Universität des Saarlandes
Max-Planck-Institut für Softwaresysteme
Max-Planck-Institut für Informatik
IMPRS-CS

A Formal Model for Capability Machines

An Illustrative Case Study towards Secure Compilation to CHERI

Masterarbeit im Fach Informatik
Master's Thesis in Computer Science
von / by

Akram El-Korashy

angefertigt unter der Leitung von / supervised by

Deepak Garg, Ph.D.

betreut von / advised by

Marco Patrignani, Ph.D.

begutachtet von / reviewers

Deepak Garg, Ph.D.

Jan Reineke, Ph.D.

Saarland Informatics Campus (SIC), September 2016

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, September 2016 Akram El-Korashy

Abstract

Vulnerabilities in computer systems arise in part due to programmer's logical errors, and in part also due to programmer's false (i.e., over-optimistic) expectations about the guarantees that are given by the abstractions of a programming language.

For the latter kind of vulnerabilities, architectures with hardware or instruction-level support for protection mechanisms can be useful. One trend in computer systems protection is hardware-supported enforcement of security guarantees/policies. Capability-based machines are one instance of hardware-based protection mechanisms. CHERI is a recent implementation of a 64-bit MIPS-based capability architecture with byte-granularity memory protection.

The goal of this thesis is to provide a paper formal model of the CHERI architecture with the aim of formal reasoning about the security guarantees that can be offered by the features of CHERI. We first give simplified instruction operational semantics, then we prove that capabilities are unforgeable in our model. Second, we show that existing techniques for enforcing control-flow integrity can be adapted to the CHERI ISA. Third, we show that one notion of memory compartmentalization can be achieved with the help of CHERI's memory protection. We conclude by suggesting other security building blocks that would be helpful to reason about, and laying down a plan for potentially using this work for building a secure compiler, i.e., a compiler that preserves security properties.

The outlook and motivation for this work is to highlight the potential of using CHERI as a target architecture for secure compilation.

Acknowledgements

I am grateful and beholden to numerous people since embarking on my Master's study and even earlier. So this is an opportunity to express the fraction that my words can capture of my heartfelt and candid *Thank-you*'s to all the considerate, thoughtful, helpful, and supportive people that I am or ever have been blessed with in my life.

Thank you my parents, my grandma, my late grandpa, my brother, my sister, my sister-in-law, and my nephew, for your unconditional care, love, and support.

Thank you, every one of my teachers and supervisors; in school, at home, online, in the GUC, at DFKI, and in Saarland for your time, effort and friendliness, and even sometimes your very personalized advice and help. I am forever indebted to every one of you.

Thank you, friends, former colleagues at work, colleagues in Saarland, at MPI-SWS and at MPI-INF, and foremost the dear friends here in Saarbrücken with whom I share the moments of joy and stress (and food and hiking!) for your fun, care, support, honesty, and delight.

Thank you Eslam and Miguel for being amazing and humorous officemates.

Thank you Marco Patrignani for being friendly, supportive and caring, for all the time that you dedicate to my innumerable impromptu jumps that I make into your office!, and for the detailed, careful and constructive feedback that you continue to give me.

Thank you Deepak Garg for your continuous inspiration, encouragement, support, understanding and friendliness, for the great opportunities that you have offered and continue to offer me, and for the countless ideas and thoughts that I keep learning from you.

Thanks to the International Max-Planck Research School for Computer Science (IMPRS-CS) for the priceless support of my study and my administrative issues, without which I would not have been able to make it to Saarland in the first place, and for involving us in the research community at the Max-Planck Institutes.

Thank you Jan Reineke for your kind acceptance to review this thesis.

Contents

Chapter 1

Introduction

Vulnerabilities in computer systems arise in part due to programmer's logical errors [1], but in part also due to programmer's false (i.e., over-optimistic) expectations about the guarantees that are given by the abstractions of a programming language [2].

For the latter kind of vulnerabilities, architectures with hardware or instruction-level support [3, 4, 5, 6, 7, 8, 9, 10] for protection mechanisms can be useful.

One trend in computer systems protection is hardware-supported enforcement of security guarantees/policies. Capability-based machines are one instance of hardware-based protection mechanisms. CHERI [4, 11] is a recent implementation of a 64-bit MIPS-based capability architecture with byte-granularity memory protection.

1.1 Goals and Contribution

The goal of this thesis is to provide a formal model that makes it easy to reason on paper about a simplified but realistic version of the CHERI architecture. We provide simplified instruction operational semantics for the CHERI ISA [11]. We use it to show and reason about some useful security goals achievable by the architecture. The end goal of this pursuit is to make easy and evidence-supported the use of the CHERI architecture as a target of secure compilation based on full abstraction [12, 13, 14, 15] in the future. It should be made clear that our goal is distinct from the goal of verifying the actual CHERI ISA. The REMS project [16] at Cambridge is taking on such a direction with the help of tools like L3 [17, 18].

The thesis is organized into a Background chapter, followed by three main ones, and a concluding chapter discussing future work.

The background chapter (Chapter 2) highlights in some detail the features of the CHERI architecture as part of a discussion of existing literature.

Next comes the description of our formal model (Chapter 3) which provides a rewriting of the semantics of the CHERI instructions in a formal and simplified way. We show that capability unforgeability is preserved by arbitrary execution.

Then, in Chapter 4, we show that the policy of control-flow integrity [19, 20, 21] can be easily adapted to the CHERI ISA by adapting the machine code instrumentation mechanism in [19, 21] to our formal model.

Next, we introduce a basic notion of memory compartmentalization (Chapter 5) and we show that it is enforceable by CHERI's memory protection.

In the end (Chapter 6), we discuss potential development of this work, which –we think– can be used as a building block for writing a fully-abstract compiler.

Chapter 2

Background

In this chapter, we present the CHERI architecture [11] as one of the most recent capability machine models [4]. We do so in the context of discussing previous literature [22, 23, 24] on potential design choices for capability-based machines. We also briefly discuss other forms of hardware support like SAFE [5] that are built with the goal of achieving security requirements on the instruction set and memory word level. Later in the chapter, we discuss Control Flow Integrity [20] and memory compartmentalization [25, 26, 27].

2.1 Hardware Security and Capability Machines

The assumption on which the trend [3, 4, 5] of hardware-based enforcement mechanisms for protection or security goals lies is that processing power abounds [5]. This makes it appealing to provide hardware-based support for protection mechanisms [28, 29] aiming at mitigating potential risks from adversarial machine code. A general motivation for developing protection mechanisms is the attempt to achieve more fine-grained decomposition of software and of security-policies in order to realize more adherence of the various system abstraction layers to the principle of least privilege [29]. We focus on reviewing hardware architectures that have built-in support for capabilities [22]. But we first point out some alternative existing research directions on hardware support for security.

Hardware Security

Various hardware support features have been suggested that provide various kinds of support for protection mechanisms. Memory protection in the form of bounds-checking has particularly been the focus of Intel Memory Protection Extensions (iMPX) [6] and Hardbounds [7]. The protection techniques in both of these rely on the availability of the bounds information in a table. A fat-pointer model helps the memory-access instruction to access this table and make the necessary check [4]. In iMPX, bounds checking must be explicitly done by means of specific instructions [4].

Another variety of protection mechanisms that is more general than bounds checking is the idea of generalized security policy enforcement [9, 8, 10]. The PUMP machine [8] extended the work on dynamic security policy enforcement such as [10] by describing architectural support for general tag-based policy specification and dynamic enforcement. According to [9]: *"The PUMP architecture associates each piece of data in the system with a metadata tag describing its provenance or purpose (e.g., this is an instruction, this came from the network, this is secret, this is sealed with key k), propagates this metadata as instructions are executed, and checks that policy rules are obeyed throughout the computation."*

Various policies have been shown to be expressible in the micropolicies architecture (that the PUMP machine supports) [9] including Control-Flow Integrity [19], Memory Safety, Dynamic Sealing, and others.

Capability Machines

We now turn to the capability machine model that is the focus of our work. CHERI [11, 4] is one implementation of a capability-based hardware architecture.

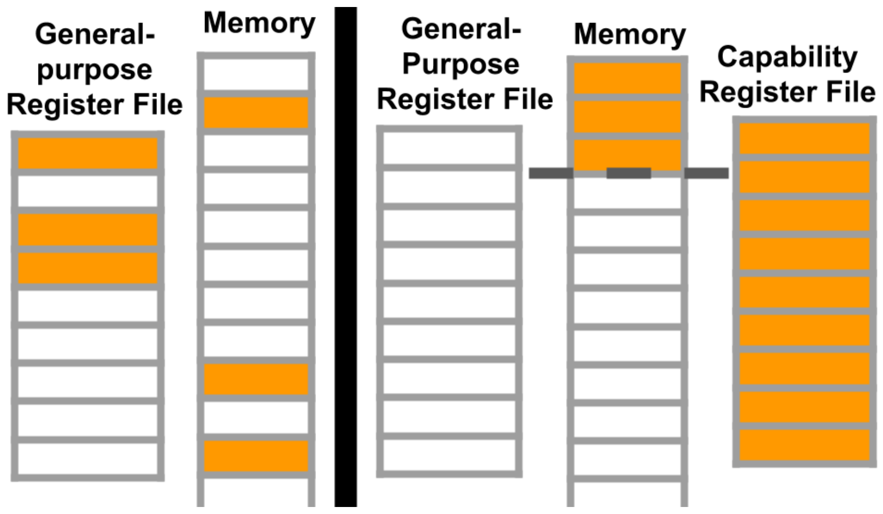
A capability is an unforgeable token that gives its owner permission(s) to access a particular entity or object in a computer system [22]. Capabilities have been the basis of operating systems protection mechanisms in various research projects like Capsicum [30], EROS [31], and other older operating systems [32]. Here we focus on reviewing capabilities as a hardware mechanism for protection goals [28].

Capabilities often serve two purposes; identifying an object, and allowing or denying particular operations to be performed on it [22]. Thinking of capabilities as an addressing mechanism has been characteristic of capability-based computer systems. Hardware support of capability-based addressing is an idea that has been around at least since 1974 when Fabry [23] discussed potential implementation ideas and argued that such addressing schemes could be an efficient solution to memory protection. Among the ideas discussed in Fabry's work [23] are ways for maintaining the integrity of capabilities, namely, the **tagged approach** and the **partition approach**.

Carter et al. [33] from MIT used the tagged approach pointed out by Fabry in [23, Section: Integrity of capabilities] and by Saltzer in [29, B. The Capability System] and introduced the idea of guarded pointers which was the basis of the capability-based addressing employed in the experimental design of the M-Machine [24], an architecture for a multicomputer by MIT. The general idea of a tagged approach is to have a tag that helps distinguish whether a word is meant to represent a normal data word or a capability. The design of the instruction set guarantees that whenever an arbitrary data operation is performed on a word, the word gets the "data" tag instead of the "capability tag". Authorization using a capability which takes place upon an addressing operation checks first that the word presented as a capability indeed carries a valid "capability" tag, not a "data" tag.

The partition approach, on the other hand, disallows confusion about what a word represents by segregating between data words and capabilities by means of, e.g., having two separate register files where each is allowed to live. A certain limited set of operations only is allowed on the capability register file. A pure extension of this approach to memory organization is that each program gets two segments of memory, one for data and one for capabilities. The operations that are allowed on the capability segment of the memory are guaranteed to be only those allowed for the capability register file. Figure 2.1 illustrates the difference between memory and register file organization by the two approaches.

FIGURE 2.1: Difference between tagged and partitioned approaches for organization of data and capabilities. In the tagged approach (left), both the memory and the general-purpose register file are legal regions for both capabilities (orange) and data (white) to live in. In the partitioned approach (right), memory is partitioned based on whether the values that are allowed to exist are valid capabilities.



The CHERI architecture [11] does not follow any one of these purely. Instead, it combines both the tagged memory approach with the partitioned register file approach. More details are discussed in the context of stating Theorem 3.1 on capability unforgeability in our simplified formal model of CHERI. Briefly, capability manipulation is guarded (i.e., only a limited set of instructions are available on the capability register file) to guarantee monotonicity and to prevent privilege escalation [11].

A feature worth noting about an instruction set that uses a capability-based addressing mechanism is that every instruction which addresses/accesses a memory word or an input/output device must specify a capability for the object to be accessed [23]. Specifying a capability might mean naming or presenting it as an argument or specifying it by any other implicit means (e.g., always look it up in a predefined register).

2.2 The CHERI Capability Machine

CHERI [4, 11] is a capability-based machine model that is based on the 64-bit MIPS ISA [34]. It combines conventional memory management unit (MMU) design choices with a capability-system model that is built on a RISC ISA [4].

Unlike conventional virtual memory, protection using capabilities is thought of as being the responsibility of the compiler, language runtime, and operating system kernel along with the properties of the capabilities themselves that automatically provide guarantees by means of bounds checking, permissions checking, and capability integrity [11].

The kinds of protection that are intended by the design of the CHERI ISA include [11]:

- spatial memory safety,
- temporal memory safety,
- software compartmentalization, and
- enforcing language-level properties by hardware assistance.

CHERI capabilities are tokens of authority that give permissions to access a specified memory region [11, 4]. The memory region is defined by a base address and a length. The permissions field specifies the kind of access, i.e., it specifies which operations this capability enables its owner to perform. Possible operations are loading, storing and executing code on a memory region. There are other operations like loading capabilities into the capability register file (as opposed to loading arbitrary memory words) and storing a capability register (as opposed to storing a general-purpose register). And there are others, in addition to some unused bits that can be used and checked arbitrarily without them having a predefined effect/authority on CHERI instructions/operations [11].

One of the other permissions that are built-in is the “non-ephemeral” permission. In newer publications about CHERI [11], this is now called the “Global” permission.

This is a permission that allows a capability to be stored anywhere in memory (i.e., globally) as opposed to only in regions that explicitly allow storing “local” capabilities. This distinction can be used to enable revocability of a capability. Local capabilities are revocable after they have been passed to a callee because they cannot be stored in memory. One possible extension to this idea of having a local/global permission bit is to have multiple security/clearance levels [35]. This possible extension is suggested in the CHERI technical report [11] but has been proposed at least since 1975 by Saltzer and Schroeder [29]. More discussion of the idea of an ephemeral capability can be found in Section 3.1.

CHERI Capabilties in detail

FIGURE 2.2: Format of a CHERI capability

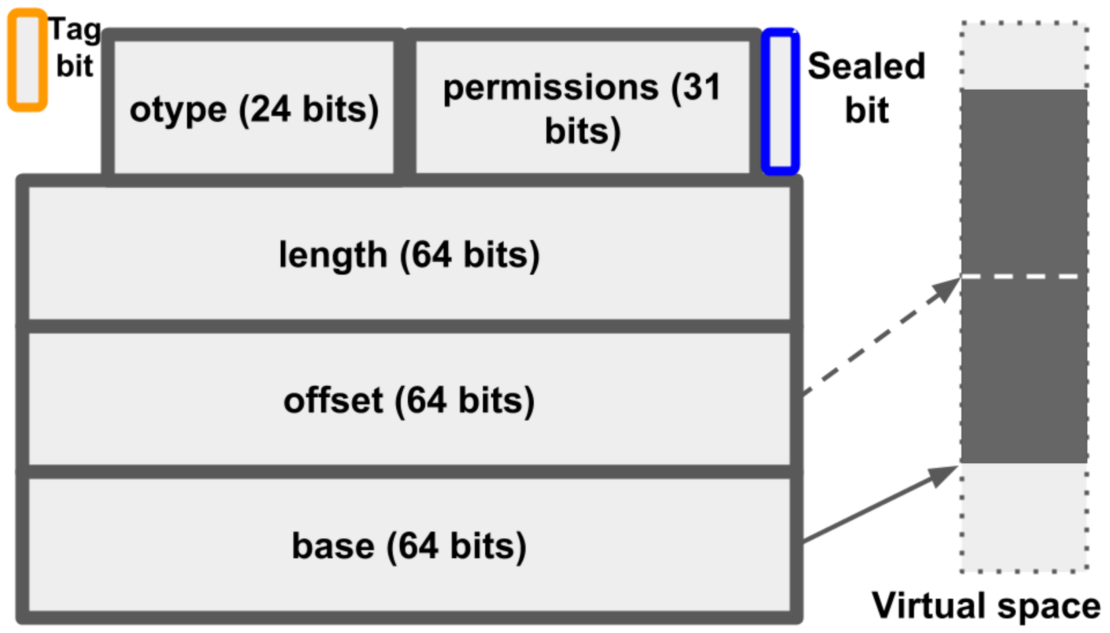


Figure 2.2 illustrates the fields of a capability in CHERI. A CHERI capability is represented as a 256-bit value [11]. The following fields constitute a capability:

- **base (64 bits)** is the start address of the virtual memory region which the capability describes.
- **length (64 bits)** is the length in bytes of the memory region which the capability describes.
- **offset or cursor (64 bits)** is a pointer (i.e., some byte address) that is used as part of computing the address on which a memory operation (one which uses the capability) will be performed.

- **permissions (31 bits: perms (15 bits) + uperms (16 bits))** is the set of hardware-defined and user-defined permissions.
- **sealed bit** indicates whether a capability is restricted from being used as a token of authority for almost all operations (except for the CCall instruction). The sealed bit is a means of having an object-capability model [36, 42] where dereferencing is prohibited except by means of transfer of domain (i.e., execution of arbitrary instructions from a callee domain is not possible). A full transition from the caller to the callee is the only legal privilege offered by a sealed capability.
- **object type (otype: 24 bits)** holds the type of a sealed capability. "*This field allows an unforgeable link to be created between associated data and object (code) capabilities.* [11]"
- 8 remaining bits (currently unused).

Thus, a capability grants a set of permissions on a virtual memory region defined by a base address and a length. The tag bit is an external bit that is attached to every aligned 256-bit value in memory or in the capability register file. It indicates validity of a capability. In other words, it indicates whether the 256 bits should be interpreted as a capability. It should be noted that an arbitrary value which was never derived from a capability will always (by design of the instruction set) have its tag bit cleared, thus preventing its usage as a token of authority on any memory operation. This is a property called capability unforgeability. In Theorem 3.1, we prove one form of capability unforgeability about our simplified formal model of the CHERI ISA, namely that no privilege escalation is possible.

In other words, a capability register or a 256-bit value in memory can be in one of the following states [11]:

- The tag bit is set, indicating that the capability is valid. Additionally, the fields of the capability are well-defined.
- The tag bit is not set, indicating that the value is not a valid capability (although all the fields may be well-defined).
- The tag bit is set, indicating that the capability is valid. However, some values of the capability are ill-defined (i.e., they indicate an inconsistency that will be detected by the corresponding operation that would attempt to use it).

It is important to note that in our formal model (Chapter 3), we do not have the notion of ill-defined values for fields in a capability. One reason is that we do not represent the offset (cursor) field.

It is worth noting that CHERI offers an alternative 128-bit compressed format for storing a capability [11]. In our formal model, we ignore such representation details (about the sizes of the fields of capabilities) because careful choices are only useful for efficiency tradeoff reasons in the real hardware model. We may avoid making simplifying assumptions for reasons of inefficiency, but only if some operation would be significantly or asymptotically less efficient (in terms of time or space complexity) under the simplifying assumption. For the case of capability sizes, we will assume that a field of a capability is represented by a natural number. And we avoid choosing representations that, e.g., restrict certain bound alignments in favor of a more compact representation.

The CHERI Instruction Set

CHERI is based on the 64-bit MIPS ISA [34]. All the ordinary MIPS instructions are available. Additionally, CHERI offers capability manipulation instructions. It also offers various capability-protected general-purpose instructions for the ordinary memory (load/store/etc..) operations. These make it easier to write more efficient machine code by offering an argument to provide the name of the capability register that authorizes the operation performed by the instruction. For backward compatibility, on the other hand, a predefined capability register is used. Every ordinary MIPS load/store instruction is only successfully executed if that predefined register contains the suitable capability.

An example of the general way instructions work is illustrated in Figure 2.3. There, we show a simplified form of a load instruction (simplified, in the sense of eliminating details about offsets and invalid representations of address values) which supports specifying the capability that authorizes the load operation as an argument.

