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

Gongqi Huang
Princeton University
Princeton, USA
gongqih@princeton.edu

Leon Schuermann
Princeton University
Princeton, USA
lschuermann@princeton.edu

Amit Levy
Princeton University
Princeton, USA
aalevy@princeton.edu

Abstract

Embedded and Internet of Things (IoT) devices are increasingly ubiquitous and process increasingly sensitive data. As a result, such devices must uphold security in addition to functional safety to avoid unintended information leaks. To react this change of environment, developers deploy conventional mechanisms such as memory isolation and priority scheduling to achieve aforementioned goals. While such techniques are resilient against attacks that endanger a device's functional safety, they are less effective in maintaining security as they ignore information leaks through timing channels, such as through scheduling policy and implicit microarchitectural state. Recent advances in timing-safe systems, in turn, limit themselves to time-shared systems without parallelism. This is problematic in the face of responsiveness and real-time constraints which are often found in embedded devices.

This paper explores timing-safety in the space of parallel systems. We introduce Bridge, a new system architecture featuring multiple tasks with different security concerns that can execute in parallel without leaking information due to timing interference.

CCS Concepts: • **Security and privacy → Operating systems security**; • **Computer systems organization → Embedded systems**; **Multicore architectures**.

ACM Reference Format:
Gongqi Huang, Leon Schuermann, and Amit Levy. 2024. Bridge: A Leak-Free Hardware-Software Architecture for Parallel Embedded Systems. In *Kernel Isolation, Safety and Verification (KISV '24), November 4–6, 2024, Austin, TX, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3698576.3698765

1 Introduction

Embedded and Internet of Things (IoT) devices are increasingly ubiquitous and connected. For example, connected cars stream metrics to manufacturers to schedule predictive maintenance appointments, and personal medical devices constantly monitor a user's important health parameters while informing their medical provider of any deviations. With increasing connectivity and growing sensitivity of processed data, it is clear that these devices must no longer just uphold *functional safety—security*, such as being resilient against external attacks, and *privacy*, by avoiding leakage of personal information, become equally important goals.

Developers of embedded systems react to this changed environment by strengthening their system's security through a variety of mechanisms. Apart from software fortification efforts like code auditing, fuzzing, and formal verification, many systems adopt a fundamentally new architecture that divides their responsibilities into multiple, independent and isolated *security domains*. For example, a personal health monitor may be implemented as two separate processes: one running its sensor evaluation and alerting logic, and another implementing its wireless communication subsystem. In practice, these architectures are implemented using well-established techniques of computer systems to meet security and functional safety requirements, such as using memory isolation and priority scheduling. Modern and secure embedded operating systems help enforce these policies [11, 21, 22].

While these changes increase a device's resilience against attacks that endanger their safety, they are less effective when it comes to maintaining *privacy* or *secrecy*. All of fuzzing, verification, and compartmentalization test and model explicit data flows and permissions in these systems, but do not take into account that their behavior may also leak sensitive information through *implicit* channels. This deficiency is often rooted in the implementation of these basic isolation mechanisms themselves: for instance, a memory isolation primitive does not address microarchitectural timing channels in, e.g., caches, and a system's high-level scheduling policy can reveal how much time a computation over secret data takes. Spectre and Meltdown are prominent examples of such hardware-based information leaks [9, 12].

Recent contributions focus on providing *leak-free* or *timing-safe* isolation across security domains within *time-shared* systems [5, 14, 20]. These approaches extend a system's security guarantees by reasoning about implicit information leaks in addition to explicit data flows. For instance, seL4 extends classical memory isolation (for functional safety)

KISV '24, November 4–6, 2024, Austin, TX, USA

ACM ISBN 979-8-4007-1301-9/24/11
https://doi.org/10.1145/3698576.3698765

with additional time protection mechanisms to eliminate both scheduling and microarchitectural timing channels.

While these approaches are promising for upholding secrecy in time-shared systems, they compromise on some properties important for embedded systems. For instance, such leak-free systems often employ a scheduling policy that time-shares hardware according to fixed scheduling quanta or prioritizes less-secret security domains. This can be problematic in the face of real-time requirements. And importantly, these solutions do not translate beyond time-shared hardware towards parallel systems, where contention due to concurrent hardware accesses introduces an unmitigated point of information leakage.

