTALS-Kyber and CRYSTALS-Dilithium).

1 Introduction

For over a decade, modern processors have faced a myriad of microarchitectural side-channel attacks, e.g., through the caches [63, 91], TLBs [42, 78, 82], branch predictors [6, 35], on-chip interconnects [31, 64, 85], memory management units [43, 50, 81], speculative execution [51, 54], voltage-frequency scaling [77, 87, 88] and more.

The most prominent class of these attacks occurs when the program's memory access pattern becomes dependent on secret data. For example, cache and TLB side-channel attacks arise when the program's data memory access pattern becomes secret dependent. Other attacks, e.g., those monitoring on-chip interconnects, can be viewed similarly with respect to the program's instruction memory access pattern. This has led to the development of a wide range of defenses—including the ubiquitous constant-time programming model [52, 61], information flow-based tracking [41, 79, 94], and more—all of which seek to prevent secret data from being used as an address to memory/control-flow instructions.

Recently, however, Augury [83] demonstrated that Apple m-series CPUs undermine this programming model by introducing a Data Memory-dependent Prefetcher (DMP) that will attempt to prefetch addresses found in the contents of program memory. Thus, in theory, Apple's DMP leaks memory contents via cache side channels, even if that memory is never passed as an address to a memory/control-flow instruction.

Despite the Apple DMP's novel leakage capabilities, its restrictive behavior has prevented it from being used in attacks. In particular, Augury reported that the DMP only activates in the presence of a rather idiosyncratic program memory access pattern (where the program streams through an array of pointers and architecturally dereferences those pointers). This access pattern is not typically found in security critical software such as side-channel hardened constant-time code—hence making that code impervious to leakage through the DMP. With the DMP's full security implications unclear, in this paper we address the following questions:

Do DMPs create a critical security threat to high-value software? Can attacks use DMPs to bypass side-channel countermeasures such as constant-time programming?

