CAPSTONE: A Capability-based Foundation for Trustless Secure Memory Access

Jason Zhijingcheng Yu, National University of Singapore; Conrad Watt, University of Cambridge; Aditya Badole, Trevor E. Carlson, and Prateek Saxena, National University of Singapore

https://www.usenix.org/conference/usenixsecurity23/presentation/yu-jason

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.
August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.
Abstract

Capability-based memory isolation is a promising new architectural primitive. Software can access low-level memory only via capability handles rather than raw pointers, which provides a natural interface to enforce security restrictions. Existing architectural capability designs such as CHERI provide spatial safety, but fail to extend to other memory models that security-sensitive software designs may desire. In this paper, we propose CAPSTONE, a more expressive architectural capability design that supports multiple existing memory isolation models in a trustless setup, i.e., without relying on trusted software components. We show how CAPSTONE is well-suited for environments where privilege boundaries are fluid (dynamically extensible), memory sharing/delegation are desired both temporally and spatially, and where such needs are to be balanced with availability concerns. CAPSTONE can also be implemented efficiently. We present an implementation sketch and through evaluation show that its overhead is below 50% in common use cases. We also prototype a functional emulator for CAPSTONE and use it to demonstrate the runnable implementations of six real-world memory models without trusted software components: three types of enclave-based TEEs, a thread scheduler, a memory allocator, and Rust-style memory safety—all within the interface of CAPSTONE.

1 Introduction

Hardware isolation primitives for privilege separation play a fundamental role in security designs. Several security extensions to the memory access interfaces provided by commodity processors have been proposed and deployed. For example, trusted execution environments (TEEs) support “enclaves” which are isolated memory regions accessible only to certain user applications, but not to privileged software [11, 27, 34]. TrustZone [1] partitions all software, including privileged software, into separate secure and normal worlds. Similarly, extensions that improve spatial memory safety, e.g., via pointer integrity (e.g., ARM PAC [29, 42]), bounds checks (e.g., Intel MPX [37, 39]), and so on, have emerged.

While each of those extensions is individually promising, ultimately they are each designed for achieving rather specialized and rigid forms of memory access restrictions. They cannot be easily configured for achieving memory protections substantially different from their original intent efficiently. This makes it difficult for hardware architecture designers to pick between security extensions to support natively, which in turn leads to splintering: Different architectures support different security extensions, so software protections cannot rely on the availability of most of them ubiquitously. We therefore ask: Can a hardware-based memory access model enable several existing memory isolation and protection models simultaneously? Such “one for many” memory access models would be a natural solution to splintering.

Two common paradigms for enforcing isolation of memory accesses exist: access control [4, 5] and capabilities [8, 14, 53, 56]. Hardware-based memory protections based on the classical access control paradigm, where a security monitor enforces the access policy (read, write, execute) on every access, are ubiquitous. For example, privilege rings, enclaves, segmentation, virtualization extensions are all based on access checks during address translation via memory management units (MMUs) or memory protection units (MPUs). However, the access control paradigm requires tracking a currently executing authority (e.g., current privilege ring) and an explicit access policy for that authority. In contrast, capability-based designs allow any software to access memory if and only if it presents a capability, an unforgeable token which grants its holder rights to access a specific memory region. A memory region cannot be accessed without a capability, which reinforces the principle of least privilege and reduces ambient authorities [21, 36]. Capabilities do not require explicit per-context access control policies—they are implicit in how accessors transfer capabilities between each other.

CHERI is an example of a capability-based architecture [53]. In CHERI, each capability encodes both the bounds of the memory region to access and the types of permitted
operations on it (e.g., read, write, execute). Software can only create new capabilities from existing ones in a monotonic way. This means newly created capabilities cannot grant access rights beyond those of the currently held ones. CHERI capabilities already provide fine-grained spatial memory safety: All memory accesses are bounded by the capabilities a piece of software holds. This feature is readily useful in software fault isolation or memory bounds-checking [14, 33, 46, 57].

However, CHERI-style capabilities are limited in their power to enforce memory safety in different scenarios. Firstly, software can use such capabilities in the same way as normal pointers, creating aliases in different locations. This can lead to temporal safety violations, wherein code forgets to clean up some capabilities pointing to sensitive data, increasing the chance of a capability leak [2]. Second, delegating capabilities across trust boundaries temporarily is inherently unsafe in CHERI-style capabilities. After delegating a capability temporarily to a component, software has no way to ensure that it no more has access to the memory region, since it can make copies of the received capability. In general, CHERI-style capabilities only allow for irrevocable delegation. Third, CHERI-style capabilities do not directly provide exclusivity, i.e., the holder of a capability is not guaranteed exclusive access to the memory region. However, exclusivity is often needed, for example, in TEEs [1, 11, 27, 34] and for executing critical sections in shared memory systems [54].

Our approach. We present CAPSTONE, a new capability-based low-level memory access interface intended as an instruction set architecture (ISA) extension. CAPSTONE shares the basic notion of capabilities with CHERI, but adds a novel combination of improvements which enable it to support many more memory isolation models at the architectural level—a step towards the goal of avoiding splintering in systems with finer-grained privilege separation.

Consider an abstract model of a capability machine which runs $N$ security domains. Each security domain has a register file that holds data and capabilities, and can choose to pass them to others through shared memory regions. A security domain can also create a new domain by specifying its initial state and supplying the necessary data and capabilities. CAPSTONE provides the following security properties. Firstly, capabilities can be linear$^1$ [19, 46, 49, 53]. Beyond granting memory access permissions like ordinary capabilities, linear capabilities are guaranteed to be alias-free, meaning that no other capability grants a set of memory accesses that overlaps with that of a linear capability. Therefore, a domain that holds a linear capability in a register can be sure that it has exclusive access to the associated memory region. Linear capabilities can be derived from one another through spatial split and merge. Secondly, CAPSTONE enables revocable delegation of capabilities across trust boundaries. If $D$ holds a linear capability and passes it to $E$, $D$ can also choose to revoke this capability at any time. $D$ can be assured that $E$ has no access to the memory associated with the capability immediately after revocation. This prevents $E$ from keeping its access permissions to memory indefinitely, or leaking capabilities by keeping extra copies of them. Thirdly, CAPSTONE supports an extensible hierarchy of privileges. Linear capabilities to regions containing other such capabilities and the above properties hold transitively. Let us say $D$ holds a capability $c_1$ in a register and the memory region corresponding to $c_1$ contains a capability $c_2$, which in turn contains capabilities $c_3$, and so on. $D$ can ensure that it has exclusive access to all memory regions corresponding to $c_1, c_2, \ldots, c_n$, can delegate access to any suffix of the chain of such capabilities and/or immediately revoke access to all delegated capabilities at once if it so desires. This considerably simplifies the management of sharing and delegation of memory, and minimizes the risk of temporal safety bugs or capability leaks.

Applications. We present a proof-of-concept prototype of CAPSTONE consisting of an ISA emulator and a compiler for a language with a C-like syntax. We demonstrate with runnable implementations how CAPSTONE can express multiple memory isolation or protection models without relying on trusted software components. We implemented three different TEE models: spatially-isolated enclaves [11, 27, 34], temporally-isolated enclaves [59], and nested enclaves [43]. A spatially-isolated enclave resembles an Intel SGX enclave [11, 34]: It has a private memory region accessible only to itself, and a public memory region also accessible to the operating system (OS). The boundaries of those regions are fixed upon enclave creation. Temporally-isolated TEEs allow dynamic adjustment of access permissions of memory regions to different enclaves. This provides a means for secure and efficient memory sharing across enclaves [17, 45, 59]. Nested enclaves follow a hierarchical structure, wherein an enclave can create enclaves inside itself and exchange data with its parent enclave through a shared memory region [43]. CAPSTONE supports all those demonstrated TEE models trustlessly, i.e., without involving any trusted software component. Most notably, enclaves do not need to trust the memory allocator or thread scheduler incorporated in our implementations.

CAPSTONE is also useful in non-enclave applications. Recall that CHERI capabilities provide spatial safety directly, but not temporal safety. On CAPSTONE, we show that one can mimic a Rust-style ownership and delegation model to achieve both forms of memory safety. Our implementation enforces such restrictions through the correct use of capabilities during runtime, rather than through static type checking, offering a dynamic alternative for achieving memory safety.

We formally define the operational semantics of CAPSTONE. To analyse its security, we define an abstract model that trivially provides the desired properties (see “Our approach” above), and prove that CAPSTONE refines it and also provides the properties therein. Our main focus is on the superior expressiveness of CAPSTONE to support several desirable

---

$^1$Linear as in “linear type systems” and “linear logic”.
memory isolation models at once as compared to CHERI.