In this paper, we explore the space of leak-free, *parallel* systems. We design Bridge, a new system architecture for devices featuring multiple security domains that can execute simultaneously. In this architecture, security domains can remain responsive while avoiding resource starvation in the presence of external events. To avoid timing channels in asynchronous cross-domain interactions of such a parallel system, Bridge introduces a leak-free inter-core message passing protocol. Notably, Bridge does not rely on its processes to provide any security or secrecy guarantees. Instead, we introduce a Trusted Computing Base (TCB) that maintains secrecy while supporting arbitrary applications.

2 Threat Model

Bridge maintains its security even in the face of adversarial applications. In particular, Bridge ensures that, apart from a controlled declassification mechanism, any secret data maintained within a high-security domain must not be exposed to other low-security domains or external observers.

We assume that an attacker seeks to expose secret data from a high-security domain. Such an attacker is assumed to not have direct control over what data can be declassified but otherwise full control over this domain. Furthermore, an attacker may observe cycle-level timing behavior of applications executing in other low-security domains. For this, an attacker can either take over control of applications running in these low-security domains, or monitor behavior of low-security domain applications through external interactions with peripherals (like radio transmissions).

Bridge does not explicitly concern itself with orthogonal implicit channels, such as power, temperature, or electromagnetic information leaks [1, 8, 13, 19]. Instead, Bridge is complimentary to existing mitigations.

3 Design

In this section, we present the Bridge system architecture. Key to this architecture is Bridge's leak-free inter-core message passing protocol. We first reason about the implications for timing channels in a multi-core environment. Based on these findings, we construct Bridge's inter-core message passing protocol. Finally, we present a system architecture encompassing this protocol and using it for cross-domain communication.

3.1 Leak-Free Inter-Core Communication

To establish a leak-free inter-core communication channel, we first have to reason about how existing, time-shared systems can be made leak-free. In time-shared systems, hardware is shared between different tasks, potentially running within different security domains. These domains may either be *mutually distrustful*, in which case no communication between any two such domains is allowed, or they may exist in a so-called *low–high* relation: in this case, information can *flow* from a low-secrecy to a high-secrecy domain, but not the other way around [4].

In addition to explicit communication, leak-free systems also consider implicit information exchanged through timing variations. For example, when a system uses a dynamic scheduling policy that is influenced by the runtime behavior of a task, these schedule differences can then be observed by tasks in other security domains. Such an implicit channel is mitigated through one of two approaches: either the system uses a fixed schedule that does not change depending on these dynamic factors, or it uses a scheduling policy that reflects the system's security policy. Such a policy must encompass the relations of security domains: for instance, timing variations of a security domain must not be exposed to any mutually distrustful domains. Instead, timing variations of one *low* domain may only be exposed to any *higher* domains. In practice, this implies a priority schedule where low-secrecy tasks take precedence over higher-secrecy tasks.

Apart from changes to the system's high-level schedule, access to time-shared hardware can implicitly leak information about tasks' behavior across security domains. This is because hardware contains state influenced by past system behavior, and observable through future system behavior. Such state is ubiquitous throughout hardware; examples being CPU registers, caches, and peripheral state. In a time-shared hardware system, this issue can be mitigated by explicitly *flushing* all such observable state between switches of security domains in order to reset the hardware into a well-known state. Crucially, this is enabled by the fact that there is a defined point in time between the execution of two security domains—namely a context switch—where neither domain can observe such hardware state.

Unfortunately, this primitive breaks down once two security domains have concurrent access to shared hardware. For instance, a bus arbiter controlling access to a shared RAM block must maintain some state for correct behavior. When a CPU core changes security domains, this hardware state cannot be flushed unconditionally as another core may access this component in parallel. Furthermore, stalling the first CPU core because of concurrent accesses by a second core will implicitly leak timing information towards the first core, and thus establish an implicit information flow.

