

CAPSTONE: A Capability-based Foundation for Trustless Secure Memory Access (Extended Version)

Jason Zhijingcheng Yu
National University of Singapore

Conrad Watt
University of Cambridge

Aditya Badole
National University of Singapore

Trevor E. Carlson
National University of Singapore

Prateek Saxena
National University of Singapore

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, \dots, 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

¹Linear 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 fulfil 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 enclaved 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 how 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.

Model	P1	P2	P3	P4
Rust-like abstraction [32]	•	•	○	○
Spatially-isolated enclaves [11, 34]	•	•	○	•
Temporally-isolated enclaves [59]	•	•	○	•
Nested enclaves [43]	•	•	•	•
Trustless memory allocation	•	•	○	○
Trustless thread scheduling	•	○	○	•
CHERI [53]	○	○	○	○
CAPSTONE (this work)	•	•	•	•

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 $CAPSTONE_{abs}$. 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 ($tstate_{sup}$ and $tstate_{sub}$). 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_{abs}) to manipulate these capabilities or update the memory. $CAPSTONE_{abs}$ directly enforces desirable properties of CAPSTONE, and we characterize CAPSTONE's security as a standard refinement theorem:

Main theorem. $CAPSTONE$ *refines* $CAPSTONE_{abs}$.

We discuss $CAPSTONE_{abs}$ 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

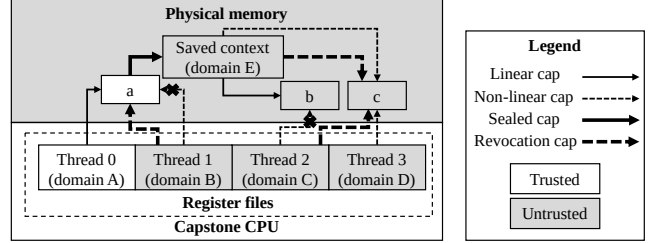


Figure 1: Overview of CAPSTONE. An arrow from X to Y represents a capability located inside X that grants access to Y . Crossed-out arrows are capabilities not allowed to exist.

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 have separate 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

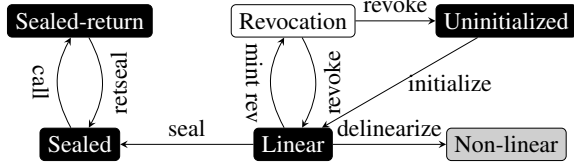


Figure 2: Overview of different types of capabilities in CAPSTONE and the operations that change the type of a capability. Capability types with black backgrounds are alias-free, non-linear capabilities can overlap among themselves, whereas revocation capabilities can overlap with any other capabilities.

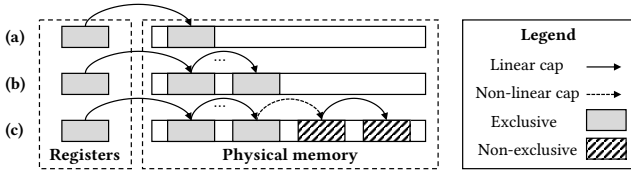


Figure 3: Exclusive access guarantees in different scenarios: (a) Through a linear capability; (b) Through a chain of linear capabilities; (c) No exclusive access guarantee through a chain with non-linear capabilities.

capability type called *linear capabilities* [19, 46, 49, 53]. 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.²

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_u for the memory region R_u , and inside R_u resides another linear capability c_v for another memory region R_v , then besides R_u , the domain also has exclusive access to R_v . 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

²Moving a linear capability from location A to B destroys the copy in A .

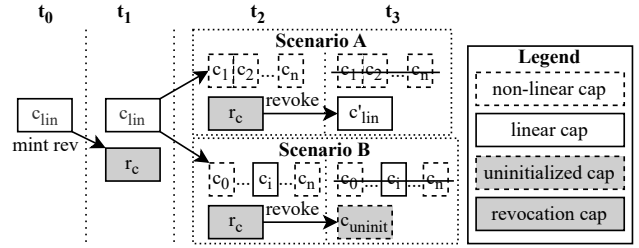


Figure 4: Overview of the operations on revocation capabilities. Strikethrough capabilities are invalid. When c_{lin} has derived non-linear capabilities only at t_3 , revocation converts r_c into a linear capability c'_{lin} (scenario A). Otherwise, r_c is converted into an uninitialized capability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

to reach the memory region (Figure 3(c)).

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.

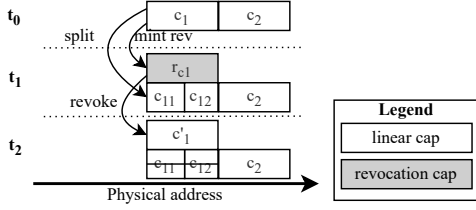


Figure 5: Split/merge. Strikethrough capabilities are invalid.

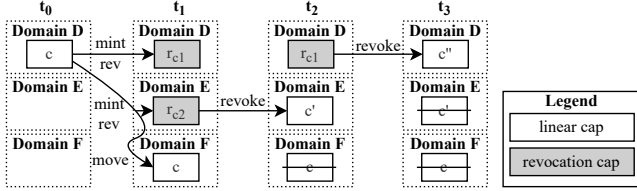


Figure 6: Extensible capability delegation hierarchy through revocation capabilities. Strikethrough capabilities are invalid.

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., guaranteedly 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 Ψ consisting of the states of physical threads Θ , physical memory state mem , and additional capability-related data structures. Each physical thread has a distinct register file which includes the program counter pc , special registers ret and epc , and M general-purpose registers r_1, r_2, \dots, r_M . By indexing Θ , we can obtain the state θ 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

n, N, d, b, e, a	\in	\mathbb{N}
$Reg \ni r$	$:=$	$pc \mid ret \mid epc \mid r_1 \mid r_2 \mid \dots \mid r_M$
$CapType \ni t$	$:=$	$Lin \mid Non \mid Rev \mid Sealed(d) \mid SealedRet(d, r) \mid Uninit$
$Perms \ni p$	$:=$	$NA \mid R \mid RW \mid RX \mid RWX$
$RNodeType \ni nt$	$:=$	$RLin \mid RNon$
$RevParent \ni pr$	$:=$	$n \mid root \mid null$
$RevTree \ni rt$	$:=$	$\mathbb{N} \mapsto RevParent \times RNodeType$
$Cap \ni c$	$:=$	(t, b, e, a, p, n)
$Word \ni w$	$:=$	$c \mid n \mid i$
$RegFile \ni regs$	$:=$	$Reg \mapsto Word$
$Memory \ni mem$	$:=$	$\mathbb{N} \mapsto Word$
$ThreadState \ni \theta$	$:=$	$regs \mid error$
$Threads \ni \Theta$	$:=$	$\mathbb{N} \mapsto ThreadState \times \mathbb{N}$
$State \ni \Psi$	$:=$	(Θ, mem, rt, N)
$Insn \ni i$	$:=$	$mov \ r \ r \mid ld \ r \ r \mid sd \ r \ r \mid tighten \ r \ r \mid shrink \ r \ r \ r \mid split \ r \ r \mid delin \ r \mid scc \ r \ r \mid lcc \ r \ r \mid mrev \ r \ r \mid drop \ r \mid seal \ r \mid call \ r \ r \mid return \ r \ r \mid retseal \ r \ r \mid revoke \ r \mid init \ r \mid except \ n \mid jmp \ r \mid jnz \ r \ r \mid li \ r \ n \mid add \ r \ r \mid lt \ r \ r \ r \mid invalid$

Figure 7: Syntax of the CAPSTONE model.

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 Ψ 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 **lt** are omitted, as they are almost identical to their counterparts in common existing architectures. Interested readers may find the full definition of the state transitions in Appendix A.

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

location (a register or a memory location) to $\text{Moved}(w)$:

$$\text{Moved}(w) = \begin{cases} 0 & w = (t, b, e, a, p, n) \wedge t \in \text{LinearTypes} \\ w & \text{otherwise,} \end{cases}$$

where (parameters are omitted here for brevity)

$$\text{LinearTypes} = \{\text{Lin}, \text{Rev}, \text{Uninit}, \text{Sealed}, \text{SealedRet}\}.$$

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 reparents 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. CAPSTONE 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 CAPSTONE 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, **sc** sets the cursor to a given 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 c_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 `r1` is reserved for argument passing. In other words, a second operand to **call** is directly loaded into E ’s `r1` 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 := user | sup | sub
 mem_{abs} := Addr \mapsto (Word | uninit)
 $range$:= $\{n \mid x \leq n < x + y\}$
 cap_{abs} := range
 $tstate$:= cap_{abs} set
 $pstate$:= $(mem_{abs}, tstate_{user}, tstate_{sup}, tstate_{sub})$

Figure 8: CAPSTONE_{abs} state.

$\text{Refines}_{(d, D_{sub})}(\Psi, (mem_{abs}, tstate_{user}, tstate_{sup}, tstate_{sub}))$
 where: $ind_{uninit} = \bigcup_{d \in \mathbb{N}} \text{woranges}(\mathcal{R}_w(\Psi, d))$
 $mem_{abs} = \Psi.mem[ind_{uninit} := \text{uninit}]$
 $tstate_{user} = \text{ranges}(\mathcal{X}(\Psi, d))$
 $tstate_{sub} = \bigcup_{d' \in D_{sub}} \text{ranges}(\mathcal{X}(\Psi, d'))$
 $tstate_{sup} = \bigcup_{d' \in (\mathbb{N} - D_{sub} - d)} \text{ranges}(\mathcal{X}(\Psi, d'))$

Figure 9: CAPSTONE_{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 Appendix B.