Implementation and evaluation. In order to show that CAPSTONE can be implemented with acceptable performance impact, we describe a sketch of a potential implementation, model its performance with gem5 [6], and evaluate it on the SPEC CPU 2017 intspeed benchmarks [7]. The results suggest that overall performance overhead of CAPSTONE over a traditional system is within 50%. A full hardware implementation requires additional design decisions and remains promising future work beyond the scope of this paper.

Contributions. We present CAPSTONE, the first architectural capability design that provides exclusivity, delegation, and revocation simultaneously for hardware-isolated memory. CAPSTONE enables richer memory models demanded by security applications with a single set of interfaces than prior capability-based systems.

2 Overview

Existing memory isolation models require specialized architectural support. While this enables efficient implementations ultimately, the plurality of such models has led to splintering. CAPSTONE is an effort towards finding abstractions expressive enough to be configured to support multiple useful existing isolation models without increasing the software trusted computing base (TCB).

2.1 Architectural Capabilities

A capability is a token that grants its holder memory access permissions. It typically contains the bounds of the accessible memory locations as well as the allowed access types (i.e., read, write, execute). Software presents a capability every time it needs to make a memory access. The hardware then performs checks on the memory access to make sure that it falls in the allowed bounds and is of an allowed type of the capability. Whenever the memory access is found to violate the restrictions, the hardware refuses to fulfill the memory access. Capabilities are unforgeable—software can only derive new capabilities from existing ones through a well-defined set of operations. For example, software cannot directly cast an integer into a capability. Implementations of capability-based architectures like CHERI enforce this through memory tagging [8, 14, 53] by marking memory locations and registers containing capabilities with tags that are hidden from software. The operations that create capabilities out of existing ones are monotonic, i.e., new capabilities cannot allow accesses disallowed by the original ones. This prevents privilege escalation through direct operations on a capability.

Compared to identity-based access control mechanisms, capabilities have the advantage of not relying on complex central policies, and can yield greater expressiveness. As an example, CHERI [13, 53] has been shown to enable fine-grained software compartmentalization and spatial memory safety in C/C++. However, it does not provide architectural support for temporal memory safety. To enable temporal memory safety for C/C++, for example, traditional software-based techniques such as reference counting and garbage collection must be used in conjunction [9]. For several other application scenarios, extensions to CHERI that require specialized hardware changes exist. Such examples include StkTokens [46], which enables a calling convention that guarantees well-bracketed control flow in software fault isolation, and CHERIvoke [58] and Cornucopia [55], which mitigate use-after-reallocation of heap memory for C code.

2.2 Motivating Examples

Many memory isolation models are useful in the real world but are not supportable with CHERI, motivating our work.

Trustless memory allocation. One important task of the OS and the VMM is the allocation of physical memory. Traditionally, an application has to trust privileged code when using the allocated memory. Achieving trustless memory allocation requires considering two aspects. Firstly, the application that receives an allocated memory region from the memory allocator should not trust that it will not access the memory region or delegate it to another application in the future. Secondly, the memory allocator should not overly trust applications, which may refuse to relinquish access to memory regions.

CHERI is unable to achieve trustless memory allocation. When the application receives a capability from the memory allocator, it cannot ensure that no other software component, including the memory allocator, also has access to the allocated memory region. Likewise, the memory allocator cannot be sure that the application has relinquished the capability when it wishes to reclaim it. The memory allocator needs to trust that the application has not kept or leaked copies of the reclaimed capability. As a result, both the allocator and the application would need to trust each other.

Trustless preemptive scheduling. Modern systems widely rely on preemptive scheduling to multiplex multiple domains (e.g., processes) on limited CPU resources. Preemptive scheduling relies on preempting (i.e., interrupting) the execution of a domain through timer interrupts. A scheduler then handles each interrupt and decides which domain to execute next. The scheduler is normally part of an OS and has the privilege to arbitrarily access domain execution states. On the contrary, trustless preemptive scheduling enforces the principle of least privilege and provides applications with the assurance that the scheduler is not capable of doing anything more than deciding when to execute each domain. This effectively removes the scheduler from the TCB of an application.

On CHERI, an exception or interrupt on a thread diverts the control flow to an exception handler, which can then perform scheduling. However, the execution context of the interrupted thread, including all the capabilities stored in registers, is
directly accessible to the exception handler. This gives the scheduler the ability to modify the content of the execution context of a domain (e.g., register values), for example, to hijack the domain control flow. The scheduler can also duplicate the execution context and force application code that is not designed to be thread-safe to interleave on multiple threads on shared memory regions through capabilities duplicated as part of the execution context. Therefore, CHERI requires the application to fully trust the scheduler.

**Trusted execution environments.** Traditionally, software such as OS kernels is assumed to be trustworthy and runs with high privileges. However, the growing complexity of privileged software and the increasing demand for secure remote execution have rendered this assumption increasingly unjustifiable. Trusted execution environments (TEEs) provide a promising solution: They support running security-sensitive software without requiring it to trust any other software on the system, including privileged software such as the OS. Most TEEs follow a *spatial isolation model*, where each secure application receives a private memory region called an enclave at its launch time. An enclave stores both the code and the private data of the application, and is accessible only to it. The remaining part of the memory, called the public memory and accessible to both the application and the OS, enables data exchange between them (e.g., to support system calls). Variations of the enclave TEE model exist. In the nested enclave model [43], for example, an application in a nested enclave has access to its parent enclave in the same way as an application in a top-level enclave has access to a public memory shared with the OS. Another variant, Elasticlave [59], supports temporal isolation, where each application can set time-varying access policies for its memory regions for sharing them in a controlled way. Both the nested enclave and the Elasticlave models enable greater flexibility and a wider range of application scenarios than spatial isolation.

CHERI does not support any of the above-mentioned TEE models, as it does not guarantee exclusive access for applications. Any memory region an application can access through a capability, even one intended as an enclave private memory region, can potentially be accessible to other software components as well, by passing them copies of the same capability. **Rust-like memory restrictions.** Many applications involve sharing memory across software components. An example is a Linux process sharing a buffer with the kernel in order to read data from a file. In such cases, it is important to maintain spatial and temporal safety of memory access across domains, and violations can lead to serious consequences. Memory-safe abstractions are one answer to this problem. Rust [32], a programming language that provides a memory model with spatial and temporal safety is an example of such models. However, abstractions which rely on static enforcement by compilers require that software components be written in specific languages and require additional trust assumptions, i.e., that components trust each other to have used a correct compiler implementation without taking unsafe shortcuts.

Rust enforces memory safety with the notions of ownership and lifetime. Rust programs access data objects through their references. Though an object can have more than one reference, exactly one of them is its owner, while the others are all borrowed references. Hence, the owner reference of an object can only be moved, but not duplicated. The owner’s lifetime is tied to that of the object, and a borrowed reference cannot outlive the owner. This makes sure that no reference to an object can exist after the object is destroyed (i.e., when the owner’s lifetime ends). It also implicitly guarantees that access to an object will be exclusive to the owner again after the borrowed references are destroyed.

The CHERI capability interface, however, is not expressive enough to directly enable Rust-style memory restrictions at the architectural level. Rust involves different types of references and imposes different restrictions on them. CHERI, on the other hand, provides only one type of capabilities. It also does not provide revocable delegation, as exemplified by borrowed references in Rust. Once a domain delegates certain memory access to another domain, there is no guarantee that it can get back exclusive memory access at a future point.

### 2.3 Capstone in a Nutshell

We design a new memory access interface called **Capstone** that can express the memory restrictions needed by the memory models discussed in Section 2.2. **Capstone** uses capabilities for memory access control in the physical address space. As an architecture-level interface, it is intended to be implemented in the processor, with capabilities replacing raw memory addresses. The processor enforces memory protection guarantees at runtime without assuming trusted software components. Capstone does not rely on assumptions regarding the MMU on a system by directly working with physical addresses instead of virtual ones. In the future, this could eventually enable greater flexibility and compatibility with isolation models not built on MMUs or MPUs (e.g., ARM MPU [38], RISC-V PMP [51]).

On top of the original capabilities from CHERI, **Capstone** adds new capability types with the following properties:

- **(P1) Linearity.** Domains can have exclusive access to memory regions. When a domain D holds a linear capability to a memory region, no other domain can access the region.

- **(P2) Delegation and revocation.** When a domain D holds a linear capability L, D may choose to transfer L to another domain E. Moreover, D may later choose to reclaim exclusive ownership of L, even if E has in turn transferred the capability to another domain. To protect the potentially secret data that E may have placed in locations accessed through L, when D regains ownership of L, the memory region corresponding to L will become unreadable to D until D overwrites it.

- **(P3) Dynamically extensible hierarchy.** A domain D can...
Table 1: Properties required or present in each model.

