Protecting Enclaves from Intra-Core Side-Channel Attacks through Physical Isolation

Marno van der Maas
University of Cambridge
Cambridge, United Kingdom
Marno.van-der-Maas@cl.cam.ac.uk

Simon W. Moore
University of Cambridge
Cambridge, United Kingdom
Simon.Moore@cl.cam.ac.uk

ABSTRACT

Systems that protect enclaves from privileged software must consider software-based side-channel attacks. Our system isolates enclaves on separate secure cores to stop attackers from running on the same core as the victim, which mitigates intra-core side-channel attacks. Redesigning the memory hierarchy based on enclave ownership protects enclaves against inter-core side-channel attacks. We implement this system and evaluate it in terms of communication performance, memory overhead and hardware area. Combining physical isolation and a redesigned memory hierarchy protects enclaves against all known software-based side-channel attacks.

CCS CONCEPTS

• **Security and privacy → Security in hardware**; **Systems security**; *Software and application security*; • **Hardware**;

KEYWORDS

enclave; memory protection; physical isolation; security; side channel; trusted execution;

ACM Reference Format:
Marno van der Maas and Simon W. Moore. 2020. Protecting Enclaves from Intra-Core Side-Channel Attacks through Physical Isolation. In *2nd Workshop on Cyber-Security Arms Race (CYSARM '20), November 13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3411505.3418437

1 INTRODUCTION

The concept of enclaves in trusted execution comes from the desire to protect applications from privileged software on a system. As operating systems become richer and more complicated, their attack surface increases, and it becomes more likely that they will be compromised by an attacker [7]. Previous enclave solutions show that a combination of hardware enforcement and software management is a powerful and flexible solution to allow any application to protect trusted code [3, 6, 9, 10, 15, 19, 29, 33]. However, the degree to which these solutions protect against side-channel attacks differs widely.

Protecting against side-channel attacks is hard. There are many classes of side-channel attacks like differential power analysis, fault injection using lasers and cache timing attacks. Since side-channel attacks that require physical access are less scalable, we choose to only protect against side-channel attacks that can be executed by software. Previous enclave systems do not fully protect against software-based side-channel attacks, so this requires the formulation of a new threat model that adequately protects enclaves.

Software-based side-channel attacks can be classified into two categories: intra-core and inter-core. Intra-core side-channel attacks require the attacker and the victim to be co-located on the same core, while inter-core side-channel attacks do not have this requirement. Of these two classes, intra-core side-channel attacks are the hardest to protect against, mainly due to the sheer number of *shared* micro-architectural resources in modern application cores. Additionally, the many forms of speculative execution attacks have shown that the dependencies between these micro-architectural resources leak information in unexpected ways [5]. To protect against all intra-core side-channel attacks, we learn from the use of physical isolation in a different class of trusted execution environments.

SIM cards, smart cards and trusted platform modules create a physically-isolated execution environment to make security guarantees. Our work applies this concept of physical isolation in the context of enclaves. All enclaves run on physically separate cores from application cores, and then we focus on protecting against inter-core side-channel attacks that exist due to the sharing of the memory hierarchy. Protecting against inter-core side-channel attacks is not easy because side channels can be based on contention on cache lines, request buffers, busses, DRAM row buffers, etc. However, it is much easier to create a system that protects against inter-core side-channel attacks when we do not have to worry about intra-core side-channel attacks as well.

Given that we now live in the age of dark silicon [13], we can afford to have dedicated hardware to perform key operations. For enclaves, this means that we can have dedicated processors that are designed to be highly robust against intra-core side-channel attacks and speculative execution attacks. Having dedicated secure processors for enclaves means that we do not have to pay the performance penalty of creating extra security requirements for fast application cores. We contribute to the field by:

- Formulating an enclave threat model that includes all software-based side-channel attacks.
- Showing that physical isolation protects enclaves from intra-core side-channel attacks without having to change the implementation of the main application cores.
- Using access control in the memory hierarchy to protect against direct attacks and inter-core side-channel attacks.
- Showing how Linux applications can create and interact with physically isolated enclaves.

CYSARM '20, November 13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8091-1/20/11.
https://doi.org/10.1145/3411505.3418437

2 THREAT MODEL

The motivation for exploring physical isolation in enclave systems stems from the inconsistency of protection against side-channel attacks in previous enclave systems. This section sets out a new threat model for enclaves that includes software-based side-channel attacks.

The foundation for this threat model is protection against attacks that can be launched from *privileged software*, which is similar to that of previous enclave systems [3, 9, 10]. Enclave systems generally protect the following assets from attacks launched by privileged software and other enclaves:

- Confidentiality of memory: so that secret content is not leaked across enclave boundaries.
- Integrity of memory: so that code and data cannot be tampered with from outside an enclave boundary.
- Authenticity of the enclave system and enclave: to attest that an enclave is running on and is bound to a secure environment.

Generally this means that enclave threat models include *direct attacks*, like reading from and writing to memory. These attacks can be launched by an operating system or a malicious enclave, and include writing to memory via a direct memory access (DMA) request. For example, it is trivial for privileged software on the CPU to use the GPU to make a DMA request.

Denial of service attacks are classically excluded in enclave threat models because privileged software is in charge of resource management like memory allocation and scheduling. It is trivial for privileged software to refuse to schedule an application or to refuse to allocate memory to it. *Physical attacks* are also excluded because requiring physical access to a machine is less scalable than just requiring software access.

All of the above is similar to what previous solutions have considered, but there is a lack of consensus on protection against side-channel attacks. Our threat model includes *side-channel attacks* that can be launched from privileged software, like timing of memory requests and contention of execution units. To motivate this choice of threat model, we look at previous solutions and their inconsistent coverage of side-channel attacks. These findings are summarized in Table 1 with the Praesidio column presenting our new threat model, and the details of the side channels are discussed in Section 5.4.

2.1 Trusted Computing Base

In our system we protect against this new threat model with the assumption that our trusted computing base (TCB) is not compromised. Part of the TCB is the hardware implementation of the memory hierarchy and the secure cores. Another part of the TCB is the enclave management software, which we call the management shim. We thus exclude attacks that rely on the presence of *hardware Trojans* or *software bugs* in the TCB. Notably, we do not trust the hardware implementation of the fast application cores in our system, which makes it possible to gain confidence in the hardware's security without having to verify the fast cores. The details of the TCB are discussed further in Section 5.1.

2.2 Physical Isolation

We argue that it is impractical and not good for performance to gain enough confidence that intra-core side-channel attacks are covered when an attacker shares a high-performance core with a victim. This is because there are numerous shared resources in a high-performance core, and each of those shared resources can have contention, which exposes some of the micro-architectural state. Additionally, micro-architectural resources can have complex interactions with each other causing information to leak widely throughout the core. A good example of this is speculative execution, where the speculative nature of cores is combined with a side-channel attack to extract information from a victim [5]. Additionally, simultaneous multi-threading increases the attack surface even more in increasingly complicated cores [2].

We propose to protect against intra-core side-channel attacks from a policy point of view, which enforces that all active enclaves are physically isolated on their own separate core. In Table 1, we can see that all intra-core side-channel attacks are highlighted in gray for the Praesidio column. This means that physical isolation protects against this whole class of attacks.

Another benefit of physical isolation is that it helps with the security/performance trade-off. Application cores can do speculative and out-of-order execution to improve performance of the feature-rich operating system and applications, whereas enclaves can run on cores that lean more towards security without having to cater to the performance requirements of other applications. Physical isolation allows for this trade-off between security and performance to be played out in a heterogeneous environment. It also has the benefit of applications being able to choose what part of their code they execute on high-performance cores that have fewer guarantees on isolation and what part of their code requires stricter isolation.