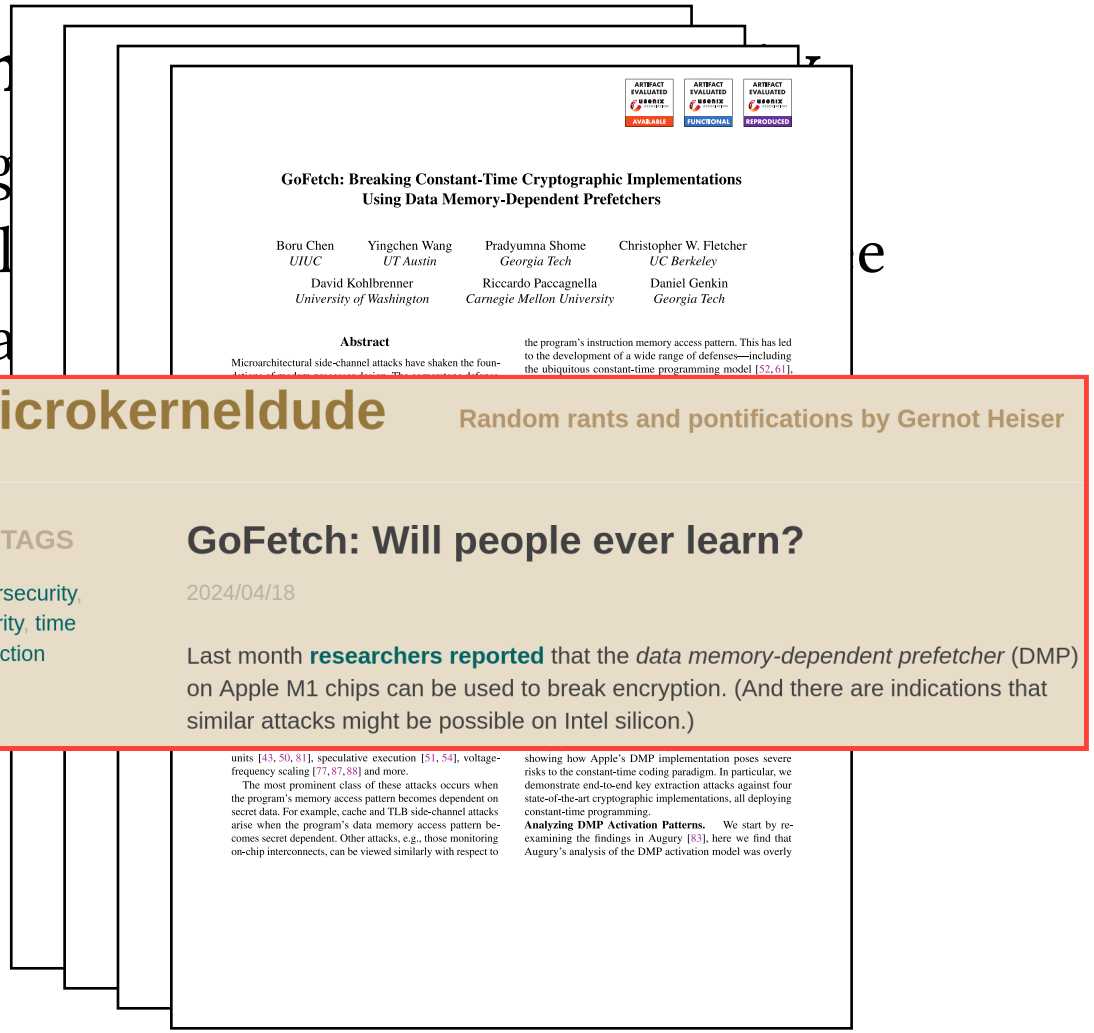
1.1 Our Contribution

This paper answers the above questions in the affirmative, showing how Apple's DMP implementation poses severe risks to the constant-time coding paradigm. In particular, we demonstrate end-to-end key extraction attacks against four state-of-the-art cryptographic implementations, all deploying constant-time programming.

Analyzing DMP Activation Patterns. We start by re-examining the findings in Augury [83], here we find that Augury's analysis of the DMP activation model was overly

Timing Chan

- Exploits *timing*
  - Fundamental
  - Mitigations a
- More and mo



ARTIFACT EVALUATED USENIX AVAILABLE FUNCTIONAL REPRODUCED

GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers

Boru Chen *UIUC* · Yingchen Wang *UT Austin* · Pradyumna Shome *Georgia Tech* · Christopher W. Fletcher *UC Berkeley*
David Kohlbrenner *University of Washington* · Riccardo Paccagnella *Carnegie Mellon University* · Daniel Genkin *Georgia Tech*

Abstract

Microarchitectural side-channel attacks have shaken the foun-

the program's instruction memory access pattern. This has led to the development of a wide range of defenses—including the ubiquitous constant-time programming model [52, 61],

microkerneldude Random rants and pontifications by Gernot Heiser

TAGS

cybersecurity, security, time protection

GoFetch: Will people ever learn?

2024/04/18

Last month **researchers reported** that the *data memory-dependent prefetcher* (DMP) on Apple M1 chips can be used to break encryption. (And there are indications that similar attacks might be possible on Intel silicon.)

units [43, 50, 81], speculative execution [51, 54], voltage-frequency scaling [77, 87, 88] and more.

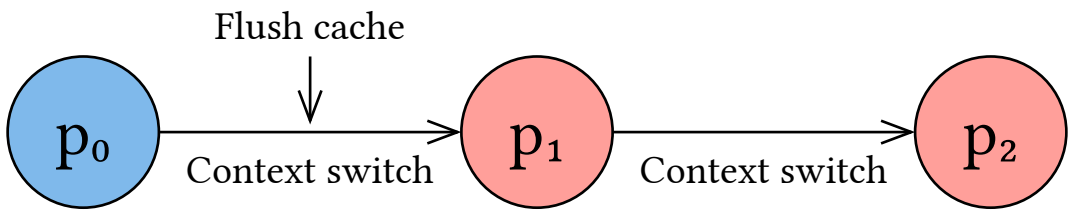
The most prominent class of these attacks occurs when the program's memory access pattern becomes dependent on secret data. For example, cache and TLB side-channel attacks arise when the program's data memory access pattern becomes secret dependent. Other attacks, e.g., those monitoring on-chip interconnects, can be viewed similarly with respect to

showing how Apple's DMP implementation poses severe risks to the constant-time coding paradigm. In particular, we demonstrate end-to-end key extraction attacks against four state-of-the-art cryptographic implementations, all deploying constant-time programming.