<table>
<thead>
<tr>
<th>Model</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rust-like abstraction [32]</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Spatially-isolated enclaves [11, 34]</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
</tr>
<tr>
<td>Temporally-isolated enclaves [59]</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>Nested enclaves [43]</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
</tr>
<tr>
<td>Trustless memory allocation</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Trustless thread scheduling</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
<tr>
<td>CHERI [53]</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>CAPSTONE (this work)</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

create another domain $E$ that is subordinate to it, in the sense that $D$ can choose to revoke any capability that $E$ holds at any time, and once this is done, $E$ cannot get back the revoked capability without $D$’s cooperation. Such a hierarchy is dynamically defined by the runtime behaviour of each security domain and can be indefinitely extended on demand.

(P4) Safe domain switching. If at a certain moment the physical thread executing a domain $D$ switches to a different domain (e.g., due to an exception/interrupt, or when calling into another domain), and $D$’s context (register file content) is $C$, then the next time $D$ is executed, its context is still $C$.

Table 1 lists the properties the example models require.

CAPSTONE abstract model & security. To capture our security properties more precisely, we define an abstract model called CAPSTONE$_\text{abs}$. Its state is defined in terms of an abstract memory store, where memory cells may be marked uninit to capture that reading them would be a security violation, together with a user domain executing in a two-part environment composed of the superordinate and the subordinate environment domains (tstate$_\text{sup}$ and tstate$_\text{sub}$). The superordinate environment represents other domains which may revoke the user domain’s capabilities arbitrarily. The subordinate environment represents domains which are guaranteed to never revoke the user domain’s linear capabilities. The user domain and the two environments each track the memory accessible to them through the capabilities they currently own, and can perform actions (act$_\text{abs}$) to manipulate these capabilities or update the memory. CAPSTONE$_\text{abs}$ directly enforces desirable properties of CAPSTONE, and we characterize CAPSTONE’s security as a standard refinement theorem:

Main theorem. CAPSTONE refines CAPSTONE$_\text{abs}$.

We discuss CAPSTONE$_\text{abs}$ and the proof of the main theorem in Section 5.

2.4 Threat Model and Scope

We assume that the security domains do not trust one another. We focus on the security of one of the security domains, and assume that the attacker can control any other domain on the system, including those in charge of managing system resources (e.g., thread scheduler, heap memory allocator, and so on). The domain of interest, on the other hand, is assumed to be benign and bug-free. We are also interested in denial-of-service (DoS) attacks from an application which attempts to hold memory resources indefinitely and thereby reject them to OS components in charge of memory management. DoS attacks from the OS against applications are out of scope.

3 Design Overview

A CAPSTONE machine runs multiple security domains multiplexed on a set of physical threads which share register files but share the physical memory (and the physical address space), as shown in Figure 1. At any given point in time, each thread is running exactly one security domain, and each security domain is running on at most one physical thread. Certain events (e.g., exceptions) can trigger a thread to switch between security domains. When a security domain is running on a physical thread, we refer to the register file content of the thread as the context of the security domain. For a security domain that is not currently running on any physical thread, we define its context as the context it will have when it next starts running. Such a context is physically stored inside a memory region, from which the content is loaded into the register file of a thread when it loads the security domain.

Similar to CHERI, CAPSTONE is an instruction set architecture (ISA) based on capabilities. For any memory access by any security domain, CAPSTONE mandates that a capability granting this memory access must be provided. Each domain context can hold capabilities that grant the domain access to memory regions, which can in turn hold more capabilities and hence grant access to yet more memory regions. To overcome the limitations of existing capability-based architectures (Section 2.2), CAPSTONE incorporates extra capability types and capabilities-related operations beyond those already present in CHERI. Some capability-related operations involve changing capability types. Figure 2 overviews the capability types and the operations. We describe them in more details below.

Linear capabilities. Central to CAPSTONE is the additional
A linear capability grants access to memory locations in the same way as an ordinary capability, but instead of only guaranteeing that certain memory accesses are allowed, linear capabilities also assure that certain accesses are disallowed. This is because linear capabilities are alias-free. Holding a linear capability not only gives a domain certain access permissions to the memory region, but is also sufficient to guarantee that access to the region is exclusive to the domain alone. This does not assume any trust in software. To maintain the alias-free property of linear capabilities, any operation in CAPSTONE that would otherwise lead to overlap between input and output capabilities will consume (i.e., invalidate) the input capabilities. For example, software can only move, but not copy linear capabilities. A linear capability does not need to be in a domain context to guarantee exclusive access. For example, when a domain holds in its context a linear capability \( c_a \) for the memory region \( R_a \), and inside \( R_a \) resides another linear capability \( c_r \) for another memory region \( R_r \), then besides \( R_a \), the domain also has exclusive access to \( R_r \). In general, exclusive access through linear capabilities can be chained indefinitely. As shown in Figure 3, when a domain can reach a memory region through a linear capability kept directly in its context (register file) (Figure 3(a)), or through a chain of linear capabilities (Figure 3(b)), its access to the memory region is guaranteed to be exclusive. On the other hand, exclusive access is not guaranteed if the domain has to involve a non-linear capability

\[ c_{lin} \text{ mint rev} \quad c_{lin} \]

\[ c_{ininit} \text{ revoke}
\]

Figure 4: Overview of the operations on revocation capabilities. Strikethrough capabilities are invalid. When \( c_{lin} \) has derived non-linear capabilities only at \( t_3 \), revocation converts \( r_c \) into a linear capability \( c_{lin}' \) (scenario A). Otherwise, \( r_c \) is converted into an uninitialized capability \( c_{uninit} \) (scenario B).

**Revocation.** CAPSTONE includes the notion of revocation capabilities. A revocation capability does not grant memory access permissions, but serves as a token for revoking all capabilities that overlap with it. As demonstrated in Figure 4, a domain can create a revocation capability only for a linear capability it currently holds (\( t_0 \) to \( t_1 \) in Figure 4, and “mint rev” in Figure 2). Creating a revocation capability does not consume the given linear capability. This does not violate its alias-free property, as the revocation capability conveys no memory access permissions. As such, the revocation capability serves as a basis for revocable delegation. Before a domain \( D \) passes a linear capability to another domain \( E \), it creates a revocation capability for the linear capability, which it later uses to revoke the delegated capability, regardless of what \( E \) has done. In order for \( D \) to reclaim exclusive access to the memory region, CAPSTONE converts the revocation capability into a corresponding capability that grants access permissions in the revocation operation (\( t_2 \) and \( t_3 \) in scenario A in Figure 4, and “revoke” in Figure 2). Since linear or non-linear capabilities that overlap with the memory region are all revoked, the new capability does not violate the alias-free property. On some occasions, CAPSTONE converts the revocation capability into an uninitialized capability instead of a linear capability to prevent secret leakage. For example, when \( D \) reclaims exclusive access, the memory region possibly holds \( E \)’s secret data. CAPSTONE identifies such situations by checking whether a linear capability has been revoked during the revocation process, which indicates a domain is still interested in maintaining its exclusive access (\( t_2 \) and \( t_3 \) in scenario B in Figure 4). An uninitialized capability represents a memory region whose content should be unavailable until written (hence effectively uninitialized). Correspondingly, an uninitialized capability only grants write access, but can be converted to a linear capability when all locations inside the region have been written at least once with it (“initialize” in Figure 2). CAPSTONE thus prevents the domain that reclaims access to the memory region from reading its original content.

---

\( ^2 \)Moving a linear capability from location A to B destroys the copy in A.
Splitting and merging. CAPSTONE allows spatially splitting a linear capability into two non-overlapping linear capabilities. Since this operation is entirely monotonic, a reverse process is needed to merge linear capabilities and stop linear capabilities from becoming increasingly fragmented. However, it is infeasible to simply allow any two linear capabilities for adjacent regions and identical permissions to be merged. Consider the scenario where a linear capability \( c_1 \) has a corresponding revocation capability \( r_{c_1} \), and is later split into two linear capabilities \( c_{11} \) and \( c_{12} \). Another linear capability \( c_2 \) neighbours \( c_{12} \) in terms of their memory regions, and has identical permissions as \( c_{12} \). If under this condition we merge \( c_2 \) and \( c_{12} \) into a new linear capability \( c_3 \), problems will arise when \( r_{c_1} \) is used to perform revocation: On the one hand, \( c_3 \) overlaps with \( r_{c_1} \), so it should be revoked; on the other, \( c_3 \) is not entirely covered by the memory region associated with \( r_{c_1} \), and if it is revoked, the \( c_2 \) part of \( c_3 \) will be lost. This introduces significant complexity in capability management and poses challenges to both implementations and applications. As a result, CAPSTONE avoids such arbitrary merging, and instead relies on the semantics of revocation for the reverse operation of splitting. Before splitting a linear capability, a domain creates a revocation capability, and later uses it to revoke the capabilities that result from this split as well as reclaim the original linear capability (Figure 5). In the example, \( c_{12} \) can only be merged with \( c_{11} \) to reverse the split of \( c_1 \), by performing revocation using \( r_{c_1} \). This enables a limited form of merging that reverses past splits, which we consider as a reasonable compromise between complexity and utility. Revocation capabilities are used to reverse other types of operations as well: for example, tightening capability permissions and delinearizing linear capabilities (i.e., converting them into non-linear capabilities, shown as “delinearize” in Figure 2).