Naïvely, we can ensure timing-safety in a parallel system by establishing a *shared-nothing* architecture—when no shared hardware state is accessible from two independent CPU cores, then no such state can implicitly communicate information between these cores. Unfortunately, such an architecture is overly restrictive for practical systems: it can only model mutually distrustful security domains. However, practical systems will need to declassify some sensitive information in a controlled (e.g., encrypted or obfuscated) manner and provide it to lower or distrustful counterparts. With no communication channel, no controlled exposure is possible.

Instead, the Bridge system architecture takes a *shared-nothing* multi-core architecture and adds a single, leak-free communication channel between two otherwise independent and isolated cores. By ensuring that this channel is timing-safe and adheres to the system's security policy, we guarantee—by construction—that the overall system is leak-free as well. Next, we describe the construction of this protocol and map it onto hardware and software primitives.

3.2 The Bridge Inter-Core Message Passing Protocol

In Bridge, a task can declassify data towards another task, potentially running in a different security domain. Unfortunately, delivering a message from one task to another task carries not just its intended data: the message's delivery time further exposes information about the sender's secret-dependent runtime behavior to the receiver. This is problematic, as the message's contents have been explicitly declassified, but not the additional timing information, violating the system's security policy.

A naïve solution to this problem is to unconditionally deliver messages at a fixed time interval. However, this is a particularly inflexible policy: the rate of declassified information cannot be adjusted at runtime. Furthermore, the system needs to send messages even when no new information can be declassified, leading to extraneous resource utilization.

Instead, we propose a different policy: Bridge takes advantage of the fact that in leak-free systems, *some* communication patterns are still allowed across security domains. In particular, any low-secrecy domain—by definition—can arbitrarily deliver information to a high-secrecy domain. Thus, it is perfectly fine for a low-secrecy task to expose information about its timing to a high-secrecy task, for instance by use of an asynchronous and uncoordinated inter-core interrupt, so long as this mechanism does not also act as a backchannel.

In Bridge, we use this primitive to establish a higher-level communication channel with request–response semantics. In particular, we use inter-core interrupts to signal a request from a low-secrecy task, running on one CPU core, to another high-secrecy task, running on a different CPU core. However, we must avoid introducing an implicit backchannel, such as a response-ready signal to the low-secrecy core. Instead, we allow the low-secrecy core to retrieve the response at any time. Because the low-secrecy task governs *when* the response will be retrieved, we must additionally avoid leaking information on *whether* the high-secrecy task was even able to prepare a complete response before it is retrieved. This is because the time it takes to prepare a response may depend on other secrets in the high-secrecy domain.

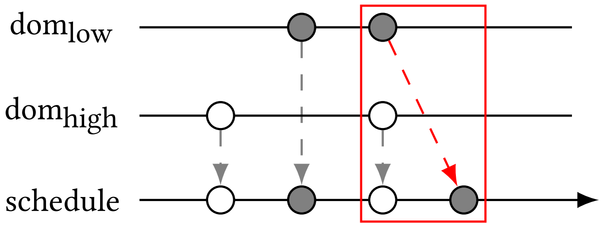

Figure 1. Parallel accesses to shared hardware can leak timing information. In this example, a low-secrecy domain can observe the presence of a parallel access from a high-secrecy domain because of a delay in its own access.

To mitigate this issue, when the high-secrecy task is unable to prepare a response in time, Bridge returns a fabricated response based on an application-defined response policy. In particular, the policy specifies *how* to construct a fabricated response that matches both the application's functional safety and security semantics. The application can choose to eliminate this leak by acquiring a policy that fabricates a response that is *indistinguishable* from a proper response. When eliminating such leak endangers the application's functional safety, the application can *explicitly* choose to leak this information by returning a default response.

We thus far only established a *signaling* mechanism between cores. To realize this channel, we need to solve two additional issues. First, passing actual declassified data requires some shared storage, accessible to low- and high-secrecy domains. This risks introducing new timing channels due to parallel accesses to this hardware. Furthermore, handling inter-core signals must not be delayed by the high-secrecy task. This is problematic in the presence of critical sections. Over the remainder of this section, we present solutions to each of these issues in detail.