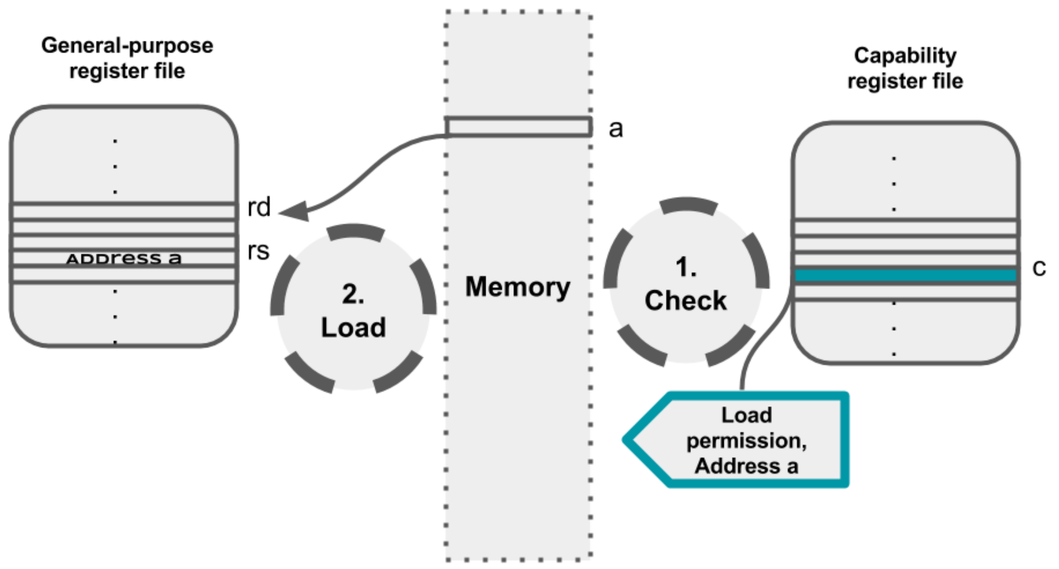
We point out again that there are two register files; a general-purpose register file, and a capability register file. The load instruction would name a capability register from the latter. The load instruction performs the following checks:

1. Check that the capability specified in the named register (register c in Figure 2.3) is a valid capability (i.e., its tag bit is set). If not, a hardware exception is raised.
2. Check that the capability in register c is not a sealed capability. If not, a hardware exception is raised.
3. Check that the load permission exists in the permissions field of the capability in register c. If not, a hardware exception is raised.

4. Check that the address specified as a source of loading a memory word into the destination register *rd* indeed lies withing the bounds of the memory region that is specified by the capability in register *c*. If not, a hardware exception is raised.

Since all of the checks are independent, and since all of them have to succeed for the instruction to execute correctly, then the order of performing these checks does not really matter if there is no interest in the type of exception that would be raised.

FIGURE 2.3: Load instruction checks the capability first then loads



As a basic precondition, the execution of every instruction has to be authorized by the **program counter capability (PCC)**. This is a special capability register that is used for indirection of the program counter (PC), and also for authorizing the execution of every instruction, i.e., the execute operation has to be specified in the permissions field of this register, and the bounds specified by this register have to contain the address that is specified by the base and cursor (offset) of PCC, and the offset (PC).

The state of the capability register file at any particular state of the machine specifies what is called a **current security domain**. The current security domain can be thought of as specified by the transitive closure of capability values that can be loaded via capabilities that exist in the capability register file [11]. It is useful to think of the different security domains of a CHERI machine state as representing the mutually distrusting components of the running software. One feature that CHERI offers for mutually distrustful components to share/delegate the right to execute APIs among each other is the feature of **sealed capabilities**. Sealed capabilities can be thought of as *immutable and non-dereferenceable [11]* capabilities. Sealed capabilities are thus the main building block for meaningful compartmentalization [25] of the memory into mutually distrustful components.

One useful instruction that emphasizes the role of the concept of a security domain in the CHERI protection model is the **CCall** instruction. This instruction is intended to be used as a trusted transfer procedure between security domains. In more familiar terms, an untrustworthy program can, by means of using the **CCall** instruction be allowed to use a library API without posing risks of it making illegal accesses or illegal jumps into the library code. This is achieved with the help of the feature of **sealed capabilities**. The library API exposes sealed capabilities (can be thought of as an object capability model [36]) to the users. The **CCall** instruction makes use of possibly a trusted stack [11] which is part of the underlying trusted computing base (TCB) in order to atomically manipulate the capability register file, and unseal the capabilities. This, in effect, provides an atomic and trusted transition (call) mechanism between mutually distrusting software components. It is important to note that since the **CCall** instruction implementation may need to vary based on the operating system/kernel implementation and also based on the assumptions about the trusted computing base, the hardware implementation is kept minimal and additionally a software trap is triggered so that additional functionality including handling the trusted stack is left to trusted software.

In Table 2.1, we give a list of the CHERI instructions along with a brief description of each. The list is exactly the one provided in the technical report written by the CHERI research team [11]. We copy it here for convenience. We point out that these are the additional instructions that CHERI provides on top of the ordinary MIPS instruction set. They are called the instructions of the CHERI capability coprocessor. However, the concept of a "coprocessor" here should not be understood as a separate ALU or that an instruction either executes on the coprocessor or the main processor. This is not true. A capability-utilizing instruction can normally operate on both the capability register file and the general-purpose register file at the same time.

TABLE 2.1: Summary of the instructions of the CHERI coprocessor

Instruction	Description
CGetBase	Move the base field of a capability to a general-purpose register
CGetOffset	Move the offset field of a capability to a general-purpose register
CGetLen	Move the length field of a capability to a general-purpose register
CGetTag	Move the tag bit of a capability register file to a general-purpose register
CGetSealed	Move the sealed bit of a capability to a general-purpose register
CGetPerm	Move the permissions field of a capability to a general-purpose register
CGetType	Move the object type field of a capability to a general-purpose register

CToPtr	Capability to pointer
CPtrCmp	Compare capability pointers
CClearRegs	Clear multiple registers
CIncBase	*Instruction removed (in favor of CSetBounds)*
CIncOffset	Increase offset
CSetBounds	Set bounds of a capability (i.e., the base and length fields)
CSetBoundsExact	Set bounds exactly. In a compressed format, if the exact bounds are not possibly representable, an exception is thrown.
CSetLen	*Instruction removed (in favor of CSetBounds)*
CClearTag	Clear the tag bit
CAndPerm	Restrict permissions
CSetOffset	Set cursor to an offset from base
CGetPCC	Move PCC to capability register
CGetPCCSetOffset	Get PCC with new offset
CFromPtr	Create capability from pointer
CSub	Subtract capabilities
CSC	Store capability register
CLC	Load capability register
CL[BHWD][U]	Load via capability register. Various load instructions are available for loading a byte, a half-word, a word, and a double-word.
CS[BHWD]	Store via capability register
CLLC	Load linked capability via capability register
CLL[BHWD][U]	Load linked via capability register
CSC[BHWD]	Store conditional via capability register
CSCC	Store conditional capability via capability
CBTU	Branch if capability tag is unset
CBTS	Branch if capability tag is set
CJR	Jump capability register. Assign to PCC the value in the given capability register, and jump to the address specified by its offset.
CJALR	Jump and link capability register. Same as CJR. Additionally, save current PCC in the given destination capability register.
CCheckPerm	Raise exception on insufficient permission
CCheckType	Raise exception if object types do not match
CSeal	Seal a capability
CUnseal	Unseal a sealed capability
CCall	Call into another security domain
CReturn	Return to the previous security domain
CGetCause	Move the capability exception cause register to a general-purpose register

CSetCause	Set the capability exception cause register

2.3 Security Goals

In this section, we list briefly security goals or applications that are the focus of this thesis.

Control Flow Integrity

The CFI [19, 20, 21] security policy enforces that the execution of software follows a stipulated control flow graph. The control flow graph for such a policy is computed by static analysis of the binary machine code or the source code [21]. The way CFI is enforced in [21, 19, 20] is by means of machine code rewriting. The rewriting procedure instruments the machine code with checks that, at run-time, ensure that the control flow adheres to the statically generated control flow graph. If it does not, then the instrumentation forces execution into jumping to some safe instruction which prevents the malicious or attacked program from further execution.

In Chapter 4, we adopt the machine code instrumentation techniques presented in [21, 20] into the MIPS instruction set, and we show that with the CHERI ISA, Theorem 4.2 about correctness of CFI enforcement by machine code instrumentation holds. The theorem is an imitation of [21, Theorem 1].

Memory Compartmentalization

Compartmentalization is a general term that refers to the practice of breaking down software into mutually distrustful components [26]. One advantageous result of assuming mutual distrust is that components get to share with each other only the necessary resources, and nothing more – in adherence to the principle of least privilege [29].

Memory compartmentalization is a useful way of modeling desirable high-level programming features such as encapsulation in object-oriented programming languages at a lower level (i.e., in machine code).

A note on encapsulation and protection Although data encapsulation is not a real memory protection guarantee in almost all object-oriented programming languages because of the possibility of its circumvention by means of reflection APIs (like in Java, C#, Ruby) or by means of name mangling (like in Python), we still use the notion of encapsulation as an analogy to protection, and we mean to realize the guarantees that encapsulation would give if only a fragment of an object-oriented Java-like language is considered. Only in such a fragment, where encapsulation would indeed guarantee protection would it make sense to talk about preserving the memory protection (compartmentalization) that is offered by the source language. Because otherwise, as in most real-world programming languages, encapsulation is considered as a design pattern, and not a concrete security guarantee. Choosing to use the term "encapsulation" to refer to memory protection spares us the effort of defining a possibly unfamiliar language with e.g., a type system for secure information flow [37]. We find such an approach unnecessary if the only purpose is defining compartmentalization. Credit for understanding that encapsulation is a feature intended to achieve good programming practice rather than a protection mechanism goes to the Wikipedia page: `https://en.wikipedia.org/wiki/Encapsulation_(computer_programming)`. It is worth mentioning though that it is indeed possible for the programmer of Java, for example, to use a feature called a security manager, by means of which they can deny permissions that allow the usage of reflection in some ways. The Stackoverflow page: `http://stackoverflow.com/questions/770635/disable-java-reflection-for-the-current-thread` and the Java documentation page: `docs.oracle.com/javase/6/docs/api/java/lang/reflect/ReflectPermission.html` both give a hint on this possibility. So, after all, to simply assume that encapsulation guarantees protection is not a naively-optimistic or unrealistic claim to make (compared to what at least one existing programming language can offer).

In Chapter 5, we follow the definition of compartmentalization which is presented in the context of evaluating the expressiveness of the micropolicies framework [9] using the PUMP machine. We use that definition as a guideline for designing a way in which permissions can be structured in memory and in the register files in order to express the notion of compartments, and we show that if such a structure of memory is guaranteed at the beginning of execution of a program, then compartmentalization behavior is guaranteed.

Secure Compilation

One notion of a secure compilation procedure is given by fully-abstract compilation [38, 26, 14, 12, 13]. A fully-abstract compiler is secure in the sense that it provides a guarantee on the possible interactions that the produced machine code could have with any arbitrary low-level context. The guarantee is that whatever behavior can be exhibited by the compiled code by means of interaction with any arbitrary low-level libraries will be producible by means of a legal interaction with some high-level context or library.

In a slightly more formal sense, a compiler is fully-abstract if it preserves and reflects observational equivalence. Reflecting observational equivalence is usually implied by correctness of the compiler. Preserving observational equivalence is what is referred to as "preserving the abstractions" (i.e., the abstractions that were introduced by means of programming in a higher-level source language).

This guarantee is useful because it means that if the programmer were able to prove that some undesired behavior is impossible to be exhibited by the guarantees of the semantics of the source language (regardless of which interactions take place with source language contexts), then if the compiler is fully-abstract, the programmer is able to also conclude that no low-level attacker will be able to trigger that undesired behavior either.

Secure compilation is the motivation for us to pursue a formal treatment of the CHERI hardware architecture. The end goal of this pursuit is to have available means of using the CHERI architecture in a way that guarantees preservation of source language properties into the produced CHERI machine code. One building block towards that goal is what is presented in Chapter 5.

Chapter 3

A Formal Model for CHERI

In this chapter, we describe a formal model for an instruction-level language that is based on CHERI [4]. The formal model is largely based on the technical report of the CHERI Instruction-Set Architecture [11], and was guided by some of the explanations in Norton's and Woodruff's PhD theses [27, 39].

3.1 Simplifying Assumptions

We intend to describe only a subset of the CHERI instruction set, and to generalize and/or simplify some details whenever we think they will not reduce security guarantees or functionality. In particular, we do not include some instructions that are only intended to achieve run-time optimizations.

Word-addressable memory One example of these ignored instructions is the case of memory load and store. We do with only one version of load and store, and disregard the need for code optimizations that may depend on loading a single byte, a half-word, etc.. But one important effect of that, on a memory-protected architecture like CHERI, is that the chosen level of granularity has to be the same (or finer) than the level of granularity of memory protection. Otherwise, a security domain that is allowed access to just one "thin" unit of memory might not be able to use the available instructions to access it alone without accessing memory that is also not part of its domain.

Absolute addressing only The offset field in an instruction that accesses memory is also neglected. The offset field can be compensated by having a sufficient size of a register word, together with the availability of instructions that can be used to perform the same arithmetic operations that are used to compute the target address from the offset. It is worth mentioning that the adoption of a base register and an offset field in memory access instructions in MIPS [34] is for practical performance and compiler optimization reasons [40].

Also, the offset field of capabilities is ignored in our formal model. We choose to not use capabilities as a base for computing addresses. So, for example, pc, the program counter is an absolute value and not a relative value to the base address of pcc, the capability on the program counter. This means that capabilities fields are never used to determine the calculation of any address in our semantics, except when a capability is explicitly used in a ccall instruction to define the memory region that determines the security domain transition.

Some instructions can be thought of as the responsibility of a compiler, like cjalr (Jump and Link Capability Register), and creturn. Those we choose to drop. They allow atomic execution of a series of loads/stores, which can be guaranteed by the compiler, and not necessarily by the instruction set architecture.

Uni-processor We also choose to keep our instruction-level language devoid of support for some synchronization primitives that are originally offered by the CHERI architecture, which essentially means we model a uni-processor system.

Unbounded memory We assume unbounded memory. This allows us to drop specifications and checks related to the sizes/values of addresses and registers.

Implicit MMU The memory hierarchy, and support of virtual address space to physical address space translation is beyond the scope of our formal model. However, to clarify that the formal model is not over-simplifying memory operations, it is worth mentioning that CHERI adopts a hybrid capability-system model [11, Section 2.2] where both a virtual-memory model and the capability-based protection on top of it apply to a memory access operation. We do not formalize the memory-management unit, but that does not mean that our formal model assumes a single global address space. Rather, since CHERI's memory operations are directed first to the capability co-processor, and then to the memory management unit (MMU) for translation, all we are assuming is an implicitly correct MMU.

Arbitrary binary operations In order to avoid the redundancy that is otherwise inevitable when describing detailed operations of an instruction set, we refer to any binary operations on registers with the arbitrary instruction BinOp in order to skip the details of which particular operations to support. As a result of that, we abandon the usage of flags that might be used in a MIPS processor to signal some side effect of a binary operation.

Memory allocation service In order to be able to reason about our formal model as a model for program execution without having to state much detail about the underlying trusted software layers, we incorporate in our semantics support for a memory allocation instruction allocate. This instruction does not strengthen our formal model compared to CHERI. All it does is, it provides a compact way of representing the implicit assumption about the existence of a trusted service of the operating system kernel that securely gives to an arbitrary caller control over part of the "free" memory that only it (the kernel) used to hold before calling the service. We again point out that an alternative representation of this trusted service would be to make use of a normal ccall operation that gives control to a trusted domain of the underlying software (the operating system kernel) in order that this domain gets access to some predefined and reserved registers and memory regions, and to perform the delegation of capabilities to the caller fulfilling the memory allocation request.

Support for "ephemeral" capabilities is available in our formal model semantics. This feature was intended to support the brief delegation of arguments from callers to callees across object-capability invocation [11]. We point out that the usage of the term "ephemeral" has been discontinued by the authors of the technical report on the CHERI instruction set [11]. The term "local" capability is now being used instead of "ephemeral" and the term "global" for "non-ephemeral".

We also point out the similarity between the concept of ephemeral capabilities and the concept of "linear types" or "linear logic" [41], and we point out that the idea of limiting copying of capabilities is at least as old as Saltzer [29, Revocation and Control of Propagation] where a "copy" bit is mentioned as a conventional solution to prevent a capability from getting stored in a memory segment.

An ephemeral capability can be seen as a revocable capability. Being ephemeral disallows storing of the capability in memory, which means that such capability can only exist in the capability register file. This allows for the possibility of overwriting (deleting) it by the caller once the call returns. If such a possibility is not available, then arbitrary reuse of a capability on some memory region seems unpreventable. So, to support temporal memory safety, one can make use of ephemeral capabilities.

An extension to this model of linearity/revocability of capabilities was pointed out by Saltzer in 1975 [29] where a depth counter is suggested to keep track of the number of successive copy operations that have been performed on a capability. The counter starts with, say, one. And the capability would then have a custom permission set that prohibits this counter to be copied, say, more than three times. And the instruction set design would guarantee then that every valid load of a capability increments the depth counter associated with the capability, and guarantees that a valid permission exists that allows each increment of the counter up to a limit indicated by the permission.

One other possible extension is to assign a security level (possible encoded as a custom permission of the store capability) on each memory region. And to allow copying a certain capability to regions on which a store capability exists that have a security level equal to or higher than the level that would be indicated by a "lowest-security" field that the capability would specify instead of just the ephemeral bit field.

3.2 Operational Semantics for reduced CHERI

We first describe the syntactic categories used in our model.

- $\mathit{Word} = \mathit{Tag} \times \mathit{Content}$
  This describes a memory word, which is a tag and some content.
- $\mathit{Tag} ::= \mathsf{is\_cap} \mid \mathsf{is\_data}$
  The tag describes whether the content is a capability or is data. We choose to use the representation 1 for is_cap and 0 for is_data.
- $\mathit{Content} = \mathbb{N}$
  The content can be interpreted as a capability or as data. In particular, we use natural numbers, and use an encoding of a capability into natural numbers. It is important to note that this allows any value of *Content* to be freely interpreted as data.
- A capability:

$$\mathit{Cap} = \{\mathsf{n\_encode}(s, \mathsf{bin}(\mathit{perms}), \mathit{addr}, \mathit{len}, \mathit{otype}) \mid$$

$$s \in \mathit{Sealed}, \mathit{perms} \in 2^{\mathit{Perm}}, \mathit{addr} \in \mathit{Addr}, \mathit{len} \in \mathit{Length}, \mathit{otype} \in \mathit{Otype}\}$$

  where $\mathit{Sealed} = \{1, 0\}$ interpreted as $\{\mathtt{true}, \mathtt{false}\}$, $\mathit{Addr} = \mathit{Length} = \mathit{Otype} = \mathbb{N}$, and $\mathsf{n\_encode} : \mathbb{N}^n \to \mathbb{N}$ is defined as $\mathsf{n\_encode}(a_1, a_2, \ldots, a_n) \overset{def}{=} p_1^{a_1} \times p_2^{a_2} \times \cdots \times p_n^{a_n} \times p_{n+1}^{(n+1)}$ where p_i is the i^{th} prime number.

The capability is a special kind of content, but it is not a distinct syntactic category from *Content*. They are only made distinct (as an interpretation) by the value of the tag. As a shorter form of writing n_encode, we simply use parentheses "(", ")" to denote the encoding of fields of a capability.

The capability is interpreted as a collection of fields; a bit indicating whether it is sealed, a set of permissions, a base address, a length (a number of words), and a type field.

- $\mathit{RegFile} = \mathit{RegName} \rightarrow \mathit{Word}$

  The register file is a mapping from register names to memory words. And $\mathit{RegName} = \{\texttt{r0}, \texttt{r1}, \ldots\}$.

- $\mathit{CapRegFile} = \mathit{CapRegName} \rightarrow \mathit{Word}$

  .. and the same for the capability register file, but the distinction between the two register files is crucial. And $\mathit{CapRegName} = \{\texttt{c0}, \texttt{c1}, \ldots\}$. We assume the register files are of unbounded size.

- $\mathit{Addr} = \mathbb{N}$

  Memory addresses are natural numbers.

- $\mathit{Mem} = \mathit{Addr} \rightarrow \mathit{Word}$

  The memory is a mapping from addresses to words.

- Perm ::= permit_load | permit_store | permit_execute | permit_seal
  | permit_load_capability | permit_store_capability | non_ephemeral
  | permit_store_ephemeral_capability | permit_extra_1 | ... | permit_extra_n

  We leave the number n of extra permissions available to be defined if necessary. We use $\mathsf{bin}(\mathit{perms})$ to refer to the binary encoding of a boolean assignment that assigns true to permissions available in a set perms, and false to each of the possible permissions that are not included in perms.

  We note the conventional correspondence between set operations on perms and bit operations on $\mathsf{bin}(\mathit{perms})$ (e.g., $\mathsf{bin}(\mathit{perms1} \cap \mathit{perms2}) \equiv \mathsf{bin}(\mathit{perms1})\&\mathsf{bin}(\mathit{perms2})$, and $p \in \mathit{perms1} \equiv$ "bit i_p is set in $\mathsf{bin}(\mathit{perms})$", where i_p is the bit at some fixed index for permission p, and the "&" symbol denotes bitwise-and).