Implicit extensible hierarchy. CAPSTONE provides an implicit and extensible hierarchy through revocation capabilities. Revocation capabilities can overlap with one another. For example, after a domain \( D \) creates a revocation capability \( r_{c_1} \) for the linear capability \( c \), it passes \( c \) to another domain \( E \), which in turn creates another revocation capability \( r_{c_2} \) for \( c \). \( E \) may pass \( c \) further to a third domain \( F \) while retaining \( r_{c_2} \) so it can later revoke \( F \)’s access permissions (\( t_0 \) to \( t_1 \) in Figure 6). In such a case, \( r_{c_1} \) and \( r_{c_2} \) should not be considered as identical. If \( r_{c_1} \) is used to perform revocation, \( r_{c_2} \) should be revoked, or \( E \) will be able to regain \( c \) through \( r_{c_2} \) afterwards, whereas if \( r_{c_2} \) is used for revocation first (\( t_2 \) in Figure 6), \( r_{c_1} \) should remain valid, so that \( D \) can still revoke \( E \)’s access regardless of this event (\( t_3 \) in Figure 6). CAPSTONE deals with such situations by assigning each revocation capability with a different strength based on seniority. The earlier a revocation capability was created, the stronger it is. A revocation operation with a revocation capability \( r \) invalidates all other weaker and overlapping revocation capabilities. Note that such other revocation capabilities can only point to spatial sub-regions (including the identical region) of the memory region that \( r \) points to, because creating a revocation capability requires a valid linear capability. Such a hierarchy of revocation strengths is implicit in how security domains delegate linear capabilities and is indefinitely extensible. A linear capability can be passed indefinitely many times across a sequence of domains. Each domain can always “roll back” passing a linear capability to the next domain, regardless of the behaviours of the domains further down the sequence.

Capability-based designs have the advantage that they can work without access control policies written to be enforced by security monitors. It frees us from defining access control policies upfront. In contrast, when capabilities are passed from program context to context, say from one process to another, they implicitly carry with it the semantics that the sender context wishes to allow the recipient access to the object. This implicit capability-passing is a form of delegation without explicit intervention or access control decisions being made. This also means that if we design capability-based models, we do not need to define privilege levels explicitly.

Safe domain switching. CAPSTONE supports safe domain switching with the help of sealed capabilities, which are present also in CHERI [53]. Similar to revocation capabilities, sealed capabilities do not grant direct memory access. Instead, a sealed capability represents the context of a security domain that is not currently running. The memory region associated with a sealed capability stores the content of a domain context. A linear capability can be converted into a sealed capability (“seal” in Figure 2), which in effect creates a security domain with a specified context. A domain can use the “call” operation on a sealed capability to switch the current physical thread to the corresponding domain. A similar operation is “return”, which also switches to a specified domain context stored inside a memory region, but is intended as the reverse
of “call”. CAPSTONE accounts for the semantic difference between “call” and “return” with a separate sealed-return capability type, which is generated in the “call” operation as shown in Figure 2. Unlike CHERI, CAPSTONE extends sealed capabilities to exception handling as well, guaranteeing that the exception handler cannot arbitrarily access the execution context of an interrupted domain. Furthermore, in CAPSTONE, sealed capabilities are linear (i.e., guaranteeably alias-free), which ties access to the stored resources behind a sealed capability to the domain. This also guarantees that only one instance of a domain exists at any time, effectively preventing potentially unsafe re-entries into the same domain.

**Alias-free capability types.** Linear capabilities are not the only capabilities with alias-free guarantees. As Figure 2 shows, sealed, sealed-return, and uninitialized capabilities are also alias-free, allowed to overlap only with revocation capabilities, whereas revocation capabilities can overlap with any capability of any type, and non-linear capabilities can overlap with other non-linear capabilities and revocation capabilities.

**Software stack.** Upon a system reset, the register file is initialized to contain capabilities that cover the full physical memory. The first piece of code to execute can then bootstrap other domains and delegate to them parts of the physical memory as linear capabilities. Multiple paths henceforth are worth exploring, from adapting a monolithic kernel such as Linux to creating a new microkernel-based software stack. Detailed software stack design is future work.

**Supporting memory protection models.** The design delineated above enables CAPSTONE to provide the desired properties discussed in Section 2.3, which as summarized in Table 1, are required to support the example memory protection models but are missing in CHERI. We describe more details on how to implement those models on CAPSTONE in Section 8.

### 4 CAPSTONE Formal Model

#### 4.1 Overview

Figure 7 defines the entities CAPSTONE involves.

**CAPSTONE machine.** The execution of a CAPSTONE machine consists of a sequence of steps. At each step, the CAPSTONE machine is in a state \( \Psi \) consisting of the states of physical threads \( \Theta \), physical memory state \( \text{mem} \), and additional capability-related data structures. Each physical thread has a distinct register file which includes the program counter pc, special registers \( \text{ret} \) and \( \text{epc} \), and \( M \) general-purpose registers \( r_1, r_2, \ldots, r_M \). By indexing \( \Theta \), we can obtain the state \( \theta \) of a specific physical thread, including its register file contents. Each register or memory location contains either a raw scalar value or a capability, referred to as a word collectively.

At each step, the machine picks any one of the threads and executes an instruction on it. The instruction can be either the one stored in the physical memory under the cursor of the capability held by \( pc \), or except, a special instruction that helps model an exception or interrupt. Executing an instruction changes the machine state. We represent the machine state immediately after executing instruction \( i \) on thread \( k \) at the machine state \( \Psi \) as Execute(\( \Psi, k, i \)).

**Capabilities.** Each capability is a tuple \( c = (t, b, e, a, p, n) \), where \( b \) and \( e \) are the base and end addresses of the memory region respectively, \( p \) is the access permissions granted by the capability, and \( t \) identifies the capability type. CAPSTONE includes new capability types in addition to the normal non-linear capability (denoted as Non). We follow CHERI [53] to include in each capability the address for the next memory access, called its cursor and denoted as \( a \). Information useful for capability revocation is recorded in \( n \).

**Operational semantics.** We discuss the semantics of each instruction in turn below. The instructions jmp,jnz, li,add, and It are omitted, as they are almost identical to their counterparts in common existing architectures. Due to the limited space, we refer interested readers to the extended version of this paper [60] for the full definition of the state transitions.

#### 4.2 Moving Capabilities

As in traditional architectures (e.g., RISC-V [51]), ld and sd perform memory load and store operations, but require a capability rather than a raw address. If the provided capability is valid, ld and sd perform the operations on its cursor. Furthermore, if the data word transferred is a capability that is linear, the original copy, be it in a register (as in sd and mov) or memory (as in ld), will be cleared to zero. Formally, let \( w \) be the data word, mov, ld, and sd set the content of the source
4.3 Capability Revocation

**Revocation.** We model the `revoke` instruction using the revocation tree. The root of the revocation tree is a special node `root`. Each valid capability `c`, regardless of its type, maps to a node with the index `c.n` (we will use an index to refer to a node for simplicity) in the revocation tree, whereas each revoked capability maps to one outside the revocation tree (i.e., disconnected from `root`). Using `revoke` on a revocation capability `r` re-parents all the children of `r.n` to `null`, cutting the subtree off the revocation tree and thus invalidating the nodes inside. Meanwhile, the type of `r` is changed to `Lin` (linear) if the only capabilities that map to nodes in the subtree are non-linear, or `Uninit` (uninitialized) otherwise.

**Creation of revocation capabilities.** The `mrev` (“mint revocation”) instruction creates a revocation capability from a linear capability `c`. The resulting revocation capability `r` receives the same region bound and access permission set in `c`. Meanwhile, a new node is created in the revocation tree for `r` between `c.n` and its parent. Since `c.n` is in the subtree of `r.n`, `c` is revoked in the process of `revoke` `r`. This process naturally captures the hierarchical strengths of revocation capabilities. Consider the example where `r_1` and `r_2` are both created using `mrev` on the same linear capability `c`, and `r_2` is created after `r_1`. In this case, `r_1.n` will be the parent of `r_2.n`, which is in turn the parent of `c.n`. Using `revoke` on `r_1` will therefore revoke both `r_2` and `c`, whereas using it on `r_2` will only revoke `c`.