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

We show that CAPSTONE’s use of uninitialized capabilities refines CAPSTONE_{abs}’s use of uninit memory values to denote memory which cannot be accessed. To this end, the line $ind_{uninit} = \bigcup_{d \in \mathbb{N}} \text{woranges}(\mathcal{R}_w(\Psi, d))$ collects the ranges of all write-only (uninitialized) capabilities. The line $mem_{abs} = \Psi.mem[ind_{uninit} := \text{uninit}]$ 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 ,

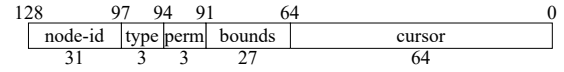
that is, the set of all concrete linear capabilities exclusively accessible (transitively) in the domain d . The abstract state of the user domain ($tstate_{user}$) is defined as the set of ranges corresponding to the capabilities in the exclusive realm of d . The abstract states of the subordinate and 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

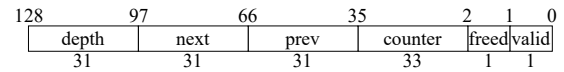
We show that CAPSTONE can be implemented with acceptable overhead. Since a complete RTL implementation requires significant engineering effort, we consider it as future work beyond the scope of this paper. Instead, we present a sketch of a potential implementation below, and evaluate it in Section 7.

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



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:



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 invalidates 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

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

Behaviour in applications	Event in CAPSTONE
<code>malloc</code>	a new linear capability
<code>free</code>	revoking on a revocation capability
overwriting an address	destroying a nonlinear capability
producing an address	creating a nonlinear capability

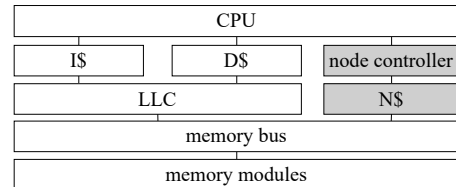


Figure 10: Overview of the CAPSTONE model implemented in `gem5`. The shaded components are added by us.

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.

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 3: Evaluation results on SPEC CPU 2017 intspeer, collected after 10 billion instructions of fast-forwarding.

Workload	Runtime (seconds)		Overhead (%)	CAPSTONE node cache			CAPSTONE revocation tree operations			
	CAPSTONE	Baseline		Misses	Hits	Miss rate (%)	#Allocation	#Query	#RC-update	#Revocation
600.perlbenc_s/0	4.893	3.632	34.710	555427	615721998	0.090	22402	96213853	259932183	18754
600.perlbenc_s/1	4.960	3.882	27.781	415766	529275069	0.078	59565	106558849	211290520	58279
600.perlbenc_s/2	5.157	3.841	34.262	997293	625836524	0.159	1293	95528533	265646893	1161
602.gcc_s/0	5.256	4.005	31.219	295175	616668184	0.048	139829	98004533	258869083	139020
602.gcc_s/1	5.259	4.007	31.263	305182	617601867	0.049	139870	98162127	259262035	139032
602.gcc_s/2	5.260	4.007	31.257	325157	617383805	0.053	139814	98120826	259183800	139011
605.mcf_s/0	5.467	5.467	0.000	0	0	0.000	0	0	0	0
620.omnetpp_s/0	7.947	5.267	50.870	12058732	902225434	1.319	430165	172949008	368670213	379251
623.xalancbmk_s/0	9.017	6.202	45.387	16891678	851610538	1.945	112500	273464398	297073283	80999
625.x264_s/0	4.434	3.679	20.516	95	377394651	0.000	38	175363916	101015339	0
625.x264_s/1	4.511	3.765	19.819	0	373093048	0.000	0	183377516	94857766	0
625.x264_s/2	4.078	3.459	17.902	33	309551709	0.000	116	146974430	81288420	2
631.deepsjeng_s/0	3.363	3.344	0.565	0	9452119	0.000	0	4280475	2585822	0
641.leela_s/0	3.386	3.105	9.072	1764704	87073176	1.986	23432	33280704	27675933	19844
648.exchange2_s/0	3.646	3.638	0.222	0	4029952	0.000	22078	1909208	972060	22078
657.xz_s/0	3.011	2.899	3.846	0	33256028	0.000	0	24748672	4253678	0
657.xz_s/1	3.522	3.078	14.414	0	221848643	0.000	0	73389965	74229339	0

```

struct capstone_runtime {
    void* malloc;
    void* free;
    void* thread_start;
    void* thread_create;
    void* join_all;
    void* enclave_create;
    void* enclave_enter;
    void* enclave_destroy;
};

struct mem_region {
    struct mem_region *left,
    *right;
    int size, leaf, free;
    void* mem;
};

struct malloc_state {
    struct mem_region* heap;
    int alloc_n;
};

```

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.