Analyzing DMP Activation Patterns. We start by re-examining the findings in Augury [83], here we find that Augury's analysis of the DMP activation model was overly

Timing-Safe Isolation in Time-Shared Systems

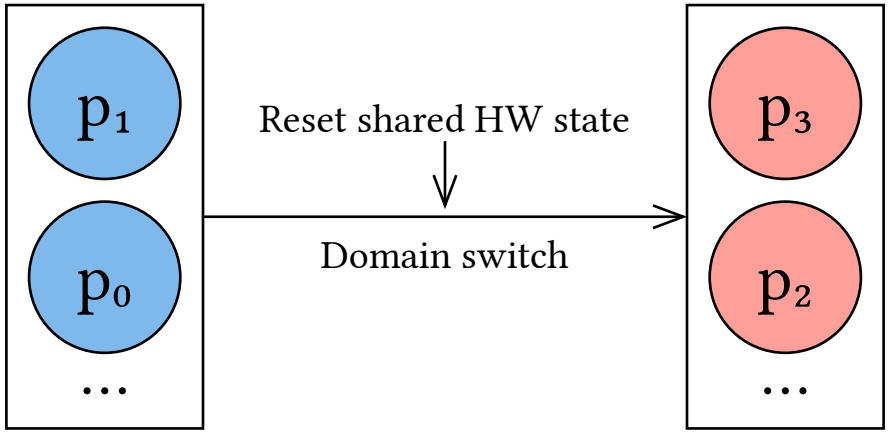
- Lattice scheduler in the VAX/VMM security kernel[1]
  - Resetting cache state when switching to another process
  - Fixed time slot for each process



[1] W.-M. Hu. 1992. Lattice scheduling and covert channels. In Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy. 52–61.

Timing-Safe Isolation in Time-Shared Systems

- Time protection in seL4 microkernel[2]
  - A set of mechanisms to maintain timing-safety across domains
    - Kernel text and data partition, deterministic data sharing, etc.



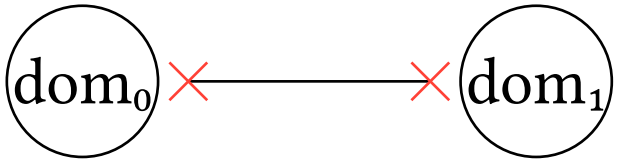
[2] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. 2019. Time Protection: The Missing OS Abstraction. EuroSys '19. Association for Computing Machinery, New York, NY, USA, Article 1, 17 pages.

Leak-Freeness and Leak-Free *Time-Shared* System

- Leak-freeness considers *timing* channels
- Leak-freeness respects *security policy*
  - Any violation of such policy is a leak, otherwise not

Leak-Freeness and Leak-Free *Time-Shared* System

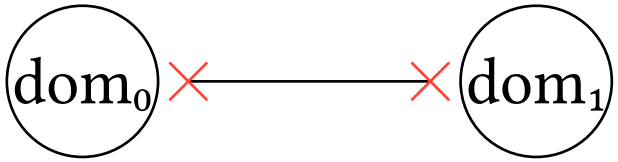
- Leak-freeness considers *timing* channels
- Leak-freeness respects *security policy*
  - Any violation of such policy is a leak, otherwise not



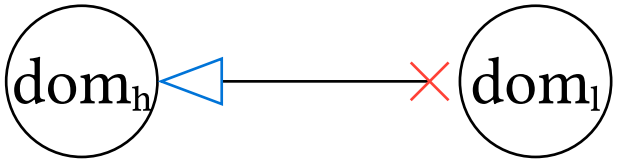
Mutually distrusted

Leak-Freeness and Leak-Free *Time-Shared* System

- Leak-freeness considers *timing* channels
- Leak-freeness respects *security policy*
  - Any violation of such policy is a leak, otherwise not