**Uninitialized capabilities.** Uninitialized capabilities always grant only write memory access regardless of the permissions recorded in them. An uninitialized capability newly generated by `revoke` always has its cursor set to its base address. Every subsequent write made with the uninitialized capability increments its cursor by one word position. The cursor effectively marks the boundary of the already initialized part of the memory region. APSTONE provides `init` to convert an uninitialized capability whose cursor has reached its end address (and thus fully initialized) to a linear capability which inherits both its memory region and access permissions.

**Capability dropping.** The `drop` instruction directly invalidates a given capability. By invoking `drop` on a linear capability, a domain effectively informs the APSTONE implementation that it is not interested in the memory region any more. This can prevent `revoke` from producing an unintended uninitialized capability. Invoking `drop` on a capability `c` (which is not non-linear) removes the node `c.n` from the revocation tree. The children of `c.n`, if any, will be adopted by its parent.

4.4 Capability Modification

Instructions `tighten`, `shrink` and `split` modify a given capability without changing its type. The `tighten` instruction changes the memory access permissions to a more restrictive subset. The `shrink` instruction sets the region bound to a specified bound fully covered by the original one.

The `split` instruction splits a capability `c` into two at the specified address `s`. Let `b` and `e` be the base and end addresses of the given capability with `b < s < e`, then the two resulting capabilities `c_1`, `c_2` will have base and end addresses `b_1 = b`, `e_1 = s` and `b_2 = s`, `e_2 = e` respectively. The original capability becomes unavailable. Meanwhile, the node `c.n` is also split in two, both inheriting the original parent. The `delin` instruction converts a linear capability into a non-linear capability by simply changing its type to `Non`. For all but sealed (including sealed-return) and uninitialized capabilities, `scx` sets the cursor to a give address. Another instruction, `lcc`, returns the cursor of a given capability.

4.5 Domain Switching

**Sealing.** The `seal` instruction converts a given linear capability to a sealed capability. The memory region associated with a sealed capability contains the context of a security domain that is not currently running. An application can prepare desired contents using a linear capability, and then `seal` it into a sealed capability. In this way, the application has effectively created a new domain with a specific initial context.

A sealed capability does not grant direct memory access. Rather, it needs to be `unsealed` into the register file of a physical thread either through the `call` instruction or as the result of an exception or interrupt, which effectively switches the physical thread to the sealed domain.

**Synchronous domain calls.** When a security domain `D` holds a sealed capability `c_E` for another security domain `E`. `D` can use `call` on `c_E` to switch the current physical thread to `E`. This unseals `E`’s context from `c_E` into the register file of the current physical thread, while sealing `D`’s context (i.e., the current content of the register file) to `E`’s associated region. The `call` instruction writes a sealed-return capability `c_r` for the same region as `c_E` to `E`’s `ret` register, so `E` is able to return to `D` later. To facilitate communication between `D` and `E`, the register `r_1` is reserved for argument passing. In other words, a second operand to `call` is directly loaded into `E`’s `r_1` register. To return to `D`, `E` invokes `return` on `c_r`. The `return` instruction is similar to `call`, except that it does not save the current context on the physical thread. Note that `c_E` has been destroyed in `D`’s context during `call`. `E` can specify a value to `return` to replace `c_E` with when returning to `D`. A variant to `return`, `retseal`, replaces `c_E` with `E`’s current context with the pc cursor switched to a specified value. This is useful for allowing `D` to invoke `E` multiple times, each time with a controlled and potentially different initial context.
dom \quad ::= \quad \text{user} \mid \text{sup} \mid \text{sub} \\
mem_{\text{abs}} \quad ::= \quad \text{Addr} \rightarrow (\text{Word} \mid \text{uninit}) \\
ranges \quad ::= \quad \{ n \mid x \leq n < x + y \} \\
cap_{\text{abs}} \quad ::= \quad \text{range} \\
tstate \quad ::= \quad \text{cap}_{\text{abs}} \mid \text{set} \\
ptstate \quad ::= \quad (\text{mem}_{\text{abs}}, \text{tstate}_{\text{user}}, \text{tstate}_{\text{sup}}, \text{tstate}_{\text{sub}})

Figure 8: CAPSTONE_{\text{abs}} state.

\text{Refines}_{d,D_{\text{sub}}}(\Psi, (\text{mem}_{\text{abs}}, \text{tstate}_{\text{user}}, \text{tstate}_{\text{sup}}, \text{tstate}_{\text{sub}})) \\
\quad \text{where:} \\
\quad \text{ind}_{\text{uninit}} = \bigcup_{d \in D_{\text{sub}}} \text{woranges}(\mathcal{R}_{w}(\Psi, d)) \\
\quad \text{mem}_{\text{abs}} = \Psi . \text{mem}[\text{ind}_{\text{uninit}} := \text{uninit}] \\
\quad \text{tstate}_{\text{user}} = \text{ranges}(\mathcal{X}(\Psi, d)) \\
\quad \text{tstate}_{\text{sup}} = \bigcup_{d' \in D_{\text{sup}}} \text{ranges}(\mathcal{X}(\Psi, d')) \\
\quad \text{tstate}_{\text{sub}} = \bigcup_{d' \in (N - D_{\text{sub}} - d)} \text{ranges}(\mathcal{X}(\Psi, d'))

Figure 9: CAPSTONE_{\text{abs}} refinement mapping.

Interrupt and exception handling. CAPSTONE uses the same mechanism for except as for call. When an interrupt or exception occurs, the current physical thread switches to a handler domain defined in the epc register of the current security domain. An interrupt or exception is hence essentially an asynchronous call on epc. The register receives the following special treatments for its role in system management:

Pinned per-thread. Except for interrupt or exception handling, epc is excluded in the part of the execution state replaced during domain switching. The end result is that the epc value is per-thread instead of per-domain;

Immutable. Unless the current value is unset, epc is immutable. In other words, epc is fixed upon first write.

5 Security Analysis

Due to the space limit, we only briefly overview the security proof. Full details are available in the extended version [60].

We define an abstract model, CAPSTONE_{\text{abs}}, and prove that CAPSTONE refines it (main theorem). The state of CAPSTONE_{\text{abs}} is defined in Figure 8. Figure 9 defines a refinement mapping between a concrete state of CAPSTONE and an abstract state of CAPSTONE_{\text{abs}} with respect to a distinct domain d which serves as the user domain, and a set of domains D_{\text{sub}} which serves as the subordinate environment.

We show that CAPSTONE’s use of uninitialized capabilities refines CAPSTONE_{\text{abs}}’s use of uninitialized memory values to denote memory which cannot be accessed. To this end, the line \text{ind}_{\text{uninit}} = \bigcup_{d \in D_{\text{sub}}} \text{woranges}(\mathcal{R}_{w}(\Psi, d)) collects the ranges of all write-only (uninitialized) capabilities. The line mem_{\text{abs}} = \Psi . \text{mem}[\text{ind}_{\text{uninit}} := \text{uninit}] indicates that the abstract memory is the same as the concrete memory, except that indices for uninitialized capabilities are set to uninit.

In the abstract model, domains are represented as sets of abstract capabilities, each being a simple range of accessible addresses. The definition \mathcal{X}(\Psi, d) is the exclusive realm of d, that is, the set of all concrete linear capabilities exclusively accessible (transitively) in the domain d. The abstract state of the user domain (tstate_{\text{user}}) is defined as the set of ranges corresponding to the capabilities in the exclusive realm of d. The abstract states of the subordinate and superordinate domains are similarly defined.

We complete the proof by showing that the refinement mapping is preserved by execution of the concrete model, and that steps in the concrete model can be mapped to zero or more abstract actions in the abstract model.

6 Implementation

We show that CAPSTONE can be implemented with acceptable overhead. Since a complete RTL implementation requires significant engineering effort, we consider it as future work beyond the scope of this paper. Instead, we present a sketch of a potential implementation below, and evaluate it in Section 7.

Capabilities. We represent each capability as 128 bits in registers and memory as follows:

<table>
<thead>
<tr>
<th>node-id</th>
<th>type</th>
<th>perm</th>
<th>bounds</th>
<th>cursor</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>3</td>
<td>5</td>
<td>27</td>
<td>64</td>
</tr>
</tbody>
</table>

The bounds field encodes the capability address range following the CHERI Concentrated scheme [53] which compresses such information into 27 bits. The type and perm fields indicate the type and associated permissions of each capability, and the identifier of the associated revocation node of a capability is recorded in node-id. Each general-purpose register is extended to 16 bytes to allow it to hold a capability. To distinguish normal data from capabilities, we follow the implementation of CHERI [53] to store a separate tag bit for each register as well as every 16 bytes-aligned location in DRAM. Existing work on implementing CHERI has shown that tag bits can be maintained and queried efficiently [25].