3 SYSTEM OVERVIEW

Praesidio is our enclave system that enforces physical isolation while still providing the same functionality as conventional enclave systems and with minimal performance loss. Our system enforces access control in the memory hierarchy without trusting the implementation of the fast application cores. This is possible because of the addition of secure cores which are designed to be highly robust against speculative execution and side-channel attacks.

Figure 1 shows the system overview of Praesidio with a clear distinction between software that runs on cores optimized for performance (left) and software that runs on cores optimized for isolation (right). The goal is for any user application to be able to launch an enclave and interact with it. To accomplish this we developed a user-level application programming interface (API), a Linux driver, a management shim and an enclave runtime. The user-level API avoids having to repeat code between applications that need enclaves. The Linux driver interacts with the management shim to set an enclave up and to reserve memory from the operating system for the enclave. The hardware is in charge of access control: making sure that any enclave can only interact with its own pages. The management shim does all the managing like creating a new enclave, scheduling an enclave, handling enclave traps and

Table 1: Comparing Enclave Threat Models by Which Inter- and Intra-Core Side-Channel Attacks They Cover

	Attack	Timber-V [33]	Intel SGX [9]	AMD SEV-SNP [19]	Sanctum [10]	MI6 [3]	Praesidio
Inter-core	Cache line [10]	○	○	○	●	●	●
	Cache request buffer [3]	○	○	○	○	●	●
	DRAM row buffer [27]	○	○	○	○	●	●
	DRAM request buffer [3]	○	○	○	○	●	●
	DRAM bit leakage [21]	○	●	●	○	○	●
Intra-core	TLB [35]	○	○	○	●	●	●
	Execution unit [2]	○	○	●	○	●	●
	Branch predictor [23]	○	○	●	○	●	●
	Hardware accelerators [14]	○	○	○	○	○	●
	Speculative execution [5]	○	○	●	○	●	●

Table legend: Providing protection is marked as ● and failing to protect is marked as ○. Physical isolation is shown in gray.

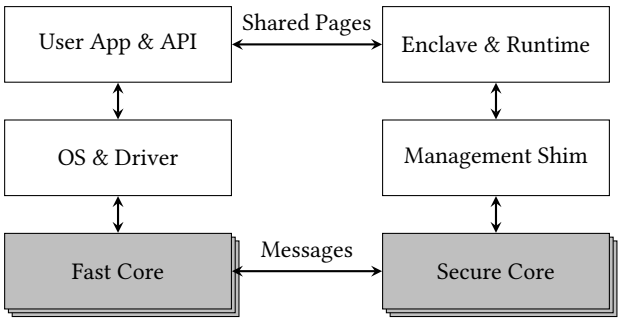

Figure 1: System overview of Praesidio. White boxes are software, gray boxes are hardware, arrows are channels of communication.

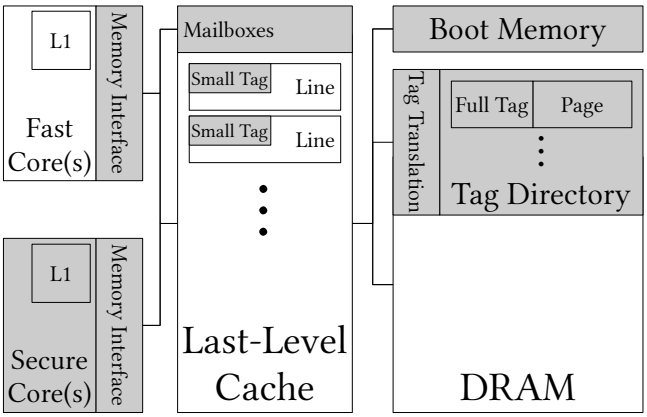

Figure 2: An overview of the hardware in our system, with white backgrounds showing already existing blocks and gray backgrounds showing added hardware.

re-assigning pages. The enclave runtime allows enclaves to interact with the rest of the system.

Another way of looking at the system is from a hardware perspective. Figure 2 gives an overview of the additional hardware (in gray) that is required by Praesidio. It also shows that fast cores stay unmodified except for a memory interface that enforces access control on the communication between the core's private caches and the shared last-level cache. We discuss optional tag translation to reduce the LLC overhead in Section 4.4.3. Finally, a trusted boot ROM is added so that the management shim can protect its pages and clear any sensitive data before untrusted code runs.

3.1 Memory Protection

To protect enclaves against direct attacks, Praesidio tags each page with an enclave identifier. Memory protection is enforced by the memory interfaces that are located between the cores and the shared cache. The tags are managed by the management shim, and the tags are stored in the last-level cache (LLC) and the tag directory. On every memory access that reaches the LLC, the tag of the memory is checked with the identifier of the currently running enclave. In this model, normal applications and other software running on the fast cores are assigned the default enclave identifier. Loading from or storing to a page that an enclave does not have access to causes a trap.

The tag directory is separate from the page table because tags protect physical pages, not virtual ones. In theory we could include the enforcement of tags in the page table of the enclave cores because the management shim is responsible for installing the memory mapping on those cores. However, the page logic of the fast cores is controlled by privileged software that is potentially hostile. Even though privileged software controls memory management, it should not be able to access pages it does not own. This means that the checking of tags must be enforced on the interface between the fast core's private caches and the LLC. For consistency, we choose to keep tags separate from the page directory in the whole system.

To support communication, our system allows a page owner to add a reader tag to a page. This reader enclave can then read, but

not modify, the content of the page. We choose to only give the second enclave read access because this one-way communication can create two-way communication by setting up two pages in opposite directions. Setting up two unidirectional communication channels is a well-understood concept due to the frequent use of Unix pipes for communicating between parallel processes. Additionally, the principle of least privilege justifies only giving read access where that suffices. These one-way shared pages are used to implement ring buffers for communication (see Section 3.2.7).

The tag directory contains all pages with their associated tags. For each page there is an owner tag and an optional reader tag. The absence of a reader tag means that the page is private to the owner. Any page that is not in the tag directory is considered to be owned by the rich operating system, which operates under the default enclave identifier. Each line in the last-level cache is tagged with an owner and optionally a reader. The following steps describe the way memory requests are handled in Praesidio and how to enforce access control:

(1) First, the core's local memory is checked. Any hit at this level of the memory hierarchy means that the memory request can succeed without checking any tags because, depending on the core type, one of these two applies:
   (a) For fast cores, all code is considered to run under the default enclave identifier, so the content of the translation look-aside buffer (TLB), store buffer and first-level cache only contain entries of memory that is not owned by another enclave.
   (b) For secure cores, the content of the TLB, store buffer and first-level cache can either be private to each enclave or partitioned securely. When a context switch occurs on a secure core, all of the local state related to that enclave must be flushed (see Section 3.2.4).

(2) In the case of a first-level cache miss, the memory hierarchy checks the LLC. The memory interfaces for both fast and secure cores ensure that each memory request is accompanied by the enclave identifier of the requester. Checking the LLC leads to one of two scenarios:
   (a) If the LLC hits, the tag for the cache line of the LLC is checked with the requester's identifier as shown in Figure 3. If smaller tags are used in the LLC, then the requesting enclave identifier must first be translated to the smaller tag format.
   (b) If the LLC misses, the request must first go through the tag directory before going to DRAM. The tag directory contains all mappings from physical pages to owner enclave identifiers and optionally reader enclave identifiers. The request is evaluated using the process shown in Figure 3. If the request is allowed by the tag directory, the tag from the tag directory is optionally translated to a smaller version for in the LLC.