Mutually distrusted

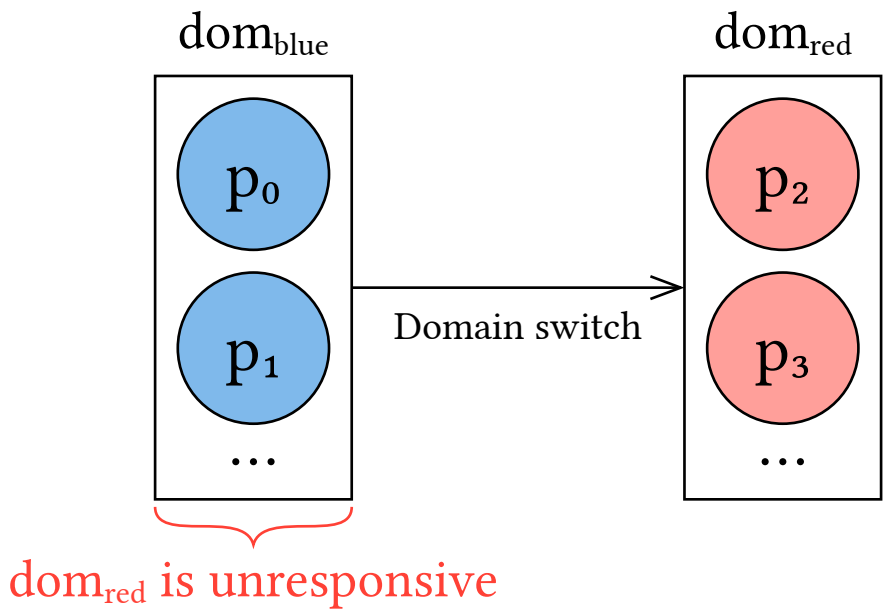


High—low relation

Leak-Freeness and Leak-Free *Time-Shared* System

- Leak-freeness considers *timing* channels
- Leak-freeness respects *security policy*
  - Any violation of such policy is a leak, otherwise not
- A time-shared system that ensures no leak across *security domains*

Existing Leak-Free *Time-Shared* Systems Compromise *Responsiveness*



BRIDGE: Exploring Leak-Free, *Parallel* Systems

- A hardware-software architecture design
  - Executing multiple domains simultaneously
    - Domains can remain responsive
  - Enabling *explicit* communication without additional leakage
    - Practical to support real applications

Naïve Leak-Free Parallel System

- *Shared-nothing* multi-core architecture
  - Each security domain runs in a *completely* independent and isolated slice of machine

Naïve Leak-Free Parallel System

- *Shared-nothing* multi-core architecture
  - Each security domain runs in a *completely* independent and isolated slice of machine

Overly restrictive — practical systems allow declassification in a ***controlled*** *manner (e.g., encryption, aggregation)*

BRIDGE Leak-Free Parallel System

- *Shared-nothing* multi-core architecture
  - Each security domain runs in a *completely* independent and isolated slice of machine

+ Extended with a single, *leak-free* communication channel between two machine slices

BRIDGE Leak-Free Parallel System

- *Shared-nothing* multi-core architecture
  - Each security domain runs in a *completely* independent and isolated slice of machine

+ Extended with a single, *leak-free* communication channel between two machine slices

Ensures leak-freeness ***by construction***

Bridge's Inter-Core Message Passing Mechanism

- Use a shared resource + a signaling mechanism
-

Bridge's Inter-Core Message Passing Mechanism

- Use a shared resource + a signaling mechanism
- carry message

Bridge's Inter-Core Message Passing Mechanism

- Use a shared resource + a signaling mechanism
- carry message deliver message
  + arbitrate parallel accesses

Bridge's Inter-Core Message Passing Mechanism

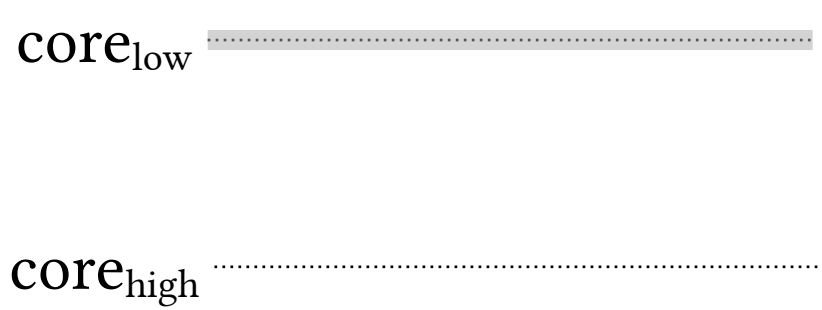
- Use a shared resource + a signaling mechanism
- A request—response communication scheme
  -