Revocation tree. On top of this, a CAPSTONE implementation also needs to record the validity of each capability which might change due to revocations. This concerns the maintenance of the revocation tree (Section 4.3). Similarly to the tag bits, the nodes of the revocation tree are stored in a DRAM region inaccessible to software. Each revocation tree node is represented using the format below:

<table>
<thead>
<tr>
<th>depth</th>
<th>next</th>
<th>prev</th>
<th>counter</th>
<th>freed</th>
<th>valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>31</td>
<td>31</td>
<td>35</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Whenever a capability is used in a memory access, the node associated with it (node-id) needs to be retrieved from DRAM to query its validity. To hide the latency of this query, we perform it in parallel to the actual memory access.

Revocation. A revocation operation involves traversing a subtree of a given node (the one associated with the revocation capability) and invalidate each node within the subtree. Invalidated nodes are removed from the revocation tree so each node can be visited and invalidated at most once. This entails
a constant amortized overhead of the revocation operation. To facilitate subtree traversals, we maintain the revocation tree as a doubly-linked list of nodes in the depth-first order, with the depth recorded in each node. Each subtree corresponds to a contiguous range within the linked list. We unlink nodes from the linked list to remove them from the revocation tree.

**Deallocation of revocation nodes.** We cannot make a revocation node available for allocation again immediately after invalidating it, as capabilities that reference it may still exist. However, we do need to free it at some point because the DRAM region for storing revocation nodes is limited in size. We have two potential solutions to this issue. One is a garbage collection mechanism based on memory sweeping: Whenever free nodes run out, we scan the whole memory to discover and free such nodes that are not referenced by any capability. The other option is to include a reference count in each node and free a node when its reference count is zero. We adopt the latter option, as we expect the former to introduce large latencies in unpredictable locations. This can be avoided with memory sweeps in parallel to the pipeline execution, which, however, can be tricky to implement. In comparison, reference counting would require updating the counter when a capability is created or overwritten but we expect the implementation to be straightforward and the overhead acceptable. We free a revocation node by adding it to a free-nodes linked list.

### 7 Evaluation

We aim to answer the following question: **How does the performance of a CAPSTONE implementation compare with that of a traditional platform?** We use the gem5 simulator [6] to model the most performance-relevant aspects of the implementation described in Section 6, namely the operations on the revocation tree, including allocations, revocations, queries, and reference count updates. Other parts such as bound and permission checking and tagged memory are either trivial or already examined in previous work in terms of implementation and performance impact [15, 16, 24, 25].

**Setup.** In the absence of applications written for CAPSTONE, we map runtime behaviours of existing RISC-V applications to the expected corresponding events in their CAPSTONE ports. The details of this mapping are shown in Table 2. As shown in Figure 10, compared to a traditional system, our gem5-based CAPSTONE model has no MMU but incorporates a node controller and a node cache for revocation node storage. While out-of-order CPU models are more accurate for modern mainstream high-performance systems, we choose an in-order core for its lower complexity which is conducive to a first-step evaluation. Evaluating on out-of-order models is future work. We use a clock frequency of 1 GHz, a 2-way set associative L1 instruction (16 kB) and data cache (64 kB), and an 8-way set associative last level cache of 256 kB. The node cache (NS) is an 8 kB 2-way set associative cache with a 32 GB DRAM region for storing revocation nodes is limited in size. However, we do need to free it at some point because the DRAM region for storing revocation nodes is limited in size.

We have two potential solutions to this issue. One is a garbage collection mechanism based on memory sweeping: Whenever free nodes run out, we scan the whole memory to discover and free such nodes that are not referenced by any capability. The other option is to include a reference count in each node and free a node when its reference count is zero. We adopt the latter option, as we expect the former to introduce large latencies in unpredictable locations. This can be avoided with memory sweeps in parallel to the pipeline execution, which, however, can be tricky to implement. In comparison, reference counting would require updating the counter when a capability is created or overwritten but we expect the implementation to be straightforward and the overhead acceptable. We free a revocation node by adding it to a free-nodes linked list.

#### 8 Case Studies

The expressiveness of CAPSTONE enables the memory isolation models discussed in Section 2.2. To demonstrate this, we have implemented a functional prototype of CAPSTONE in the form of an ISA emulator (CAPSTONEEmu) and a simple compiler (CAPSTONECC), and, on top of them, a runtime library (CAPSTONELib) that encapsulates runnable imple-

---

**Table 2: Mappings from existing RISC-V application behaviours to events in CAPSTONE used in our evaluation.**

<table>
<thead>
<tr>
<th>Behaviour in applications</th>
<th>Event in CAPSTONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>malloc</td>
<td>a new linear capability</td>
</tr>
<tr>
<td>free</td>
<td>revoking on a revocation capability</td>
</tr>
<tr>
<td>overwriting an address</td>
<td>destroying a nonlinear capability</td>
</tr>
<tr>
<td>producing an address</td>
<td>creating a nonlinear capability</td>
</tr>
</tbody>
</table>

**Figure 10: Overview of the CAPSTONE model implemented in gem5. The shaded components are added by us.**

1600 MHz DDR3 DRAM.

**Benchmarks.** We use the SPEC CPU 2017 intspeed benchmark suite [7], ref inputs. Instead of a full detailed simulation, which would take months to years to complete, The MMU is removed from both simulations.

**Results.** As shown in Table 3, the workloads vary widely in their use of revocation tree operations, ranging from no use (605.mcf_s) to few to no allocations after initial setup (625.x264_s, 631.deepsjeng_s, and 657.xz_s), to significant use of all operations. Correspondingly, the overhead varies from 0 to 50%. As expected, the overhead roughly correlates with the number of misses in the node cache, as each of them involves accessing the DRAM. The results also show that reference count updates are often the dominating revocation tree operations in terms of frequency, and their frequency strongly correlates with the displayed overhead. We believe that this is partly caused by the inability to hide the latency when a reference count update results from a non-load/store instruction (e.g., move or pointer arithmetic). In an actual CAPSTONE program, we expect such cases to be considerably less frequent because of the ubiquity of linear capabilities (for example, moving a linear capability between registers does not change the reference count). This also points to potential future work of exploring such optimizations as delayed updates to better hide the latency and improve the performance.
Table 3: Evaluation results on SPEC CPU 2017 intspeed, collected after 10 billion instructions of fast-forwarding.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Runtime (seconds)</th>
<th>Baseline</th>
<th>Overhead (%)</th>
<th>CAPSTONE node cache Hits</th>
<th>CAPSTONE revocation tree operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>600.perlbenc_s/0</td>
<td>4.893</td>
<td>3.632</td>
<td>34.710</td>
<td>555427</td>
<td>615721998</td>
</tr>
<tr>
<td>600.perlbenc_s/1</td>
<td>4.960</td>
<td>3.882</td>
<td>27.781</td>
<td>415766</td>
<td>529725069</td>
</tr>
<tr>
<td>600.perlbenc_s/2</td>
<td>5.157</td>
<td>3.841</td>
<td>34.262</td>
<td>4997293</td>
<td>625836524</td>
</tr>
<tr>
<td>602.gcc_s/0</td>
<td>5.258</td>
<td>4.005</td>
<td>31.219</td>
<td>295175</td>
<td>616668184</td>
</tr>
<tr>
<td>602.gcc_s/1</td>
<td>5.259</td>
<td>4.007</td>
<td>31.263</td>
<td>305182</td>
<td>676016867</td>
</tr>
<tr>
<td>602.gcc_s/2</td>
<td>5.260</td>
<td>4.007</td>
<td>31.257</td>
<td>325157</td>
<td>617388305</td>
</tr>
<tr>
<td>605.mcf_s/0</td>
<td>5.467</td>
<td>5.467</td>
<td>0.000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>620.omnetpp_s/0</td>
<td>7.947</td>
<td>5.267</td>
<td>50.870</td>
<td>12058732</td>
<td>902225434</td>
</tr>
<tr>
<td>623.talambnk_s/0</td>
<td>9.017</td>
<td>6.202</td>
<td>45.857</td>
<td>16891678</td>
<td>851610538</td>
</tr>
<tr>
<td>625.x264_s/0</td>
<td>4.434</td>
<td>3.679</td>
<td>20.916</td>
<td>95</td>
<td>377394651</td>
</tr>
<tr>
<td>625.x264_s/1</td>
<td>4.511</td>
<td>3.765</td>
<td>19.819</td>
<td>0</td>
<td>373093048</td>
</tr>
<tr>
<td>625.x264_s/2</td>
<td>4.078</td>
<td>3.459</td>
<td>17.902</td>
<td>33</td>
<td>309551709</td>
</tr>
<tr>
<td>631.deepsjeng_s/0</td>
<td>3.363</td>
<td>3.344</td>
<td>0.565</td>
<td>0</td>
<td>9452119</td>
</tr>
<tr>
<td>641.leela_s/0</td>
<td>3.386</td>
<td>3.105</td>
<td>9.072</td>
<td>1764704</td>
<td>87073176</td>
</tr>
<tr>
<td>648.iceberg3_s/0</td>
<td>3.646</td>
<td>3.638</td>
<td>0.222</td>
<td>0</td>
<td>4029952</td>
</tr>
<tr>
<td>657.xz_s/0</td>
<td>3.011</td>
<td>2.899</td>
<td>3.846</td>
<td>0</td>
<td>33256028</td>
</tr>
<tr>
<td>657.xz_s/1</td>
<td>3.522</td>
<td>3.078</td>
<td>14.414</td>
<td>0</td>
<td>221848643</td>
</tr>
</tbody>
</table>

Table 5: LoC (input to CAPSTONECC) of our case study implementations in CAPSTONELib. Abbreviations: MA (memory allocator), TS (thread scheduler), Enc (enclaves).