CAPSTONE-	Emu	CC	Lib
LoC	1081	2319	529

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

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

Case study	MA	TS	Encl	Rust
LoC	128	242	75	5

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 those 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 implementation, the domain creating an enclave can also be an enclave itself. To enable nested enclaves, we only need to expose the

```
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);
scco(emissary_code, 0);
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* d = CAPSTONE_ATTR_HAS_METAPARAM
    runtime->shared[0]; void* setup_shared() {
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.

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-

Table 6: Rust-like abstraction on CAPSTONE.

Op.	Rust	CAPSTONE
Move	let a = b;	mov ra rb
Immutable borrow	let a = &b;	mrev rr rb; delin rb; li r0 0; tighten rb r0; mov ra rb; (use ra) revoke rr; mov rb rr
Mutable borrow	let a = &mut b;	mrev rr rb; mov ra rb; (use ra) revoke rr; mov rb rr

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 im-

proved 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, C³ 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, 26, 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 calling 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.

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, CHERIvoke [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 reallocated (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

- [1] Arm trustzone technology. <https://developer.arm.com/ip-products/security-ip/trustzone>.
- [2] Adam Barth, Joel Weinberger, and Dawn Song. Cross-origin javascript capability leaks: Detection, exploitation, and defense. In *Proceedings of the 18th Conference on USENIX Security Symposium, SSYM'09*, page 187–198, USA, 2009. USENIX Association.
- [3] Andrew Baumann, Paul Barham, Pierre-Evariste Dagdand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhanina. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP '09*, page 29–44, New York, NY, USA, 2009. Association for Computing Machinery.
- [4] A. Bensoussan, C. T. Clingen, and R. C. Daley. The multics virtual memory: Concepts and design. *Commun. ACM*, 15(5):308–318, may 1972.
- [5] H Bingham. Access controls in burroughs large systems. *Privacy and Security in Computer Systems*, pages 42–45, 1974.
- [6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, aug 2011.
- [7] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute

- benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE '18*, page 41–42, New York, NY, USA, 2018. Association for Computing Machinery.
- [8] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. In Forest Baskett and Douglas W. Clark, editors, *ASPLOS-VI Proceedings - Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 4-7, 1994*, pages 319–327. ACM Press, 1994.
- [9] David Chisnall, Colin Rothwell, Robert N. M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. Beyond the PDP-11: architectural support for a memory-safe C abstract machine. In Özcan Özturk, Kemal Ebcioğlu, and Sandhya Dwarkadas, editors, *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*, pages 117–130. ACM, 2015.
- [10] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 2015*, page 1–12, New York, NY, USA, 2015. Association for Computing Machinery.
- [11] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, 2016:86, 2016.
- [12] Jinhua Cui, Jason Zhijiangcheng Yu, Shweta Shinde, Praateek Saxena, and Zhiping Cai. Smashex: Smashing SGX enclaves using exceptions. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 779–793. ACM, 2021.
- [13] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Marketos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. Cheriabi: Enforcing valid pointer provenance and minimizing pointer privilege in the posix c runtime environment. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19*, page 379–393, New York, NY, USA, 2019. Association for Computing Machinery.
- [14] Joseph Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: architectural support for spatial safety of the c programming language. In *ASPLOS 2008*, 2008.
- [15] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Benjamin C. Pierce, and Andre DeHon. Architectural support for software-defined metadata processing. *SIGARCH Comput. Archit. News*, 43(1):487–502, mar 2015.
- [16] Udit Dhawan, Nikos Vasilakis, Raphael Rubin, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Benjamin C. Pierce, and André DeHon. Pump: A programmable unit for metadata processing. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy, HASP '14*, New York, NY, USA, 2014. Association for Computing Machinery.
- [17] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Scalable memory protection in the PENGLAI enclave. In Angela Demke Brown and Jay R. Lorch, editors, *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 275–294. USENIX Association, 2021.
- [18] Aina Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. Efficient and provable local capability revocation using uninitialized capabilities. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021.
- [19] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.
- [20] Dick Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, October 2012.
- [21] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, October 1988.
- [22] Norman Hardy. Keykos architecture. *SIGOPS Oper. Syst. Rev.*, 19(4):8–25, oct 1985.
- [23] Merle E. Houdek, Frank G. Soltis, and Roy L. Hoffman. Ibm system/38 support for capability-based addressing. In *Proceedings of the 8th Annual Symposium on Computer Architecture, ISCA '81*, page 341–348, Washington, DC, USA, 1981. IEEE Computer Society Press.
- [24] Samuel Jero, Nathan Burow, Bryan Ward, Richard Skowrya, Roger Khazan, Howard Shrobe, and Hamed Okhravi. Tag: Tagged architecture guide. *ACM Comput. Surv.*, 55(6), dec 2022.

- [25] 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 Mazinghi, Alex Richardson, Stacey Son, and A. Theodore Marketos. Efficient tagged memory. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 641–648, 2017.
- [26] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, February 2014.
- [27] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. Keystone: an open framework for architecting trusted execution environments. In *EuroSys*. ACM, 2020.
- [28] Michael LeMay, Joydeep Rakshit, Sergej Deutsch, David M. Durham, Santosh Ghosh, Anant Nori, Jayesh Gaur, Andrew Weiler, Salmin Sultana, Karanvir Grewal, and Sreenivas Subramoney. Cryptographic capability computing. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO ’21, page 253–267, New York, NY, USA, 2021. Association for Computing Machinery.
- [29] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards pointer integrity using ARM pointer authentication. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 177–194, Santa Clara, CA, August 2019. USENIX Association.
- [30] Simon Marlow. Haskell 2010 language report. <https://www.haskell.org/onlinereport/haskell2010/>, 2010.
- [31] Daniel Marshall, Michael Vollmer, and Dominic Orchard. Linearity and uniqueness: An entente cordiale. In *Programming Languages and Systems: 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings*, page 346–375, Berlin, Heidelberg, 2022. Springer-Verlag.
- [32] Nicholas D. Matsakis and Felix S. Klock II. The rust language. In Michael Feldman and S. Tucker Taft, editors, *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*, pages 103–104. ACM, 2014.
- [33] Alfredo Mazinghi, Ripduman Sohan, and Robert N. M. Watson. Pointer provenance in a capability architecture. In *Proceedings of the 10th USENIX Conference on Theory and Practice of Provenance, TaPP’18*, page 2, USA, 2018. USENIX Association.
- [34] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP@ISCA*, page 10. ACM, 2013.
- [35] Mark S. Miller, E. Dean Tribble, and Jonathan Shapiro. Concurrency among strangers. In Rocco De Nicola and Davide Sangiorgi, editors, *Trustworthy Global Computing*, pages 195–229, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
- [36] Mark S Miller, Ka-Ping Yee, Jonathan Shapiro, et al. Capability myths demolished. Technical report, Technical Report SRL2003-02, Johns Hopkins University Systems Research . . . , 2003.
- [37] Intel® 64 and ia-32 architectures software developer manual, 2018.