Communication as Ownership Transfer. One fundamental issue of existing concurrently-accessed hardware is that it incorporates a predefined scheduling policy which is agnostic to the system's security policy. For instance, as illustrated in Figure 1, such a scheduling policy may choose to schedule an event from a high-secrecy domain in the presence of two parallel events from both low- and high-secrecy domains. This leaks information. Existing approaches that share hardware components force all parallel accesses to be from the same security domain, or require specialized hardware that adheres to a fixed scheduling policy, independent of runtime behavior [5].

Bridge circumvents these limitations by ensuring that a resource is accessed by at most one security domain at a time. Bridge models cross-domain communication as *moving* resources between security domains, and using their inherent state to exchange data. Bridge then enforces that any resource is exclusively owned by a single security domain, and only accessed by its current owner.

We can use these semantics to build a request–response communication scheme as illustrated in Figure 2: first, a low-secrecy requestor uses its currently owned shared resource to prepare a request payload. Because it holds ownership over the shared resource, it can arbitrarily modify its state. It then moves this resource towards the high-secrecy task. Over the time that the high-secrecy task holds ownership of this resource, it can read the request payload, perform requested computations, and write results back to the resource. Recall that the high-secrecy task has no way to signal the completion of this request. Instead, the low-secrecy task will estimate the completion time of a request, and reclaim ownership of the shared resource once this time is reached. It can then read the high-secrecy task's response from the resource.

With this communication scheme, Bridge sidesteps any parallel accesses to shared hardware. Furthermore, so long as hardware-components return to a well-defined, single state after all accesses complete, it is safe to move such hardware between security domains. Thus, we can safely apply this policy onto a plethora of existing hardware components, such as a shared memory bus and interconnects.

Resource Reclamation without Backchannels. The Bridge communication scheme must further avoid backchannels as part of its inter-core signaling mechanism. For instance, this manifests in the low-secrecy task actively reclaiming ownership of the shared resource, regardless of whether the high-secrecy task has finished preparing a response. This reclamation is realized through a signal delivered from the low-secrecy to high-secrecy core. However, this poses another problem: how can the low-secrecy core *know* whether and when the high-secrecy core has given up acccess over this resource, without creating a backchannel?

This issue is exacerbated by the fact that the low-secrecy task has no knowledge over the runtime behavior of the high-secrecy task. For instance, the high-secrecy task may have entered a critical section while receiving the aforementioned signal. In this case, handling of the signal will be delayed until after the high-secrecy task exits the critical section. More generally, practical systems are not preemptible in every clock cycle; they either feature critical sections in software, or stall CPUs on interrupt entry[1].

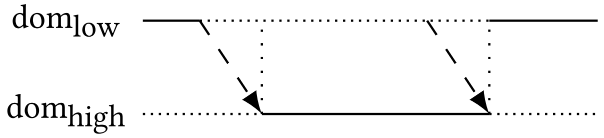

Figure 2. A request–response communication scheme with ownership transfer between security domains. Solid lines represent ownership over a shared resource. Ownership is moved through an initial operation by the low-secrecy domain and later reclaimed using a second operation. This scheme ensures the resource is only accessed by at most one security domain at a time without backchannels.

To work around this issue, Bridge analyzes the worst-case execution time (WCET) of all non-preemptible critical sections of the high-secrecy codebase. As described in Section 3.3, our system does not allow application code to introduce new critical sections, and we thus only need to analyze a small, well-defined TCB. WCET provides with us an overestimate of both the maximum time it takes for a high-secrecy domain to *start* handling a cross-domain signal (taking precedence over all other signals), referred to as

$$t_{\text{delay}} = \max_{c \in \text{CriticalSections}} \text{WCET}(c),$$

as well as the time it takes to *complete* handling this signal

$$t_{\text{handle}} = \text{WCET}(\text{inter-core signal handler}).$$

Thus, the minimum wait-time of a low-secrecy task before it can assume ownership after a reclamation request is

$$t_{\text{wait}} = t_{\text{delay}} + t_{\text{handle}}.$$

We argue that a WCET analysis is practical in this setting as it is required *only* for critical sections. Notably, these sections only require local reasoning as they are single-threaded and do not involve concurrent accesses: the ownership transfer guarantees exclusive access to the shared hardware for its respective current owner. This reduces the amount of code to verify and also eliminates sources of timing interference.