<table>
<thead>
<tr>
<th>Case study</th>
<th>MA</th>
<th>TS</th>
<th>Encl</th>
<th>Rust</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>242</td>
<td>75</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Listing 1: Left: Data structure that exposes runtime interfaces. Right: Data structures in the memory allocator.

Table 4: LoC of each component of the prototype implementation: CAPSTONEEmu, CAPSTONECC, and CAPSTONELib.

<table>
<thead>
<tr>
<th>Component</th>
<th>CAPSTONE Emu</th>
<th>CC</th>
<th>Lib</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoC</td>
<td>1081</td>
<td>2319</td>
<td>529</td>
</tr>
</tbody>
</table>

Structs `capstone_runtime` and `mem_region` for implementations of different memory isolation models. Table 4 summarizes the lines of code in each component.

Unlike our gem5 model (see Sections 6 and 7), CAPSTONEEmu is intended purely for exploring the expressiveness of the CAPSTONE `interface`. Therefore, we keep its implementation high-level and straightforward. CAPSTONECC compiles a C-like language into CAPSTONE. CAPSTONELib exposes interfaces to memory isolation models through sealed capabilities inside a `capstone_runtime` object (Listing 1, left). We have open-sourced those tools and our case study implementations (see Availability). We summarize the lines of code (LoC) of each case study in Table 5.

Trustless memory allocation. Our heap memory allocator exposes two interfaces to applications: `malloc` and `free`. The `malloc` interface receives the size of the memory region to be allocated, and returns a valid linear capability if the allocation succeeds. The `free` interface receives a linear capability for a previously allocated memory region, and makes it available for future allocations. An application does not need to trust the memory allocator. After obtaining a linear capability from `malloc`, the application is guaranteed exclusive access to the memory region. The allocator may revoke the capability, but cannot read the original memory content as long as the application holds its linear or revocation capability.

The `malloc_state` object maintains the state of a memory allocator (Listing 1, right). It contains a `heap` capability that covers all memory available for allocation. Since the capability to `malloc_state` is sealed inside `malloc` and `free` in `capstone_runtime`, an application can only access the allocator state indirectly through these well-defined interfaces.

In summary, our implementation guarantees that the memory allocator can reclaim memory whenever it wants, but cannot access any allocated region, or read private data in a region after reclaiming it. Trust between applications and the memory allocator is thus unnecessary. Since CAPSTONE does not include a centrally-managed MMU, mechanisms commonly relying on it (e.g., swapping, copy-on-write) become non-trivial. Enabling them with capabilities is future work.

Trustless thread scheduling. We implemented a thread scheduler that requires no trust from applications. The scheduler belongs to a different domain and has no access to the linear capabilities held by the application domain (and, in turn, the data they point to). CAPSTONE ensures that the application domain context is safely saved and restored when an exception occurs and when the domain resumes execution.
Part of the data involved is critical in the sense that at most one thread can safely manipulate it at any time. To protect such data, we encapsulate them in a `sched_critical_state` object, and include a linear capability to it inside the scheduler state `sched_state`. Before a thread accesses such data, it needs to load the linear capability into a register. The linearity of the capability subsequently guarantees that no other thread can access the data structure.

Besides preventing the thread scheduler from accessing application data during context switches, our implementation has several more security benefits. Since the exception handler is defined by a normal sealed capability, an application can attest to the identity of the exception handler or thread scheduler (when sealed capabilities are extended with cryptographic checksums). This mitigates attacks that involve attacker-controlled exception handlers or thread schedulers, such as Game of Threads [44] and SmashEx [12]. By safely storing the domain context upon a context switch, CAPSTONE also allows better control of domain re-entries, as re-entries that accesses overlapping resources are impossible, which improves the security of custom exception handling (e.g., in-enclave exception handling in Intel SGX [11, 34]).

**Spatially-isolated enclaves.** We implemented a basic set of interfaces for a TEE with spatially-isolated enclaves similar to Intel SGX [11, 34] and Keystone [27]: `enclave_create`, `enclave_enter`, and `enclave_destroy`. The central data structures include `enclave`, which is available to the software creating and using an enclave, and `enclave_runtime`, which is available to the enclave itself.

The `enclave_create` interface creates a new enclave from two input linear capabilities for its code and data respectively. Both capabilities are then sealed together in a sealed capability, alongside an enclave private stack and an enclave runtime object. Sealing protects the corresponding memory regions from direct access outside the enclave itself, similar to the enclave setup in Intel SGX [11, 34]. To facilitate data exchange between the host application and the enclave, `enclave_create` creates a shared memory region between them, and its capability is placed in both `enclave_runtime` and `enclave`. The `enclave_enter` interface calls into the sealed capability contained in a given enclave structure, effectively executing the enclave. The `enclave_destroy` interface reclaims and frees the memory resources of an enclave with the revocation capabilities inside `enclave`.

This case study focuses on memory isolation. A complete TEE platform usually also includes a hardware root of trust, memory encryption, and local and remote attestation [1, 11, 27, 34]. We consider the hardware root of trust and memory encryption as orthogonal to CAPSTONE. For attestation, future work may explore attaching measurements to uninitialized capabilities and extending them upon each memory store. The measurement is frozen when the uninitialized capability is initialized, and henceforth invalidated upon further stores.

**Nested enclaves.** In our spatially-isolated enclave implemen-

----

```
void* shared_mem = capstone_runtime->malloc(128);
shared_mem[0] = 42; // just some dummy data
void* shared_rev = mrev(shared_mem);
void* shared_rev_shared = mrev(shared_mem);
drop(shared_mem); // drop to share runtime->heap[1] = shared_rev;
void* emissary =
  capstone_runtime->malloc(CAPSTONE_SEALED_SIZE);
  emissary[CAPSTONE_OFFSET_PC] = emissary_code;
  emissary[CAPSTONE_OFFSET_EPC] = 0;
  emissary[CAPSTONE_OFFSET_DEDICATED_STACK] = 0;
  emissary[CAPSTONE_OFFSET_METAPARAM] = shared_rev_shared;
  seal(emissary); runtime->shared[0] = emissary;

Listing 2: Preparing a shared memory region.
```

```
void* setup_shared() {
  void* runtime->shared[0];
  void* shared_mem = d();
  void* d = CAPSTONE_METAPARAM;
  revoke(shared_mem);
  // check measurement of ret
  return d;
}
```

Listing 3: Left: accessing a shared memory region. Right: domain that performs authentication and returns revocation capabilities for shared memory regions.

tation, the domain creating an enclave can also be an enclave itself. To enable nested enclaves, we only need to expose the enclave creation interfaces to enclaves. This is easily achieved by passing the `capstone_runtime`structure to each enclave inside the `enclave_runtime` structure. The nesting structure can be extended indefinitely during runtime on demand. Each enclave can be sure that a memory region shared with a child enclave is only accessible to this same child enclave or those nested inside it and can be reclaimed at any time.

**Temporally-isolated enclaves.** Since CAPSTONE does not rely on identity-based access control, an enclave `D` cannot directly share a memory region exclusively with another enclave `E`, unless `E` is created by `D` or `D` can access the sealed capability of `E` through other means. In general, `D` needs to pass a capability to `E` and `E` alone. To achieve this on CAPSTONE, `D` can create a domain `C` specially for communicating with `E` and then pass `C`’s sealed capability to `E` (Listing 2). The host then marshalls `C`’s sealed capability to `E`, which obtains access to the shared memory region by invoking `C` and then performing revocation with the returned revocation capability (Listing 3, left). `C` can then perform authentication, e.g., by examining the measurement of the sealed-return capability in `ret`, to make sure that it is invoked by `E` before provisioning a revocation capability for the shared memory region (Listing 3, right). Hardware-generated cryptographic checksums are beyond the scope of this paper.