- [38] Armv8-m memory model and memory protection user guide. <https://developer.arm.com/documentation/107565/latest>, 2022. Accessed on 27 Feb 2023.
- [39] Intel® memory protection extensions enabling guide. <https://www.intel.com/content/www/us/en/developer/articles/guide/intel-memory-protection-extensions-enabling-guide.html>, 2016.
- [40] Armv8.5-a memory tagging extension white paper. <https://developer.arm.com/documentation/102925/0100>, 2022. Accessed on 30 January, 2023.
- [41] Karl Naden, Robert Bocchino, Jonathan Aldrich, and Kevin Bierhoff. A type system for borrowing permissions. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’12*, page 557–570, New York, NY, USA, 2012. Association for Computing Machinery.
- [42] Learn the architecture: Providing protection for complex software. <https://developer.arm.com/documentation/102433/latest/>, 2022.
- [43] Joongun Park, Naegyeong Kang, Taehoon Kim, Youngjin Kwon, and Jaehyuk Huh. Nested enclave: Supporting fine-grained hierarchical isolation with sgx. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 776–789, 2020.

- [44] Jose Rodrigo Sanchez Vicarte, Benjamin Schreiber, Riccardo Paccagnella, and Christopher W. Fletcher. *Game of Threads: Enabling Asynchronous Poisoning Attacks*, page 35–52. Association for Computing Machinery, New York, NY, USA, 2020.
- [45] Moritz Schneider, Aritra Dhar, Ivan Puddu, Kari Kostainen, and Srdjan Čapkun. Composite enclaves: Towards disaggregated trusted execution. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):630–656, Nov. 2021.
- [46] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. Stktokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019.
- [47] Sjaak Smetsers, Erik Barendsen, Marko C. J. D. van Eekelen, and Marinus J. Plasmeijer. Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. In *Proceedings of the International Workshop on Graph Transformations in Computer Science*, page 358–379, Berlin, Heidelberg, 1993. Springer-Verlag.
- [48] Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. Linear capabilities for fully abstract compilation of separation-logic-verified code. *Proc. ACM Program. Lang.*, 3(ICFP), jul 2019.
- [49] Philip Wadler. Linear types can change the world! In *PROGRAMMING CONCEPTS AND METHODS*. North, 1990.
- [50] Philip Wadler. Is there a use for linear logic? In *Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '91, page 255–273, New York, NY, USA, 1991. Association for Computing Machinery.
- [51] Andrew Waterman, Krste Asanović, and John Hauser. *The RISC-V Instruction Set Manual: Volume II: Privileged Architecture*, 2021.
- [52] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for UNIX. In *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*, pages 29–46. USENIX Association, 2010.
- [53] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav H. Dave, Brooks Davis, Khilani Gudka, Ben Laurie, Steven J. Murdoch, Robert M. Norton, Michael Roe, Stacey D. Son, and Munraj Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 20–37. IEEE Computer Society, 2015.
- [54] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. Asyncshock: Exploiting synchronisation bugs in intel sgx enclaves. In Ioannis Askoxylakis, Sotiris Ioannidis, Sokratis Katsikas, and Catherine Meadows, editors, *Computer Security – ESORICS 2016*, 2016.
- [55] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilani Gudka, Alexandre Joannou, A. Theodore Marketos, Alfredo Mazinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. Cornucopia: Temporal safety for cheri heaps. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 608–625, 2020.
- [56] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 304–316, 2002.
- [57] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert M. Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *ISCA*, pages 457–468. IEEE Computer Society, 2014.
- [58] Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel Wesley Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. Cherivoke: Characterising pointer revocation using CHERI capabilities for temporal memory safety. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*, pages 545–557. ACM, 2019.
- [59] Jason Zhijiangcheng Yu, Shweta Shinde, Trevor Carlson, and Prateek Saxena. Elasticlave: An efficient memory model for enclaves. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association.

A Complete CAPSTONE Operational Semantics

Table 7 describes the operational semantics of the CAPSTONE model. The definitions of auxiliary functions are listed in Table 8. For simplicity, we omit the thread executing the instruction (i.e., k) from both the instruction and the auxiliary function arguments.

Table 7: Operational semantics of CAPSTONE.

Instruction	State transition Conditions
mov $r_d r_s$	$\Psi \mapsto \text{UpdatePC}(\Psi[\Theta.k.\theta.r_d \mapsto w][\Theta.k.\theta.r_s \mapsto \text{Moved}(w)])$ $\Psi = (\Theta, \text{mem}, rt, N) \wedge \Theta.k.\theta.r_s = w$
ld $r_d r_s$	$\Psi \mapsto \text{UpdatePC}(\Psi[\Theta.k.\theta.r_d \mapsto w][\text{mem.addr} \mapsto \text{Moved}(w)])$ $\Psi = (\Theta, \text{mem}, rt, N) \wedge \Theta.k.\theta.r_s = c \wedge \text{ValidCap}(rt, c) \wedge \neg \text{Revoked}(rt, c) \wedge \text{InBoundCap}(c) \wedge \text{AccessibleCap}(c) \wedge \text{ReadableCap}(c) \wedge c.a = \text{addr} \wedge \text{mem.addr} = w \wedge (\text{LinearCap}(w) \implies \text{WritableCap}(c))$
sd $r_d r_s$	$\Psi \mapsto \text{UpdatePC}(\Psi[\text{mem.addr} \mapsto w][\Theta.k.\theta.r_s \mapsto \text{Moved}(w)][\Theta.k.\theta.r_d \mapsto \text{UpdateCursor}(c)])$ $\Psi = (\Theta, \text{mem}, rt, N) \wedge \Theta.k.\theta.r_d = c \wedge \text{ValidCap}(rt, c) \wedge \text{InBoundCap}(c) \wedge \text{AccessibleCap}(c) \wedge \text{WritableCap}(c) \wedge c.a = \text{addr} \wedge \Theta.k.\theta.r_s = w$
tighten $r_d r_s$	$\Psi \mapsto \text{UpdatePC}(\Psi[\Theta.k.\theta.r_d \mapsto c'])$ $\Psi = (\Theta, \text{mem}, rt, N) \wedge \Theta.k.\theta.r_d = c \wedge \text{ValidCap}(rt, c) \wedge c = (t, b, e, a, p, n) \wedge \Theta.k.\theta.r_s = n \in \mathbb{N} \wedge c' = (t, b, e, a, \text{TightenPerm}(p, \text{DecodePerm}(n)))$
shrink $r_d r_b r_e$	$\Psi \mapsto \text{UpdatePC}(\Psi[\Theta.k.\theta.r_d \mapsto c'])$ $\Psi = (\Theta, \text{mem}, rt, N) \wedge \Theta.k.\theta.r_b = b' \in \mathbb{N} \wedge \Theta.k.\theta.r_e = e' \in \mathbb{N} \wedge \Theta.k.\theta.r_d = c \wedge \text{ValidCap}(rt, c) \wedge c = (t, b, e, a, p, n) \wedge t \in \{\text{Lin}, \text{Non}\} \wedge b \leq b' < e' \leq e \wedge c' = (t, b', e', a, p, n)$
split $r_d r_s r_p$	$\Psi \mapsto \text{UpdatePC}(\Psi[\Theta.k.\theta.r_d \mapsto c'_1][\Theta.k.\theta.r_s \mapsto c'_2][rt \mapsto rt'])$ $\Psi = (\Theta, \text{mem}, rt, N) \wedge \Theta.k.\theta.r_d = c \wedge \text{ValidCap}(rt, c) \wedge c = (\text{Lin}, b, e, a, p, n) \wedge \Theta.k.\theta.r_p = pv \wedge b < pv < e \wedge c'_1 = (\text{Lin}, b, pv, a, p, n) \wedge c'_2 = (\text{Lin}, pv, e, a, p, n) \wedge n' \in \mathbb{N} \wedge n' \notin \text{dom}(rt) \wedge rt' = rt[n' \mapsto rt.n]$
delin r	$\Psi \mapsto \text{UpdatePC}(\Psi[\Theta.k.\theta.r \mapsto c', rt \mapsto rt'])$ $\Psi = (\Theta, \text{mem}, rt, N) \wedge \Theta.k.\theta.r = c \wedge \text{ValidCap}(rt, c) \wedge c = (\text{Lin}, b, e, a, p, n) \wedge c' = (\text{Non}, b, e, a, p, n) \wedge rt.n = (n', nt) \wedge rt' = rt[n \mapsto (n', \text{RNon})]$
scc $r_d r_s$	$\Psi \mapsto \text{UpdatePC}(\Psi[\Theta.k.\theta.r_d \mapsto c'])$ $\Psi = (\Theta, \text{mem}, rt, N) \wedge \Theta.k.\theta.r_s = n \wedge n \in \mathbb{N} \wedge \Theta.k.\theta.r_d = c \wedge c \in \text{Cap} \wedge c' = c[a \mapsto n]$
lcc $r_d r_s$	$\Psi \mapsto \text{UpdatePC}(\Psi[\Theta.k.\theta.r_d \mapsto n])$ $\Psi = (\Theta, \text{mem}, rt, N) \wedge \Theta.k.\theta.r_s = c \wedge c \in \text{Cap} \wedge n = c.a$
revoke r	$\Psi \mapsto \text{UpdatePC}(\Psi[\Theta.k.\theta.r \mapsto c'][rt \mapsto rt'])$ $\Psi = (\Theta, \text{mem}, rt, N) \wedge \Theta.k.\theta.r = c \wedge \text{ValidCap}(rt, c) \wedge c = (\text{Rev}, b, e, a, p, n) \wedge rt' = \text{Reparent}(rt, n, \text{null}) \wedge c' = \begin{cases} (\text{Lin}, b, e, a, p, n) & (\mathbb{N} \times \{(n, \text{RLin})\}) \cap rt = \emptyset \vee p \in \{\text{NA}, \text{R}, \text{RX}\} \\ (\text{Uninit}, b, e, b, p, n) & \text{otherwise} \end{cases}$
mrev $r_d r_s$	$\Psi \mapsto \text{UpdatePC}(\Psi[rt \mapsto rt'][\Theta.k.\theta.r_d \mapsto c'])$ $\Psi = (\Theta, \text{mem}, rt, N) \wedge \Theta.k.\theta.r_s = c \wedge \text{ValidCap}(rt, c) \wedge c = (\text{Lin}, b, e, a, p, n) \wedge n' \in \mathbb{N} \wedge n' \notin \text{dom}(rt) \wedge c' = (\text{Rev}, b, e, a, p, n') \wedge rt' = (rt[n \mapsto (n', \text{RLin})]) \cup \{(n', rt.n)\}$