- $\mathit{MachineState} = \mathit{Mem} \times \mathit{RegFile} \times \mathit{CapRegFile} \times \{\mathsf{pc}\} \times \{\mathsf{pcc}\} \times \{\mathsf{next\_free}\}$

  with $\mathsf{pc} \in \mathit{Addr}$, $\mathsf{pcc} \in \mathit{Word}$, $\mathsf{next\_free} \in \mathit{Addr}$.

  It is worth noting that pc, pcc are kept separate from the domains of *RegFile*, *CapRegFile* respectively.

  One scenario that is prohibited by this separation is a sequential block of code potentially causing the program to get stuck.

If pcc were part of the register file, then using a normal load instruction with a malicious value for the argument, pcc can be made to contain an invalid capability, for example, which causes an exception on the next instruction.

So this separation gives the possibility for programs to manipulate capability registers without posing the risk that execution gets stuck because of accidentally writing pcc.

We denote the capability register names *CapRegName* with literal names starting with the letter $\texttt{c}$, e.g., $\texttt{cb}, \texttt{cd}, \texttt{ct}, \ldots$, and general-purpose register names *RegName* with literal names starting with the letter $\texttt{r}$, e.g., $\texttt{rb}, \texttt{rd}, \texttt{rt}, \ldots$. Other literal names in the context of interpreting a memory word (which is not an instruction) should be understood as denoting the binary encoding of a natural number unless otherwise stated.

We define the following functions that are used in the rules below:

- $\mathsf{content}(\langle \texttt{t}, \texttt{c}\rangle) \overset{def}{=} \texttt{c}$
- $\mathsf{compute\_call\_address}((\texttt{s\_c}, \mathsf{bin}(\texttt{perms\_c}), \texttt{addr\_c}, \texttt{len\_c}, \texttt{otype\_c})) \overset{def}{=} \texttt{addr\_c}$
- $\mathsf{sealed}((\texttt{s}, \mathsf{bin}(\texttt{perms}), \texttt{addr}, \texttt{len}, \texttt{otype})) \overset{def}{=} (\texttt{true}, \mathsf{bin}(\texttt{perms}), \texttt{addr}, \texttt{len}, \texttt{otype})$
- $\mathsf{unsealed}((\texttt{s}, \mathsf{bin}(\texttt{perms}), \texttt{addr}, \texttt{len}, \texttt{otype})) \overset{def}{=}$
  $(\texttt{false}, \mathsf{bin}(\texttt{perms}), \texttt{addr}, \texttt{len}, \texttt{otype})$
- $\mathsf{clear\_otype}((\texttt{s}, \mathsf{bin}(\texttt{perms}), \texttt{addr}, \texttt{len}, \texttt{otype})) \overset{def}{=}$
  $(\texttt{s}, \mathsf{bin}(\texttt{perms}), \texttt{addr}, \texttt{len}, 0)$
- $\mathsf{remove\_non\_ephemeral}((\texttt{s}, \mathsf{bin}(\texttt{perms}), \texttt{addr}, \texttt{len}, \texttt{otype}),$
  $(\texttt{s\_t}, \mathsf{bin}(\texttt{perms\_t}), \texttt{addr\_t}, \texttt{len\_t}, \texttt{otype\_t})) \overset{def}{=}$
  $(\texttt{s}, \mathsf{bin}(\texttt{perms}), \texttt{addr}, \texttt{len}, \texttt{otype})$
  if $\mathsf{non\_ephemeral} \in \texttt{perms\_t}$
  $(\texttt{s}, \mathsf{bin}(\texttt{perms} \setminus \{\mathsf{non\_ephemeral}\}), \texttt{addr}, \texttt{len}, \texttt{otype})$
  otherwise

3.2.1 Helping Rules

Here we describe the rules used in the preconditions of the instructions' operational semantics.

The rule callable states preconditions on the instruction ccall. It defines when two capability registers *cc* and *cd* are valid arguments of the ccall instruction. Register *cc* in the register file *cr* should contain a valid and sealed "code capability". Register *cd* should also contain a valid and sealed non-code capability; it is called the "data capability". The "object type" field of both capabilities should match. This latter check allows the possibility (with a clever compiler) to ensure that both code and data capabilities were sealed by a single security domain before a call is allowed to execute.

$$\frac{\begin{array}{c} cr(cc) = \langle \texttt{1}, (\texttt{s\_c}, \mathsf{bin}(\texttt{perms\_c}), \texttt{addr\_c}, \texttt{len\_c}, \texttt{otype\_c})\rangle \\ cr(cd) = \langle \texttt{1}, (\texttt{s\_d}, \mathsf{bin}(\texttt{perms\_d}), \texttt{addr\_d}, \texttt{len\_d}, \texttt{otype\_d})\rangle \\ \texttt{otype\_c} = \texttt{otype\_d} \qquad \texttt{s\_c} = \texttt{true} \qquad \texttt{s\_d} = \texttt{true} \\ \{\mathsf{permit\_execute}\} \subseteq \texttt{perms\_c} \qquad \{\mathsf{permit\_execute}\} \not\subseteq \texttt{perms\_d} \end{array}}{cr \vdash \mathsf{callable}(cc, cd)}\,(\mathsf{callable})$$

The rule permits_load checks that capability *ci* allows a load from address *a*.

$$\frac{\begin{array}{c} cr(ci) = \langle \texttt{1}, (\texttt{s}, \mathsf{bin}(\texttt{perms}), \texttt{addr}, \texttt{len}, \texttt{otype})\rangle \\ \mathsf{permit\_load} \in \texttt{perms} \qquad \texttt{s} = \texttt{false} \qquad \texttt{addr} \leq a < \texttt{addr} + \texttt{len} \end{array}}{cr \vdash \mathsf{permits\_load}(ci, a)}\,(\mathsf{permits\_load})$$

The rule permits_loadcap checks that capability *ci* allows loading a capability from address *a*.

$$\frac{\begin{array}{c} cr(ci) = \langle \texttt{1}, (\texttt{s}, \mathsf{bin}(\texttt{perms}), \texttt{addr}, \texttt{len}, \texttt{otype})\rangle \\ \mathsf{permit\_load\_capability} \in \texttt{perms} \\ \texttt{s} = \texttt{false} \qquad \texttt{addr} \leq a < \texttt{addr} + \texttt{len} \end{array}}{cr \vdash \mathsf{permits\_loadcap}(ci, a)}\,(\mathsf{permits\_loadcap})$$

The rule permits_store checks that capability *ci* allows storing in address *a*.

$$\frac{\begin{array}{c} cr(ci) = \langle \texttt{1}, (\texttt{s}, \mathsf{bin}(\texttt{perms}), \texttt{addr}, \texttt{len}, \texttt{otype})\rangle \\ \mathsf{permit\_store} \in \texttt{perms} \qquad \texttt{s} = \texttt{false} \qquad \texttt{addr} \leq a < \texttt{addr} + \texttt{len} \end{array}}{cr \vdash \mathsf{permits\_store}(ci, a)}\,(\mathsf{permits\_store})$$

The rule permits_storecap checks that capability *ci* allows storing the capability *cs* in address *a*. Ephemeral capabilities are prevented from being stored except in regions where an extra permission (namely, permit_store_ephemeral_capability) is granted.

$$\frac{\begin{array}{c} cr(cb) = \langle \mathtt{1}, (\mathtt{s\_b}, \mathsf{bin}(\mathtt{perms\_b}), \mathtt{addr\_b}, \mathtt{len\_b}, \mathtt{otype\_b})\rangle \\ cr(cs) = \langle \mathtt{tag}, (\mathtt{s\_s}, \mathsf{bin}(\mathtt{perms\_s}), \mathtt{addr\_s}, \mathtt{len\_s}, \mathtt{otype\_s})\rangle \\ \mathsf{permit\_store\_capability} \in \mathtt{perms\_b} \\ (\mathsf{non\_ephemeral} \in \mathtt{perms\_s} \vee \mathtt{tag} = 0 \\ \vee \mathsf{permit\_store\_ephemeral\_capability} \in \mathtt{perms\_b}) \\ \mathtt{s\_b} = \mathtt{false} \qquad \mathtt{addr\_b} \leq a < \mathtt{addr\_b} + \mathtt{len\_b} \end{array}}{cr \vdash \mathsf{permits\_storecap}(cb, cs, a)} (\mathsf{permits\_storecap})$$

The rule permits_execute checks that capability ci will be a valid program counter capability (pcc) if address a becomes the value of the program counter (pc).

$$\frac{\begin{array}{c} cr(ci) = \langle \mathtt{1}, (\mathtt{s}, \mathsf{bin}(\mathtt{perms}), \mathtt{addr}, \mathtt{len}, \mathtt{otype})\rangle \\ \{\mathsf{permit\_execute}, \mathsf{non\_ephemeral}\} \subseteq \mathtt{perms} \\ \mathtt{s} = \mathtt{false} \qquad \mathtt{addr} \leq a < \mathtt{addr} + \mathtt{len} \end{array}}{cr \vdash \mathsf{permits\_execute}(ci, a)} (\mathsf{permits\_execute})$$

The rule executable specifies the conditions that suffice for a program counter capability to indicate that a certain program counter value is executable.

$$\frac{\begin{array}{c} pcc = \langle \mathtt{1}, (\mathtt{s}, \mathsf{bin}(\mathtt{perms}), \mathtt{addr}, \mathtt{len}, \mathtt{otype})\rangle \\ \mathsf{permit\_execute} \in \mathtt{perms} \qquad \mathtt{s} = \mathtt{false} \qquad \mathtt{addr} \leq pc < \mathtt{addr} + \mathtt{len} \end{array}}{pcc \vdash \mathsf{executable}(pc)} (\mathsf{executable})$$

The rule permits_unseal checks that capability register ct can authorize the unsealing operation on capability register cs in capability register file cr. cs must contain a valid sealed capability whose type field corresponds to the base address of the authorizing capability ct. The authorizing capability ct must grant the permit_seal permission which is required for both sealing and unsealing.

$$\frac{\begin{array}{c} cr(cs) = \langle \mathtt{1}, (\mathtt{s\_s}, \mathsf{bin}(\mathtt{perms\_s}), \mathtt{addr\_s}, \mathtt{len\_s}, \mathtt{otype\_s})\rangle \\ cr(ct) = \langle \mathtt{1}, (\mathtt{s\_t}, \mathsf{bin}(\mathtt{perms\_t}), \mathtt{addr\_t}, \mathtt{len\_t}, \mathtt{otype\_t})\rangle \\ \mathsf{permit\_seal} \in \mathtt{perms\_t} \\ \mathtt{s\_s} = \mathtt{true} \qquad \mathtt{s\_t} = \mathtt{false} \qquad \mathtt{otype\_s} = \mathtt{base\_t} \end{array}}{cr \vdash \mathsf{permits\_unseal}(cs, ct)} (\mathsf{permits\_unseal})$$

The rule permits_seal checks that capability register ct can authorize the sealing operation on capability register cs in capability register file cr. cs must contain a valid unsealed capability.

$$\dfrac{\begin{array}{c} cr(cs) = \langle 1, (\texttt{s\_s}, \mathsf{bin}(\texttt{perms\_s}), \texttt{addr\_s}, \texttt{len\_s}, \texttt{otype\_s})\rangle \\ cr(ct) = \langle 1, (\texttt{s\_t}, \mathsf{bin}(\texttt{perms\_t}), \texttt{addr\_t}, \texttt{len\_t}, \texttt{otype\_t})\rangle \\ \mathsf{permit\_seal} \in \texttt{perms\_t} \qquad \texttt{s\_s} = \texttt{false} \qquad \texttt{s\_t} = \texttt{false} \end{array}}{cr \vdash \mathsf{permits\_seal}(cs, ct)}\,(\mathsf{permits\_seal})$$

3.2.2 Instructions Semantics

We describe one rule for the transition relation on CHERI states that factors out the necessary check on pcc from the operational semantics of instructions. We denote a state transition by $\rightarrow \subseteq \mathit{MachineState} \times \mathit{MachineState}$. The rule below specifies that for any instruction to execute correctly, the program counter capability has to be a valid capability on a memory region in which this instruction lives, and the capability has to give the execute permission on this memory region.

$$\dfrac{\begin{array}{c} \langle m, r, cr, \mathsf{pc},\ \mathsf{pcc},\ \mathsf{next\_free}\rangle \xrightarrow{i} \langle m', r', cr', \mathsf{pc}',\ \mathsf{pcc}',\ \mathsf{next\_free}'\rangle \\ \mathsf{pcc}' \vdash \mathsf{executable}(\mathsf{pc}') \end{array}}{\langle m, r, cr, \mathsf{pc},\ \mathsf{pcc},\ \mathsf{next\_free}\rangle \rightarrow \langle m', r', cr', \mathsf{pc}',\ \mathsf{pcc}',\ \mathsf{next\_free}'\rangle}\,(\text{legal-transition})$$

Next we give the format, a text description, and the operational semantics of all the instructions we include in our model of the CHERI machine.

The semantics are specified by means of rule(s) for each instruction that define the machine transition relation $\xrightarrow{i} \subseteq \mathit{MachineState} \times \mathit{MachineState}$.

binop `rd` `rs1` `rs2`: An arbitrary binary operation. Source operands are in registers `rs1` and `rs2`. The result is stored in destination register `rd`.

$$\dfrac{\begin{array}{c} m(\mathsf{pc}) = \langle 0, \mathsf{binop}\ \texttt{rd}\ \texttt{rs1}\ \texttt{rs2}\rangle \\ r' = r[\texttt{rd} \mapsto r(\texttt{rs1})\ [\mathsf{BinOp}]\ r(\texttt{rs2})] \qquad \mathsf{pc}' = \mathsf{pc} + 1 \end{array}}{\langle m, r, cr, \mathsf{pc},\ \mathsf{pcc},\ \mathsf{next\_free}\rangle \xrightarrow{i} \langle m, r', cr, \mathsf{pc}',\ \mathsf{pcc},\ \mathsf{next\_free}\rangle}\,(\mathsf{BinOp})$$

cload `rd` `rt` `cb`: Load the value at the memory location given in `rt` into register `rd` if the capability in capability register `cb` gives the load permission on this location. Indirect non-relative addressing is used. ("**Indirect**" means the address is given in a register, not as an immediate value. "**Non-relative**" means that the value in the register is the effective address, not an offset.)

$$\dfrac{\begin{array}{c} m(\mathsf{pc}) = \langle 0, \mathsf{cload}\ \mathtt{rd}\ \mathtt{rt}\ \mathtt{cb}\rangle \\ cr(\mathtt{cb}) = \langle 1, (\mathtt{s\_b}, \mathsf{bin}(\mathtt{perms\_b}), \mathtt{addr\_b}, \mathtt{len\_b}, \mathtt{otype\_b})\rangle \\ cr \vdash \mathsf{permits\_load}(\mathtt{cb}, \mathsf{content}(r(\mathtt{rt}))) \\ r' = r[\mathtt{rd} \mapsto m(\mathsf{content}(r(\mathtt{rt})))] \qquad \mathsf{pc}' = \mathsf{pc} + 1 \end{array}}{\langle m, r, cr, \mathsf{pc},\ \mathsf{pcc},\ \mathsf{next\_free}\rangle \xrightarrow{i} \langle m, r', cr, \mathsf{pc}',\ \mathsf{pcc},\ \mathsf{next\_free}\rangle}\ (\mathsf{cload})$$

cstore `rs` `rt` `cb`: Store the value in register `rs` into the memory location given in register `rt` if the capability in capability register `cb` gives the store permission on this location. Indirect non-relative addressing is used.

$$\dfrac{\begin{array}{c} m(\mathsf{pc}) = \langle 0, \mathsf{cstore}\ \mathtt{rs}\ \mathtt{rt}\ \mathtt{cb}\rangle \\ cr(\mathtt{cb}) = \langle 1, (\mathtt{s\_b}, \mathsf{bin}(\mathtt{perms\_b}), \mathtt{addr\_b}, \mathtt{len\_b}, \mathtt{otype\_b})\rangle \\ cr \vdash \mathsf{permits\_store}(\mathtt{cb}, \mathsf{content}(r(\mathtt{rt}))) \qquad r(\mathtt{rs}) = \langle \mathtt{tag\_s}, \mathtt{content\_s}\rangle \\ m' = m[\mathsf{content}(r(\mathtt{rt})) \mapsto \langle 0, \mathtt{content\_s}\rangle] \qquad \mathsf{pc}' = \mathsf{pc} + 1 \end{array}}{\langle m, r, cr, \mathsf{pc},\ \mathsf{pcc},\ \mathsf{next\_free}\rangle \xrightarrow{i} \langle m', r, cr, \mathsf{pc}',\ \mathsf{pcc},\ \mathsf{next\_free}\rangle}\ (\mathsf{cstore})$$

cloadcap `cd` `rt` `cb`: Load the capability value at the memory location given in `rt` into capability register `cd` if the capability in capability register `cb` gives the load capability permission on this location. Indirect non-relative addressing is used.

$$\dfrac{\begin{array}{c} m(\mathsf{pc}) = \langle 0, \mathsf{cloadcap}\ \mathtt{cd}\ \mathtt{rt}\ \mathtt{cb}\rangle \\ cr(\mathtt{cb}) = \langle 1, (\mathtt{s\_b}, \mathsf{bin}(\mathtt{perms\_b}), \mathtt{addr\_b}, \mathtt{len\_b}, \mathtt{otype\_b})\rangle \\ cr \vdash \mathsf{permits\_loadcap}(\mathtt{cb}, \mathsf{content}(r(\mathtt{rt}))) \\ cr' = cr[\mathtt{cd} \mapsto m(\mathsf{content}(r(\mathtt{rt})))] \qquad \mathsf{pc}' = \mathsf{pc} + 1 \end{array}}{\langle m, r, cr, \mathsf{pc},\ \mathsf{pcc},\ \mathsf{next\_free}\rangle \xrightarrow{i} \langle m, r, cr', \mathsf{pc}',\ \mathsf{pcc},\ \mathsf{next\_free}\rangle}\ (\mathsf{cloadcap})$$

cstorecap `cs` `rt` `cb`: Store the capability value in register `cs` at the memory location given in register `rt` if the capability in capability register `cb` grants the store capability permission. And if the capability to be stored is valid and ephemeral, then `cb` has to grant the store-ephemeral-capability permission.

$$\dfrac{\begin{array}{c} m(\mathsf{pc}) = \langle 0, \mathsf{cstorecap}\ \mathtt{cs}\ \mathtt{rt}\ \mathtt{cb}\rangle \\ cr(\mathtt{cb}) = \langle 1, (\mathtt{s\_b}, \mathsf{bin}(\mathtt{perms\_b}), \mathtt{addr\_b}, \mathtt{len\_b}, \mathtt{otype\_b})\rangle \\ cr \vdash \mathsf{permits\_storecap}(\mathtt{cb}, \mathtt{cs}, \mathsf{content}(r(\mathtt{rt}))) \\ m' = m[\mathsf{content}(r(\mathtt{rt})) \mapsto cr(\mathtt{cs})] \qquad \mathsf{pc}' = \mathsf{pc} + 1 \end{array}}{\langle m, r, cr, \mathsf{pc},\ \mathsf{pcc},\ \mathsf{next\_free}\rangle \xrightarrow{i} \langle m', r, cr, \mathsf{pc}',\ \mathsf{pcc},\ \mathsf{next\_free}\rangle}\ (\mathsf{cstorecap})$$

cjr rt cb: Jump to the address specified in register rt if the capability in register cb is a valid capability that allows execution of the jump destination. Indirect non-relative addressing is used.

$$\frac{m(\mathsf{pc}) = \langle 0, \mathsf{cjr}\ \mathtt{rt}\ \mathtt{cb}\rangle \qquad cr \vdash \mathsf{permits\_execute}(\mathtt{cb}, \mathsf{content}(r(\mathtt{rt}))) \\ \mathsf{pc}' = \mathsf{content}(r(\mathtt{rt})) \qquad \mathsf{pcc}' = cr(\mathtt{cb})}{\langle m, r, cr, \mathsf{pc},\ \mathsf{pcc},\ \mathsf{next\_free}\rangle \xrightarrow{i} \langle m, r, cr, \mathsf{pc}',\ \mathsf{pcc}',\ \mathsf{next\_free}\rangle}\text{(cjr)}$$

cjrzero cb rz OFFS: Jump if and only if register rz contains the value 0 and the capability in register cb is a valid execute capability on the jump destination. Direct (immediate) relative addressing is used.