Note that `D` can limit `E`’s access permissions to the memory region through the permissions in the revocation capability passed to `E`. In addition, since `D` holds another revocation capability created before the one passed to `E`, it can revoke
the delegated access at any time. To establish a non-exclusive shared memory region with $E, D$ may have $C$ pass to $E$ a non-linear capability instead of a revocation capability. By passing linear capabilities back and forth through the non-exclusive shared memory between the two enclaves, they can take turns to have exclusive access to other memory regions in multiple rounds with the non-exclusive region as a trampoline.

Our implementation prevents unintended enclaves from accessing a temporarily shared memory region. Through revocation capabilities, it also allows an accessor to obtain exclusive access. Moreover, the owner enclave of a memory region can limit what each accessor can do to it.

**Rust-like memory restrictions.** CAPSTONE can enforce Rust-like memory restrictions across security domains at runtime without assuming trusted software components.

Table 6 summarizes the mapping from Rust operations to the corresponding CAPSTONE primitives. Owner references in Rust are directly mapped to linear capabilities, as they are similarly alias-free and non-duplicable. However, CAPSTONE has no direct equivalent to the mutable borrowed reference. Instead, we pass the linear capability itself for mutable borrowing, and utilize the revocation capability to ensure its return (in Rust, the owner reference becomes usable again after the lifetime of the borrowed references ends). Immutable borrowing is supported through read-only non-linear capabilities created by delinearizing the linear capability and then tightening the permissions to read-only, which can then be shared in arbitrarily many copies, matching the behaviours of immutable borrowed references in Rust. Again, the domain uses revocation capabilities to ensure that its exclusive access (owner reference) can be reclaimed.

### 9 Related Work

**Architectural capabilities.** Early computer architectures with capability-based memory addressing can be traced back to the early 1980s, but failed to see widespread adoption due to significant performance overhead [23]. M-machine [8] improved the performance through tagged memory words and a shared address space across all protection domains. Hard-Bound [14] proposed a limited form of architectural capabilities without unforgeability to improve the performance of bounds-checking in C programs. More recently, CHERI [53] follows the tagged memory design of M-machine with improved memory region granularity and compatibility with traditional page-based memory protections. Unlike CAPSTONE, all those designs assume a trusted OS kernel, and are unable to express exclusive access guarantees or hierarchical capability revocation. Instead of relying on capability metadata, C uses pointer encryption and memory encryption to prevent secret leakage and predictable memory tampering [28], which helps reduce its performance overhead. However, this trades off its flexibility in expressing more sophisticated rules such as those associated with different capability types. Capability-based security has also seen adoption in software designs, including OS kernels [3, 22, 52], programming languages [10, 35], and web services [20]. Such designs deal with higher-level notions of resources rather than memory.

**Linear capabilities.** Naden et al. proposed a type system with “unique permissions”, a concept similar to linear capabilities, to achieve efficient flexible borrowing [41]. This is different from CAPSTONE which provides linear capabilities at the lower architectural level, and enforces restrictions during runtime. StkTokens [46] is a culling convention that utilizes architectural linear capabilities to provide control flow integrity in the context of software fault isolation. StkTokens is focused on a specific memory model, whereas CAPSTONE intends to support multiple models at the same time. Moreover, StkTokens does not discuss scenarios with asynchronous exceptions or when untrusted software refuses to relinquish a linear capability. It is also unclear how linear and non-linear capabilities interact. Van Strydonck et al. proposed capturing spatial separation logic predicates during runtime through compiling verified C code into a low-level language with linear capabilities [48]. We consider their work as orthogonal to ours, as CAPSTONE is focused exclusively on low-level interfaces. The CHERI ISA document [53] briefly discusses an incomplete linear capability design as an experimental feature to replace garbage collection. It is unclear from the document what interfaces related to linear capabilities are available. Moreover, CHERI relies on a trusted OS kernel to manage linear capabilities. They propose that the OS kernel be allowed to violate linearity to this purpose. This fundamentally contrasts the goal of CAPSTONE.

**Uninitialized capabilities.** Georges et al. introduced the notion of uninitialized capability as a mechanism to improve the performance of capability revocation [18]. Converting a normal capability into an uninitialized capability allows invalidating the capabilities that reside in a large memory region without requiring a scan through it. The application can hence gain the guarantee that a memory region does not contain any capability with a small constant overhead. Unlike their work, CAPSTONE generalizes uninitialized capabilities to the generic role of preventing secret leakage, where the secrets include, but are no more limited to, capabilities.

### Table 6: Rust-like abstraction on CAPSTONE.

<table>
<thead>
<tr>
<th>Op.</th>
<th>Rust</th>
<th>CAPSTONE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move</td>
<td><code>let a = b;</code></td>
<td><code>mov ra rb</code></td>
</tr>
<tr>
<td>Immutable</td>
<td><code>let a = &amp;b;</code></td>
<td><code>mrev rr rb; delin rb;</code></td>
</tr>
<tr>
<td>Borrow</td>
<td></td>
<td><code>lb r0 0; tighten rb;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>mov ra rb;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>(use ra) revoke rr;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>mov rb rr</code></td>
</tr>
<tr>
<td>Mutable</td>
<td><code>let a = &amp;mut b;</code></td>
<td><code>mrev rr rb; mov ra rr;</code></td>
</tr>
<tr>
<td>Borrow</td>
<td></td>
<td><code>(use ra) revoke rr;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>mov rb rr</code></td>
</tr>
</tbody>
</table>
Linear and uniqueness type systems. Some high-level programming languages have adopted linear type systems [10, 30–32, 50] or uniqueness type systems [31, 47]. Linear types require that values be used exactly once in the future, whereas uniqueness types require that the values have never been duplicated in the past. They both allow conversions in single directions: from an unrestricted (i.e., non-linear) type into a linear type, or from a uniqueness type into an unrestricted type [31, 47, 50]. In contrast, by providing a mechanism for revocation, CAPSTONE enables conversions (i.e., linearization and delinearization) in both directions. Unlike high-level programming languages, CAPSTONE is intended as a low-level interface that enforces similar restrictions during runtime.

Capability revocation. Prior work on CHERI capability revocation adopts different semantics to the term “revocation” than CAPSTONE does. Whereas revocation in CAPSTONE immediately invalidates a set of capabilities defined by the revocation hierarchy to reclaim a linear capability, CHERI revoke [58] and Cornucopia [55] lazily invalidate capabilities through memory sweeping to ensure that objects do not have stale capabilities associated with them when they are relocated (i.e., no use-after-reallocation). While CAPSTONE is stricter in requiring invalidations to immediately take effect, its more constrained capability provisioning operations help simplify tracking of capability derivations and facilitate capability revocation.

Tagged architectures. Similar to CHERI [25, 57], CAPSTONE is a tagged architecture [24] as it uses hardware-maintained word-granular metadata to distinguish capabilities and enforce corresponding memory access isolation policies. Other prior tagged architectures mostly focus on accelerating specific security policies such as control flow integrity (CFI) and spatial memory safety [40], whereas CAPSTONE aims at providing a novel capability-based memory access model. Designs for general-purpose software-defined tag computations have been proposed [15, 16], but as the tag computations in CAPSTONE are concrete and sufficiently simple to implement in hardware, have a huge input space (e.g., address ranges), and are sometimes not purely functional (e.g., revocation nodes), those designs are ill-suited for supporting CAPSTONE.

## 10 Conclusions

We have proposed CAPSTONE, a new capability-based architectural design that provides the flexibility to support multiple memory isolation models without assuming trusted software components. We pointed out that existing designs are insufficient to achieve such a goal and described the additions needed to overcome those limitations, specifically through a careful design with linear and revocation capabilities. Our evaluation results suggest that CAPSTONE can be implemented with acceptable overhead. As future work, we plan to explore hardware implementations of CAPSTONE and to experiment with a wider range of use cases.

### Acknowledgments

We thank the anonymous reviewers, Shweta Shinde, and Bo Wang for their suggestions on earlier drafts of this paper. This research is supported by the Ministry of Education, Singapore, under its Academic Research Fund Tier 1, T1 251RES2023 (A-0008125-00-00). Trevor E. Carlson is supported by a research grant from Huawei. Conrad Watt is supported by a Research Fellowship from Peterhouse, University of Cambridge. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors only.

### Availability

The code produced in this work is publicly available at https://github.com/jasonyu1996/capstone.

### References


[37] Intel® 64 and ia-32 architectures software developer manual, 2018.


[49] Philip Wadler. Linear types can change the world! In PROGRAMMING CONCEPTS AND METHODS. North, 1990.