init r	$\Psi \mapsto \text{UpdatePC}(\Psi[\Theta.k.\theta.r = c'])$ $\Psi = (\Theta, \text{mem}, rt, N) \wedge \Theta.k.\theta.r = c \wedge \text{ValidCap}(rt, c) \wedge c = (\text{Uninit}, b, e, a, p, n) \wedge a = e \wedge c' = (\text{Lin}, b, e, a, p, n)$
drop r	$\Psi \mapsto \text{UpdatePC}(\Psi[rt \mapsto rt'][\Theta.k.\theta.r \mapsto 0])$ $\Psi = (\Theta, \text{mem}, rt, N) \wedge \Theta.k.\theta.r = c \wedge \text{ValidCap}(rt, c) \wedge c = (t, b, e, a, p, n) \wedge t \in \{\text{Lin}, \text{Rev}, \text{Uninit}, \text{Sealed}, \text{SealedRet}\} \wedge rt' = \text{Remove}(\text{Reparent}(rt, n, rt.n), n)$
seal r	$\Psi \mapsto \text{UpdatePC}(\Psi[\Theta.k.\theta.r \mapsto c'][N \mapsto \Psi.N + 1])$ $\Psi = (\Theta, \text{mem}, rt, N) \wedge \Theta.k.\theta.r = c \wedge \text{ValidCap}(rt, c) \wedge \text{ReadableCap}(c) \wedge \text{WritableCap}(c) \wedge c = (\text{Lin}, b, e, a, p, n) \wedge c' = (\text{Sealed}(\Psi.N), b, e, a, p, n)$
call $r_d r_s$	$\Psi \mapsto \Psi[\Theta.k \mapsto (\text{regs}', d)][\text{mem} \mapsto \text{mem}']$

	$\Psi = (\Theta, mem, rt, N) \wedge \Theta.k.\theta.r_d = c_i \wedge \Theta.k.\theta.r_s = w \wedge \text{ValidCap}(rt, c_i) \wedge c_i = (\text{Sealed}(d), b_i, e_i, a_i, p_i, n_i) \wedge b_i + M + 4 \leq e_i \wedge regs' = \text{LoadContext}(\Theta.k.\theta, b_i, mem)[ret \mapsto c_i[t \mapsto \text{SealedRet}(\Psi.k.d, r_d)]] [r_1 \mapsto w] \wedge mem' = \text{SaveContext}(mem, b_i, \Theta.k.\theta[r_s \mapsto \text{Moved}(w)])[r_d \mapsto 0]$
return $r_d r_s$	$\Psi \mapsto \Psi[\Theta.k \mapsto (regs', d)][mem \mapsto mem']$ $\Psi = (\Theta, mem, rt, N) \wedge \Theta.k.\theta.r_d = c_i \wedge \Theta.k.\theta.r_s = w \wedge \text{ValidCap}(rt, c_i) \wedge c_i = (\text{SealedRet}(d, r), b_i, e_i, a_i, p_i, n_i) \wedge b_i + M + 4 \leq e_i \wedge regs' = \text{LoadContext}(regs, mem, b_i)[r \mapsto w] \wedge mem' = \text{ClearSealed}(mem, b_i)$
retseal $r_d r_s$	$\Psi \mapsto \Psi[\Theta.k \mapsto (regs', d)][mem \mapsto mem']$ $\Psi = (\Theta, mem, rt, N) \wedge \Theta.k.\theta.r_d = c_i \wedge \Theta.k.\theta.r_s = w \wedge \text{ValidCap}(rt, c_i) \wedge c_i = (\text{SealedRet}(d, r), b_i, e_i, a_i, p_i, n_i) \wedge b_i + M + 4 \leq e_i \wedge regs' = \text{LoadContext}(\Theta.k.\theta, mem, b_i)[r \mapsto c_i[t \mapsto \text{Sealed}(\Psi.k.d)]] \wedge mem' = \text{SaveContext}(mem, b_i, \Theta.k.\theta[pc.a \mapsto w])[r_d \mapsto 0]$
except r	$= \text{call epc } r$ (without incrementing PC)
jmp r	$\Psi = \Psi[\Theta.k.\theta.pc \mapsto c']$ $\Psi = (\Theta, mem, rt, N) \wedge \Theta.k.\theta.r = n \wedge n \in \mathbb{N} \wedge c' = \Theta.k.\theta.pc[a \mapsto n]$
jnz $r_d r_s$	$\Psi = \Psi[\Theta.k.\theta.pc \mapsto c']$ $\Psi = (\Theta, mem, rt, N) \wedge \Theta.k.\theta.r_s = n_s \wedge \Theta.k.\theta.r_d = n_d \wedge n_s, n_d \in \mathbb{N} \wedge c' = \begin{cases} \text{IncrementCursor}(\Theta.k.\theta.pc) & n_s = 0 \\ \Theta.k.\theta.pc[a \mapsto n_d] & \text{otherwise} \end{cases}$
li $r n$	$\Psi \mapsto \text{UpdatePC}(\Psi[\Theta.k.\theta.r \mapsto n])$
add $r_d r_s$	$\Psi \mapsto \text{UpdatePC}(\Psi[\Theta.k.\theta.r_d \mapsto n'_d])$ $\Psi = (\Theta, mem, rt, N) \wedge \Theta.k.\theta.r_d = n_d \wedge \Theta.k.\theta.r_s = n_s \wedge n_d, n_s \in \mathbb{N} \wedge n'_d = n_d + n_s$
lt $r_d r_a r_b$	$\Psi = \text{UpdatePC}(\Psi[\Theta.k.\theta.r_d \mapsto n])$ $\Psi = (\Theta, mem, rt, N) \wedge \Theta.k.\theta.r_a = n_a \wedge \Theta.k.\theta.r_b = n_b \wedge n_a, n_b \in \mathbb{N} \wedge n = \begin{cases} 1 & n_a < n_b \\ 0 & \text{otherwise} \end{cases}$
invalid	$\Psi \mapsto \Psi[\Theta.k.\theta \mapsto \text{error}]$

The relation $\preceq \subseteq$ Perms \times Perms is defined as

$$\preceq = \begin{aligned} & \{NA\} \times \text{Perms} \cup \\ & \{R\} \times \{R, RW, RX, RWX\} \cup \\ & \{RW\} \times \{RW, RWX\} \cup \\ & \{RX\} \times \{RX, RWX\} \cup \{(RWX, RWX)\} \end{aligned} \quad (1)$$

Table 8: Auxiliary functions used in the operational semantics definition of CAPSTONE. k is the currently executing thread, and is omitted in the function arguments below for simplicity. It will be supplied as the first argument when necessary.

Function	Definition
LinearTypes	$\{\text{Lin}, \text{Uninit}, \text{Rev}, \text{Sealed}, \text{SealedRet}\}$
UpdatePC(Ψ)	$\begin{cases} \Psi[\Theta.k.\theta.pc.a \mapsto a + 1] & \Psi = (\Theta, mem, rt, N) \wedge \Theta.k.\theta.pc = (t, b, e, a, p, n) \\ \text{error} & \text{otherwise} \end{cases}$
ValidPC(Ψ)	$\Psi = (\Theta, mem, rt, N) \wedge \Theta.k.\theta.pc \in \text{Cap} \wedge \text{ExecutableCap}(\Theta.k.\theta.pc) \wedge \text{ValidCap}(rt, \Theta.k.\theta.pc) \wedge \text{InBoundCap}(\Theta.k.\theta.pc) \wedge \text{AccessibleCap}(\Theta.k.\theta.pc)$
ReadableCap(c)	$c = (t, b, e, a, p, n) \wedge p \in \{\text{R}, \text{RX}, \text{RWX}\} \wedge t \neq \text{Uninit}$
WritableCap(c)	$c = (t, b, e, a, p, n) \wedge p \in \{\text{RW}, \text{RWX}\}$
ExecutableCap(c)	$c = (t, b, e, a, p, n) \wedge p \in \{\text{RX}, \text{RWX}\} \wedge t \neq \text{Uninit}$
ValidCap(rt, c)	$c = (t, b, e, a, p, n) \wedge \neg \text{Revoked}(rt, n)$
Revoked(rt, n)	$\begin{cases} \text{F} & rt.n.pr = \text{root} \\ \text{T} & rt.n.pr = \text{null} \\ \text{Revoked}(rt, rt.n) & \text{otherwise} \end{cases}$
InBoundCap(c)	$c = (t, b, e, a, p, n) \wedge b \leq a < e$
AccessibleCap(c)	$c = (t, b, e, a, p, n) \wedge t \in \{\text{Lin}, \text{Non}, \text{Uninit}\}$
LinearCap(c)	$c = (t, b, e, a, p, n) \wedge t \in \text{LinearTypes}$
Moved(w)	$\begin{cases} 0 & \text{LinearCap}(w) \\ w & \text{otherwise} \end{cases}$
Reparent(rt, n, n')	$(rt - \mathbb{N} \times \{n\} \times \text{RNodeType}) \cup \{(k, n', nt) \mid (k, n, nt) \in rt\}$
Remove(rt, n)	$rt - \{n\} \times \text{RevParent} \times \text{RNodeType}$
DecodePerm(n)	$\begin{cases} \text{R} & 0 \\ \text{RW} & 1 \\ \text{RX} & 2 \\ \text{RWX} & 3 \\ \text{NA} & \text{otherwise} \end{cases}$
TightenPerm(p, p')	$\begin{cases} p' & p' \preceq p \\ \text{NA} & \text{otherwise} \end{cases}$
IncrementCursor(c)	$c[a \mapsto c.a + 1]$
UpdateCursor(c)	$\begin{cases} c & c \in \text{Cap} \wedge c.t \neq \text{Uninit} \\ \text{IncrementCursor}(c) & c \in \text{Cap} \wedge c.t = \text{Uninit} \\ 0 & \text{otherwise} \end{cases}$
FetchInsn(Ψ)	$\begin{cases} i & \Theta.k.\theta \in \text{RegFile} \wedge \text{ValidPC}(\Psi) \wedge \Psi.\Theta.k.\theta.pc = (t, b, e, a, p, n) \wedge i = \Psi.mem.a \wedge i \in \text{Insn} \\ \text{invalid} & \text{otherwise} \end{cases}$
LoadContext($regs, mem, b$)	$regs[pc \mapsto mem.b][epc \mapsto mem.(b + 2)][ret \mapsto mem.(b + 3)][r_1 \mapsto mem.(b + 4)][r_2 \mapsto mem.(b + 5)] \cdots [r_M \mapsto mem.(b + M + 3)]$
SaveContext($mem, b, regs$)	$mem[b \mapsto regs.pc][b + 2 \mapsto regs.epc][b + 3 \mapsto regs.ret][b + 4 \mapsto regs.r_1][b + 5 \mapsto regs.r_2] \cdots [b + M + 3 \mapsto regs.r_M]$
ClearSealed(mem, b)	$mem[b \mapsto 0][b + 1 \mapsto 0] \cdots [b + M + 3 \mapsto 0]$

B CAPSTONE_{abs}

B.1 Notations and Definitions

For any set $S \subset \mathbb{N}$, we use \bar{S} to denote its complement, i.e., $\mathbb{N} - S$.

We define the state transition function below.

Definition 1 (F, R). $F : \text{State} \times \mathbb{N} \rightarrow \text{State} \times \mathbb{N}$:

$$F(\Psi, \sigma) = (\Psi', \sigma'), \quad (2)$$

where

$$\Psi' = \begin{cases} \text{Execute}(\Psi, k, \text{FetchInsn}(\Psi, k)) & s = 0 \\ \text{Execute}(\Psi, k, \mathbf{except}) & \text{otherwise,} \end{cases} \quad (3)$$

and

$$(s, k, \sigma') = R(\sigma), \quad (4)$$

where

$$R : \mathbb{N} \rightarrow \{0, 1\} \times \mathbb{N} \times \mathbb{N}. \quad (5)$$

Definition 2 (Auxiliary functions). We define the following functions:

- $\text{SealedRegs}(c) = \{(\text{pc}, c.b), (\text{epc}, c.b + 2), (\text{ret}, c.b + 3), (\text{r}_1, c.b + 4), \dots, (\text{r}_M, c.b + M + 3)\}$.

Definition 3 (DLoc). We use DLoc to denote the location of a piece of data. There are two possibilities: it can be a memory location identified by its address or a register identified by the corresponding domain and the register name, hence

$$\text{DLoc} \ni \text{loc} := \text{DLocMem}(n) \mid \text{DLocReg}(n, r). \quad (6)$$

B.2 Abstract Model

- $\text{dom} := \text{user} \mid \text{sup} \mid \text{sub}$