We now proceed to outline Bridge's leak-free inter-core communication protocol in detail.

The Bridge Protocol. Figure 3 illustrates the Bridge protocol. Initially, the low-secrecy core holds ownership over the shared memory block. ① The low-secrecy core first prepares a request in the shared memory. When the request is ready to be sent, it ② releases its ownership over the shared memory and ③ sends a signal to the high-secrecy core. ④ Upon receiving the signal, the high-secrecy core acquires ownership over the memory, at which point it is safe for the core to access it. The high-secrecy core reads the request from the shared memory, processes it, and writes back a response. ⑤ Following an initial grace period of t_{wait}, the low-secrecy core may—at any time—send a reclamation request. The high-secrecy core is *guaranteed* to handle this reclamation request within t_{wait}. During this time, the high-secrecy core will finish execution of any prior critical section, ⑥ finalize its response, and ⑦ release ownership over the

[1] To illustrate, even Linux's PREEMPT_RT_FULL preemption model does not allow reentrance in top half interrupt handlers [17].

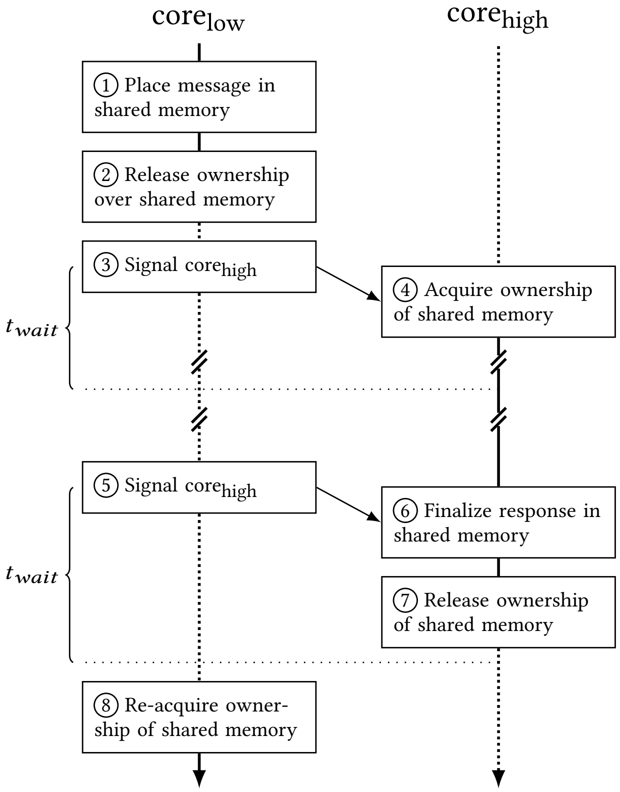
core$_{low}$
core$_{high}$
① Place message in shared memory
② Release ownership over shared memory
③ Signal core$_{high}$
④ Acquire ownership of shared memory
t_{wait}
⑤ Signal core$_{high}$
⑥ Finalize response in shared memory
t_{wait}
⑦ Release ownership of shared memory
⑧ Re-acquire ownership of shared memory

Figure 3. The Bridge protocol. Solid lines indicate ownership over a shared resource.

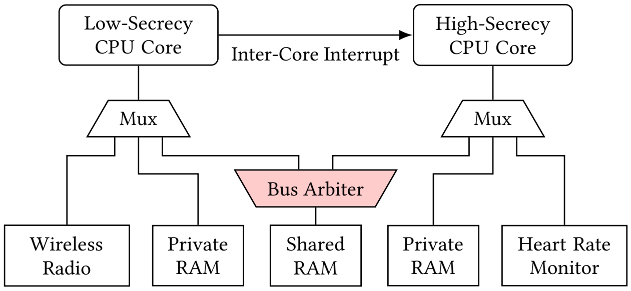

Figure 4. A simplified Bridge hardware architecture. The system features multiple, independent CPU cores executing different security domains. Each core has a dedicated memory bus, RAM and peripherals. Both cores further have access to a shared memory region, mediated through a bus arbiter, and raise asynchronous inter-core interrupts at the other core. Concurrent accesses to the shared RAM through its arbiter can result in contention, which Bridge avoids.

memory block. ⑧ Finally, the low-secrecy core is able to process this response. Note that the low-secrecy core may carry out other computations between steps ③ and ⑧.