Our memory tags protect the confidentiality and integrity of enclave memory without using encryption. A software attacker simply cannot access the memory it wants to read or write, which is similar to the guarantee that an attacker cannot disclose or tamper with the content of a ciphertext.

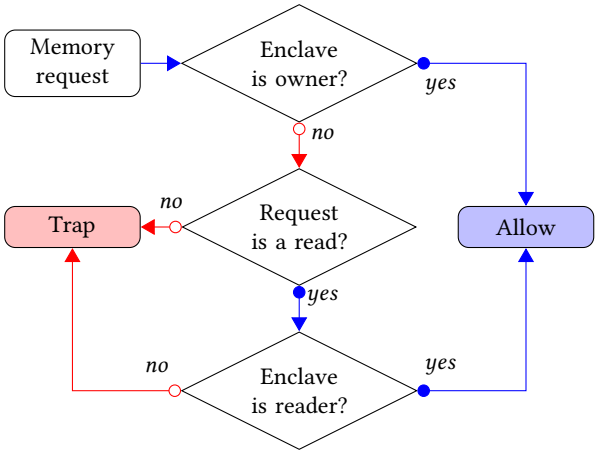

Figure 3: Process for checking whether a memory request is valid or not. This process is followed when checking whether a requester is allowed to access an LLC cache line or a page in DRAM.

3.1.1 Preventing Inter-Core Side Channels. Praesidio ensures protection against all intra-core side-channel attacks by segregating security domains onto their own cores. However, there are resources that are shared across cores, which introduces the possibility of inter-core side-channel attacks. Contention in shared caches, request buffers and DRAM are examples of inter-core side-channel attacks.

Cache-line contention causes information leakage between security domains because one domain can evict a line from another domain. To solve this issue, previous work has introduced static partitioning [3, 10], flexible partitioning [11, 31] or injecting noise [24]. Our system is configurable to use no partitioning scheme, a static partitioning scheme or a flexible partitioning scheme [31].

Because partitions are inherently assigned to different enclaves, we assume that there is no shared memory between any two enclaves except for when a reader is assigned to a piece of memory. For memory that does not have a reader enclave, we do not need to worry about cache coherence between enclaves because only the owner enclave can access that memory. This means that unless there is an explicit reader, the cache coherency mechanism will not cause any information flow between enclaves. The coherence protocol does introduce information flow between enclaves that share memory, but this is acceptable since it will only leak information about them accessing the shared memory.

Inter-core side-channel attacks that rely on DRAM, like row-buffer contention [27] or request-buffer contention [3], are also important to be protected against. From the hardware overview in Figure 2 we can see that DRAM is not adjusted, so we can use existing mitigations for these problems. In Section 5, we discuss the security of Praesidio in more detail and how known inter-core side-channel attacks on enclaves can be mitigated.

3.1.2 Bootstrapping Shim. To have a root of trust from which the management shim can start to manage the tags that protect memory, we need to ensure that at least one secure core boots into the

management shim and that all other cores do not interact with the memory. The memory interface of each core can pause all memory requests while the management shim initializes. During initialization, the management shim clears any sensitive data from DRAM and tags all the pages that are owned by the management shim. It is also at boot that the management shim creates the measurement of its own pages for attestation purposes. Bootstrapping trusted code at boot time is important to guarantee the integrity of the management shim code as well as to protect the confidentiality of the platform key that is needed for attestation.

3.2 Management Shim

The management shim is the software part of Praesidio's trusted computing base (TCB). The hardware enforces that enclaves can only access pages that they are authorized to, and the management shim is the only piece of code that is allowed to change the values in the tag directory. The management shim is not an operating system and only implements the least amount of logic necessary to securely maintain enclave lifetime and transfer ownership of pages.

In Praesidio, the resource management is still done by the rich operating system, which has the benefit of leaving this complexity outside of the TCB. The management shim is only there to ensure that the operating system does not break any security rules. For example, once a page is donated to an enclave, the operating system cannot access it unless the enclave is deleted. Upon deletion, the management shim fills the corresponding pages with zeroes and gives the pages back to the rich operating system.

3.2.1 Messaging. Communication in Praesidio is done via shared memory. Section 3.1 describes how pages can be tagged with a reader as well as an owner for communication. Section 3.2.7 describes how this communication mechanism is exposed to enclaves. To securely set up these pages and as a trusted way to manage enclave life-cycles, we provide a messaging system meant to send small, infrequent messages.

We implement this messaging system by having a portion of the last-level cache that has a mailbox for each core (see Figure 2). A core can put a message into its own mailbox with a recipient identifier, and it can check whether any of the other cores have posted a message for them. The main purpose of this messaging system is for cross-core communication within the management shim and to allow the operating system to send requests to the management shim. These mailboxes are memory mapped, so to write a message an enclave simply has to store to a special physical address.

In essence a message is sent from one enclave to another. The management shim has a dedicated set of enclave identifiers. Each instance of a management shim on a core has its own identifier; this allows scheduling messages to be sent to specific cores. Each message has a destination enclave identifier, a message type and a payload of maximum two arguments. The different message types are summarized in Table 2. The return values in the table are themselves messages in the opposite direction.

3.2.2 Enclave Life Cycle. Sending messages to the management shim allows an operating system to manage enclave life cycles while the management shim upholds the security requirements. Figure 4 shows the different states that an enclave can be in and which messages cause the state transitions. Our state diagram is similar to that of previous work by Costan [10] with the difference being that their system does not have explicit building and error states. The management shim keeps track of which state the enclave is in through an enclave data entry. This means that the management shim can enforce security properties. For example, pages cannot be donated to an enclave that has already run.

An enclave data entry starts out being empty and goes to the created state when a create message is received. Once it is created, the enclave can start accepting pages. The first page is assumed to be the entry point, which is why the created and building states are separate. Once in the building state, the enclave can accept an arbitrary number of pages until the enclave is finalized. In the process of finalizing, the management shim creates the attestation measurement of the content of all the enclave's initial pages. After the enclave is finalized it goes into the live state and can no longer accept any pages. In the live state, the enclave can be run, can receive shared pages and can be attested. Upon deletion all enclave pages are filled with zeroes and given back to the default enclave.

Table 2: Definition of Different Messages that Make Requests of Management Shim or Share Memory Between Enclaves

Type	Arguments		Return
Create	–	–	Enclave ID
Donate	Enclave ID	Page	Success
Finalize	Enclave ID	–	Success
Attest	Enclave ID	Nonce	PubKey, Signature
Run	Enclave ID	Core ID	Success
Share	–	Page	Success
Delete	Enclave ID	–	Success

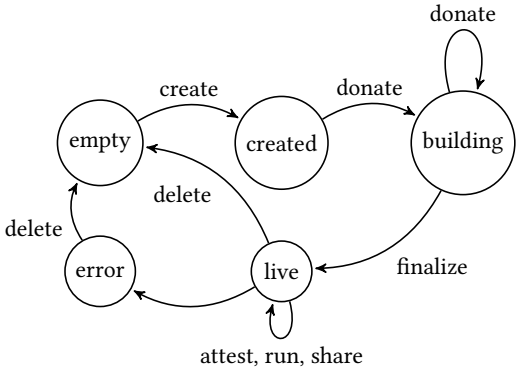

Figure 4: Life cycle state diagram that is tracked in enclave data entries. The transitions are labelled with the messages that cause the transition. Any message type not shown for a particular state is not allowed. If a trap occurs from which recovery is impossible, then the management shim changes the state from live to error.