$$\frac{m(\mathsf{pc}) = \langle 0, \mathsf{cjrzero}\ \mathtt{cb}\ \mathtt{rz}\ \mathtt{OFFS}\rangle \\ \mathsf{content}(r(\mathtt{rz})) = 0 \qquad cr \vdash \mathsf{permits\_execute}(\mathtt{cb}, \mathsf{pc} + \mathtt{OFFS} + 1) \\ \mathsf{pc}' = \mathsf{pc} + \mathtt{OFFS} + 1 \qquad \mathsf{pcc}' = cr(\mathtt{cb})}{\langle m, r, cr, \mathsf{pc},\ \mathsf{pcc},\ \mathsf{next\_free}\rangle \xrightarrow{i} \langle m, r, cr, \mathsf{pc}',\ \mathsf{pcc}',\ \mathsf{next\_free}\rangle}\text{(cjrcond-true)}$$

$$\frac{m(\mathsf{pc}) = \langle 0, \mathsf{cjrzero}\ \mathtt{cb}\ \mathtt{rz}\ \mathtt{OFFS}\rangle \qquad \mathsf{content}(r(\mathtt{rz})) \neq 0 \qquad \mathsf{pc}' = \mathsf{pc} + 1}{\langle m, r, cr, \mathsf{pc},\ \mathsf{pcc},\ \mathsf{next\_free}\rangle \xrightarrow{i} \langle m, r, cr, \mathsf{pc}',\ \mathsf{pcc},\ \mathsf{next\_free}\rangle}\text{(cjrcond-false)}$$

cbts cb OFFS: Jump to the address pc + OFFS + 1 if and only if the tag of the word in register cb is set (i.e., if it is a valid capability).

$$\frac{m(\mathsf{pc}) = \langle 0, \mathsf{cbts}\ \mathtt{cb}\ \mathtt{OFFS}\rangle \\ cr(\mathtt{cb}) = \langle 1, \mathtt{content}\rangle \qquad \mathsf{pc}' = \mathsf{pc}\ + \mathtt{OFFS} + 1}{\langle m, r, cr, \mathsf{pc},\ \mathsf{pcc},\ \mathsf{next\_free}\rangle \xrightarrow{i} \langle m, r, cr, \mathsf{pc}',\ \mathsf{pcc},\ \mathsf{next\_free}\rangle}\text{(cbts-true)}$$

$$\frac{m(\mathsf{pc}) = \langle 0, \mathsf{cbts}\ \mathtt{cb}\ \mathtt{OFFS}\rangle \qquad cr(\mathtt{cb}) = \langle 0, \mathtt{content}\rangle \qquad \mathsf{pc}' = \mathsf{pc} + 1}{\langle m, r, cr, \mathsf{pc},\ \mathsf{pcc},\ \mathsf{next\_free}\rangle \xrightarrow{i} \langle m, r, cr, \mathsf{pc}',\ \mathsf{pcc},\ \mathsf{next\_free}\rangle}\text{(cbts-false)}$$

ccall cc cd cdd: Takes a sealed code capability cc, and a sealed data capability cd that should have the same *otype*, and unseals both of them. Then the code capability is moved into pcc, and the data capability is moved into cdd. ccall is used to secure transition between security domains. Control is transferred to the base address of the code capability.

$$\frac{\begin{array}{c} m(\mathsf{pc}) = \langle 0, \mathsf{ccall}\ \texttt{cc}\ \texttt{cd}\ \texttt{cdd}\rangle \\ cr \vdash \mathsf{callable}(\texttt{cc}, \texttt{cd}) \qquad cc' = \mathsf{unsealed}(cr(\texttt{cc})) \qquad cd' = \mathsf{unsealed}(cr(\texttt{cd})) \\ \mathsf{pcc}' = cc' \qquad cr' = cr[cdd \mapsto cd'] \qquad \mathsf{pc}' = \mathsf{compute\_call\_address}(cc') \end{array}}{\langle m, r, cr, \mathsf{pc},\ \mathsf{pcc},\ \mathsf{next\_free}\rangle \xrightarrow{i} \langle m, r, cr', \mathsf{pc}',\ \mathsf{pcc}',\ \mathsf{next\_free}\rangle}\ (\mathsf{ccall})$$

allocate `rs` `cd`: Allocate a new memory region of the size indicated by the value in the register `rs`, and stores a capability with full permissions on the newly allocated region in the capability register `cd`.

$$\frac{\begin{array}{c} m(\mathsf{pc}) = \langle 0, \mathsf{allocate}\ \texttt{rs}\ \texttt{cd}\rangle \\ \texttt{perms\_new} = \{p \mid p \in \mathit{Perm}\} \qquad \texttt{s\_new} = \mathsf{false} \\ \texttt{addr\_new} = \mathsf{next\_free} \qquad \texttt{len\_new} = r(\texttt{rs}) \qquad \texttt{otype\_new} = 0 \\ \mathit{newcap} = \langle 1, (\texttt{s\_new}, \mathsf{bin}(\texttt{perms\_new}), \texttt{addr\_new}, \texttt{len\_new}, \texttt{otype\_new})\rangle \\ cr' = cr[\mathsf{cd} \mapsto \mathit{newcap}] \\ \mathsf{pc}' = \mathsf{pc} + 1 \qquad \mathsf{next\_free}' = \mathsf{next\_free} + \texttt{len\_new} \end{array}}{\langle m, r, cr, \mathsf{pc},\ \mathsf{pcc},\ \mathsf{next\_free}\rangle \xrightarrow{i} \langle m, r, cr', \mathsf{pc}',\ \mathsf{pcc},\ \mathsf{next\_free}'\rangle}\ (\mathsf{allocate})$$

cseal `cd` `cs` `ct`: Store in capability register `cd` a sealed copy of the capability in `cs` where the sealing type is given by the address of the capability in `ct`. The capability in `ct` has to grant the sealing permission.

$$\frac{\begin{array}{c} m(\mathsf{pc}) = \langle 0, \mathsf{cseal}\ \texttt{cd}\ \texttt{cs}\ \texttt{ct}\rangle \\ cr(\texttt{cs}) = \langle 1, (\texttt{s\_s}, \mathsf{bin}(\texttt{perms\_s}), \texttt{addr\_s}, \texttt{len\_s}, \texttt{otype\_s})\rangle \\ cr(\texttt{ct}) = \langle 1, (\texttt{s\_t}, \mathsf{bin}(\texttt{perms\_t}), \texttt{addr\_t}, \texttt{len\_t}, \texttt{otype\_t})\rangle \\ \texttt{otype\_d} = \texttt{addr\_t} \\ cr' = cr[\mathsf{cd} \mapsto \langle 1, (\texttt{true}, \mathsf{bin}(\texttt{perms\_s}), \texttt{addr\_s}, \texttt{len\_s}, \texttt{otype\_d})\rangle] \\ cr \vdash \mathsf{permits\_seal}(\mathsf{cs}, \mathsf{ct}) \qquad \mathsf{pc}' = \mathsf{pc} + 1 \end{array}}{\langle m, r, cr, \mathsf{pc},\ \mathsf{pcc},\ \mathsf{next\_free}\rangle \xrightarrow{i} \langle m, r, cr', \mathsf{pc}',\ \mathsf{pcc},\ \mathsf{next\_free}\rangle}\ (\mathsf{cseal})$$

cunseal `cd` `cs` `ct`: Store in capability register `cd` an unsealed copy of the capability in `cs` where the sealing type has to match the address of the capability in `ct`. The capability in `ct` has to grant the unsealing permission (which is the permission permit_seal).

$$\frac{\begin{array}{c} m(\mathsf{pc}) = \langle 0, \mathsf{cunseal}\ \texttt{cd}\ \texttt{cs}\ \texttt{ct}\rangle \qquad cr \vdash \mathsf{permits\_unseal}(\texttt{cs}, \texttt{ct}) \\ cr' = cr[\mathsf{cd} \mapsto \mathsf{clear\_otype}(\mathsf{remove\_non\_ephemeral}(\mathsf{unsealed}(cr(\texttt{cs})), cr(\texttt{ct})))] \\ \mathsf{pc}' = \mathsf{pc} + 1 \end{array}}{\langle m, r, cr, \mathsf{pc},\ \mathsf{pcc},\ \mathsf{next\_free}\rangle \xrightarrow{i} \langle m, r, cr', \mathsf{pc}',\ \mathsf{pcc},\ \mathsf{next\_free}\rangle}\ (\mathsf{cunseal})$$

$\mathsf{cmove}\ \mathtt{cd}\ \mathtt{cb}$: Move the capability in register $\mathtt{cb}$ into capability register $\mathtt{cd}$.

$$\frac{m(\mathsf{pc}) = \langle 0, \mathsf{cmove}\ \mathtt{cd}\ \mathtt{cb}\rangle \qquad cr' = cr[\mathtt{cd} \mapsto cr(\mathtt{cb})] \qquad \mathsf{pc}' = \mathsf{pc} + 1}{\langle m, r, cr, \mathsf{pc},\ \mathsf{pcc},\ \mathsf{next\_free}\rangle \xrightarrow{i} \langle m, r, cr', \mathsf{pc}',\ \mathsf{pcc},\ \mathsf{next\_free}\rangle}\ (\mathsf{cmove})$$

$\mathsf{movei}\ \mathtt{rd}\ \mathtt{imm}$: Move an immediate value $\mathtt{imm}$ into register $\mathtt{rd}$.

$$\frac{m(\mathsf{pc}) = \langle 0, \mathsf{movei}\ \mathtt{rd}\ \mathtt{imm}\rangle \qquad r' = r[\mathtt{rd} \mapsto \mathtt{imm}] \qquad \mathsf{pc}' = \mathsf{pc} + 1}{\langle m, r, cr, \mathsf{pc},\ \mathsf{pcc},\ \mathsf{next\_free}\rangle \xrightarrow{i} \langle m, r', cr, \mathsf{pc}',\ \mathsf{pcc},\ \mathsf{next\_free}\rangle}\ (\mathsf{movei})$$

$\mathsf{movepc}\ \mathtt{rd}$: Move the value in pc into the register $\mathtt{rd}$.

$$\frac{m(\mathsf{pc}) = \langle 0, \mathsf{movepc}\ \mathtt{rd}\rangle \qquad r' = r[\mathtt{rd} \mapsto \mathsf{pc}] \qquad \mathsf{pc}' = \mathsf{pc} + 1}{\langle m, r, cr, \mathsf{pc},\ \mathsf{pcc},\ \mathsf{next\_free}\rangle \xrightarrow{i} \langle m, r', cr, \mathsf{pc}',\ \mathsf{pcc},\ \mathsf{next\_free}\rangle}\ (\mathsf{movepc})$$

$\mathsf{cincbase}\ \mathtt{cd}\ \mathtt{cb}\ \mathtt{rt}$: Create in $\mathtt{cd}$ a capability on a reduced memory region compared to the capability in $\mathtt{cb}$ by increasing the base address of the capability in register $\mathtt{cb}$ by an increment of the value in register $\mathtt{rt}$.

$$\frac{\begin{array}{c} m(\mathsf{pc}) = \langle 0, \mathsf{cincbase}\ \mathtt{cd}\ \mathtt{cb}\ \mathtt{rt}\rangle \\ cr(\mathtt{cb}) = \langle 1, (\mathtt{s\_b}, \mathsf{bin}(\mathtt{perms\_b}), \mathtt{addr\_b}, \mathtt{len\_b}, \mathtt{otype\_b})\rangle \\ s\_b = \mathtt{false} \qquad r(\mathtt{rt}) \leq \mathtt{len\_b} \qquad cr' = cr[\mathtt{cd} \mapsto \\ \langle 1, (\mathtt{s\_b}, \mathsf{bin}(\mathtt{perms\_b}), \mathtt{addr\_b} + r(\mathtt{rt}), \mathtt{len\_b} - r(\mathtt{rt}), \mathtt{otype\_b})\rangle] \\ \mathsf{pc}' = \mathsf{pc} + 1 \end{array}}{\langle m, r, cr, \mathsf{pc},\ \mathsf{pcc},\ \mathsf{next\_free}\rangle \xrightarrow{i} \langle m, r, cr', \mathsf{pc}',\ \mathsf{pcc},\ \mathsf{next\_free}\rangle}\ (\mathsf{cincbase})$$

$\mathsf{csetlen}\ \mathtt{cd}\ \mathtt{cb}\ \mathtt{rt}$: Create in $\mathtt{cd}$ a capability on a reduced memory region compared to the capability in $\mathtt{cb}$ by setting the length field to a value smaller than the length field of the capability in register $\mathtt{cb}$.

$$\frac{\begin{array}{c} m(\mathsf{pc}) = \langle 0, \mathsf{csetlen}\ \mathtt{cd}\ \mathtt{cb}\ \mathtt{rt}\rangle \\ cr(\mathtt{cb}) = \langle 1, (\mathtt{s\_b}, \mathsf{bin}(\mathtt{perms\_b}), \mathtt{addr\_b}, \mathtt{len\_b}, \mathtt{otype\_b})\rangle \\ s\_b = \mathtt{false} \qquad r(\mathtt{rt}) \leq \mathtt{len\_b} \\ cr' = cr[\mathtt{cd} \mapsto \langle 1, (\mathtt{s\_b}, \mathsf{bin}(\mathtt{perms\_b}), \mathtt{addr\_b}, r(\mathtt{rt}), \mathtt{otype\_b})\rangle] \\ \mathsf{pc}' = \mathsf{pc} + 1 \end{array}}{\langle m, r, cr, \mathsf{pc},\ \mathsf{pcc},\ \mathsf{next\_free}\rangle \xrightarrow{i} \langle m, r, cr', \mathsf{pc}',\ \mathsf{pcc},\ \mathsf{next\_free}\rangle}\ (\mathsf{csetlen})$$

ccleartag `cd cb`: Create an invalid copy of the capability in register `cb` by having the tag unset, and place it in register `cd`.

$$\frac{m(\mathsf{pc}) = \langle 0, \mathsf{ccleartag}\ \mathtt{cd}\ \mathtt{cb}\rangle \qquad cr(\mathtt{cb}) = \langle \mathtt{tag\_b}, \mathtt{content\_b}\rangle \\ cr' = cr[\mathtt{cd} \mapsto \langle 0, \mathtt{content\_b}\rangle] \qquad \mathsf{pc}' = \mathsf{pc} + 1}{\langle m, r, cr, \mathsf{pc},\ \mathsf{pcc},\ \mathsf{next\_free}\rangle \xrightarrow{i} \langle m, r, cr', \mathsf{pc}',\ \mathsf{pcc},\ \mathsf{next\_free}\rangle}\ (\mathsf{ccleartag})$$

candperm `cd cb rt`: Create in `cd` a capability with reduced permissions compared to the capability in `cs` by intersecting the permissions in `cs` with the set of permissions represented by the value in the register `rt`.

$$\frac{m(\mathsf{pc}) = \langle 0, \mathsf{candperm}\ \mathtt{cd}\ \mathtt{cb}\ \mathtt{rt}\rangle \\ cr(cb) = \langle 1, (\mathtt{s\_b}, \mathsf{bin}(\mathtt{perms\_b}), \mathtt{addr\_b}, \mathtt{len\_b}, \mathtt{otype\_b})\rangle \\ \mathtt{s\_b} = \mathtt{false} \qquad cr' = cr[\mathtt{cd} \mapsto \\ \langle 1, (\mathtt{s\_b}, \mathsf{bin}(\mathtt{perms\_b}) \cap r(\mathtt{rt}), \mathtt{addr\_b}, \mathtt{len\_b}, \mathtt{otype\_b})\rangle] \qquad \mathsf{pc}' = \mathsf{pc} + 1}{\langle m, r, cr, \mathsf{pc},\ \mathsf{pcc},\ \mathsf{next\_free}\rangle \xrightarrow{i} \langle m, r, cr', \mathsf{pc}',\ \mathsf{pcc},\ \mathsf{next\_free}\rangle}\ (\mathsf{candperm})$$

ccheckperm `cs rt`: Check that the permissions represented by the value in `rt` exist in the capability in register `cs`. If not, then block execution.

$$\frac{m(\mathsf{pc}) = \langle 0, \mathsf{ccheckperm}\ \mathtt{cs}\ \mathtt{rt}\rangle \\ cr(\mathtt{cs}) = \langle 1, (\mathtt{s\_s}, \mathsf{bin}(\mathtt{perms\_s}), \mathtt{addr\_s}, \mathtt{len\_s}, \mathtt{otype\_s})\rangle \\ \mathtt{perms\_s} \subseteq \mathsf{content}(r(\mathtt{rt})) \qquad \mathsf{pc}' = \mathsf{pc} + 1}{\langle m, r, cr, \mathsf{pc},\ \mathsf{pcc},\ \mathsf{next\_free}\rangle \xrightarrow{i} \langle m, r, cr, \mathsf{pc}',\ \mathsf{pcc},\ \mathsf{next\_free}\rangle}\ (\mathsf{ccheckperm})$$

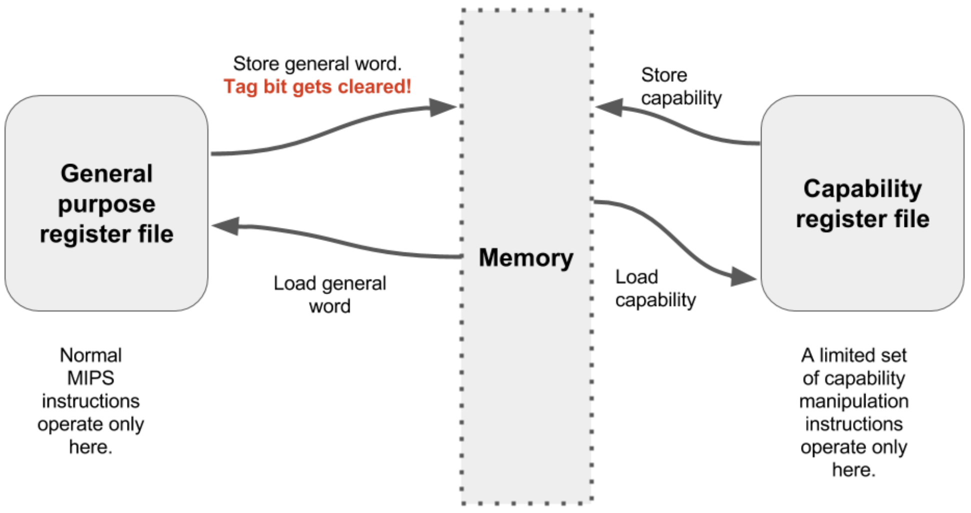
cchecktype `cs cb`: Compare the types in the two capabilities in registers `cs` and `cb`. If they are not equal, then block execution.

$$\frac{m(\mathsf{pc}) = \langle 0, \mathsf{cchecktype}\ \mathtt{cs}\ \mathtt{cb}\rangle \\ cr(\mathtt{cs}) = \langle 1, (\mathtt{s\_s}, \mathsf{bin}(\mathtt{perms\_s}), \mathtt{addr\_s}, \mathtt{len\_s}, \mathtt{otype\_s})\rangle \\ cr(\mathtt{cb}) = \langle 1, (\mathtt{s\_b}, \mathsf{bin}(\mathtt{perms\_b}), \mathtt{addr\_b}, \mathtt{len\_b}, \mathtt{otype\_b})\rangle \\ \mathtt{s\_s} = \mathtt{s\_b} = \mathtt{false} \qquad \mathtt{otype\_s} = \mathtt{otype\_b} \qquad \mathsf{pc}' = \mathsf{pc} + 1}{\langle m, r, cr, \mathsf{pc},\ \mathsf{pcc},\ \mathsf{next\_free}\rangle \xrightarrow{i} \langle m, r, cr, \mathsf{pc}',\ \mathsf{pcc},\ \mathsf{next\_free}\rangle}\ (\mathsf{cchecktype})$$

3.2.3 Useful Properties of the Formal Model

A main useful feature of the operational semantics of CHERI is that valid capabilities that exist in the capability register file and in the memory cannot have their set of

FIGURE 3.1: CHERI ISA is meant to guarantee unforgeability of capabilities


Store general word.
Tag bit gets cleared!
Store capability
General purpose register file
Memory
Capability register file
Load general word
Load capability
Normal MIPS instructions operate only here.
A limited set of capability manipulation instructions operate only here.

permissions increased or their address ranges enlarged. If a modification with such an effect (increasing of permissions, or enlarging the address range) happens, then it has to happen in the general-purpose register file, and then whenever the result of such an operation gets stored in memory, the tag bit is cleared so that the memory word is not a valid capability. This is illustrated in Figure 3.1 which describes the different operations that control the flow of data among the capability register file, memory, and the general-purpose register file.