3.3 The Bridge System Architecture

Thus far, we focused our design on the key mechanism used to achieve Bridge's leak-free property in a parallel system: its inter-core communication protocol. This protocol makes a set of assumptions about its surrounding system behavior, which can be realized through a combination of hardware and software modifications. We conclude our design by presenting a high-level overview of one particular system architecture meeting the Bridge protocol requirements. Our system architecture places an emphasis on staying close to existing hardware components and configurations, moving most responsibilities into a software-based TCB.

Hardware Architecture. Figure 4 depicts a simplified view of Bridge's hardware architecture. As described above, we base Bridge on a *shared-nothing* architecture featuring independent CPU cores with a dedicated set of peripherals. These CPU cores further feature a signal delivery mechanism, such as inter-core interrupts. We only require that low-secrecy cores are able to send signals to high-secrecy cores. Apart from their dedicated peripherals, both cores have access to a shared resource for communication (such as a shared RAM block). Simultaneous access to this resource may result in contention, which Bridge avoids. Our software TCB ensures that only the current owner of a resource can access it, instead of mediating accesses in hardware.

While Figure 4 depicts an architecture where CPU cores share no peripherals and no memory buses except for the single, shared RAM block, in practice we can be more permissive: Bridge only requires security domains to not interfere with each other when performing memory or peripheral accesses. We thus solely have to ensure that cores do not experience contention while accessing their designated peripherals. Furthermore, when performing inter-core communication as part of Bridge's protocol, a resource is temporarily moved between two cores. Thus, this invariant must also hold under this temporary ownership change.

These constraints are practical. For instance, the Raspberry Pi RP2040 ARM Cortex-M0+ SoC features two CPU cores, connected to the chip's peripherals using a fully-connected crossbar, with a dedicated upstream port per CPU core [16]. Memory is divided into individual RAM blocks attached to this crossbar, and can thus be logically moved in between cores without access contention. The crossbar further features sufficient downstream ports to logically divide other peripherals in the system between the two cores without contention[2]. Furthermore, the RP2040 architecture features a synchronous cross-core mailbox infrastructure which can be set to take priority over any other interrupts, and thus meets the requirements of the Bridge's signaling mechanism. As such, this chip meets all of Bridge's hardware requirements.

[2] In the case of the RP2040, this is not possible for peripherals that are attached to APB Bridge component [16] which acts as a bus arbiter with a single upstream port.

Software Architecture. Bridge's software is architected as a shared-nothing multi-kernel system [2]. Each core runs its own kernel instance, and shares only a buffer located in the shared RAM with other instances. Each instance is further assigned to a static security domain. Userspace processes are untrusted and isolated by the kernel, and can always be preempted—userspace cannot introduce new critical sections that would need to be subject to WCET analysis. Furthermore, the kernel must handle the request–response signal immediately (or after returning from any currently entered critical section). In practice, we achieve this by having cross-core interrupts take precedence over all other events.

Bridge does not mandate that a kernel instance use a particular scheduling policy or isolation mechanism amongst its local processes. Any interference between local resources is acceptable as they belong to the same security domain.

4 Current State

We validate the feasibility and the correctness of Bridge's protocol and system architecture through a proof of concept implementation. We model our system's hardware using a dual-core RISC-V SMP cluster virtualized in QEMU. We base our TCB software on the Tock OS kernel [11]. Tock is a secure and lightweight embedded operating system that supports running untrusted userspace applications in a single-core platform. To support execution on an SMP CPU cluster, we extend Tock to run multiple kernel instances in a multi-kernel configuration (adding approx. 2300 LOC). We further implement Bridge's leak-free inter-core message passing protocol to facilitate communication between two Tock kernel instances in a low–high security domain relation (adding approx. 300 LOC). This prototype implementation is able to successfully send a request from a low-secrecy kernel instance to a high-secrecy kernel instance, and return a response as part of the corresponding reclamation request.