core_{low}

$\text{core}_{\text{high}}$

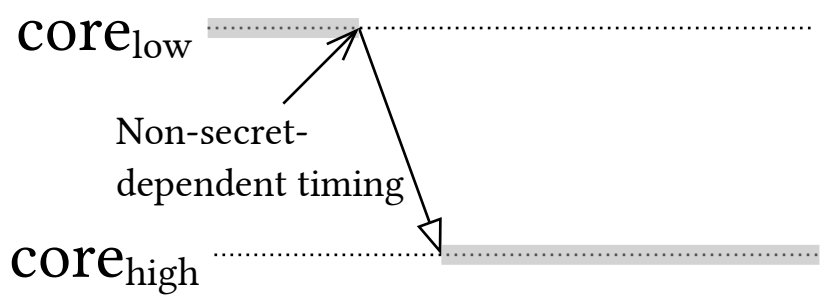
Bridge's Inter-Core Message Passing Mechanism

- Use a shared resource + a signaling mechanism
- A request—response communication scheme
  - Initially, $core_{low}$ owns the resource



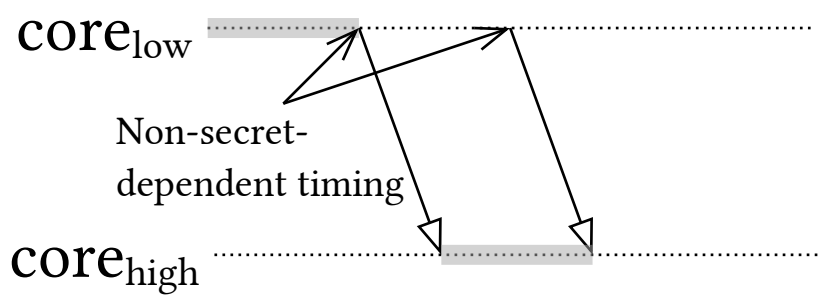
Bridge's Inter-Core Message Passing Mechanism

- Use a shared resource + a signaling mechanism
- A request—response communication scheme
  - $core_{low}$ initiates the communication



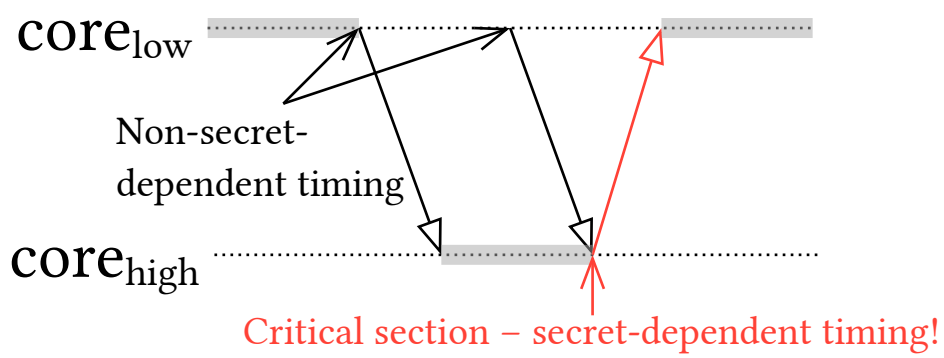
Bridge's Inter-Core Message Passing Mechanism

- Use a shared resource + a signaling mechanism
- A request—response communication scheme
  - $core_{low}$ polls the response from $core_{high}$