Capability unforgeability states that if execution starts in a CHERI machine state in which a particular permission p is not granted on a particular allocated memory address a (characterized by the absence of a valid capability on a that simultaneously grants p), then in no reachable CHERI machine state can the permission p exist on the address a. In other words, Theorem 3.1 states that the set of permissions (granted by the union of valid capabilities in a CHERI machine state) on an already allocated address is monotonically decreasing over the allowed execution steps.

By this notion of unforgeability, we capture a notion that can be seen as similar to the concepts of "only connectivity begets connectivity" and "no authority amplification" which are pointed out by Maffeis et al. [42]. In fact, the guarantees provided by compartmentalization (Definition 5.4 and Theorem 5.5 in Chapter 5) lay out a framework for organizing code into sections in a way that makes interaction between such code sections roughly follow constraints similar to those implied by the two concepts mentioned.

We now state the following theorem about capability unforgeability:

Theorem 3.1. *(Capability Unforgeability - No Privilege Escalation)*

If

1. $\langle m, r, cr, \mathit{pc}, \mathit{pcc}, \mathit{next\_free}\rangle \rightarrow^* \langle m', r', cr', \mathit{pc}', \mathit{pcc}', \mathit{next\_free}'\rangle$,
2. $a \in \mathit{Addr}$, $p \in \mathit{Perm}$,
3. $a < \mathit{next\_free}$,
4. $\forall a' \in \mathit{Addr}.\, m(a') = \langle 1, (\_, \mathit{bin}(\mathtt{perms}), \mathtt{st}, \mathtt{len}, \_)\rangle \Rightarrow$
   $(p \notin \mathtt{perms} \vee a \notin [\mathtt{st}, \mathtt{st} + \mathtt{len}))$,
5. $\forall c \in \mathit{CapRegName}.\, cr(c) = \langle 1, (\_, \mathit{bin}(\mathtt{perms}), \mathtt{st}, \mathtt{len}, \_)\rangle \Rightarrow$
   $(p \notin \mathtt{perms} \vee a \notin [\mathtt{st}, \mathtt{st} + \mathtt{len}))$,
6. $\mathit{pcc} = \langle 1, (\_, \mathit{bin}(\mathtt{perms}), \mathtt{st}, \mathtt{len}, \_)\rangle \Rightarrow$
   $(p \notin \mathtt{perms} \vee a \notin [\mathtt{st}, \mathtt{st} + \mathtt{len}))$,

then

(a) $\forall a' \in \mathit{Addr}.\, m'(a') = \langle 1, (\_, \mathit{bin}(\mathtt{perms}), \mathtt{st}, \mathtt{len}, \_)\rangle \Rightarrow$
$(p \notin \mathtt{perms} \vee a \notin [\mathtt{st}, \mathtt{st} + \mathtt{len}))$,

and

(b) $\forall c \in \mathit{CapRegName}.\, cr'(c) = \langle 1, (\_, \mathit{bin}(\mathtt{perms}), \mathtt{st}, \mathtt{len}, \_)\rangle \Rightarrow$
$(p \notin \mathtt{perms} \vee a \notin [\mathtt{st}, \mathtt{st} + \mathtt{len}))$,

and

(c) $\mathit{pcc}' = \langle 1, (\_, \mathit{bin}(\mathtt{perms}), \mathtt{st}, \mathtt{len}, \_)\rangle \Rightarrow$
$(p \notin \mathtt{perms} \vee a \notin [\mathtt{st}, \mathtt{st} + \mathtt{len}))$,

Proof. We prove it by induction on the number n of transition steps $\rightarrow$. The base case $n = 0$ is trivial; it follows directly from the assumptions.

For the inductive case, we have a state $\langle m'', r'', cr'', \mathrm{pc''}, \mathrm{pcc''}, \mathrm{next\_free''}\rangle$ satisfying the induction hypothesis (conditions 2, 3, 4, 5, 6).

And we need to show that if $\langle m'', r'', cr'', \mathrm{pc''}, \mathrm{pcc''}, \mathrm{next\_free''}\rangle \rightarrow \langle m', r', cr', \mathrm{pc'}, \mathrm{pcc'}, \mathrm{next\_free'}\rangle$ then propositions a, b, and c hold.

We do it by case distinction on the rules for instruction execution $\xrightarrow{i}$:

- **Case cchecktype, ccheckperm, movepc, movei, cbts-true, cbts-false, cjrcond-false, cload, BinOp:**

We observe that in all of these rules, $m' = m'', cr' = cr'', \texttt{pcc}' = \texttt{pcc}''$.

So we have that propositions a, b, and c follow directly from induction hypotheses 4, 5, and 6 respectively.

Case proved.

- **Case cmove**

  - We observe that $m' = m'', \texttt{pcc}' = \texttt{pcc}''$.

    So we have that propositions a and c follow directly from induction hypotheses 4 and 6.
  - To show proposition b, we obtain the necessary precondition $cr' = cr''[\texttt{cd} \mapsto cr''(\texttt{cb})]$ of the rule cmove.
  - By case distinction on the condition $c = \texttt{cd}$, we conclude that proposition b follows immediately from induction hypothesis 5 on cr'' with the instantiation $c = \texttt{cb}$ in case $c = \texttt{cd}$ holds of the goal, and that proposition b follows immediately from induction hypothesis 5 in case $c \neq \texttt{cd}$.

  Case proved.

- **Case cstore:**

  - We observe that $cr' = cr'', \texttt{pcc}' = \texttt{pcc}''$.

    So we have that propositions b and c follow directly from induction hypotheses 5 and 6 respectively.
  - To show that a holds, we proceed by case distinction on the tag bit of memory words in the range of m' for an arbitrary $a' \in Addr$:

    * Case $m'(a') = \langle \texttt{0}, \_ \rangle$:

      Then we have that proposition a holds vacuously.
    * Case $m'(a') = \langle \texttt{1}, \_ \rangle$:

      Then we know from the rule cstore that necessarily $m'(a') = m''(a')$ (inferred from the necessary precondition that defines m':
      $m' = m''[\texttt{content}(r''(\texttt{rt})) \mapsto \langle \texttt{0}, \texttt{content\_s} \rangle])$, and so for that case, proposition a follows from induction hypothesis 4.

  Case proved.

- **Case cloadcap**

  - We observe that $m' = m''$ and $\texttt{pcc}' = \texttt{pcc}''$.

    Then we have that propositions a and c follow directly from induction hypotheses 4 and 6 respectively.

- To show proposition b, we proceed by case distinction on the register name c.
  - Case $c \neq \texttt{cd}$
    Here, proposition b follows directly from induction hypothesis 5.
  - Case $c = \texttt{cd}$
    We obtain the necessary precondition $cr' = cr''[\texttt{cd} \mapsto m''(\mathsf{content}(r''(\texttt{rt})))]$, and we use induction hypothesis 4 on m'' with the instantiation $a' = \mathsf{content}(r''(\texttt{rt}))$ to conclude that proposition b follows immediately for this case.

Case proved.

- **Case cstorecap**
  - We observe that $cr' = cr''$ and $\mathsf{pcc}' = \mathsf{pcc}''$.
    Then we have that propositions b and c follow directly from induction hypotheses 5 and 6 respectively.
  - To show proposition a, we obtain the necessary precondition $m' = m''[\mathsf{content}(r''(\texttt{rt})) \mapsto cr''(\texttt{cs})]$ of the rule $\mathsf{cstorecap}$, and we proceed by case distinction on the memory address a' in statement a.
    - Case $a' \neq \mathsf{content}(r''(\texttt{rt}))$
      Then, proposition a follows directly from induction hypothesis 4 about m''.
    - Case $a' = \mathsf{content}(r''(\texttt{rt}))$
      Then, proposition a follows directly from induction hypothesis 5 about cr'' with the instantiation $c = \texttt{cs}$.

Case proved.

- **Case ccleartag**
  - We observe that $m' = m'', \mathsf{pcc}' = \mathsf{pcc}''$.
    So we have that propositions a and c follow directly from induction hypotheses 4 and 6 respectively.
  - To show that b holds, we obtain the necessary preconditions $cr''(\texttt{cb}) = \langle \texttt{tag\_b}, \texttt{content\_b} \rangle$ and $cr' = cr''[\texttt{cd} \mapsto \langle \texttt{0}, \texttt{content\_b} \rangle]$ of the rule $\mathsf{ccleartag}$. We then proceed by case distinction over the register name c that is mentioned in proposition b:
    - Case $c \neq \texttt{cd}$
      Then proposition b follows directly from induction hypothesis 5 about cr''.

    * Case $c = \texttt{cd}$
      Then statement b becomes vacuously true (because the tag bit is 0).

  Case proved.

- **Case candperm:**

  We observe that $m' = m''$, $\mathsf{pcc}' = \mathsf{pcc}''$.

  So we have that propositions a and c follow directly from induction hypotheses 4 and 6 respectively.

  To show that b holds, we distinguish between the register name `cd` mentioned in the preconditions of the rule candperm and the other registers.

  - Case $c \neq \texttt{cd}$:

    We observe that $cr'|_{CapRegName\setminus\{\texttt{cd}\}} = cr''|_{CapRegName\setminus\{\texttt{cd}\}}$, and hence proposition b follows immediately from induction hypothesis 5.

  - Case $c = \texttt{cd}$:

    Let $cr''(\texttt{cb}) = \langle \texttt{1}, (\texttt{s\_b}, \mathsf{bin}(\texttt{perms\_b}), \texttt{addr\_b}, \texttt{len\_b}, \_)\rangle$ and observe from the precondition:

    $cr' = cr''[\texttt{cd} \mapsto \langle \texttt{1}, (\texttt{s\_b}, \mathsf{bin}(\texttt{perms\_b}) \cap r''(\texttt{rt}), \texttt{addr\_b}, \texttt{len\_b}, \texttt{otype\_b})\rangle]$

    of the rule candperm that:

    $cr'(\texttt{cd}) = \langle \texttt{1}, (\texttt{s\_b}, \mathsf{bin}(\texttt{perms\_d}), \texttt{addr\_b}, \texttt{len\_b}, \_)\rangle$ with the condition that $p \in \texttt{perms\_d} \Rightarrow p \in \texttt{perms\_b}$.

    So we have:

    * Either $p \notin \texttt{perms\_d}$; in which case the consequent of statement b holds.
    * Or $p \in \texttt{perms\_d}$; in which case we know that $p \in \texttt{perms\_b}$.
      But this means that (since we know that induction hypothesis 5 must hold for $c = \texttt{cb}$) $a \notin [\texttt{addr\_b}, \texttt{addr\_b} + \texttt{len\_b})$.
      Hence, proposition b holds on cr' with the instantiation $c = \texttt{cd}$ because $cr'(\texttt{cd})$ consists of the same values `addr_b` and `len_b` for the corresponding fields. So the proposition $a \notin [\texttt{addr\_b}, \texttt{addr\_b} + \texttt{len\_b})$ follows from the induction hypothesis making the consequent of statement b also true.

  Case proved.

- **Case cincbase**

  This case is analogous to **Case candperm**.

  The same arguments hold for propositions a and c. For proposition b, we have the same case distinction. Case $c \neq \texttt{cd}$ is the same.

We show Case $c = \texttt{cd}$:

This case is dual to the corresponding case of candperm.

Let $cr''(\texttt{cb}) = \langle \texttt{1}, (\texttt{s\_b}, \mathsf{bin}(\texttt{perms\_b}), \texttt{addr\_b}, \texttt{len\_b}, \_) \rangle$ and observe from the preconditions:

(i) $cr' = cr''[\texttt{cd} \mapsto$
$\langle \texttt{1}, (\texttt{s\_b}, \mathsf{bin}(\texttt{perms\_b}), \texttt{addr\_b} + r''(\texttt{rt}), \texttt{len\_b} - r''(\texttt{rt}), \texttt{otype\_b}) \rangle]$, and

(ii) $r''(\texttt{rt}) \leq \texttt{len\_b}$

of the rule cincbase and from the fact that:

(iii) $r''(\texttt{rt})$ is non-negative which follows from the type information $\mathit{Content} = \mathbb{N}$

that $\texttt{addr\_b} \leq \texttt{addr\_d} \leq \texttt{addr\_d} + \texttt{len\_d} \leq \texttt{addr\_b} + \texttt{len\_b}$

where

$cr'(\texttt{cd}) = \langle \texttt{1}, (\texttt{s\_b}, \mathsf{bin}(\texttt{perms\_b}), \texttt{addr\_d}, \texttt{len\_d}, \_) \rangle$.

Hence we conclude that $a \in [\texttt{addr\_d}, \texttt{addr\_d} + \texttt{len\_d}) \Rightarrow a \in [\texttt{addr\_b}, \texttt{addr\_b} + \texttt{len\_b})$.

So we have:

– Either $a \notin [\texttt{addr\_d}, \texttt{addr\_d} + \texttt{len\_d})$; in which case the consequent of statement b holds.

– Or $a \in [\texttt{addr\_d}, \texttt{addr\_d} + \texttt{len\_d})$; in which case we know that $a \in [\texttt{addr\_b}, \texttt{addr\_b} + \texttt{len\_b})$ holds (by the implication concluded above). But this means that $p \notin \texttt{perms\_b}$ (by disjunctive syllogism since we know that induction hypothesis 5 must hold with the instantiation $c = \texttt{cb}$). Hence, proposition b holds on cr' with the instantiation $c = \texttt{cd}$ because $cr'(\texttt{cd})$ consists of the same value $\texttt{perms\_b}$ for the permissions field. So the proposition $p \notin \texttt{perms\_b}$ follows from the induction hypothesis making the consequent of statement b also true.

Case proved.

- **Case csetlen:**

This case is similar to **Case cincbase**. The main observation is that we can conclude the same proposition $\texttt{addr\_b} \leq \texttt{addr\_d} \leq \texttt{addr\_d} + \texttt{len\_d} \leq \texttt{addr\_b} + \texttt{len\_b}$ from the preconditions:

(i) $cr' = cr''[\texttt{cd} \mapsto \langle \texttt{1}, (\texttt{s\_b}, \mathsf{bin}(\texttt{perms\_b}), \texttt{addr\_b}, r''(\texttt{rt}), \texttt{otype\_b}) \rangle]$, and

(ii) $r''(\texttt{rt}) \leq \texttt{len\_b}$

of the rule csetlen and from the fact that:

(iii) $r''(\texttt{rt})$ is non-negative which follows from the type information $\mathit{Content} = \mathbb{N}$.

Case can hence be proved with the rest of the argument being identical to cincbase. We avoid repetition.

- **Case cseal:**
  - We observe that $m' = m'', \mathsf{pcc}' = \mathsf{pcc}''$.

    So we have that propositions a and c follow directly from induction hypotheses 4 and 6 respectively.
  - Similarly to all cases where $cr' \neq cr''$, we distinguish between the cases $c \neq \texttt{cd}$ and $c = \texttt{cd}$. For the former case, proposition b follows directly from induction hypothesis 5.

    For case $c = \texttt{cd}$, we obtain the necessary precondition $cr' = cr''[\texttt{cd} \mapsto \langle 1, (\texttt{true}, \mathsf{bin}(\texttt{perms\_s}), \texttt{addr\_s}, \texttt{len\_s}, \texttt{otype\_d})\rangle]$ of the rule cseal and notice that the fields $\texttt{perms\_s}, \texttt{addr\_s}, \texttt{len\_s}$ are the same from $cr''(\texttt{cs})$, and hence we satisfy proposition b for $c = \texttt{cd}$ directly based on induction hypothesis 5 applied with the instantiation $c = \texttt{cs}$.

  Case proved.
- **Case cunseal:**
  - We observe that $m' = m'', \mathsf{pcc}' = \mathsf{pcc}''$.

    So we have that propositions a and c follow directly from induction hypotheses 4 and 6 respectively.
  - Similarly to all cases where $cr' \neq cr''$, we distinguish between the cases $c \neq \texttt{cd}$ and $c = \texttt{cd}$. For the former case, proposition b follows directly from induction hypothesis 5.

    For case $c = \texttt{cd}$, we obtain the necessary precondition $cr' = cr''[\texttt{cd} \mapsto \mathsf{clear\_otype}(\mathsf{remove\_non\_ephemeral}(\mathsf{unsealed}(cr''(\texttt{cs})), cr''(\texttt{ct})))]$ of the rule cunseal and notice from the definition of remove_non_ephemeral that we can distinguish two cases:
    * Case $\mathsf{non\_ephemeral} \in \texttt{perms\_t}$: In this case, looking at the definition of unsealed and noticing that it does not change the permissions field of the base and length fields, we deduce that proposition b follows directly from induction hypothesis 5 about cr'' instantiated with $c = cs$.
    * Case $\mathsf{non\_ephemeral} \notin \texttt{perms\_t}$: In this case, either $p = \mathsf{non\_ephemeral}$, in which case proposition b holds because the disjunct $p \notin \texttt{perms}$ will hold for the instantiation $c = \texttt{cd}$; or $p \neq \mathsf{non\_ephemeral}$, in which case

again proposition b follows directly from induction hypothesis 5 about cr'' instantiated with $c = cs$.

- **Case cjrcond-true, cjr:**
  - We observe that $m' = m'', cr' = cr''$.

    So we have that propositions a and b follow directly from induction hypotheses 4 and 5 respectively.
  - To show that proposition c holds, we obtain the necessary precondition $\mathsf{pcc}' = cr''(\mathsf{cb})$ of both rules cjrcond-true and cjr. We then note that proposition c follows directly from the validity of induction hypothesis 5 on cr'' with the instantiation $c = \mathsf{cb}$.

    Case proved.
- **Case ccall:**
  - We observe that $m' = m''$.

    So we have that proposition a follows directly from induction hypothesis 4.
  - To show that proposition b holds:

    Similarly to all cases where $cr' \neq cr''$, we distinguish between the cases $c \neq \mathsf{cdd}$ and $c = \mathsf{cdd}$. For the former case, proposition b follows directly from induction hypothesis 5. For the latter, we obtain the necessary preconditions:

    (i) $cd' = \mathsf{unsealed}(cr''(\mathsf{cd}))$, and

    (ii) $cr' = cr''[cdd \mapsto cd']$

    of the rule ccall.

    We then notice from the definition of unsealed that the permissions, base, and length fields all do not change.

    So we conclude that proposition b follows directly from induction hypothesis 5 on cr'' with the instantiation $c = \mathsf{cd}$.
  - To show that proposition c holds, we obtain the necessary preconditions:

    (i) $cc' = \mathsf{unsealed}(cr''(\mathsf{cc}))$, and

    (ii) $\mathsf{pcc}' = cc'$

    of the rule ccall.

    Using the same observation mentioned above about the definition of unsealed, we conclude that proposition c follows directly from induction hypothesis 5 on cr'' with the instantiation $c = \mathsf{cc}$.
- **Case allocate**

- We observe that $m' = m''$, $\mathsf{pcc}' = \mathsf{pcc}''$.

  So we have that propositions a and c follow directly from induction hypotheses 4 and 6.

- To show that proposition b holds, similarly to all cases where $cr' \neq cr''$, we distinguish between the cases $c \neq \mathtt{cd}$ and $c = \mathtt{cd}$. For the former case, proposition b follows directly from induction hypothesis 5.

  For the latter case ($c = \mathtt{cd}$), we obtain the necessary preconditions:

  (i) $\mathtt{addr\_new} = \mathsf{next\_free}$,

  (ii) $newcap =$
  $\langle 1, (\mathtt{s\_new}, \mathsf{bin}(\mathtt{perms\_new}), \mathtt{addr\_new}, \mathtt{len\_new}, \mathtt{otype\_new})\rangle$, and

  (iii) $cr' = cr''[\mathtt{cd} \mapsto newcap]$

  of the rule allocate,

  and we also use induction hypothesis 3 which states that $a < \mathsf{next\_free}$ to conclude that $a \notin [\mathtt{addr\_new}, \mathtt{addr\_new} + \mathtt{len\_new})$ and hence that the consequent of statement b holds.

Case proved.

All cases covered. □

The theorem below states that the memory address that points to the next free memory location in a CHERI machine state can only increase or remain constant upon execution of CHERI machine instructions. This can be shown by observing that the only instruction that changes the value of this address is the `allocate` instruction, and that it always increments it. All other instructions do not change the value of next_free.

An obvious extension of such a model of the instruction set is to have a more realistic data structure for memory management with operations/instructions that allow deallocating memory.

Theorem 3.2. *(next_free is non-decreasing)*

$$\langle m, r, cr, pc, pcc, next\_free\rangle \to^* \langle m', r', cr', pc', pcc', next\_free'\rangle \Rightarrow$$

$$next\_free' \geq next\_free$$

Proof. We prove it by induction on execution steps. Base case ($n = 0$) is trivial (by reflexivity of $\leq$). For the inductive case, we proceed by case distinction on the instruction step $\xrightarrow{i}$.