Our prototype further confirms that existing memory protection infrastructure, such as RISC-V's Physical Memory Protection (PMP), is sufficiently expressive to model our *ownership* and *move* semantics. Finally, our analysis shows that there are only four critical sections in the existing Tock codebase which are all suitable for WCET analysis. Integrating the Bridge protocol does not add a new critical section, and only extends Tock's top-half trap handler.

At this point, we have not yet carried out the required automated WCET analysis of our implementation but instead used conservative estimates. Furthermore, we are currently transitioning this system towards an SoC hardware configuration featuring two VexRiscv CPU cores, which can be synthesized for FPGAs or used in a cycle-accurate simulator to verify Bridge's timing guarantees [15].

Nonetheless, these initial results are promising and show that Bridge's system architecture and the requirements of its protocol are feasible.

5 Related Work

Bridge builds on a wide range of related work. Its foundation is rooted in the concepts of information flow modeling [4]. We borrow its notion of high- and low-secrecy labels and relations and map them onto our security domains. In particular, we reason about both explicit data *and* implicit timing interference as information flows between security domains.

Timing-safe time-shared systems. Prior work addresses Bridge's problem domain in the context of non-parallel time-shared systems. For example, the seL4 microkernel has been extended with time protection mechanisms to eliminate microarchitectural timing channels by resetting hardware state when switching between security domains and using a fixed time domain scheduler to prevent high-level timing channels across security domains [5, 14, 20]. The VAX/VMM security kernel implements a so-called lattice process scheduler that flushes cache state only when necessary, i.e., when switching from a high-secrecy process to a low-secrecy process [7]. The lattice scheduler also eliminates scheduling-based timing channels by only allowing a high-secrecy process to run so long as no lower-secrecy processes are schedulable.

Timing-safe parallel systems. To avoid leaking information due to both microarchitectural state and contention while accessing shared resources, *core slicing* runs mutually distrustful workloads on individual, independent CPU cores, memory, and peripherals [24]. However, this work does not permit direct communication between these domains. Furthermore, it does not explicitly consider potential timing interference due to shared components in the underlying hardware, for instance, when each domain accesses its dedicated peripherals through a shared memory bus.

Other approaches. The emergence of side-channels in widely used hardware has further prompted the development of various policy-agnostic mitigations. For instance, a large class of work introduces additional entropy into a system to prevent meaningful extraction of information, without addressing the underlying channels [6, 23]. Other approaches use time quantization to provide an upper bound on the amount of information that can be extracted, while often compromising efficiency or responsiveness [3, 10, 18].

6 Conclusion

In this paper, we explore the space of timing-safe parallel systems. To that end, we present Bridge, a new system architecture for parallel tasks running across multiple security domains. This architecture allows maintaining responsiveness and avoids resource starvation of one security domain in the presence of external events towards other security domains. Our initial prototype demonstrates its feasibility and represents a promising step towards a complete implementation. In the future, we hope to extend Bridge's principles towards more dynamic and complicated security policies. We view Bridge as a stepping stone towards this direction.

References

[1] Dakshi Agrawal, Bruce Archambeault, Josyula R Rao, and Pankaj Rohatgi. 2003. The EM side—channel (s). In *Cryptographic Hardware and Embedded Systems-CHES 2002: 4th International Workshop Redwood Shores, CA, USA, August 13–15, 2002 Revised Papers 4*. Springer, 29–45.

[2] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 29–44. https://doi.org/10.1145/1629575.1629579

[3] David Cock, Qian Ge, Toby Murray, and Gernot Heiser. 2014. The Last Mile: An Empirical Study of Timing Channels on seL4. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. Association for Computing Machinery, New York, NY, USA, 570–581. https://doi.org/10.1145/2660267.2660294

[4] Dorothy E. Denning. 1976. A lattice model of secure information flow. *Commun. ACM* 19, 5 (may 1976), 236–243. https://doi.org/10.1145/360051.360056

[5] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. 2019. Time Protection: The Missing OS Abstraction. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 1, 17 pages. https://doi.org/10.1145/3302424.3303976