Once the enclave pages are given back, the enclave data entry goes to the empty state. If at any point the enclave experiences a trap that cannot be recovered from it moves to the error state. From the error state the enclave can no longer run and is just waiting for deletion. Keeping track of these states allows the management shim to know which messages are allowed in each state of the enclave's life cycle.

3.2.3 Scheduling on Enclave Cores. Scheduling is managed by the run message specified in Table 2. The operating system can request an enclave to be run on a specific core. The management shim has a data structure that remembers which enclave is running on which core. This data structure makes it possible for the management shim to check whether an enclave is running when deletion is requested and to perform a secure context switch between enclaves.

The complexity of deciding which enclave is run on which core is delegated to the operating system and driver. This adheres to our threat model because it only allows the operating system to perform a denial of service attack, and it reduces the size of the TCB. In general, the idea is that an enclave runs on a core until another one is scheduled there. To allow the management shim to interpose, we introduce periodic interrupts where the management shim can check whether it needs to perform a context switch or not.

3.2.4 Context Switches. The secure context switch on enclave cores is an important part of the security model. Our physical isolation protects against all intra-core side-channel attacks as long as enclaves do not share a secure core. Context switching introduces time-multiplexed sharing of a core, so we must mitigate intra-core side-channel attacks upon context switch on the secure cores. In essence this requires a flush of all the micro-architectural state like the first-level cache, TLB, store buffer, register file, branch target buffer, in-flight instructions, etc. Previous work has shown the intricacies of purging the micro-architectural state and the difficulty in identifying all the state [3, 34], so the simpler the secure cores are the easier this will be. The execution time of the secure context switch must not vary based on the core's state to avoid leaking information about enclave execution. All of the software logic for this secure context switch resides in the management shim for security reasons.

3.2.5 Handling Traps. Handling traps that occur within enclaves needs to be done by the management shim because traps allow for side-channel attacks if handling them is delegated to an untrusted operating system. One example of such an attack is tracking enclave access patterns through page faults [35]. To avoid this, the management shim installs a simple trap handler.

One example of a trap is when an enclave attempts to access memory that it does not own nor have read access to. Another example is a trap caused by a management shim interrupt, in which case the management shim interposes to check for scheduling messages and either performs a context switch or resumes the enclave. When the trap handler cannot recover from a trap, it puts the enclave into the error state (see Figure 4).

3.2.6 Attestation. Attestation is an important part of any enclave system because it allows a remote party to verify that an enclave is running securely on a trusted platform. Our goal is to show that existing attestation methods can be easily ported to a system with physically-isolated enclaves.

Table 2 references a finalize message, which can be sent to the management shim. This message prevents any more pages from being donated to that enclave and creates a measurement of all the pages in the initial state. This measurement can be made using a secure hash function like SHA3 [12]. Another important part of attestation is proving to the remote party that the enclave is running on a trusted platform. To do this, we must measure and sign the current version of the management shim.

Signing can be done with any secure digital signature algorithm, which can be based on RSA (like in previous enclave systems [10]), but can also be based on elliptic curves which require a smaller signature size for the same security level. The signature is created over a quote when the attest message is sent to the management shim. The quote includes the measurement of the management shim, the measurement of the enclave, "an ephemerally generated public key to be used by the challenger for communicating secrets back to the enclave" [1] and a nonce (number used once) to ensure liveness. More details of what should be included in a quote can be found in the specification of attestation in SGX [1].

The message return value is sent in multiple messages because the management shim can only return a maximum of 96 bits per message, and attestation requires a bigger ECDSA signature, which is 256 bits if NIST P-256 parameters are used [20]. Additionally another 256 bits are necessary to communicate the enclave's ephemeral public key. The signature gets sent along with a certificate that proves the authorization of the signing key of the device using a publicly trusted authority, for example that of the manufacturer. This certificate is public information and does not need to be sent from the management shim to the driver. Our solution is compatible with the certification scheme that has been used in previous work [1, 9, 10].

There are also ways to do runtime attestation where the current state of the stack and heap are taken into account [30]. Praesidio lends itself to attestation of both the initial state and the runtime state.

3.2.7 Enclave Runtime. The main purposes of the enclave runtime are to facilitate communication between enclaves and to maximize code re-use. It has an API to set up communication pages and provides functionality to implement a ring buffer on the shared pages that are described in Section 3.1. The first thing most enclaves will do is set up a communication channel, which is done using the following function call:

```
setupCommunicationPages(receiverID)
```

This sets the reader tag on one of the pages owned by the enclave to the receiver specified. It also waits for that enclave to dedicate a page to communicate over in the other direction. This is similar to setting up two Unix pipes to create bidirectional communication.

The enclave runtime uses pointers to the sending and receiving pages and uses these pages in a ring-buffer fashion. To implement our ring buffer, we need to indicate the status of an entry and the length of an entry. Since our ring-buffer is implemented on a page, we only need 12 bits to encode the length, and we need 1 bit to encode the status. We combine both of these into two bytes; if the most significant bit is set then the entry is not ready yet and if the

most significant bit is unset then these two bytes are equal to the length of the entry. The payload follows these first two bytes.

The following call takes the next entry in the ring buffer, writes the payload after the first two bytes and sets the most significant bit of the next entry to 1. Finally, it writes the length for the current entry to indicate to the receiver that it is ready.

```
sendMessage(message, length)
```

The following call waits until the most significant bit of the status is unset and then returns the corresponding message and length.

```
(message, length) readMessage()
```

Both `sendMessage` and `readMessage` update the pointers for the current position in the communication pages. The implementation of a ring buffer allows new entries to start being written to the start of the page again once the end of the page is reached.

Besides sending and receiving from a ring buffer, Praesidio also allows complete pages to be transferred by the following two commands. The first sets the reader tag for a page and communicates the address of this page via a share message:

```
giveReadPersmission(receiverID, page)
```

On the receiving end, the enclave can get the address of the shared page via the following function:

```
page getReadOnlyPage(senderID)
```

This mechanism allows for a quick transfer of data that is larger than a page, while the ring buffer is an efficient way to send smaller messages and reusing space.

3.3 Linux Driver and User API

The purpose of the Linux driver is to show how a rich operating system can expose enclave functionality to its applications. The driver is primarily in charge of sending messages to the management shim about enclave life cycles and setting up a communication channel between the application and its enclave. The purpose of the user API is to provide an abstraction layer for the application. In essence the user API provides functions to send messages and functions to manage enclave lifetime like the ones below:

```
device create(elfFile)
delete(device)
(publicKey, signature) attest(nonce, device)
```

The `create` function returns a `device`, which is a file descriptor of the character device made for that enclave. It is important to know that the Praesidio driver consists of a base driver and an enclave driver. First, the user API requests an enclave device to be made for them by using an `ioctl` to the base driver, which returns the filename of a new instance of the enclave driver. Having a separate device for each enclave allows the driver to limit access to the enclave based on process identifiers. The user API opens the new device file and requests an enclave to be created using the contents of an ELF file. Internally, the Linux driver converts the create enclave call into individual messages sent to the management shim: Each page is donated individually, the enclave is finalized and the enclave is run at the end of the creation process.

After this, the user application sets up a communication channel by calling `setupCommunicationPages`. The application can then call the `sendMessage` and `readMessage` functions described in Section 3.2.7. Similar to the enclave runtime, the user API also provides ways to donate complete pages using two calls. The first is `getSendPage`, which requests the driver to allocate a page to send over, gives the enclave permission to read it and maps the page into the application's virtual memory space. The second is `getReadOnlyPage`, which waits for an enclave to donate a page and then maps this page into the user's virtual memory.