- **Case allocate:**

  We notice from the type information $\mathit{Content} = \mathbb{N}$ and from the obtained preconditions $\texttt{len\_new} = r(\texttt{rs})$ and $\mathsf{next\_free}' = \mathsf{next\_free} + \texttt{len\_new}$ of the rule allocate that $\mathsf{next\_free}' \geq \mathsf{next\_free}$ follows immediately.

- All other cases follow trivially from the observation that $\mathsf{next\_free} = \mathsf{next\_free}'$.

$\square$

Chapter 4

Control Flow Integrity based on CHERI MIPS

Some attacks arise in the form of exploiting vulnerabilities in a computer program that enable attackers to subvert the expected flow of the program intended by the programmer. So constraining the possible behaviors or flows of a program is a known goal for mitigation techniques.

The set of possible legal behaviors of a program can be approximated by means of static analysis of binary or source code (like in CFI [19, 20, 21] or in [43] where a program is forced to execute only system calls that are accepted by some computed automaton) or by other techniques that are intended for mitigating known vulnerabilities (like mitigating buffer-overflows in C [44]) that arise due to compiler implementations together with programmer errors. One other way of achieving the goal of constraining flow of a program is to preserve the behavior of a program upon the compilation (translation) to machine code. Typed Assembly Language [45] is one example of this direction.

Control Flow Integrity (CFI) [19, 20, 21] aims at restricting the flow transfers of a machine-code program. The restriction is that flow transfers (jumps) should be taking place only to target addresses that are valid destinations with respect to a predetermined control flow graph. Enforcement of such a policy is implemented by inline instrumentation of the machine code. Statically, the control flow graph is computed by analysis of the binary code. Dynamically, the instrumentation guarantees that every indirect jump takes place to one of a set of expected target destinations before it (the instrumentation) allows the jump to take place. Direct jumps or instructions that target a constant destination can be inspected statically, so there is no need to enforce any policy on their execution.

The instrumentation modifies each source and destination of an indirect jump. So, based on a Control Flow Graph (CFG), an indirect jump instruction is allowed to take place only if the destination of the jump complies with an existing edge in the graph. The graph is constructed in such a way that for some source instruction, all possible destinations are represented as one node. Also, as an over-approximation that helps for efficiency concerns, two destinations are considered equivalent (i.e., represented using one node) if they share a common source.

Control Flow Integrity (CFI) is a safety property whose enforcement provides protection even against adversaries that are able to control the data memory of the executing program [19]. It rests on three main assumptions:

- unique IDs for "different" code destinations (UNQ),
- non-writable code (NWC), and
- non-executable data (NXD).

In this chapter we illustrate briefly that all of these assumptions are achievable by a compilation procedure that uses CHERI as a target architecture. We show a simple way by which one can achieve CFI enforcement by means of simple machine code rewriting. In this we follow the footsteps of the formal model given by the authors of Control Flow Integrity in [20], which itself is based on the language given in [46]. We point out that there are other techniques [47] for CFI enforcement, and that we choose the machine code instrumentation technique for simplicity of replicating the work done [20] for the x86 architecture on MIPS [34] and because no architectural support would be required more than what is available by MIPS. Nevertheless, it is important to make sure that the additional CHERI instructions still leave it possible to enforce CFI by machine code instrumentation. In Section 4.4, we prove Theorem 4.2 about our simplified formal model which is almost identical to the theorem in [20].

NXD and NWC are straightforward to achieve by any CHERI-based operating system or compiler. The permissions field already offers the separation between the load, store, and execute privileges [11]. One can potentially make use of the other unused bits (or in our formal model, simply the permissions set) in order to assign unique IDs to the different code destinations (UNQ). Or we can alternatively make use of the *otype* field of the capability. Both of these are candidate choices because they have a corresponding check instruction, namely `ccheckperm` and `cchecktype` respectively.

However, more obviously, CFI enforcement by machine code rewriting does not use any specific features of the x86 architecture (the architecture of choice in [19]), so having a look at [20, Section 4.1], we realize that the only steps necessary to show CFI enforcement on CHERI, is to do the machine code rewriting without the help of any specific CHERI instructions and to only use ordinary MIPS [34] instructions, which are already anyway the general-purpose computing basis of the CHERI architecture [11]. In particular, even though a label instruction (which serves as a colored checkpoint by the enforcement mechanism) is not part of a MIPS ISA [34], one can imagine a code rewriting mechanism which uses an immediate movei or addi instruction, and just makes sure that the destination register is never used by the called code sequence.

We adapt the conditions mentioned in Section 4.1 in [20] that are enforced on the “code memory” by the formal model of CFI to our model of CHERI, and we adapt Theorem 1 in Section 4.2 in [20] to the new conditions.

Informally, the theorem should then guarantee that, given an initial state that satisfies the new conditions on a CFG on the code memory, any change of the state is either a malicious modification of data memory, accompanied with no change in pc, or that the next pc value is a valid successor corresponding to the code memory, the CFG, and the pc value. This would mean, in particular, that despite attacks on data memory, the program flow respects the computed CFG [20].

4.1 NXD and NWC using CHERI

Work has been done to achieve application compartmentalization [25] supported by CHERI in the form of compiler-directed memory protection. See section 5.2 for an illustrative example of how memory organization using capabilities can be thought of in terms of compartments, and how this guarantees NXD and NWC.

4.2 Constructing the Control Flow Graph by static analysis

Following the convention in [20], we denote the control flow graph by the function succ.

In our treatment, we can let the function succ map a memory address to a set of memory addresses, which are all valid jump destinations.

As in [20, Section 3], all destinations that are potentially valid for a particular source address are treated as equivalent. In particular, a static analysis constructing a control flow graph needs to guarantee that $\mathsf{succ}(w_0) \cap \mathsf{succ}(w_1) = \emptyset$ or $\mathsf{succ}(w_0) = \mathsf{succ}(w_1)$ for arbitrary instruction memory locations w_0, w_1.

The requirement above which actually describes an approximation that reduces the precision of the analysis is crucial for feasibility of an enforcement mechanism that uses colors.

A more precise Control Flow Graph

It is worth noting that for an enforcement mechanism that uses colors (See 4.3), having a "color" represent a set of destinations (in particular, a color per jump destination representing the subset of code memory which is a possible successor of that destination), yields an exponential decrease in the number of possible destinations in a code sequence that can possibly be protected by an n-bit color. An n-bit color in that case would be able to protect a code base with just n destinations, instead of 2^n destinations when the approximation mentioned in [20, Section 3] is employed. So, raising the precision of CFI enforcement may require other less obvious mechanisms if large code bases are to be protected with a feasible overhead (If the code base is on the order of a few hundreds (let alone a few thousands) of potential jump destinations, then an overhead of a few hundred bits protecting each jump destination is not practical on any existing ISA. A different technique from the one in [20, Section 4] would have to be used).

So, we require that the succ function on memory addresses have the following properties:

For a particular CHERI machine state $\langle m, r, cr, \mathsf{pc}, \mathsf{pcc}, \mathsf{next\_free}\rangle$,

- $\{\mathsf{pc}+1, \mathsf{pc}+\mathtt{OFFS}+1\} = \mathsf{succ}(pc)$ if $m(\mathsf{pc}) = \langle 0, \mathsf{cbts}\ \mathtt{cb}\ \mathtt{OFFS}\rangle$.
- $\{\mathsf{pc}+1, \mathsf{pc}+\mathtt{OFFS}+1\} = \mathsf{succ}(pc)$ if $m(\mathsf{pc}) = \langle 0, \mathsf{cjrzero}\ \mathtt{cb}\ \mathtt{rz}\ \mathtt{OFFS}\rangle$.
- $\{\mathsf{pc}+1, \mathsf{pc}+\mathtt{OFFS}+1\} = \mathsf{succ}(pc)$ if $m(\mathsf{pc}) = \langle 0, \mathsf{cjrnotzero}\ \mathtt{cb}\ \mathtt{rz}\ \mathtt{OFFS}\rangle$.
- $\mathsf{succ}(\mathsf{pc}) \neq \emptyset$ if $m(\mathsf{pc}) = \langle 0, \mathsf{cjr}\ \mathtt{rt}\ \mathtt{cb}\rangle$.

  This is to ensure non-triviality of the static analysis. A trivially sound analysis can disallow any indirect jump address by simply including only edges between consecutive instructions.
- $\mathsf{succ}(\mathsf{pc}) \neq \emptyset$ if $m(\mathsf{pc}) = \langle 0, \mathsf{ccall}\ \mathtt{cc}\ \mathtt{cd}\ \mathtt{cdd}\rangle$.

  This is to ensure non-triviality.
- $\mathsf{succ}(\mathsf{pc}) = \{\mathsf{pc}+1\}$ if $m(\mathsf{pc})$ is any other valid instruction encoding than the ones above.

- $\mathsf{succ}(\mathsf{pc}) = \emptyset$ if $m(\mathsf{pc})$ is not a valid instruction encoding.

  This is not necessary for correctness of the static analysis, since the protection it provides is automatically guaranteed by CHERI's checks on pcc. We include it just to make sure the static analysis does not add useless information to the graph.

Note that the condition that we require on the indirect jump instruction cjr is just a sanity requirement. This instruction is the very instruction that CFI seeks to protect.

Like in [21, Section 6.3], $\mathsf{dst}(\mathsf{pc})$ or $\mathsf{dst}(m, G, \mathsf{pc})$ are used to denote an ID of the equivalence class of $\mathsf{succ}(\mathsf{pc})$.

Note that the function succ is defined for a particular machine state. Another way to refer to this parametrization of succ on the machine state is to instead write $\mathsf{succ}(\mathsf{pc}, m)$ because the only interesting component of the machine state that should affect the definition of succ is the memory. In particular for a fixed non-writable code memory (i.e., memory on which whenever there exists an execute permission, it does not also lie under a write permission), succ function can have a fixed definition for all the CHERI machine states that follow from a state with such code memory.

4.3 Enforcing CFI by machine code rewriting

We assume we have a valid CFG, and show the properties required of a transformed (re-written) sequence of CHERI machine code so that a safety theorem holds.

We follow the treatment in [20, Section 4.1].

The verification criteria of a code rewriting procedure that enforces CFI are:

1. Every memory location that is computed by the static analysis to be a potential destination of a cjr instruction should be a location that contains the special instruction movei with its argument $\mathtt{r}_{\mathtt{reserved}}$. In other words, for a fixed and chosen $\mathtt{r}_{\mathtt{reserved}}$ (a register which is not used elsewhere in the program code[1]), all jump destinations should be of the form:

$$m(\mathsf{jump\_destination}) = \langle 0, \mathsf{movei}\ \mathtt{r}_{\mathtt{reserved}}\ \mathtt{ID}\rangle$$

   for some value $\mathtt{ID}$ which is unique for the equivalence class of this jump destination.

[1]Note that one can ensure that a certain register is reserved by the binary code instrumentation procedure itself.

2. Every cjr instruction should be preceded by the following instructions of this form:

$$\begin{array}{l}
\mathsf{movei}\ \mathtt{rc}\ \mathtt{IMM} \\
\mathsf{cload}\ \mathtt{ri}\ \mathtt{rt}\ \mathtt{cc} \\
\mathsf{sub}\ \mathtt{rz}\ \mathtt{ri}\ \mathtt{rc} \\
\mathsf{cjrnotzero}\ \mathtt{cc}'\ \mathtt{cz}\ \mathtt{HALT} \\
\mathbf{cjr}\ \mathtt{rt}\ \mathtt{cc}
\end{array}$$

The immediate value `IMM` in register `rc` should equal the value of the instruction at the jump destination (i.e., the legal entry point). This instrumentation checks this fact by loading in register `ri` the actual instruction at the jump destination (i.e., the memory word at the memory address specified by `rt`), and checking that both values in `rc` and `ri` are equal. If not, it forces a jump to the offset `HALT` which would be the address of a safe instruction.

That instruction at the jump destination can be a dummy instruction, but it should be one that is unique to the equivalence class of destinations defined by the CFG. Since there is no dedicated label instruction in the standard MIPS instruction set, movei can be used, with the destination register being a fixed register that is not used elsewhere in the code, and the "color" denoting the equivalence class can be described by the immediate value of movei.

In particular, by condition 1, we require that all and only the memory words at any valid jump destination address $\mathsf{pc}_{\mathsf{jump\_destination}}$ have the form:

$$m(\mathsf{pc}_{\mathsf{jump\_destination}}) = \langle 0, \mathsf{movei}\ \mathtt{r}_{\mathtt{reserved}}\ \mathtt{IMM} \rangle$$

where $\mathtt{r}_{\mathtt{reserved}}$ is a register name that is not used anywhere else in the executing program, and $\mathtt{IMM} = \mathsf{dst}(m, G, \mathsf{pc})$ for all and only pc values in the same equivalence class $\mathsf{succ}(\mathsf{pc})$.

Note that as far as the formal semantics are concerned, it suffices to compare the value `IMM` to the full memory word at the jump destination because both are unbounded natural numbers. However, practically, only part of the memory word should be compared against `IMM`.

The immediate value `HALT` should equal the address of a safe instruction that will handle the violation of the control flow graph that is detected, and `cc'` should be a capability register that contains a valid capability which allows execution of this safe instruction at `HALT`.

sub is one instance of a binop instruction.

cjrnotzero is dual to cjrzero. We use it here for convenience, and to avoid more checks.

Note that the cload instruction uses the same capability register `cc` that is used for the jump (cjr) instruction. This is not necessary. Another correct rewriting of the machine code could also use separate capability registers, and additionally check that they both have the same base value (if not, jump to `HALT`). This latter way would be beneficial when the code at the jump destination is intended to be executable but not readable. A capability for reading only the first (artificially added) instruction will then be all that is needed for cload.

3. Every cjrzero or cjrnotzero instruction should have its destination address not holding a cjr instruction or any of the occurrences of the instructions:

```
cload ri rt cc
sub rz ri rc
cjrnotzero cc' cz HALT
```

that precede a cjr instruction according to condition 2. Note that the destination address of both of these instructions is statically computable because its is an immediate value representing an offset to the current pc value.

Note: Following the treatment in [21, 20], we write $I(m)$ or $I(m, G)$ to denote that a certain memory (in particular, code memory) m satisfies the instrumentation conditions mentioned above for an expected control-flow graph G.

4.4 CFI Theorem

We follow the treatment in [20, Section 4.2] and state the following theorem about control flow integrity.

Separate syntactic representation of code memory

For convenience, we stick to the syntactic representation of a separate code memory M_c, where for a given CHERI machine state $\langle m, r, cr, \text{pc}, \text{pcc}, \text{next\_free}\rangle$, we assume that for any memory address $a < \text{next\_free}$,

- **Either** all the capabilities cap that are available in the capability register file cr, or *anywhere* in the memory m over this address (i.e., cap with $cap.addr \leq a \leq cap.addr + cap.len$) are subject to the constraint that **permit_execute** $\in cap.perms$ **and permit_store** $\notin cap.perms$
- **Or** that all of them are subject to the constraint that **permit_store** $\in cap.perms$ **and permit_execute** $\notin cap.perms$.

And hence, we define code memory to be the executable (and, hence, non-writable) part of memory

$$M_c := m|_{exec\_a}$$

where

$$exec\_a := \{a \mid \exists\ cap.\ \ cap.addr \leq a \leq cap.addr + cap.len \wedge \mathsf{permit\_execute} \in cap.perms\}$$

where the existential quantifier is over the contents of memory words in the range of the capability register file and the range of allocated memory until next_free.

We stress that the syntactic representation of a separate code memory is a matter of convenience for the purposes of stating and proving the theorem, and that it is not necessary for ensuring NWC. And we may still use m in the context of a related Control Flow Graph instead of using the above definition of M_c.

Similar to [21], we use the notation $\rightarrow_n$ to denote normal execution steps. As in [21], we deliberately choose to model the normal execution step as an incomplete relation between machine states. In particular, we model it as execution within just one program. And we assume that whenever the instruction ccall is executed, then arbitrary manipulation of the data memory and the register files is possible, except for the reserved registers. In CHERI, one realistic way of assuming a register is reserved is to use what are called system registers [11]. We, hence, define this normal execution step as any instruction except the ccall instruction. This matches the expectation of the definition of this relation that is given in [21]. So the rules for $\rightarrow_n$ are the same as the rules for the CHERI instruction transition relation $\xrightarrow{i}$ with the exclusion of the rule (ccall).

Also similar to [21], we also use the notation $\rightarrow_a$ to denote a memory attacker step. This is defined as an arbitrary transition where data memory can be changed arbitrarily. Protection against such an attack is perfectly possible in CHERI, but we still assume that such an attack is a threat because we need to model the same protection mechanism of CFI [19, 21] under the same attacker model. One subtle simplification that we also make similar to [21] is that we assume that attacker steps do not change the pc value. If we think of the CHERI machine state as a model of the program state, which we indeed realize by defining only the "normal" program execution steps $\rightarrow_n$ which exclude the $\textsf{ccall}$ instruction (which is used to switch context), then modeling an attacker step that does not change the value of $\textsf{pc}$ corresponds to thinking of the program code as a coherent flow of code (eliminating the $\textsf{ccall}$ instructions). This is a simplifying assumption that we make in order to be able to adapt/model the possibility of the subtle memory attack that is assumed as a potential threat against enforcement of the policy of CFI [19, 21]. Hence, we define an attack step $\rightarrow_a \subseteq \mathit{MachineState} \times \mathit{MachineState}$ as $\rightarrow_a \overset{\mathit{def}}{=} \{(S, S') \mid S.\textsf{pc} = S'.\textsf{pc} \wedge S.M_c = S'.M_c \wedge S.r(\texttt{r}_{\texttt{reserved}}) = S'.r'(\texttt{r}_{\texttt{reserved}})\}$.

Following the treatment in [21, Appendix A.1], we start by stating a proposition about the sequence of instrumentation instructions that precede a $\textsf{cjr}$ instruction.

Proposition 4.1. *[21, Proposition 3] (Jump destination is valid if checking instructions pass) Let S_0, S_1, S_2, S_3, S_4 be CHERI machine states with memory m, where NWC is assumed about the program executing. (See Section 4.1 for illustration that this assumption is reasonable to make about a CHERI machine state and a program without the need for syntactic separation between the code memory and data memory constructs as in the formal model of [21].)*

And let:

$$\begin{aligned}
m(S_0.\textsf{pc}) &= \langle \textsf{0},\ \textsf{movei}\ \texttt{rc IMM}\rangle \\
m(S_1.\textsf{pc}) &= \langle \textsf{0},\ \textsf{cload}\ \texttt{ri rt cc}\rangle \\
m(S_2.\textsf{pc}) &= \langle \textsf{0},\ \textsf{sub}\ \texttt{rz ri rc}\rangle \\
m(S_3.\textsf{pc}) &= \langle \textsf{0},\ \textsf{cjrnotzero}\ \texttt{cc}'\ \texttt{cz HALT}\rangle \\
m(S_4.\textsf{pc}) &= \langle \textsf{0},\ \textbf{cjr}\ \texttt{rt cc}\rangle
\end{aligned}$$

and

$$S_0 \rightarrow_n \rightarrow_a^* S_1 \rightarrow_n \rightarrow_a^* S_2 \rightarrow_n \rightarrow_a^* S_3 \rightarrow_n \rightarrow_a^* S_4$$

where $\rightarrow_a^*$ is the reflexive transitive closure of $\rightarrow_a$.

Assume that $S_4.r(\texttt{rt})$ is a valid jump destination with respect to the capability $S_4.cr(\texttt{cc})$, i.e., assume that $S_4.cr \vdash \textsf{permits\_execute}(\texttt{cc})$, and let $w = S_4.r(\texttt{rt})$. Then we have that $m(w) = \texttt{IMM}$, and if $I(m, G)$, then $m(w) = \langle \textsf{0}, \textsf{movei}\ \texttt{r}_{\texttt{reserved}}\ \texttt{dst}(\texttt{m}, \texttt{G}, \texttt{S}_4.\textsf{pc})\rangle$.

This proposition states what it means for the mentioned sequence of instructions to execute sequentially. It infers that such execution, when guarded by the assumption that $I(m, G)$ must mean that the jump destination is as expected by the control flow graph.