- $\text{act}_{\text{abs}} :=$
 - load_linear addr payload |
 - store_linear addr payload |
 - load addr payload |
 - store addr payload |
 - split cap nat |
 - shrink cap nat |
 - send cap dom |
 - discard cap |
 - claim cap |
 - revoke cap
- $\text{mem}_{\text{abs}} := \text{Addr} \mapsto (\text{Word} \mid \text{unit})$
- $\text{range} := \{n \mid x \leq n < x + y\}$
- $\text{payload} := \text{Word}$
- $\text{cap}_{\text{abs}} := \text{range}$
- $\text{tstate} := \text{cap}_{\text{abs}} \text{ set}$
- $\text{pstate} := (\text{mem}_{\text{abs}}, \text{tstate}_{\text{user}}, \text{tstate}_{\text{sup}}, \text{tstate}_{\text{sub}})$

B.3 Abstract Semantics

$\text{Owns}(\text{tstate}, a) = \exists c. c \in \text{tstate} \wedge a \in c$

$\text{Split}(\{n \mid x \leq n < x + y\}, a) = \{\{n \mid x \leq n < a\}, \{n \mid a \leq n < x + y\}\}$

$\text{pstate} \xrightarrow[\text{dom}]{\text{act}_{\text{abs}}} \text{pstate}'$

load linear:

$(\text{mem}_{\text{abs}}, \text{tstate}_{\text{user}}, \text{tstate}_{\text{sup}}, \text{tstate}_{\text{sub}}) \xrightarrow[\text{dom}]{\text{load_linear addr payload}} (\text{mem}_{\text{abs}}, \text{tstate}_{\text{user}}, \text{tstate}_{\text{sup}}, \text{tstate}_{\text{sub}})$
 if $\text{payload} = \text{mem}_{\text{abs}}[\text{addr}] \wedge$

$\text{Owns}(tstate_{dom}, addr)$

store linear:

$(mem_{abs}, tstate_{user}, tstate_{sup}, tstate_{sub}) \xrightarrow[\text{dom}]{\text{store_linear } addr \text{ payload}} (mem'_{abs}, tstate_{user}, tstate_{sup}, tstate_{sub})$
 if $mem'_{abs} = mem_{abs}[addr := \text{payload}] \wedge$
 $\text{Owns}(tstate_{dom}, addr)$

load:

$(mem_{abs}, tstate_{user}, tstate_{sup}, tstate_{sub}) \xrightarrow[\text{dom}]{\text{load } addr \text{ payload}} (mem_{abs}, tstate_{user}, tstate_{sup}, tstate_{sub})$
 if $\text{payload} = mem_{abs}[addr] \wedge$
 $\nexists dom'. \text{Owns}(tstate_{dom'}, addr)$

store:

$(mem_{abs}, tstate_{user}, tstate_{sup}, tstate_{sub}) \xrightarrow[\text{dom}]{\text{store_linear } addr \text{ payload}} (mem'_{abs}, tstate_{user}, tstate_{sup}, tstate_{sub})$
 if $mem'_{abs} = mem_{abs}[addr := \text{payload}] \wedge$
 $\nexists dom'. \text{Owns}(tstate_{dom'}, addr)$

split:

$(mem_{abs}, tstate_{user}, tstate_{sup}, tstate_{sub}) \xrightarrow[\text{dom}]{\text{split } cap \ n} (mem_{abs}, tstate'_{user}, tstate'_{sup}, tstate'_{sub})$
 if $cap \in tstate_{dom} \wedge$
 $nat \in cap \wedge$
 $tstate'_{dom} = (tstate_{dom} - \{cap\}) \cup \text{Split}(cap, n) \wedge$
 $\forall dom' \neq dom. tstate'_{dom'} = tstate_{dom'}$

shrink:

$(mem_{abs}, tstate_{user}, tstate_{sup}, tstate_{sub}) \xrightarrow[\text{dom}]{\text{split } cap \ n} (mem_{abs}, tstate'_{user}, tstate'_{sup}, tstate'_{sub})$
 if $cap \in tstate_{dom} \wedge$
 $nat \in cap \wedge$
 $tstate'_{dom} = (tstate_{dom} - \{cap\}) \cup \text{Shrink}(cap, n) \wedge$
 $\forall dom' \neq dom. tstate'_{dom'} = tstate_{dom'}$

send:

$(mem_{abs}, tstate_{user}, tstate_{sup}, tstate_{sub}) \xrightarrow[\text{dom}]{\text{send } cap \ dom'} (mem_{abs}, tstate'_{user}, tstate'_{sup}, tstate'_{sub})$
 if $cap \in tstate_{dom} \wedge$
 $tstate'_{dom} = tstate_{dom} - \{cap\} \wedge$
 $tstate'_{dom'} = tstate_{dom'} \cup \{cap\} \wedge$
 $tstate'_{dom'' \neq dom \neq dom'} = tstate_{dom''}$

discard:

$(mem_{abs}, tstate_{user}, tstate_{sup}, tstate_{sub}) \xrightarrow[\text{dom}]{\text{discard } cap} (mem_{abs}, tstate'_{user}, tstate'_{sup}, tstate'_{sub})$
 if $cap \in tstate_{dom} \wedge$
 $tstate'_{dom} = tstate_{dom} - \{cap\} \wedge$
 $\forall dom' \neq dom. tstate'_{dom'} = tstate_{dom'}$

claim:

$(mem_{abs}, tstate_{user}, tstate_{sup}, tstate_{sub}) \xrightarrow[\text{dom}]{\text{claim } cap} (mem_{abs}, tstate'_{user}, tstate'_{sup}, tstate'_{sub})$
 if $cap \notin tstate_{user \cup sub \cup sup} \wedge$
 $tstate'_{dom} = tstate_{dom} \cup \{cap\} \wedge$
 $\forall dom' \neq dom. tstate'_{dom'} = tstate_{dom'}$

revoke:

$(mem_{abs}, tstate_{user}, tstate_{sup}, tstate_{sub}) \xrightarrow[\text{dom}]{\text{revoke } cap} (mem'_{abs}, tstate'_{user}, tstate'_{sup}, tstate'_{sub})$

Table 9: Actions in $\text{CAPSTONE}_{\text{abs}}$.

Action	Description
load_linear <i>addr payload</i>	load <i>payload</i> from memory
store_linear <i>addr payload</i>	store <i>payload</i> to memory
load <i>addr payload</i>	load <i>payload</i> from memory
store <i>addr payload</i>	store <i>payload</i> to memory
split <i>cap nat</i>	split <i>cap</i> in two at <i>nat</i>
shrink <i>cap nat</i>	shrink <i>cap</i> to length <i>nat</i>
send <i>cap dom</i>	send <i>cap</i> to another domain
discard <i>cap</i>	remove <i>cap</i> from the domain
claim <i>cap</i>	take ownership of unowned <i>cap</i>
revoke <i>cap</i>	wrest ownership of <i>cap</i> , setting relevant memory to uninit

if $tstate'_{dom} = tstate_{dom} \cup \{cap\} \wedge$
 $\forall dom' \neq dom. tstate'_{dom'} = tstate_{dom'} - \{cap\} \wedge$
 $(dom = \text{sub} \rightarrow cap \notin tstate_{\text{user}}) \wedge$
 $(mem'_{\text{abs}} = mem_{\text{abs}}[cap := \text{uninit}])$

An intuitive summary of the semantics of each abstract is provided in Table 9.

B.4 Refinement Mapping

Function from Ψ to $pstate$, with respect to a distinguished user domain $d \in \mathbb{N}$ and a distinguished set of subordinate domains $D_{\text{sub}} \subseteq (\mathbb{N} - d)$.

$\text{Refines}_{(d, D_{\text{sub}})}(\Psi, (mem_{\text{abs}}, tstate_{\text{user}}, tstate_{\text{sup}}, tstate_{\text{sub}}))$

where:

$ind_{\text{uninit}} = \bigcup_{d \in \mathbb{N}} \text{woranges}(\mathcal{R}_w(\Psi, d))$
 $mem_{\text{abs}} = \Psi.mem[ind_{\text{uninit}} := \text{uninit}]$
 $tstate_{\text{user}} = \text{ranges}(\mathcal{X}(\Psi, d))$
 $tstate_{\text{sub}} = \bigcup_{d' \in D_{\text{sub}}} \text{ranges}(\mathcal{X}(\Psi, d'))$
 $tstate_{\text{sup}} = \bigcup_{d' \in (\mathbb{N} - D_{\text{sub}} - d)} \text{ranges}(\mathcal{X}(\Psi, d'))$
 $\text{Sub}(\Psi, D_{\text{sub}})$

B.4.1 Subordinate Environment Invariant

$\text{Sub}(\Psi, D_{\text{sub}})$

where:

$\forall c \in \mathcal{X}(\Psi, d). \nexists c' \in (\bigcup_{k \in D_{\text{sub}}} \mathcal{C}(\Psi, k)). c'.t = \text{Rev} \wedge \text{range}(c) \cup \text{range}(c') \neq \emptyset$

Intuitively, a subordinate domain may never locally hold a revocation capability for a capability in the exclusive realm of the user domain. This is an *assumption* rather than a proof obligation. By default, all domains other than the user domain must be assumed to be part of the superordinate environment, unless the user has some trustable information regarding their behaviour (for example, because it created the other domain itself).

Table 10 presents a summarizing sketch of the mapping from instructions in the concrete model to abstract actions.

Table 10: Mapping from instructions to abstract actions.

Instruction	revoke r	delin r	shrink $r_d r_b r_e$
Action	revoke	discard	shrink
split $r_d r_s r_p$	drop r		
split	discard if r is linear, otherwise none		
ld $r_d r_s$	load_linear if r_s is linear, otherwise load and claim of any linear capability newly reachable through r_d after execution. In addition, store_linear or store as appropriate if a memory location was cleared after a linear capability was moved out of it.		
sd $r_d r_s$	store_linear if r_s is linear, otherwise store and discard of any linear capability moved from exclusive realm into shared memory.		
call $r_d r_s$ and except r	send of any linear capabilities transitively reachable through r_s , load_linear as appropriate to capture LoadContext, store_linear as appropriate to capture SaveContext.		
return $r_d r_s$	send of any linear capabilities transitively reachable through r_s , load_linear as appropriate to capture LoadContext, store_linear as appropriate to capture ClearSealed.		
retseal $r_d r_s$	load_linear as appropriate to capture LoadContext, store_linear as appropriate to capture ClearSealed and SaveContext (since retseal returns a sealed capability, no transfer of linear capabilities takes place in the abstract model).		

B.5 Required Concrete Well-Formedness Invariants

WF(Ψ)

if:

$$\begin{aligned}
 caps &= \bigcup_{d \in \mathbb{N}} \mathcal{R}(\Psi, d) \\
 \forall (loc, c) \in caps. \text{LinearCap}(c) &\longrightarrow \\
 \forall (loc', c') \in (caps - (loc, c)). & \\
 \bigcup \text{ranges}(\{(loc, c)\}) \cap \bigcup \text{ranges}(\{(loc', c')\}) &= \emptyset \vee (c'.t = \text{Rev} \wedge) \\
 \forall (loc, c) \in caps. c.t = \text{Rev} &\longrightarrow \\