Bridge's Inter-Core Message Passing Mechanism

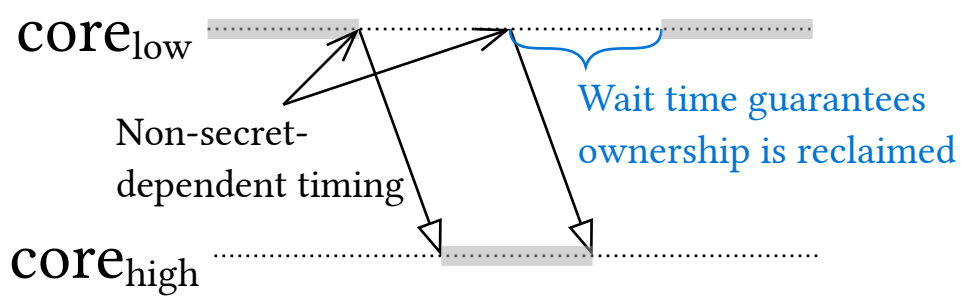
- Use a shared resource + a signaling mechanism
- A request—response communication scheme
  - Critical sections can create backchannels[3]



[3]Critical sections are unavoidable: for instance, Linux's PREEMPT_RT_FULL model does not allow reentrance in top half interrupt handlers.

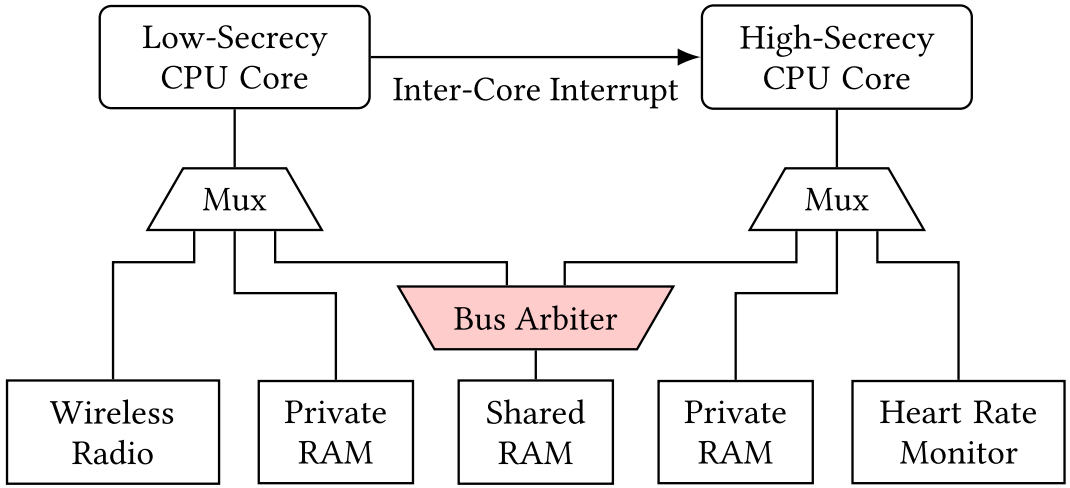
Bridge's Inter-Core Message Passing Mechanism

- Use a shared resource + a signaling mechanism
- A request—response communication scheme
  - Worst case execution time (WCET) over all critical sections



Bridge System Architecture: Hardware

- *Shared-nothing* multi-core architecture + a shared RAM block



A simplified Bridge hardware architecture

Bridge's Hardware Constraints are Practical

- Raspberry Pi RP2040 ARM Cortex-M0+ SoC
  - Two CPU cores
  - Most peripherals have a dedicated upstream port per core
  - Inter-core interrupts can take the highest priority

Bridge System Architecture: Software

- *Shared-nothing* multi-kernel architecture + a leak-free channel
  - Userspace processes can always be preempted
  - Kernel instance must handle inter-core signal immediately[4]
- No requirements regarding kernel-local policy and mechanism

[4] Or after returning from any currently entered critical section.

Current State

- Prototype
  - Extending Tock OS kernel to a multi-kernel architecture (approx. 2300 LOC)
  - Inter-core messaging mechanism (approx. 300 LOC)
- In progress
  - Cross-kernel IPC support
  - Bridge SoC with two VexRiscv CPU cores
  - WCET analysis

Current State

- Prototype
  - Extending Tock OS kernel to a multi-kernel architecture (approx. 2300 LOC)
  - Inter-core messaging mechanism (approx. 300 LOC)
- In progress
  - Cross-kernel IPC support
  - Bridge SoC with two VexRiscv CPU cores
  - WCET analysis

Thank you!