Theorem 4.2. *[21, Theorem 1] (Every execution step is either a memory attack or a transition to an expected successor instruction)*

Let S_0 be a state $\langle m, r, cr, \mathsf{pc}, \mathsf{pcc}, \mathsf{next\_free}\rangle$ such that $\mathsf{pc} = 0$ and G is a CFG of m, and both m and G satisfy the requirements listed in Sections 4.2 and 4.3 (on CFG construction and machine code rewriting); we denote such state of satisfaction by the predicate $I(M_c, G)$, and let $S_1, ..., S_n$ be CHERI machine states such that $S_0 \to S_1 \to ... \to S_n$. Then, for all $i \in 0...(n-1)$, either $S_i \to_a S_{i+1}$ (the step is a memory attack) and $S_{i+1}.pc = S_i.pc$, or $S_{i+1}.pc \in \mathsf{succ}(S_i.pc)$ for succ defined according to G and M_c.

The proof is by induction on executions. But we state the following lemma, and then Theorem 4.2 is a straightforward corollary.

Using proposition 4.1, the following lemma can be proved:

Lemma 4.3. *[21, Lemma 4] Let S_0 be a state $\langle m, r, cr, \mathsf{pc}, \mathsf{pcc}, \mathsf{next\_free}\rangle$ such that $I(m, G)$ where G is a control-flow graph of m, and let $S_1, ..., S_n$ be states such that $S_0 \to S_1 \to ... \to S_n$, with $n \geq 0$. Then:*

1. *$S_n.M_c = S_0.M_c$;*
2. *$S_n.\mathsf{pc} \in dom(S_0.M_c)$;*

   and if $n > 0$,
3. *either $S_{n-1} \to_a S_n$ and $S_n.\mathsf{pc} = S_{n-1}.\mathsf{pc}$, or $S_n.\mathsf{pc} \in succ(S_0.m, G, S_{n-1}.\mathsf{pc})$;*
4. *if there exists $k \in \{0..3\}$ such that $m(S_n.\mathsf{pc} + k)$ holds a cjr instruction, then $S_{n-1}.\mathsf{pc} = S_n.\mathsf{pc}$ or $S_{n-1}.\mathsf{pc} + 1 = S_n.\mathsf{pc}$.*

This lemma focuses on the execution step $S_{n-1} \to S_n$ rather than on an arbitrary execution step $S_i \to S_{i+1}$.

Conjunct 1 means that code memory does not change in the course of execution, which immediately implies that $I(S_n.M_c, G)$.

Conjunct 2 means that execution does not leave code memory.

Conjunct 4 forbids jumps past or into the middle of the checking sequences of instructions that precede cjr and cjrcond instructions.

Similar to what is claimed in [21], conjuncts 1 and 2 always follow immediately from the definition of the operational semantics, and from the assumption that we start with compartmentalized memory.

So we write M_c instead of $S_i.M_c$ for $i \in \{0..n\}$, and use that $I(M_c, G)$ and that $S_n.\mathsf{pc} \in dom(M_c)$.

Theorem 4.2 is a corollary of Lemma 4.3 because the hypotheses of the lemma are those of the theorem, and so is conjunct 3 of the conclusion of the lemma except that the lemma focuses on the last step of execution $S_{n-1} \rightarrow S_n$ for any n states. But this obviously is generalizable to an arbitrary execution step $S_i \rightarrow S_{i+1}$ by applying the lemma to all prefixes of a given sequence of states.

So now we just show the proof of Lemma 4.3.

Proof. Following the proof in [21, Appendix A], we establish conjuncts 3 and 4 by strong induction[2] on n. For $n = 0$, all conjuncts are either trivially or vacuously true. For $n > 0$, we consider the two cases of an attack step or a normal step. For the case of an attack step, we know that code memory and program counter are not changed by definition, so conjuncts 3 and 4 trivially follow. For the case of a normal step, we argue by cases on the instruction $S_{n-1}.\mathsf{pc}$.

- For cjzero, cjnotzero, cbts instructions, we have that $S_n.\mathsf{pc} \in \{S_{n-1}.\mathsf{pc}+1, S_{n-1}.\mathsf{pc}+\mathtt{OFFS}+1\}$ (by the operational semantics), and $S_n.\mathsf{pc} \in dom(M_c)$, and therefore $S_n.\mathsf{pc} \in \mathsf{succ}(M_c, G, S_{n-1}.\mathsf{pc})$ (by the requirements on the CFGs), so conjunct 3 holds.

  If $S_n.\mathsf{pc} = S_{n-1}.\mathsf{pc} + 1$, then conjunct 4 holds immediately. On the other hand, for the case when $S_n.\mathsf{pc} = S_{n-1}.\mathsf{pc} + \mathtt{OFFS} + 1$, we have that $m(S_{n-1}.\mathsf{pc} + \mathtt{OFFS} + k + 1)$ cannot be a cjr instruction for $k \in \{0..3\}$, by the definition of the predicate I in the assumption of the lemma, so conjunct 4 holds vacuously.

- For cjr `rt` `cc`, we observe that it must be preceded by the sequence of instructions

  ```
  movei rc IMM
  cload ri rt cc
  sub rz ri rc
  cjrnotzero cc′ cz HALT
  ```

  according to the definition of the predicate I. With the exception of movei `rc` `IMM`, none of these instructions could be at memory address 0 (i.e., none could be the starting point of the execution). Similarly the cjr instruction in question cannot be the starting point for the execution either.

[2]Proof of the equivalence of strong and regular induction can be found here: http://www.oxfordmathcenter.com/drupal7/node/485

Moreover by the induction hypothesis (conjunct 4), the execution could not have jumped past or into the middle of these instructions (here we make use of strong induction because the induction hypothesis on $n-1$ only would not suffice).

From the operational semantics and conjuncts 1 and 2, we have that $S_n.\mathsf{pc} \in dom(M_c)$ and $I(M_c, G)$. Therefore Proposition 4.1 applies, and yields $m(S_{n-1}.r(\mathtt{rt})) = \langle 0, \mathsf{movei}\ \mathtt{r_{reserved}}\ \mathtt{dst}(\mathtt{m}, \mathtt{G}, \mathtt{S_{n-1}}.\mathsf{pc})\rangle$. From the operational semantics we also have that $S_n.\mathsf{pc} = S_{n-1}.r(\mathtt{rt})$, so $m(S_n.\mathsf{pc}) = \langle 0, \mathsf{movei}\ \mathtt{r_{reserved}}\ \mathtt{dst}(\mathtt{m}, \mathtt{G}, \mathtt{S_{n-1}}.\mathsf{pc})\rangle$, which implies that $S_n.\mathsf{pc}$ is a destination and that $\mathtt{dst}(\mathtt{m}, \mathtt{G}, \mathtt{S_{n-1}}.\mathsf{pc})$ is its ID. Since the instruction at $S_{n-1}.\mathsf{pc}$ is a cjr instruction, $\mathtt{dst}(\mathtt{m}, \mathtt{G}, \mathtt{S_{n-1}}.\mathsf{pc})$ is the ID of the elements of $\mathsf{succ}(M_c, G, S_{n-1}.\mathsf{pc})$ (by $I(M_c, G)$). Hence, we conclude that $S_n.\mathsf{pc} \in \mathsf{succ}(M_c, G, S_{n-1}.\mathsf{pc})$, and so conjunct 3 holds.

But since $S_n.\mathsf{pc}$ contains a $\mathsf{movei}\ \mathtt{r_{reserved}}$ instruction with $\mathtt{r_{reserved}}$ not used anywhere else in the code, so in particular this destination cannot be one of the checking instructions that are prohibited by conjunct 4, so conjunct 4 holds vacuously.

- For all the other cases binop, movei, $\mathsf{cincbase}$, cload, $\mathsf{cstorecap}$, etc.., we have that $S_n.\mathsf{pc} = S_{n-1}.\mathsf{pc} + 1$ by the operational semantics, so conjunct 4 holds. As $S_n.\mathsf{pc} \in dom(M_c)$, we have that $\mathsf{succ}(S_{n-1}.\mathsf{pc}) = S_n.\mathsf{pc}$ (by the requirements on the CFGs), so conjunct 3 holds as well.

$\square$

Chapter 5

Security Goals using CHERI

Example Micro-policies applications shown on CHERI

In [39, Chapter 6], three general uses of the CHERI capability mechanism are demonstrated to argue for the utility of the CHERI architecture. In this section, we illustrate more applications, and we elicit some relevant security examples from ones that were shown to be expressible using micropolicies [9]. In particular, we focus on formally showing that a basic notion of memory compartmentalization [25, 26, 27, 9] is achievable using CHERI.

Micropolicies [9] provide a generic framework for verifying arbitrary policies enforceable by the PUMP architecture.

In this section, we show how various security goals that are enforceable using micropolicies can also be simulated on the CHERI architecture.

- Control Flow Integrity [9, Section 6] is one of the micropolicies that are enforceable by even more primitive mechanisms on general architectures (See section 4.3).
- Dynamic Sealing [9, Section 4] is another example of a security primitive that can be easily simulated on CHERI. According to the requirements stated in [48], the semantics of the ccall instruction together with csealcode and csealdata constitute a dynamic sealing mechanism. Capabilities constitute an additional layer of indirection to the memory object that is being sealed.
- Compartmentalization [9, Section 5] is based on software fault isolation (SFI) [49]. A compartment can be seen as analogous to the "security domain" defined by the transitive closure of legal memory accesses allowed by the current state of

the capability register file (where a legal memory access is one that is allowed by the ownership of a valid, unsealed capability). (In [9, Section 5], a compartment is defined as an address space, legal Jump addresses outside the address space, and legal load/store targets outside the address space. And the guarantee is that no memory access should disrespect the address space $\cup$ the load/store targets, and analogously for jumps.) See section 5.2 for an illustrative example of how compartments can be modeled using capabilities.

- Temporal memory safety [9, Section 7] can be supported by capabilities, if we treat capabilities as pointers. Support for ephemeral capabilities may be helpful for ensuring a limited temporal scope on the validity of a capability [25]. We present only an informal and unproved claim of how ephemeral capabilities can support temporal memory safety in Section 6.1.

5.1 Dynamic Sealing using CHERI

Dynamic sealing is readily implemented by the CHERI ISA, with the restriction that only valid capabilities can be sealed.

This can be seen as a general sealing mechanism of any memory location if only ephemeral capabilities are used. In other words, if a memory location is accessible using only an ephemeral capability, then sealing this capability is effectively a sealing of the memory location (with the capability being an extra level of indirection).

But regardless of the desired domain of the sealing function, correctness of a sealing mechanism is defined by two requirements:

1. Sealed values are not usable by any operation except by the operation that unseals them.
2. The unseal operation correctly retrieves all and only values that were sealed using correct usage of the seal operation.

Note that the cseal instruction for sealing capabilities, and the cunseal/ccall instructions for unsealing capabilities constitute a correct sealing mechanism.

Non-usability by every instruction except cunseal/ccall/move can be shown easily by case distinction over all instructions and showing that no machine state can evaluate if any of the operands of the current instruction is a register or memory location that contains a sealed capability.

Correctness of cunseal and ccall then follows directly from non-usability of sealed capabilities.

5.2 Compartmentalization using CHERI

A basic assumption to start with is that there is a secure call/return mechanism. In particular, there is a way that the operating system or the kernel can save registers in memory and return them back as part of a non-interruptible routine. And we assume that such a routine is trusted. The details of such a call routine are provided partly by the semantics of the ccall instruction. But saving the registers on a trusted stack is not part of the described semantics in our formal model. Such details are available in [11], as well as the details of the return routine.

The notion of compartments can be thought of as a more abstract analogue of the notion of classes in an object-oriented programming language.

Formally, a compartment can be defined as [9, Section 5]:

- an instruction memory, and data memory regions that are mutually exclusive of one another (and hence, NXD and NWC [4.1] follow) and that are not accessible by any other compartment,
- a set of legal jump addresses, which are legal entry points [50] into other compartments' instruction memories,
- and a set of valid load/store addresses, which are additional legal data memory regions (possibly part of other compartments' data memories)

Consider the following Java-like code snippet:

```
class A {
  private fieldA1, ..., fieldAn;

  public methodA1(B obj) { foo(obj.fieldB1); }
  private methodA2() { bar(); }
}

class B {
  public fieldB1;
  private fieldB2, ..., fieldBm;

  public methodB1() {
    a_in_B1 = new A();
    a_in_B1.methodA1(this);
  }
  private methodB2() { baz(); }
}
...
```

```
class Main {
  main () {
    bMain = new B();
    bMain.methodB1();
  }
}
```

Encapsulation in the object-oriented programming sense is a realization of the compartmentalization principles (requirements) described above (Please see the note in Section 2.3 for a discussion of the usage of the term “encapsulation”.).

A compartment can be seen as a class definition together with data memory containing all of the instances of this class.

The legal jump addresses outside a compartment consist in the public methods available from other compartments, and the store addresses that are available to a compartment consist in the public fields of objects that are instantiated within the scope of the compartment in question.

So, for the code snippet shown above, we can show at least three compartments:

- Compartment ⟨ `Main` ⟩ consists of:
  - Instruction memory: Capability(`main(), EXECUTE`)
  - Data memory: empty
  - External Data memory: Capability(`bMain.fieldB1, READ/WRITE`)
  - External Instruction memory: Capability(`methodB1(), etc.., EXECUTE`)
- Compartment ⟨ `A` ⟩ consists of:
  - Instruction memory: Capability(`methodA1(), methodA2(), EXECUTE`)
  - Data memory: Capability(`a_in_B1.fieldA1, ..., a_in_B1.fieldAn, READ/WRITE`)
  - External Data memory: Capability(`bMain.fieldB1, READ/WRITE`)
  - External Instruction memory: Capability(`methodB1(), foo(), bar(), EXECUTE`)
- Compartment ⟨ `B` ⟩ consists of:
  - Instruction memory: Capability(`methodB1(), etc.., EXECUTE`)
  - Data memory: Capability(`bMain.fieldB1, ..., bMain.fieldBn, READ/WRITE`)
  - External Data memory: empty
  - External Instruction memory: Capability(`methodA1(), baz(), EXECUTE`)

Assuming that we have a correct and trusted call/return mechanism, we describe the CHERI "security domain" that corresponds to the call sequence (ignoring constructor calls) starting with the "`main()`" method shown in the code snippet above, thus illustrating informally how CHERI capabilities can be used by a compilation procedure to implement compartmentalization (we indicate the new compartments entered at each method call):

1. `main():` Compartment ⟨ `Main` ⟩
   (a) `B():` Compartment ⟨ `B` ⟩
   (b) `methodB1():` Compartment ⟨ `B` ⟩
      i. `A():` Compartment ⟨ `A` ⟩
      ii. `methodA1():` Compartment ⟨ `A` ⟩
         A. `foo():` Compartment ⟨ `compartment containing foo` ⟩

The compiler is responsible for correctly using the call/return mechanism in order to switch between compartments, i.e., to make sure that the security domain at each call consists of all and only the capabilities available for the corresponding compartment indicated, and to return to exactly the parent compartment on each method return. So, the correct sequence of compartment switches should look like: `Main, B, Main*, B, A, B*, A, foo, A*, B*, Main*`, where the (`*`) denotes a compartment switch due to return.

Compartmentalized behavior of CHERI execution

We choose to reason about a definition of compartmentalization that is different from the one in [9, Section 5] in three ways:

1. We have separate load and store targets.
2. Dynamically adding a new jump target or a new load or store target is disallowed.
3. Creating subcompartments dynamically is disallowed.

   The last two points can be described as having *the compartments fixed statically* (except that if there is no trusted stack for compartments switching, a compartment may still choose to reduce its own address space, either by design or accidentally).

Organizing memory into compartments

When reasoning about compartments of a program, there are at least three different assumptions/decisions that could have been made:

1. Either assume that compartments are fixed at the beginning of a program. (i.e., a trusted procedure assigns the compartments just once with fixed address spaces, then we consider an initial CHERI state from which subsequent execution will be evaluated against that predetermined compartments set),
2. Or assume that compartments exist from the beginning but are possibly not fixed. They can be expanded (e.g., the target jump addresses J or store addresses S) by passing of capabilities between each other. However jumps to outside the address space should lead to the switching of the current compartment. So the only way code capabilities (i.e., execute capabilities) should be passed is that they are sealed capabilities, so that they are only usable by the ccall instruction
3. Or maybe also have a form (though restricted) of creating new subcompartments by enabling a compartment to give away part of its own address space.

We choose to go with the first assumption. And will then need to reason about transitions that do not use the allocate instruction.

ccall can be defined in such a way to be responsible for register saving. It can do the register saving on the data memory of the current active compartment. This procedure for register saving is necessary for preventing leakage of capabilities upon switching from a compartment to another (or in more general CHERI terminology, from security domain to another). However, for the purposes of achieving compartmentalization as described here, it is not necessary that ccall perform this procedure. A trusted stack is also not necessary for compartmentalization. It is interesting for other stronger guarantees though. **It would be easiest to define a semantics for ccall that deletes all permissions from all the capability register file except that it uses `cc` and `cd` as follows:**

$$\frac{\begin{array}{c} m(\mathsf{pc}) = \langle 0, \mathsf{ccall}\ \mathtt{cc}\ \mathtt{cd}\ \mathtt{cdd}\rangle \\ cr \vdash \mathsf{callable}(\mathtt{cc}, \mathtt{cd}) \qquad cc' = \mathsf{unsealed}(cr(\mathtt{cc})) \\ cd' = \mathsf{unsealed}(cr(\mathtt{cd})) \qquad \mathsf{pcc}' = cc' \qquad \boldsymbol{cr' = \{\mathtt{cdd} \mapsto \mathsf{cd}'\}} \\ \mathsf{pc}' = \mathsf{compute\_call\_address}(cc') \qquad \mathsf{pcc} \vdash \mathsf{executable}(\mathsf{pc}) \end{array}}{\langle m, r, cr, \mathsf{pc},\ \mathsf{pcc},\ \mathsf{next\_free}\rangle \rightarrow \langle m, r, cr', \mathsf{pc}',\ \mathsf{pcc}',\ \mathsf{next\_free}\rangle}\ (\mathsf{ccall})$$

It is obvious that a simple technique for achieving the precondition $\boldsymbol{cr'} = \{\texttt{cdd} \mapsto \texttt{cd'}\}$, which basically requires deleting all the capability registers in the cr' map, is to have a constant predefined value (say 0) with the tag bit of each register (except cdd) cleared so that the register is not usable as a valid capability. We use the notation of clearing the map for brevity.

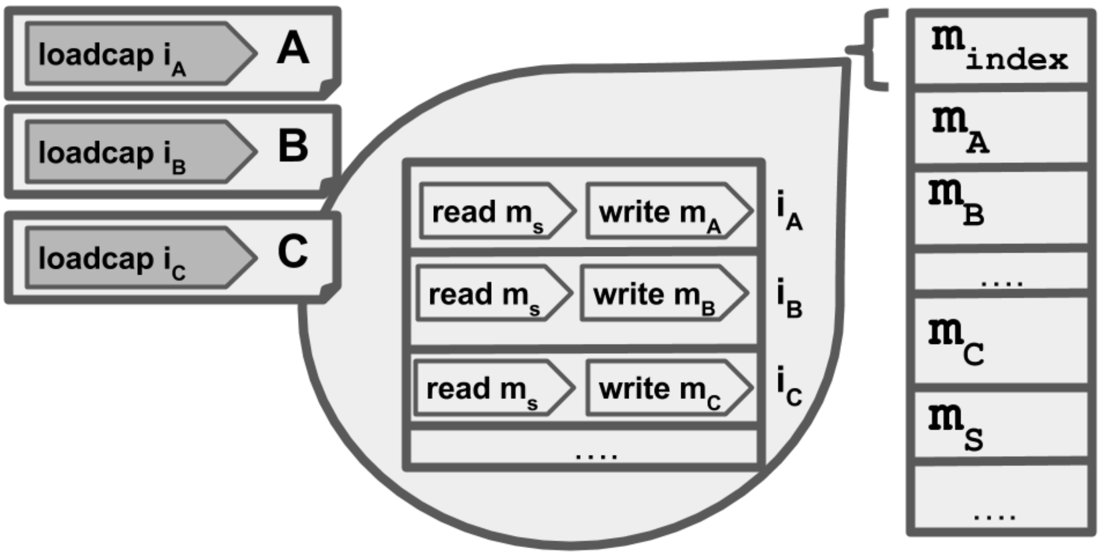
This way, we defer the responsibility of preserving ownership of capabilities during the operation of compartment switching to each compartment's implementation (realistically to the compiler). So, in effect a compartment can choose to reduce its own address space by giving up capabilities by not properly saving them on data memory.