Table 3: Cache Parameters for Parts of the Cache Hierarchy [26]

Cache	Sets	Ways	Line size (bytes)	Total size (KiB)
Data	64	8	64	32
Instruction	64	8	64	32
Last-level	1024	8	64	512

4 EVALUATION

As Praesidio is a new system that physically isolates enclaves from applications, we need to evaluate whether it is a valid approach with regard to performance and overhead. We implement Praesidio on Spike, a RISC-V instruction set simulator. Spike includes a cache simulator, which we configure with the parameters shown in Table 3. The cache parameters are the same as Berkeley's out-of-order RISC-V processor [26]. The data and instruction cache sizes align well with Intel's ones, but our last-level cache is twice the size of Intel's L2 and a sixteenth of the size of the L3 cache [8].

Besides the cache configuration, we add a tag directory to Spike and enforce access control on each memory access. For all of our experiments we change the interleave to 6, which means that each core takes a turn executing six instructions before the next one. The reason for using six is that for values lower than that Linux no longer boots reliably. The closer the interleave value is to one the more realistic the simulation is compared to simultaneously executing instructions. All of our experiments run on Linux 4.15.0 using buildroot release 2016.05 and RISC-V GNU tool-chain v20171107. All the code necessary to run our experiments is open source[1].

4.1 Enclave Communication

Enclave communication is the most common interaction with enclaves. We evaluate this communication in two ways: by sending messages that are smaller than a page using a ring buffer and by donating complete pages to enclaves.

4.1.1 Ring Buffer. To show how the communication via ring buffer over shared memory performs between enclaves, we send messages of varying sizes. Figure 5 shows the instruction count and last-level cache accesses for varying packet sizes up to just below a page. For sending data that is larger than a page we provide a way to donate complete pages. The instruction count increases linearly with packet size, which is expected since the instruction count is dominated by the loops that write to and read from shared memory. Cache accesses are also linear because they are dominated by the

[1]Source code available at: https://github.com/marnovandermaas/praesidio-sdk

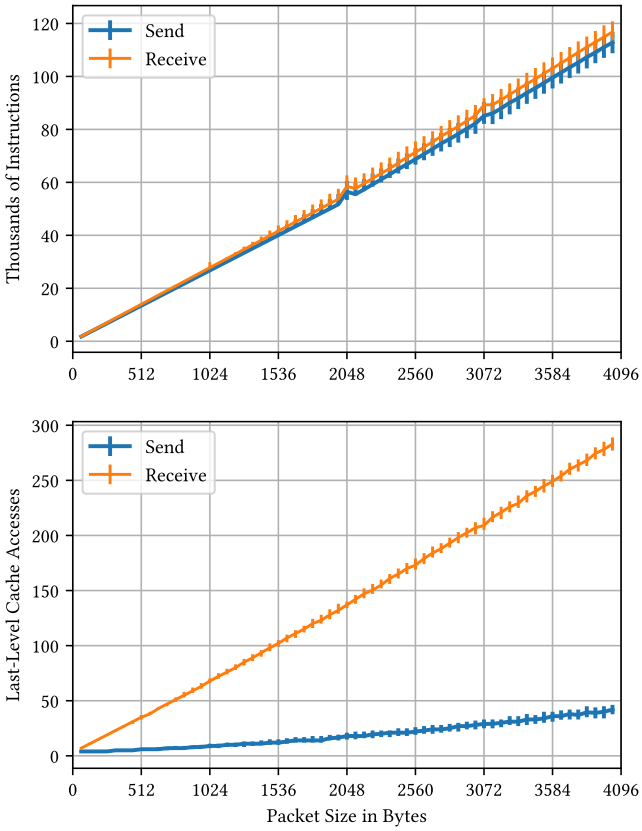

Figure 5: Ring buffer performance over shared pages between enclaves. Each packet size is sent 256 times, and the graph shows a line of the median value and error bars from the first quartile to the third quartile.

loads and stored to shared memory. Receiving dominates the cache accesses because a read causes the dirty line from the writer's cache to be sent to the LLC and causes an access by the reader. In essence, a write does not immediately incur the cache access penalty while the read causes both a write-back and a read access.

4.1.2 Page Donation. Besides using a single page to send messages through a ring buffer, Praesidio allows sending complete pages to enclaves. To measure the performance of this, we create a benchmark in which a user application fills a page with data, sends that page and waits for acknowledgement from the enclave. The enclave waits for the page to be sent, then reads the content of the page and sends an acknowledgement to the user application. This micro-benchmark takes a median of $22,400 \pm 200$ instructions and 390 ± 20 last-level cache accesses, where the spread is determined by the first and third quartile using the following formula: $\text{Max}(\text{median} - Q_1,\ Q_3 - \text{median})$.

This benchmark uses significantly fewer instructions than the ring buffer benchmark because the ring buffer writes one byte at a time, while this page benchmark writes in eight byte chunks. Multiplying the instructions to donate a page by 8 gives approximately $180,000$, which is close to the trend shown in Figure 5.

Table 4: Setup Cost for the Different Phases of the Enclave Creation Process in Instructions and Cache Accesses

	Instructions	Cache accesses
Prepare memory	2.8 %	4.1 %
Driver setup	95.7 %	92.8 %
Enclave setup	1.5 %	3.1 %
Total (thousands)	$9,466 \pm 8$	81.0 ± 0.4

This benchmark can also be compared to doing a similar benchmark using Unix Pipes, which is a commonly used form of unidirectional communication. Sending 4 KiB over a Unix pipe on our system takes $74,900 \pm 100$ instructions and 310 ± 30 cache accesses. Unix pipes take nearly 5 times as many instructions due to the overhead of the operating system, but the cache accesses are of the same order of magnitude as our benchmark. The cache accesses are similar to our page benchmark because Praesidio enforces that enclaves cannot share first-level caches, while this requirement is not true for two Unix processes communicating through pipes. Sending individual pages can be repeated for larger pieces of memory that need to be sent to enclaves.

4.2 Enclave Creation

Creating enclaves is a one-time cost per enclave. The creation of enclaves is done as described in Section 3.3 and can be split into three stages: preparing the enclave memory, setting up the driver and setting up the enclave. Firstly, preparing the enclave memory involves opening an ELF file and reading the contents into user-owned pages. Secondly, setting up the driver involves creating a dedicated character device for each enclave. Finally, setting up the enclave itself involves copying the content of user pages to kernel pages that allow DMA, sending messages to the management shim and setting up the communication channel. Table 4 shows the number of instructions as well as the number of LLC cache accesses in the different phases of creating an enclave. We can see that the driver setup takes the most instructions and cache accesses. This means that we may gain only marginal performance improvements from optimizing the management shim and user API.

4.3 Hardware Area and Performance

Our performance evaluation is implemented on an instruction level simulator, which gives limited information about the hardware costs of our solution. This section calculates the hardware costs based on the design proposal we made and data from previous work.

The main added hardware costs are the added area required for the extra cores and the additional memory required. To put these costs into perspective, we looked at previous enclave systems and how much additional hardware they usually add to each core. We then compare these costs with adding relatively simple cores. These simple cores are placeholders for our secure cores, which we envisage to be small in-order cores with minimal micro-architectural state so that securely context switching between isolation domains is easier. Table 5 summarizes our findings, where the number of

Table 5: Comparison of Enclave Hardware Cost with "Little" Cores

Solution name	Number of thousand gates
AEGIS [29]	205.0
Bastion [6]	30.1
Iso-X [15]	1.0
Sanctum [10]	5.6
Cortex M0 [4]	12.0
Twine [25]	9.4

Table 6: Area of Apple A12 System on Chip Using TSMC N7 Process Node [17]

Block	Area (mm^2)	Percentage of total
Total die	83.27	100.0%
CPU complex	11.90	14.3%
Big core	2.07	2.5%
Little core	0.43	0.5%

gates for previous work is compared to adding a simple core. We can see that the additional hardware needed to allow enclave to run on an existing core is similar to adding a small core that is dedicated to enclaves.