[6] W.-M. Hu. 1991. Reducing timing channels with fuzzy time. In *Proceedings. 1991 IEEE Computer Society Symposium on Research in Security and Privacy*. 8–20. https://doi.org/10.1109/RISP.1991.130768

[7] W.-M. Hu. 1992. Lattice scheduling and covert channels. In *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*. 52–61. https://doi.org/10.1109/RISP.1992.213271

[8] Michael Hutter and Jörn-Marc Schmidt. 2014. The Temperature Side Channel and Heating Fault Attacks. In *Smart Card Research and Advanced Applications: 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers* (Berlin, Germany). Springer-Verlag, Berlin, Heidelberg, 219–235. https://doi.org/10.1007/978-3-319-08302-5_15

[9] 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. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[10] Boris Köpf and Markus Dürmuth. 2009. A Provably Secure and Efficient Countermeasure against Timing Attacks. In *2009 22nd IEEE Computer Security Foundations Symposium*. 324–335. https://doi.org/10.1109/CSF.2009.21

[11] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 234–251. https://doi.org/10.1145/3132747.3132786

[12] 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. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[13] Rita Mayer-Sommer. 2000. Smartly Analyzing the Simplicity and the Power of Simple Power Analysis on Smartcards. In *Cryptographic Hardware and Embedded Systems - CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1965)*. Springer, 78–92. https://doi.org/10.1007/3-540-44499-8_6

[14] Marcelo Orenes-Vera, Hyunsung Yun, Nils Wistoff, Gernot Heiser, Luca Benini, David Wentzlaff, and Margaret Martonosi. 2023. AutoCC: Automatic Discovery of Covert Channels in Time-Shared Hardware. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 871–885. https://doi.org/10.1145/3613424.3614254

[15] Charles Papon. 2024. VexRiscv. https://github.com/SpinalHDL/VexRiscv. Accessed: 2024-08-16.

[16] Raspberry Pi Ltd. 2024. RP2040 Datasheet. https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf Revision `576cee3-clean`, retrieved at 2024-08-16.

[17] Federico Reghenzani, Giuseppe Massari, and William Fornaciari. 2019. The Real-Time Linux Kernel: A Survey on PREEMPT_RT. *ACM Comput. Surv.* 52, 1, Article 18 (feb 2019), 36 pages. https://doi.org/10.1145/3297714

[18] Ryan Torok and Amit Levy. 2023. Only Pay for What You Leak: Leveraging Sandboxes for a Minimally Invasive Browser Fingerprinting Defense. In *2023 IEEE Symposium on Security and Privacy (SP)*. 1023–1040. https://doi.org/10.1109/SP46215.2023.10179385

[19] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. 2022. Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 679–697. https://www.usenix.org/conference/usenixsecurity22/presentation/wang-yingchen

[20] Nils Wistoff, Moritz Schneider, Frank K. Gürkaynak, Luca Benini, and Gernot Heiser. 2021. Microarchitectural Timing Channels and their Prevention on an Open-Source 64-bit RISC-V Core. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 627–632. https://doi.org/10.23919/DATE51398.2021.9474214

[21] WITTENSTEIN High Integrity Systems Ltd. 2024. SafeRTOS. https://www.highintegritysystems.com/safertos/. Accessed: 2024-07-01.

[22] Zephyr Project Contributors. 2024. The Zephyr Project. https://www.zephyrproject.org/. Accessed: 2024-07-03.

[23] Rui Zhang, Xiaojun Su, Jianping Wang, Cong Wang, Wenyin Liu, and Rynson W. H. Lau. 2015. On Mitigating the Risk of Cross-VM Covert Channels in a Public Cloud. *IEEE Trans. Parallel Distrib. Syst.* 26, 8 (aug 2015), 2327–2339. https://doi.org/10.1109/TPDS.2014.2346504

[24] Ziqiao Zhou, Yizhou Shan, Weidong Cui, Xinyang Ge, Marcus Peinado, and Andrew Baumann. 2023. Core slicing: closing the gap between leaky confidential VMs and bare-metal cloud. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 247–267. https://www.usenix.org/conference/osdi23/presentation/zhou-ziqiao