 \exists (loc', c') \in (caps - (loc, c)). \bigcup \text{ranges}(\{(loc, c)\}) \cap \bigcup \text{ranges}(\{(loc', c')\}) &\neq \emptyset \wedge c'.t \in \{\text{Sealed}, \text{SealedRet}, \text{Lin}\} \longrightarrow \\
 (\mathbb{N} \times \{(n, \text{RLin})\}) \cap rt &\neq \emptyset
 \end{aligned}$$

Intuitively: (1) a capability may never overlap with any other capability, unless it is non-linear and (2) every linear capability in the state must have a corresponding entry in the revocation tree.

B.6 Statement of Correctness of Refinement Mapping

Define a function from concrete state and domain information to abstract action.

$$\text{Step}_{(d, D_{sub})}(\Psi, \sigma) :: (act_{abs}, dom)$$

Show that if

WF(Ψ) \wedge

Refines_(d, D_{sub})($\Psi, pstate$) \wedge

Step_(d, D_{sub})(Ψ, σ) = (act_{abs}, dom) \wedge

$F(\Psi, \sigma) = (\Psi', \sigma')$ \wedge

Refines_(d, D_{sub})($\Psi', pstate'$)

then

WF(Ψ') \wedge

$pstate \xrightarrow{act_{abs}^*}_{dom} pstate'$

B.7 Proof of Correctness of Refinement Mapping

We must define our step function and prove the above. We can proceed by case analysis on the instruction cases of our $F(\Psi, \sigma) = (\Psi', \sigma')$ and the underlying definition of Execute, defining the cases of Step inline by picking appropriate act_{abs} . Note that in each case the currently executing domain dom is given by the following, where $(s, k, \sigma') = R(\sigma)$:

$$dom = \begin{cases} \text{user} & \Psi.k.d = d \\ \text{sub} & \Psi.k.d \in D_{\text{sub}} \\ \text{sup} & \text{otherwise} \end{cases}$$

B.7.1 Trivial cases

mov $r_d r_s$
li $r n$
add $r_d r_s$
jnz $r_d r_s$
lt $r_d r_a r_b$
lcc $r_d r_s$
scc $r_d r_s$
invalid

These cases do not involve any changes to (the domains of) the capabilities in the system, nor to the state of the heap memory. Therefore $WF(\Psi')$ and $pstate = pstate'$ and $act_{abs}^* = \varepsilon$ and $pstate \xrightarrow{\varepsilon}_{dom} pstate'$

B.7.2 Interesting cases

ld $r_d r_s$

Assume

$\Psi = (\Theta, mem, rt, N)$

$pstate = (mem_{abs}, tstate_{user}, tstate_{sup}, tstate_{sub})$

$\Psi' = \text{UpdatePC}(\Psi[\Theta.k.\theta.r_d \mapsto w][mem.addr \mapsto \text{Moved}(w)])$

$\Theta.k.\theta.r_s = c$

$\text{ValidCap}(rt, c)$

$\neg \text{Revoked}(rt, c)$

$\text{InBoundCap}(c)$

$\text{AccessibleCap}(c)$

$\text{ReadableCap}(c)$

$c.a = addr$

$mem.addr = w$

(i.e. a successful concrete reduction step in the operational semantics)

Have $WF(\Psi')$ from assumptions+definitions (in particular, Moved case for linear caps ensures no overlap).

For the second top-level proof obligation we must consider two cases:

(1) consider the case $\neg \text{LinearCap}(c)$.

have $\forall d. w \notin \mathcal{X}(\Psi, d)$ from $WF(\Psi)$ and $mem.addr = w$ and definitions. (i.e. w can't be in an exclusive realm because it falls within the range of a non-linear cap, and no linear cap can overlap with the non-linear cap)

moreover, have $\forall d. addr \notin \text{woranges}(\mathcal{R}_w(\Psi, d))$ from $WF(\Psi)$, $\text{InBoundCap}(c)$, and definitions.

therefore $w = mem_{abs}[addr]$ and $\nexists dom'$. $\text{Owns}(tstate_{dom'}, addr)$

Two sub-cases:

(1.a) $\text{LinearCap}(w)$

(i.e. we are loading a linear cap from non-linear memory - in this case the loaded cap and all newly transitively-reachable linear caps enter our exclusive realm)

In this case we must also collect the linear capabilities transitively reachable from w , given by $\hat{w} = \text{cl}_{\text{Lin}}(\Psi, \{w\})$.

We choose $\text{act}_{\text{abs}}^* = (\text{load } \text{addr } w) (\text{claim } (\text{ranges}(\hat{w}))) (\text{store } \text{addr } \text{Moved}(w))$, and must therefore additionally show that $\text{ranges}(\hat{w}) \notin \text{tstate}_{\text{user} \cup \text{sup} \cup \text{sub}}$, from above and the definitions of Refines and \mathcal{X} .

(1.b) otherwise we have $\neg \text{LinearCap}(w)$ and $\text{act}_{\text{abs}}^* = (\text{load } \text{addr } w) (\text{store } \text{addr } \text{Moved}(w))$

(2) second top-level case, if $\text{LinearCap}(c)$ then $\text{act}_{\text{abs}} = (\text{load_linear } \text{addr } w) (\text{store_linear } \text{addr } \text{Moved}(w))$

have $w \in \mathcal{X}(\Psi, \text{dom})$ from $\text{WF}(\Psi)$ and $\text{mem.addr} = w$ and definitions. (i.e. w must be in the current domain's exclusive realm, as it's accessed through a linear cap held in a local register)

moreover, have $\forall d. \text{addr} \notin \text{woranges}(\mathcal{R}_w(\Psi, d))$ from $\text{WF}(\Psi)$, $\text{InBoundCap}(c)$, and definitions.

therefore $w = \text{mem}_{\text{abs}}[\text{addr}]$ and $\text{Owns}(\text{tstate}_{\text{dom}}, \text{addr})$

therefore $\text{pstate} \xrightarrow{\text{act}_{\text{abs}}}_{\text{dom}} \text{pstate}'$

QED

sd $r_d r_s$

Assume:

$\Psi = (\Theta, \text{mem}, \text{rt}, N)$

$\text{pstate} = (\text{mem}_{\text{abs}}, \text{tstate}_{\text{user}}, \text{tstate}_{\text{sup}}, \text{tstate}_{\text{sub}})$

$\Psi' = \text{UpdatePC}(\Psi[\text{mem.addr} \mapsto w][\Theta.k.\theta.r_s \mapsto \text{Moved}(w)][\Theta.k.\theta.r_d \mapsto \text{UpdateCursor}(c)])$

$\Theta.k.\theta.r_d = c$

$\text{ValidCap}(\text{rt}, c)$

$\text{InBoundCap}(c)$

$\text{AccessibleCap}(c)$

$\text{WritableCap}(c)$

$c.a = \text{addr} \wedge \Theta.k.\theta.r_s = w$

have $\text{WF}(\Psi')$ from definitions (in particular, Moved case for linear caps ensures no overlap).

For the second top-level proof obligation we must consider two cases:

(1) first, consider the case $\neg \text{LinearCap}(c)$

have $\forall d. w \notin \mathcal{X}(\Psi, d)$ from $\text{WF}(\Psi)$ and $\text{mem.addr} = w$ and definitions. (i.e. w can't be in an exclusive realm because it falls within the range of a non-linear cap, and no linear cap can overlap with the non-linear cap)

moreover, have $\forall d. \text{addr} \notin \text{woranges}(\mathcal{R}_w(\Psi, d))$ from $\text{WF}(\Psi)$, $\text{InBoundCap}(c)$, and definitions.

therefore $w = \text{mem}_{\text{abs}}[\text{addr}]$ and $\nexists \text{dom}'. \text{Owns}(\text{tstate}_{\text{dom}'}, \text{addr})$

Two sub-cases:

(1.a) $\text{LinearCap}(w)$

(i.e. we are storing a linear cap into non-linear memory - in this case the stored cap and all transitively-reachable linear caps leave our exclusive realm)

In this case we must also collect the linear capabilities transitively reachable from w , given by $\hat{w} = \text{cl}_{\text{Lin}}(\Psi, \{w\})$.

We choose $\text{act}_{abs}^* = (\text{store } \text{addr } w) (\text{discard } (\text{ranges}(\hat{w})))$.

(1.b) otherwise we have $\neg \text{LinearCap}(w)$ and $\text{act}_{abs}^* = \text{store } \text{addr } w$

(2) second top-level case, if $\text{LinearCap}(c)$ then $\text{act}_{abs} = \text{store_linear } \text{addr } w$

have $w \in \mathcal{X}(\Psi, \text{dom})$ from $\text{WF}(\Psi)$ and $\text{mem.addr} = w$ and definitions. (i.e. w must be in the current domain's exclusive realm, as it's held in a local register)

moreover, have $\forall d. \text{addr} \notin \text{woranges}(\mathcal{R}_w(\Psi, d))$ from $\text{WF}(\Psi)$, $\text{InBoundCap}(c)$, and definitions.

therefore $w = \text{mem}_{abs}[\text{addr}]$ and $\text{Owns}(t\text{state}_{\text{dom}}, \text{addr})$

therefore $p\text{state} \xrightarrow{\text{act}_{abs}^*} p\text{state}'$

QED

seal r

This case is trivial, with $\text{WF}(\Psi')$ and $p\text{state} = p\text{state}'$ and $\text{act}_{abs}^* = \epsilon$, however it's worth noting *why* explicitly - sealing a capability does not alter the domains of any capabilities reachable through it. Note that a domain sealed inside a linear capability has no control in general over when the capability is unsealed, so the abstract model simply over-approximates that all sealed capabilities are immediately available to the sealed domain.

call $r_d r_s$

Assume:

$\Psi = (\Theta, \text{mem}, \text{rt}, N)$

$p\text{state} = (\text{mem}_{abs}, \text{tstate}_{\text{user}}, \text{tstate}_{\text{sup}}, \text{tstate}_{\text{sub}})$

$\Psi' = \Psi[\Theta.k \mapsto (\text{regs}', d)][\text{mem} \mapsto \text{mem}']$

$\Theta.k.\theta.r_d = c_i$

$\Theta.k.\theta.r_s = w$

$\text{ValidCap}(rt, c_i)$

$c_i = (\text{Sealed}(d), b_i, e_i, a_i, p_i, n_i)$

$b_i + M + 4 \leq e_i$

$\text{regs}' = \text{LoadContext}(\Theta.k.\theta, b_i, \text{mem})[\text{ret} \mapsto c_i[t \mapsto \text{SealedRet}(\Psi.k.d, r_d)]][\text{r}_1 \mapsto w]$