The gate comparisons made in Table 5 are based on research hardware or non-application cores, so to get a better idea of what the area costs would be in a real system, we look at the Apple A12 system on chip, which implements a big-little scheme where bigger high-performance cores are paired with smaller power-efficient cores [17]. The area of the two types of cores is compared to the total die area in Table 6. The CPU complex includes all the cores (2 big and 4 little) as well as hardware shared between cores. The first things to note are that the CPU complex takes only a fraction of the complete die and that the little cores are about a fifth of the size of a big core.

If we assume that our secure cores in Praesidio are similar in size to the little cores and that our fast cores are the big cores, we can get an idea of how much it would cost to add physical isolation to an SoC like the Apple A12. Let us assume that we add 4 little cores for enclaves. The total cost would then be $4 \times 0.43 \approx 1.7$ mm^2. This would increase the CPU complex by $1.72/11.90 \approx 14.5\%$ and the total SoC size by $1.72/83.27 \approx 2.07\%$. This seems acceptable in the era of dark silicon, where we expect most of the die to be turned off due to power constraints [13]. Additionally, this measured overhead is likely to be an overestimate because we expect using even smaller cores for enclaves is better for security.

The same article [17] compares the performance difference between the big and little cores of the Apple A12, which are summarized in Table 7. In essence it shows that for an approximate 4 times slow-down an enclave can experience increased security. Additionally, offloading tasks to smaller cores has the benefits of less energy consumption for the same workload and freeing up the bigger cores for other tasks.

Table 7: SPECInt2006 Performance Comparison Between Apple's A12 Big and Little Cores [17]

Core Type	Performance (SPECSpeed)	Energy (kJ)
Big	44.92	9.521
Little	12.12	3.968
Ratio (Big/Little)	3.71	2.40

4.4 Memory Overhead

We judge that the area and performance overhead due to logic is acceptable, but there is also an overhead incurred by needing additional memory. Memory overhead is caused by the additional tags in the tag directory and the LLC as well as the storage needed for the mailboxes and the management shim.

4.4.1 Enclave Identifier. First, we determine the size of the enclave identifier. If we would like to have a unique identifier for the maximum number of enclaves that can concurrently exist on a system, that is the same as how many pages can exist on the system. RISC-V allows a maximum of 56-bit physical addresses [32] and pages that are equal to or greater than 4 KiB. This means that to uniquely address each page in a system we need at most $56 - 12 = 44$ bits. This is an upper bound because it is dependent on how much physical memory there is on a system and the size of pages.

Another way to estimate the size of an enclave identifier is by how many processes a modern operating system can manage concurrently. On Ubuntu 18.04 the maximum process identifier is $32,768 = 2^{15}$, so each running process can be uniquely identified with a 15-bit identifier. However, process identifiers can be reused, so uniquely identifying all processes that have ever run may require more than that.

In our system we choose an enclave identifier of 32 bits (4 bytes). This identifier fits nicely in between our upper estimate of 44 bits and our lower estimate of 15 bits.

4.4.2 Tag Directory. Next we estimate the cost of storing the tag directory in DRAM. For each page in the system, there are potentially two enclave identifiers: one for the owner and one for the reader. The worst-case size of the tag directory is thus $2 \times \text{idSize} \times \text{numPages}$; we can compare this to the size of DRAM which is $\text{pageSize} \times \text{numPages}$. The DRAM size would thus increase by $\frac{2 \times \text{idSize}}{\text{pageSize}}$. If we assume that pages are 4 KiB, this means that DRAM increases by $\frac{2 \times 4\ \text{B}}{4\ \text{KiB}} = \frac{2 \times 2^2}{2^2 \times 2^{10}} = 2^{-9} \approx 0.2\%$. This is a worst case estimate since we can limit the number of pages that can be tagged.

4.4.3 Last-Level Cache. The memory overhead in the LLC is more critical than in DRAM; this is because the size of tag compared to a cache-line size is more significant than compared to a page size. In our system a cache line is 512 bits [8, 26], and the overhead for the LLC is $\frac{2 \times \text{idSize}}{\text{cacheLineSize}} = \frac{2 \times 32}{512} = 12.5\%$. This is quite significant, but we can decrease this memory cost if we are willing to add extra administrative tasks to the management shim. For example, we can map the 32 bit enclave identifier to a smaller LLC tag, which only has to differentiate between all running enclaves. This tag can be as small as the number of cores in the system. Even if we have

large 32-core or 64-core systems, this would reduce the tag bits to $2 \times \log_2 32 = 10$ and $2 \times \log_2 64 = 12$ and the overhead to about 2%. However, this would also mean that on each enclave context switch the management shim must adjust the tag mapping and invalidate LLC lines.

To minimize context-switch costs, we figure out how many enclaves are likely to be running simultaneously, which is unlikely to be more than the number of processes allowed in the rich operating system. Again, on Ubuntu 18.04 the maximum process identifier is $32,768 = 2^{15}$. To represent this we need 15 bits, which decreases the LLC cost to $\frac{2 \times \text{smallIdSize}}{\text{cacheLineSize}} = \frac{2 \times 15}{512} \approx 6\%$. This requires the management shim to keep track of which LLC tags are mapped to which full enclave identifiers. Costs in the LLC can range from 2% to 13%, and 6% seems like a good middle ground that balances both LLC usage and the frequency of needing to change the tag mappings.

Besides the tag overhead in the LLC, we must calculate the overhead caused by the mailboxes. Since each core has one slot, the memory overhead is $\frac{\text{coreNumber} \times \text{messageSize}}{\text{cacheSize}}$. The message size can be derived from Table 2 by taking the biggest message payload and adding a sender and receiver identifier: messageSize = typeSize + idSize + nonceSize + $2 \times$ idSize $= 8 + 32 + 64 + 2 \times 32 = 168$ bits. Assuming the number of cores is limited to 64, this would make the overhead $64 \times 168 = 10,752$ bits. Compared to the LLC size of 512 KiB $= 512 \times 8 \times 2^{10}$ bits $= 4,194,304$ bits, the LLC overhead for this messaging system is $\frac{10,752}{4,194,304} \approx 0.3\%$. This calculation is a worst-case because it does not account for the smaller tags used in the LLC, and even then the calculated overhead is an order of magnitude less than the overhead of the tags. Thus, the total memory overhead in the LLC is dominated by the tags.

4.4.4 Management Shim. Another source of memory overhead is the size of the management shim. At the moment we put the complete management shim in trusted boot memory, although the shim can also be split into multiple parts using a secure boot process. The management shim is 2.8 KiB and 561 lines of code. This is comparable the 1,600 lines of code of Lee's security monitor [22], which has similar functionality. Assuming that binary size is directly proportional to lines of code, this would make the boot memory between 2 and 8 KiB.

5 SECURITY DISCUSSION

This section assesses the security of the Praesidio system within the context of the threat model described in Section 2. The goal of this paper is to show that physical isolation is a feasible model to protect enclaves from intra-core side-channel attacks, and this section discusses the realism of protecting such a system from these and other important attacks on enclaves.

5.1 Trusted Computing Base

As our threat model defines, Praesidio explicitly trusts certain parts of the system. The following parts are included in the trusted computing base:

- The tag directory, the LLC and the translation of tags between the two.
- The memory interfaces between the fast cores and the LLC.
- The implementation of the secure cores.
- The management shim and securely booting into the boot memory.

The reasons for these parts being trusted are to ensure that access control is enforced (memory interfaces), to guarantee that the tags are preserved across the memory hierarchy (tag directory, LLC and tag translation) and to make sure that changing tags is done in a correct way (management shim). The management shim is also trusted to ensure that enclaves can securely bootstrap with the system and prove this through remote attestation. To do this there are secret platform keys that can only be accessed by the management shim. The secrecy of these keys and the integrity of the management shim is ensured by the tagging system and by the boot process guaranteeing that the management shim is the first code to run on the system. Finally, the hardware implementation of the secure cores is trusted to reliably execute management shim code and provide functionality to securely context switch between enclaves.

In accordance with the threat model there are also parts of the system that are explicitly *excluded* from the trusted computing base in order to protect against the defined attacker model:

- The hardware of the fast cores.
- The privileged software running on fast cores (including operating systems and hypervisors).
- The software of our Linux driver.
- The user applications and user API.
- The enclaves and enclave runtime.

Excluding all of these from the trusted computing base allows engineers the freedom to pursue performance optimizations on fast cores and to create rich features in operating systems without having to consider the enclave threat model.

It is also important that the Linux driver is untrusted because this is the piece of code that includes complex tasks like memory management and scheduling of enclaves. The management shim contains the logic to check whether the actions of the driver are authorized or not, but it does not need to implement all the logic to do the tasks themselves. This is essential to decrease the size of the trusted computing base, which is often used as an indicator of how easy it is to gain confidence in the security of a system.

5.2 Direct Accesses

Directly reading from or writing to protected pages are examples of attacks using direct accesses. Praesidio protects against these attacks by tagging physical pages and enforcing access control using the LLC and the tag directory (see Section 3.1). Additionally, the management shim fills pages with zeroes when deleting enclaves, so there is no way for an attacker to reclaim a page and then read sensitive content.

Another way of performing the same attack is by accessing protected memory through another memory device using DMA requests. For example, an operating system can write a shader for a GPU to access enclave memory. To solve this problem, each memory device must adhere to the tags in the tag directory, and an input-output memory management unit (IOMMU) must enforce tags for removable devices.

5.3 DRAM Bit Leakage

DRAM bit leakage happens because accessing a row causes charge to leak from adjacent rows [27]. It can be used in an active eavesdropping attack to read confidential information like enclave memory [21]. Recent attacks have shown that the current row refresh techniques implemented by DRAM vendors are insufficient to protect against these attacks [16]. As the DRAM industry finds adequate mitigations to these attacks, physically isolating enclaves is an orthogonal technique and will work together with these mitigations in future enclave systems.

5.4 Side Channels

Table 1 is split between intra-core side-channel attacks that require the victim to be running on the same core as the attacker and inter-core side-channel attacks that do not have this requirement. All of the side-channel attacks can be launched by privileged software, usually using timing to measure contention of shared resources.

Intra-core side-channel attacks rely on contention of resources that are local to a core. Sharing a translation look-aside buffer (TLB), for example, allows for contention that leaks memory access patterns [35]. Execution units [2] and hardware accelerators [14] leak information on what operations other threads are doing, while the branch predictor leaks information on control flow [23]. Speculative execution attacks are a special class of attacks that exploit the information leakage caused by processors speculatively executing instructions [5]. Intra-core side-channel attacks are included in our threat model and are the main motivation to explore physical isolation as a defence mechanism.

Physical isolation provides protection against intra-core side-channel attacks by making sure that core resources are not shared between enclaves and attackers. Even though we do not have simultaneous sharing of resources, there is still the opportunity for time-multiplexed sharing of resources because of context switching between different enclaves on secure cores. It is imperative that we scrub all the local micro-architectural state when context switching (see Section 3.2.4).

Inter-core side-channel attacks rely on contention on resources that are shared by different cores like caches and DRAM. In caches, lines and request buffers are shared and cause a timing dependence between cores [10]. In DRAM, row buffers containing the currently open row [27] and request buffers containing a queue of DRAM requests [3] cause timing dependencies between cores. All of these attacks are included in the Praesidio threat model and a memory hierarchy must be carefully designed to ensure isolation.

Since the memory hierarchy is the main resource that is shared between attackers and victims in our system, contention in the memory hierarchy is an important aspect to solve. Contention happens when two applications compete for the same resource, which creates information leakage because it creates a time dependency between the attacker and the victim. As an example, enclave systems have mitigated cache-line contention by using static partitioning schemes [3, 10], and flexible partitioning schemes [11, 31] are good alternatives for applications that heavily use the LLC. Physical isolation can be paired with these cache mitigations as well as with other memory contention mitigations like partitioning memory request buffers [3].

6 RELATED WORK

Praesidio sets itself apart from other enclave systems through physical isolation. Table 1 references a number of previous enclave systems with regard to their protection against software-based side-channel attacks. The physical isolation that is explored in Praesidio composes well with insights from other enclave work like using ring oscillators [29], coloring the cache [10] and purging the pipeline [3]. Physical isolation is a powerful mitigation that contributes to the field of trusted execution environments.

Memory protection in Praesidio is based on tagged memory. This is different from classic enclave systems which either use cryptography to enforce access control [9] or keep track of enclave ownership in a shadow page table [10]. Timber-V uses the idea of tagged memory to protect enclaves but does not consider side channels as a part of the threat model [33]. Praesidio shows how tagged memory can be used in conjunction with physically isolated enclaves to protect enclaves from side-channel attacks. The performance of memory tagging in enclave systems can be further optimized by using existing optimizations like tag caches [18].

Praesidio also introduces a user API to interface with enclaves. Previous enclave APIs are based on the synchronous enclave model, where an enclave runs on the same core as the application [3, 9]. Praesidio provides a similar programming model to the synchronous interfaces by allowing shared memory for communication. The main difference is that enclaves run in parallel to applications. Existing asynchronous interfaces for trusted execution environments, like those based on the Global Platform specification [28], can be implemented on top of the functionality that Praesidio already provides.

7 CONCLUSION

In this paper, we argue that protection against side-channel attacks is an essential part of an enclave threat model. We show that physical isolation, in conjunction with secure context switching, mitigates all known software-based intra-core side-channel attacks. We examine the performance, memory and hardware implications of this design decision using Praesidio. Our evaluation shows acceptable performance cost in three main ways. Firstly, communicating with enclaves causes a similar number of cache accesses as communicating over Unix pipes. Secondly, adding secure cores to a modern system on chip would increase the hardware area by less than 2%. Lastly, storing the extra tag data increases the size of the last-level cache by 2 to 13%. We show that, through physical isolation, an enclave system can be created without changing the implementation of fast application cores and thus minimally affecting the performance on those cores. Physical isolation is a powerful technique to add enclave capabilities to multi-core systems.

ACKNOWLEDGMENTS

Marno van der Maas is a PhD candidate who is funded by a Qualcomm European doctoral fellowship and an EPSRC doctoral studentship. A big thank you to everyone who helped in revising this paper and discussing the ideas at the University of Cambridge.

REFERENCES

[1] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, Vol. 13. Citeseer, ACM, New York, NY, USA, 7.