$\text{mem}' = \text{SaveContext}(\text{mem}, b_i, \Theta.k.\theta[\text{r}_s \mapsto \text{Moved}(w)]][\text{r}_d \mapsto 0]$

We must show that the capability ownership changes in the abstract domains are congruent with the changes in the concrete domains. First, note that the only way that a capability switches domains as a result of executing **call** is if the argument register r_s contains a capability, since the cap in **sc** is linear, so **SaveContext** will only move capabilities within the same domain. Moreover, **LoadContext** loads capabilities into the registers from c_i , a sealed capability with domain d , but the currently executing domain is also switched to d , so no transfer takes place. If w is a linear capability, we define $\hat{w} = \text{cl}_{\text{Lin}}(\Psi, \{w\})$, otherwise we define $\hat{w} = \epsilon$. Then $\text{act}_{abs}^* = \text{send } (\text{ranges}(\hat{w})) d$, plus appropriate load/store actions to handle **SaveContext** and **LoadContext** (trivial details of these omitted).

To show $p\text{state} \xrightarrow{\text{act}_{abs}^*} p\text{state}'$, we must show that the capability ownership changes in the abstract domains are congruent with the changes in the concrete domains. We can show this by observing that all registers in the current domain are saved to a sealed capability for the same domain (i.e. no domain transfer), except r_d and r_s which are cleared appropriately, with the contents of r_d being loaded into the newly-executing domain, and the transfer of the contents of r_s via r_1 captured by our choice of **send** action.

return $r_d r_s$

Essentially the same as **call**, with appropriate memory actions for the LoadContext, ClearSealed, and SaveContext operations in the concrete semantics.

retseal $r_d r_s$

Essentially the same as **call**, with appropriate memory actions for the LoadContext, ClearSealed, and SaveContext operations in the concrete semantics.

drop r

Maps straightforwardly to the discard operation.

revoke r

Assume:

$$\Psi = (\Theta, mem, rt, N)$$

$$pstate = (mem_{abs}, tstate_{user}, tstate_{sup}, tstate_{sub})$$

$$\Psi' = \text{UpdatePC}(\Psi[\Theta.k.\theta.r \mapsto c'][rt \mapsto rt'])$$

$$\Theta.k.\theta.r = c \wedge \text{ValidCap}(rt, c)$$

$$c = (\text{Rev}, b, e, a, p, n)$$

$$rt' = \text{Reparent}(rt, n, \text{null})$$

$$c' = \begin{cases} (\text{Lin}, b, e, a, p, n) & (\mathbb{N} \times \{(n, \text{RLin})\} \cap rt) = \emptyset \\ (\text{Uninit}, b, e, b, p, n) & \text{otherwise} \end{cases}$$

Have $\text{WF}(\Psi')$ from assumptions+definitions (in particular, following the definition of Reparent to ensure the revocation tree is updated correctly).

Two cases:

$$\text{First, } (\mathbb{N} \times \{(n, \text{RLin})\} \cap rt) = \emptyset$$

$$\text{We pick } act_{abs}^* = \text{claim}(\text{ranges}(\{c\}))$$

Must show that $\text{ranges}(\{c\} \notin tstate_{user} \cup sub \cup sup$. This follows from $\text{WF}(\Psi)$ and the above (i.e. since there is no entry in the revocation tree, by the definition of WF there can be no linear capability in any exclusive realm that overlaps with c).

$$\text{Second, } (\mathbb{N} \times \{(n, \text{RLin})\} \cap rt) \neq \emptyset$$

$$\text{We pick } act_{abs}^* = \text{revoke}(\text{ranges}(\{c\}))$$

To establish $pstate \xrightarrow[dom]{act_{abs}^*} pstate'$, we must show

$$(a) \text{ } (dom = \text{sub} \longrightarrow cap \notin tstate_{user})$$

$$(b) \text{ } (mem'_{abs} = mem_{abs}[cap := \text{uninit}])$$

(a) follows from the invariant $\text{Sub}(\Psi, D_{sub})$ that is contained within Reifies, which enforces that c cannot overlap with any user domain capability.

(b) holds by the definition of Reifies, as we know in this case that $c'.t = \text{Uninit}$

delin r

Essentially the same as the `Lin` case of **drop**.

mrev $r_d r_s$

This case is trivial, with $\text{WF}(\Psi')$ and $pstate = pstate'$ and $act_{abs}^* = \epsilon$, however it's worth noting *why* explicitly - the abstract model permits domains to revoke at any time, essentially over-approximating that they have already minted all of the revocation capabilities that they possibly can (with reference to the restrictions on the subordinate environment).

tighten $r_d r_s$

This case is trivial, with $\text{WF}(\Psi')$ and $pstate = pstate'$ and $act_{abs}^* = \epsilon$, however it's worth noting *why* explicitly - the abstract model over-approximates that owning a linear capability grants full permissions to the relevant underlying memory. This is acceptable since we don't allow both a separate read-only and write-only linear cap for the same memory location to exist at once.

split $r_d r_s r_p$

Maps straightforwardly to the `split` abstract action.

shrink $r_d r_b r_e$

Maps straightforwardly to the `shrink` abstract action.

init r

Another trivial case, since the abstract model over-approximates that uninitialized capabilities confer arbitrary access to any memory that is not `uninit`.

except r

The concrete model defines exceptions in terms of calls, so identical to the `call` case above.

B.8 Auxiliary Definitions

Definition 4 (Capability closure). For any $\Psi \in \text{State}$, $S \subseteq \text{DLoc} \times \text{Cap}$ and $T \subseteq \text{CapType}$, we inductively define the capability closure of S with respect to T at Ψ , denoted as $\text{cl}_T(\Psi, S)$, as follows

- $\{s \mid s \in S \wedge s.t \in T\} \subseteq \text{cl}_T(\Psi, S)$
- If $\text{ValidCap}(\Psi.rt, \Psi.mem.u) \wedge \Psi.mem.u.t \in T \wedge b \leq u < e \wedge c' = (t, b, e, a, p, n)$ where $(loc, c') \in \text{cl}_T(\Psi, S)$, then $(\text{DLocMem}(u), \Psi.mem.u) \in \text{cl}_T(\Psi, S)$

Definition 5 (Execution context). For any domain $d \in \mathbb{N}$, $\Psi \in \text{State}$, its execution context $\mathcal{C}(\Psi, d)$ is defined as

$$\mathcal{C}(\Psi, d) = \begin{cases} \{(\text{DLocReg}(k, u), \Psi.\Theta.k.\theta.u) \mid u \in \mathbb{N}\} & \exists k \in \mathbb{N}, \Psi.\Theta.k.d = d \\ \{(\text{DLocMem}(u, \Psi.mem.u)) \mid (r, u) \in \text{SealedRegs}(c)\} & \exists c \in \text{Dom}(\Psi.mem) \cup \bigcup_{k \in \mathbb{N}} \text{Dom}(\Psi.\Theta.k.\theta), \text{ValidCap}(c) \wedge \\ & c.t \in \{\text{Sealed}(d), \text{SealedRet}(d)\} \\ \emptyset & \text{otherwise.} \end{cases} \quad (7)$$

Definition 6 (Realm). For any domain $d \in \mathbb{N}$, $\Psi \in \text{State}$, its realm is defined as

$$\mathcal{R}(\Psi, d) = \text{cl}_T(\Psi, S), \quad (8)$$

where

$$T = \{\text{Lin}, \text{Non}\}, \quad (9)$$

and

$$S = \{(loc, c) \mid \text{ValidCap}(\Psi.rt, c) \wedge c.t \in T, (loc, c) \in \mathcal{C}(\Psi, d)\}. \quad (10)$$

Definition 7 (Exclusive realm). For any domain $d \in \mathbb{N}$, $\Psi \in \text{State}$, its exclusive realm is defined as

$$\mathcal{X}(\Psi, d) = \text{cl}_{T_x}(\Psi, S_x), \quad (11)$$

where

$$T_x = \{\text{Lin}\}, \quad (12)$$

and

$$S_x = \{(loc, c) \mid \text{ValidCap}(\Psi.rt, c) \wedge c.t \in T_x, (loc, c) \in \mathcal{C}(\Psi, d)\}. \quad (13)$$

We can easily define a function that converts realms and exclusive realms to sets of ranges:

$$\text{ranges}(\mathcal{R}) = \{\{b, b+1, \dots, e-1\} \mid (loc, (t, b, e, a, p, n)) \in \mathcal{R}\}. \quad (14)$$

This following is the counterpart of a realm for write-only memory (for uninitialized capabilities).

Definition 8 (Write-only realm). For any $d \in \mathbb{N}$, $\Psi \in \text{State}$, the write-only realm of d in Ψ is defined as

$$\mathcal{R}_w(\Psi, d) = \{(loc, c) \mid (loc, c) \in \hat{\mathcal{R}}(\Psi, d), \text{ValidCap}(\Psi.rt, c), c.t = \text{Uninit}\}, \quad (15)$$

where

$$\hat{\mathcal{R}}(\Psi, d) = \{(\text{DLocMem}(u), \Psi.mem.u) \mid u \in \text{range} \in \text{ranges}(\mathcal{R}(\Psi, d))\} \cup \mathcal{C}(\Psi, d). \quad (16)$$

Definition 9 (Exclusive write-only realm). For any $d \in \mathbb{N}$, $\Psi \in \text{State}$, the exclusive write-only realm of d in Ψ is defined as

$$\mathcal{X}_w(\Psi, d) = \{(loc, c) \mid (loc, c) \in \hat{\mathcal{X}}(\Psi, d), \text{ValidCap}(\Psi.rt, c), c.t = \text{Uninit}\}, \quad (17)$$

where

$$\hat{\mathcal{X}}(\Psi, d) = \{(\text{DLocMem}(u), \Psi.mem.u) \mid u \in \text{range} \in \text{ranges}(\mathcal{X}(\Psi, d))\} \cup \mathcal{C}(\Psi, d). \quad (18)$$

We can define a function that converts a write-only or exclusive write-only realm to ranges:

$$\text{woranges}(\mathcal{R}_w) = \{\{a, a+1, \dots, e-1\} \mid (loc, (t, b, e, a, p, n)) \in \mathcal{R}_w\}. \quad (19)$$