[2] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: exploiting speculative execution through port contention. *CoRR* abs/1903 (2019), 01843. arXiv:1903.01843 https://arxiv.org/abs/1903.01843

[3] Thomas Bourgeat, Ilia A. Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. 2019. MI6: Secure Enclaves in a Speculative Out-of-Order Processor. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, New York, NY, USA, 42–56. https://doi.org/10.1145/3352460.3358310

[4] S. Bush. 2009. ARM's Cortex-M0 processor – how it works. https://www.electronicsweekly.com/news/products/micros/arms-cortex-m0-processor-how-it-works-2009-03/

[5] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2018. A Systematic Evaluation of Transient Execution Attacks and Defenses. *CoRR* abs/1811 (2018), 05441. arXiv:1811.05441 https://arxiv.org/abs/1811.05441

[6] David Champagne and Ruby B. Lee. 2010. Scalable architectural support for trusted software. In *16th International Conference on High-Performance Computer Architecture (HPCA-16 2010), 9-14 January 2010, Bangalore, India*, Matthew T. Jacob, Chita R. Das, and Pradip Bose (Eds.). IEEE Computer Society, Los Alamitos, CA, USA, 1–12. https://doi.org/10.1109/HPCA.2010.5416657

[7] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. 2011. Linux kernel vulnerabilities: state-of-the-art defenses and open problems. In *APSys '11 Asia Pacific Workshop on Systems, Shanghai, China, July 11-12, 2011*, Haibo Chen, Zheng Zhang, Sue Moon, and Yuanyuan Zhou (Eds.). ACM, New York, NY, USA, 5. https://doi.org/10.1145/2103799.2103805

[8] Intel Coorporation. 2016. Intel 64 and IA-32 architectures optimization reference manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf

[9] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016 (2016), 86. https://eprint.iacr.org/2016/086

[10] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, https://www.usenix.org/, 857–874. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan

[11] Max Doblas, Ioannis-Vatistas Kostalabros, Miquel Moretó, and Carles Hernández. 2020. Enabling Hardware Randomization Across the Cache Hierarchy in Linux-Class Processors. *ISCA 2020* 47th edition (May 2020), 1–7.

[12] Morris J. Dworkin. 2015. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.* Technical Report FIPS PUB 202. National Institute of Standards and Technology, 100 Bureau Drive, Stop 8900, Gaithersburg, MD 20899-8900. https://www.nist.gov/publications/sha-3-standard-permutation-based-hash-and-extendable-output-functions

[13] Hadi Esmaeilzadeh, Emily R. Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *38th International Symposium on Computer Architecture (ISCA 2011), June 4-8, 2011, San Jose, CA, USA*, Ravi Iyer, Qing Yang, and Antonio González (Eds.). ACM, New York, NY, USA, 365–376. https://doi.org/10.1145/2000064.2000108

[14] Dmitry Evtyushkin. 2017. *Secure Program Execution Through Hardware-Supported Isolation.* Ph.D. Dissertation. Graduate School of State University of New York at Binghamton. https://search.proquest.com/docview/2007563667

[15] Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry V. Ponomarev, Nael B. Abu-Ghazaleh, and Ryan Riley. 2014. Iso-X: A Flexible Architecture for Hardware-Managed Isolated Execution. In *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom, December 13-17, 2014*. IEEE Computer Society, Los Alamitos, CA, USA, 190–202. https://doi.org/10.1109/MICRO.2014.25

[16] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. TRRespass: Exploiting the Many Sides of Target Row Refresh. *CoRR* abs/2004 (2020), 01807. arXiv:2004.01807 https://arxiv.org/abs/2004.01807

[17] Andrei Frumusanu. 2018. The iPhone XS & XS Max Review: Unveiling the Silicon Secrets. https://www.anandtech.com/show/13392/the-iphone-xs-xs-max-review-unveiling-the-silicon-secrets/

[18] Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W. Moore, Alex Bradbury, Hongyan Xia, Robert N. M. Watson, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alfredo Mazzinghi, Alex Richardson, Stacey D. Son, and A. Theodore Markettos. 2017. Efficient Tagged Memory. In *2017 IEEE International Conference on Computer Design, ICCD 2017, Boston, MA, USA, November 5-8, 2017*. IEEE Computer Society, Los Alamitos, CA, USA, 641–648. https://doi.org/10.1109/ICCD.2017.112

[19] David Kaplan, Jeremy Powell, and Tom Woller. 2020. *AMD SEV-SNP: Strengthening VM Isolationwith Integrity Protection and More.* Technical Report. Advanced Micro Devices Inc. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf

[20] Cameron F. Kerry, Acting Secretary, and Charles Romine Director. 2013. FIPS PUB 186-4 Digital Signature Standard (DSS). https://csrc.nist.gov/publications/detail/fips/186/4/final

[21] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. 2020. RAMBleed: Reading Bits in Memory Without Accessing Them. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, Los Alamitos, CA, USA, 695–711. https://doi.org/10.1109/SP40000.2020.00020

[22] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: an open framework for architecting trusted execution environments. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo Seltzer (Eds.). ACM, New York, NY, USA, 38:1–38:16. https://doi.org/10.1145/3342195.3387532

[23] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2016. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. *CoRR* abs/1611 (2016), 06952. arXiv:1611.06952 https://arxiv.org/abs/1611.06952

[24] Naiwei Liu, Wanyu Zang, Songqing Chen, Meng Yu, and Ravi Sandhu. 2019. Adaptive Noise Injection against Side-Channel Attacks on ARM Platform. *EAI Endorsed Trans. Security Safety* 6, 19 (2019), e1. https://doi.org/10.4108/eai.25-1-2019.159346

[25] M. Naylor. 2018. POETSII Twine. https://github.com/POETSII/twine

[26] University of California. 2019. The Berkeley Out-of-Order Machine (BOOM). https://docs.boom-core.org/en/latest/sections/intro-overview/boom.html

[27] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, https://www.usenix.org/, 565–581. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/pessl

[28] Global Platform. 2017. TEE Client API Specification Version 1.0. https://globalplatform.org/specs-library/tee-client-api-specification/

[29] G. Edward Suh, Charles W. O'Donnell, and Srinivas Devadas. 2007. Aegis: A Single-Chip Secure Processor. *IEEE Design & Test of Computers* 24, 6 (2007), 570–580. https://doi.org/10.1109/MDT.2007.179

[30] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. 2018. OEI: Operation Execution Integrity for Embedded Devices. *CoRR* abs/1802 (2018), 03462. arXiv:1802.03462 https://arxiv.org/abs/1802.03462

[31] Zhenghong Wang and Ruby B. Lee. 2008. A novel cache architecture with enhanced performance and security. In *41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-41 2008), November 8-12, 2008, Lake Como, Italy*. IEEE Computer Society, Los Alamitos, CA, USA, 83–93. https://doi.org/10.1109/MICRO.2008.4771781

[32] Andrew Waterman and Krste Asanović. 2019. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, v1. 11. https://riscv.org/specifications/privileged-isa/

[33] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. 2019. TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, Reston, VA, USA, 15. https://www.ndss-symposium.org/ndss-paper/timber-v-tag-isolated-memory-bringing-fine-grained-enclaves-to-risc-v/

[34] Nils Wistoff, Moritz Schneider, Frank K. Gürkaynak, Luca Benini, and Gernot Heiser. 2020. Prevention of Microarchitectural Covert Channels on an Open-Source 64-bit RISC-V Core. *CoRR* abs/2005 (2020), 02193. arXiv:2005.02193 https://arxiv.org/abs/2005.02193

[35] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, Los Alamitos, CA, USA, 640–656. https://doi.org/10.1109/SP.2015.45