It is worth noting that the restriction on having just two registers (the sealed code and the sealed data capability registers) does not deny us expressivity in terms of how a callee can specify to each of its callers ownership of multiple specific data capabilities in its own address space. As an example, assume there is some API service s which has clients $c_1, ..., c_n$. Assume that for s to correctly operate (when called), it would need to get access to a shared data segment of its address space, say the region m_s. But also, in order that it really offers meaningful service to any client c_i, it would need access to the respective regions m_i. Assume that no client c_i is supposed to trick the service s by claiming that during the call to s initiated by c_i, s has access to m_j for $i \neq j$. Then the obvious solution of s giving each client a sealed data capability on s's whole address space will not serve the purpose. Also, each client can only be given just one sealed data capability, because of the restricted form of the ccall semantics. So, a solution to this is [1] for s to have its memory organized into $m_s, m_1, ..., m_n, m_{index}$ where m_{index} will be a region of memory dedicated to holding capabilities. In this region, s can group together – in successive chunks – each client's data capabilities. Then s would give each client c_i a sealed data capability cd_i on its respective chunk. cd_i will have the permit_load_capability permission set. Figure 5.1 illustrates how the index region of a callee compartment can achieve the described organization of memory among three possible API callers A, B and C.

However, a compartment will not be allowed to give away capabilities to other compartments, or to create a subcompartment. That latter restriction (guarantee) will not follow from the ccall semantics alone. More restrictions have to be enforced on what capabilities a compartment can own. Namely, if sharing capabilities is to be prohibited, then a compartment should not own any capability that gives the permission of loading capabilities from other compartments.

[1] applying what is humorously called the "Fundamental Theorem of Software Engineering", which states that any problem is solvable by introducing an extra level of indirection (except for the problem of too many levels of indirection)

FIGURE 5.1: On the right, the address space of a service (i.e., a callee compartment) is shown. The index region of the memory is highlighted. On the left, each of compartments A, B and C gets a sealed permission to load capabilities from the corresponding segment of the index region of the service. Upon using the ccall instruction to invoke the service, this sealed capability on the segment of the index should be presented as the data capability.



Now, we use some definitions that will help towards stating a theorem about memory compartmentalization.

Definition 5.1. (Compartment)

A 5-tuple of sets of addresses, $c = (Code, Data, J, L, S) \in 2^{Addr} \times 2^{Addr} \times 2^{Addr} \times 2^{Addr} \times 2^{Addr}$ is called a compartment iff $(c.J \cup c.Code) \cap (c.S \cup c.Data) = \emptyset$. We refer to $c.Code \cup c.Data$ as the address space of c. We refer to $c.J$ as the set of legal jump targets with $c.J \cap c.Code = \emptyset$, and $c.L/c.S$ as the set of legal load/store targets.

Definition 5.2. (Disjoint compartments)

Two compartments c_i, c_j are said to be disjoint, written $c_i \cap c_j = \emptyset$ iff $(c_i.Code \cup c_i.Data) \cap (c_j.Code \cup c_j.Data) = \emptyset$.

Definition 5.3. (A valid set of compartments)

A set $C \subset 2^{Addr} \times 2^{Addr} \times 2^{Addr} \times 2^{Addr} \times 2^{Addr}$ is a valid set of compartments (denoted $\mathsf{valid}(C)$) iff every $c \in C$ is a compartment and $(\bigcup_{c_i \in C} c_i.J \cup c_i.Code) \cap (\bigcup_{c_i \in C} c_i.S \cup c_i.Data) = \emptyset$ and $\forall c_i, c_j \in C.\ i \neq j \Rightarrow c_i \cap c_j = \emptyset$ and $\bigcup_{c \in C} (c.L \cup c.S) \subseteq \bigcup_{c \in C} c.Data$ and $\bigcup_{c \in C} c.J \subseteq \bigcup_{c \in C} c.Code$.

Note that we choose to enforce Non-Executable-Data (NXD) and Non-Writable-Code (NWC) on the memory regions that constitute compartments, e.g., by the condition $(\bigcup\limits_{c_i \in C} c_i.J \cup c_i.\mathit{Code}) \cap (\bigcup\limits_{c_i \in C} c_i.S \cup c_i.\mathit{Data}) = \emptyset$ on a compartment set, and that we allow code of one compartment to be loadable by other compartments by not requiring any mutual exclusion between the code section of an address space of a compartment and the load destinations of other compartments.

Definition 5.4. (Capability register file and memory *more restrictive than* a compartment, and a valid compartment set)

A tuple of pcc, capability register file, and memory $\langle \mathsf{pcc}, cr, m\rangle$ is said to be more restrictive than a compartment c^* (called the active compartment) and a valid compartment set C (where $\mathsf{valid}(C)$ holds), written $\langle \mathsf{pcc}, cr, m\rangle \preceq \langle c^*, C\rangle$ **iff**:

1. (Every valid, unsealed, execute capability is a capability on addresses that all belong to the code region of just one compartment)

$$\begin{aligned} \forall a \in \mathit{Addr}.\ m(a) = \langle 1, (\mathtt{false}, \mathsf{bin}(\mathtt{perms}), \mathtt{st}, \mathtt{len}, \mathtt{o})\rangle \\ \wedge\ \mathsf{permit\_execute} \in \mathtt{perms} \\ \Rightarrow \exists! c \in C.\ (a \in c.\mathit{Data} \wedge \forall a' \in [\mathtt{st}, \mathtt{st}+\mathtt{len}).\ a' \in c.\mathit{Code}) \end{aligned}$$

and

2. (Every valid, sealed, execute capability is a capability on addresses that are all external legal jump destinations of or are in the code region of just one compartment.)

$$\begin{aligned} \forall a \in \mathit{Addr}.\ m(a) = \langle 1, (\mathtt{true}, \mathsf{bin}(\mathtt{perms}), \mathtt{st}, \mathtt{len}, \mathtt{o})\rangle \\ \wedge\ \mathsf{permit\_execute} \in \mathtt{perms} \\ \Rightarrow \exists! c \in C.\ (a \in c.\mathit{Data} \wedge \\ ((\forall a' \in [\mathtt{st}, \mathtt{st}+\mathtt{len}).\ a' \in c.\mathit{Code} \wedge o \in c.\mathit{Code} \cup c.\mathit{Data}) \\ \vee \exists c_j \in C. \forall a' \in [\mathtt{st}, \mathtt{st}+\mathtt{len}).\ a' \in c.J \wedge a' \in c_j.\mathit{Code} \wedge o \in c_j.\mathit{Code} \cup c_j.\mathit{Data})) \end{aligned}$$

and

3. (If a compartment owns a sealing key, then it is their own key. In other words, every valid sealing capability in memory has the property that its base address

belongs to the compartment in which it lives.)

$$\forall a \in \mathit{Addr}.\ m(a) = \langle 1, (\_, \mathsf{bin}(\mathtt{perms}), \mathtt{st}, \mathtt{len}, \mathtt{o})\rangle$$
$$\wedge\ \mathsf{permit\_seal} \in \mathtt{perms}$$
$$\Rightarrow \exists! c \in C.\ (a \in c.\mathit{Data} \wedge \mathit{st} \in c.\mathit{Data} \cup c.\mathit{Code})$$

and

4. (If a sealed capability exists in memory, then it is a capability on a region of the address space of the sealer compartment.)

$$\forall a \in \mathit{Addr}.\ m(a) = \langle 1, (\mathtt{true}, \mathsf{bin}(\mathtt{perms}), \mathtt{st}, \mathtt{len}, \mathtt{o})\rangle$$
$$\wedge\ \{\mathsf{permit\_execute},\ \mathsf{permit\_load},\ \mathsf{permit\_store},$$
$$\mathsf{permit\_load\_capability},\ \mathsf{permit\_store\_capability},\ \mathsf{permit\_seal}\} \cap \mathtt{perms} \neq \emptyset$$
$$\Rightarrow \exists! c \in C.\ ([\mathtt{st}, \mathtt{st} + \mathtt{len}) \subseteq c.\mathit{Code} \cup c.\mathit{Data} \wedge o \in c.\mathit{Code} \cup c.\mathit{Data})$$

and

5. (Every valid, store capability is a capability on addresses that all belong to the data region of just one compartment or to legal store addresses of the same compartment)

$$\forall a \in \mathit{Addr}.\ m(a) = \langle 1, (\mathsf{s}, \mathsf{bin}(\mathtt{perms}), \mathtt{st}, \mathtt{len}, \mathtt{o})\rangle$$
$$\wedge\ \mathsf{permit\_store} \in \mathtt{perms}$$
$$\Rightarrow \exists! c \in C.\ (a \in c.\mathit{Data} \wedge \forall a' \in [\mathtt{st}, \mathtt{st} + \mathtt{len}).\ a' \in c.\mathit{Data} \vee (a' \in c.S \wedge \mathtt{s} = \mathtt{true}))$$

and

6. (Every valid, load capability is a capability on addresses that all belong to the address space (code or data) of just one compartment or to legal load addresses of the same compartment)

$$\forall a \in \mathit{Addr}.\ m(a) = \langle 1, (\mathsf{s}, \mathsf{bin}(\mathtt{perms}), \mathtt{st}, \mathtt{len}, \mathtt{o})\rangle$$
$$\wedge\ \mathsf{permit\_load} \in \mathtt{perms}$$
$$\Rightarrow \exists! c \in C.\ (a \in c.\mathit{Data} \wedge \forall a' \in [\mathtt{st}, \mathtt{st} + \mathtt{len}).\ a' \in c.\mathit{Data} \vee$$
$$(a' \in c.L \wedge \mathtt{s} = \mathtt{true}) \vee a' \in c.\mathit{Code})$$

and

7. (Every valid capability that permits storing capabilities is a capability on addresses that all belong to the data region of just one compartment. Note that this restriction is necessary together with restricting imports, because if we don't talk about the permission to store capabilities at all, it means that a compartment is allowed to leak its capability to arbitrary regions of other compartments, which could be regions that the other compartment is allowed to load capabilities from.

   This requirement can be seen as an integrity guarantee for the other compartments rather than a confidentiality guarantee that prevents leakage of capabilities. It disallows a malicious compartment from being able to overwrite other compartments' memory.)

$$\begin{gathered}\forall a \in \mathit{Addr}.\ m(a) = \langle \mathtt{1}, (\_, \mathsf{bin}(\mathtt{perms}), \mathtt{st}, \mathtt{len}, \mathtt{o})\rangle \\ \land\, \mathsf{permit\_store\_capability} \in \mathtt{perms} \\ \Rightarrow \exists! c \in C.\ (a \in c.\mathit{Data} \land \forall a' \in [\mathtt{st}, \mathtt{st} + \mathtt{len}).\ a' \in c.\mathit{Data}) \\ \textbf{and}\end{gathered}$$

8. (Importing capabilities is restricted. In other words, loading capabilities is allowed from only a locally-restricted part of the address space. Note that this requirement is sufficient to prohibit a leakage resulting from successive copies that potentially start with an importing store operation because no memory-to-memory copy instruction is available.)

$$\begin{gathered}\forall a \in \mathit{Addr}.\ m(a) = \langle \mathtt{1}, (\_, \mathsf{bin}(\mathtt{perms}, \mathtt{st}, \mathtt{len}, \mathtt{o})\rangle \\ \land\, \mathsf{permit\_load\_capability} \in \mathtt{perms} \\ \Rightarrow \exists! c \in C.\ (a \in c.\mathit{Data} \land \forall a' \in [\mathtt{st}, \mathtt{st} + \mathtt{len}).\ a' \in c.\mathit{Data}) \\ \textbf{and}\end{gathered}$$

9. (The capability register file and pcc should contain capabilities to allow execute on at most the active compartment (i.e., and on no other compartment).)

$$\begin{gathered}\forall \mathit{cap} \in \{\mathsf{pcc}\} \cup \mathsf{Range}(\mathit{cr}).\ \mathit{cap} = \langle \mathtt{1}, (\mathtt{false}, \mathsf{bin}(\mathtt{perms}), \mathtt{st}, \mathtt{len}, \mathtt{o})\rangle \\ \land\, \mathsf{permit\_execute} \in \mathtt{perms} \\ \Rightarrow \forall a' \in [\mathtt{st}, \mathtt{st} + \mathtt{len}).\ a' \in c^{*}.\mathit{Code}\end{gathered}$$

and

10. (Every valid, sealed, execute capability in the capability register file is a capability on addresses that are all external legal jump destinations of or are in the code region of the active compartment.)

$$\begin{gathered}\forall cap \in \mathsf{Range}(cr).\ cap = \langle \mathtt{1}, (\mathtt{true}, \mathsf{bin}(\mathtt{perms}), \mathtt{st}, \mathtt{len}, \mathtt{o})\rangle \\ \wedge\ \mathsf{permit\_execute} \in \mathtt{perms} \\ \Rightarrow (\forall a' \in [\mathtt{st}, \mathtt{st}+\mathtt{len}).\, a' \in c^*.\mathit{Code} \wedge o \in c^*.\mathit{Code} \cup c^*.\mathit{Data} \\ \vee\ \exists c \in C.\, \forall a' \in [\mathtt{st}, \mathtt{st}+\mathtt{len}).\, a' \in c^*.J \wedge a' \in c.\mathit{Code} \wedge o \in c.\mathit{Code} \cup c.\mathit{Data})\end{gathered}$$

and

11. (Every compartment owns only keys on its own address space, i.e., every valid sealing capability in the capability register file has their base address belonging to the address space of the active compartment.)

$$\begin{gathered}\forall cap \in \mathsf{Range}(cr).\ cap = \langle \mathtt{1}, (\_, \mathsf{bin}(\mathtt{perms}), \mathtt{st}, \mathtt{len}, \mathtt{o})\rangle \\ \wedge\ \mathsf{permit\_seal} \in \mathtt{perms} \\ \Rightarrow st \in c^*.\mathit{Data} \cup c^*.\mathit{Code}\end{gathered}$$

and

12. (If a sealed capability exists in the capability register file, then it is a capability on a region of the address space of the sealer compartment.)

$$\begin{gathered}\forall cap \in \mathsf{Range}(cr).\ cap = \langle \mathtt{1}, (\mathtt{true}, \mathsf{bin}(\mathtt{perms}), \mathtt{st}, \mathtt{len}, \mathtt{o})\rangle \\ \wedge\ \{\mathsf{permit\_execute}, \mathsf{permit\_load}, \mathsf{permit\_store}, \\ \mathsf{permit\_load\_capability}, \mathsf{permit\_store\_capability}, \mathsf{permit\_seal}\} \cap \mathtt{perms} \neq \emptyset \\ \Rightarrow \exists! c \in C.\ ([\mathtt{st}, \mathtt{st}+\mathtt{len}) \subseteq c.\mathit{Code} \cup c.\mathit{Data} \wedge o \in c.\mathit{Code} \cup c.\mathit{Data})\end{gathered}$$

and

13. (The capability register file should contain capabilities to allow the legal stores of at most the active compartment (i.e., to its data address space or to its legal store

addresses and to no other compartment).)

$$\forall cap \in \mathsf{Range}(cr).\ cap = \langle \mathtt{1}, (\mathtt{s}, \mathsf{bin}(\mathtt{perms}), \mathtt{st}, \mathtt{len}, \mathtt{o})\rangle \wedge \mathsf{permit\_store} \in \mathtt{perms} \Rightarrow \forall a' \in [\mathtt{st}, \mathtt{st}+\mathtt{len}).\ a' \in c^{*}.Data \vee (a' \in c^{*}.S \wedge \mathtt{s} = \mathtt{true})$$

and

14. (The capability register file should contain capabilities to allow the legal loads of at most the active compartment (i.e., loads from its address space or from its legal load addresses but from no other illegal region of another compartment).)

$$\forall cap \in \mathsf{Range}(cr).\ cap = \langle \mathtt{1}, (\mathtt{s}, \mathsf{bin}(\mathtt{perms}), \mathtt{st}, \mathtt{len}, \mathtt{o})\rangle \wedge \mathsf{permit\_load} \in \mathtt{perms} \Rightarrow \forall a' \in [\mathtt{st}, \mathtt{st}+\mathtt{len}).\ a' \in c^{*}.Data \vee (a' \in c^{*}.L \wedge \mathtt{s} = \mathtt{true}) \vee a' \in c^{*}.Code$$

and

15. (**See the notes at conjunct 7**)

$$\forall cap \in \mathsf{Range}(cr).\ cap = \langle \mathtt{1}, (\_, \mathsf{bin}(\mathtt{perms}), \mathtt{st}, \mathtt{len}, \mathtt{o})\rangle \wedge \mathsf{permit\_store\_capability} \in \mathtt{perms} \Rightarrow \forall a' \in [\mathtt{st}, \mathtt{st}+\mathtt{len}).\ a' \in c^{*}.Data$$

and

16. (Importing capabilities is restricted.)

$$\forall cap \in \mathsf{Range}(cr).\ cap = \langle \mathtt{1}, (\_, \mathsf{bin}(\mathtt{perms}), \mathtt{st}, \mathtt{len}, \mathtt{o})\rangle \wedge \mathsf{permit\_load\_capability} \in \mathtt{perms} \Rightarrow \forall a' \in [\mathtt{st}, \mathtt{st}+\mathtt{len}).\ a' \in c^{*}.Data$$

Note that in definition 5.4, we ignore the necessity of the existence of permissions to load the capabilities that give a compartment access over its own address space. Thus, a memory organized into a strictly more restrictive set of compartments (e.g., a set of compartments where only a subset of an address space of some or every compartment is accessible to its respectively owning compartment) will indeed still satisfy the relation defined in 5.4.

Also note that another possible way to define a state that complies with a compartments set is to require that all the machine states reachable are legal with respect to the given compartments set, i.e., to require that the fixed point of the legal execution relation applied to the given state consists of only states that all satisfy the conditions (conjuncts) stated above. We instead show a similar notion by a theorem that states that legal execution preserves the $\preceq$ relation.

Theorem 5.5. *(A state that respects a compartments set steps into only states that respect the same compartment set)*

If C is a valid set of compartments ($\mathsf{valid}(C)$), $c^* \in C$,

$\langle \mathsf{pcc}, cr, m\rangle \preceq \langle c^*, C\rangle$,

$\mathsf{pc} \in c^*.Code$,

$\langle m, r, cr, \mathsf{pc}, \mathsf{pcc}, \mathsf{next\_free}\rangle \to^* \langle m', r', cr', \mathsf{pc}', \mathsf{pcc}', \mathsf{next\_free}\rangle$,

then

$\exists c \in C. \langle \mathsf{pcc}', cr', m'\rangle \preceq \langle c, C\rangle \wedge \mathsf{pc}' \in c.Code.$

Proof. We prove the theorem by induction on the execution steps $\xrightarrow{i}$. (Every legal transition $\to$ is a result of some instruction execution $\xrightarrow{i}$).

Base case follows directly from the assumptions. Namely, we are looking for a $c \in C$ with $\langle \mathsf{pcc}, cr, m\rangle \preceq \langle c, C\rangle \wedge \mathsf{pc} \in c.Code$. So c^* satisfies such c we are looking for.

We then consider the inductive case where we have a CHERI machine state $\langle m'', r'', cr'', \mathsf{pc}'', \mathsf{pcc}'', \mathsf{next\_free}''\rangle$ where $\exists c'' \in C. \langle \mathsf{pcc}'', cr'', m''\rangle \preceq \langle c'', C\rangle \wedge \mathsf{pc}'' \in c''.Code$, and we need to show that if

$\langle m'', r'', cr'', \mathsf{pc}'', \mathsf{pcc}'', \mathsf{next\_free}''\rangle \to \langle m', r', cr', \mathsf{pc}', \mathsf{pcc}', \mathsf{next\_free}'\rangle$ then $\exists c \in C.$ $\langle \mathsf{pcc}', cr', m'\rangle \preceq \langle c, C\rangle \wedge \mathsf{pc}' \in c.Code.$

We consider distinction between the transition on a ccall instruction and other transitions. We show that for only the former transition, $c \neq c''$ may hold. In the latter, $c = c''$ necessarily holds.

We consider all the possible rules for $\xrightarrow{i}$ through which $\langle m'', r'', cr'', \mathsf{pc}'', \mathsf{pcc}'', \mathsf{next\_free}''\rangle$ steps.

- **Case BinOp, cload, ccheckperm, cchecktype, movepc, movei, cbts-true, cbts-false, cjrcond-false**

  – From each of these operational semantics rules, we can deduce that no change happens to the program counter capability, the capability register file or memory.

  – Namely, $\mathsf{pcc}' = \mathsf{pcc}''$, $cr' = cr''$ and $m' = m''$,

  – and so we can choose $c = c''$ and have that $\langle \mathsf{pcc}', cr', m' \rangle \preceq \langle c, C \rangle$ holds directly from the induction hypothesis which states that $\langle \mathsf{pcc}'', cr'', m'' \rangle \preceq \langle c'', C \rangle$.

  – Next, we need to show that $\mathsf{pc}' \in c.\mathit{Code}$.

  – From the operational semantics, we see that $\mathsf{pcc}' \vdash \mathsf{executable}(\mathsf{pc}')$ must hold.

  – So using the definition of rule $\mathsf{executable}$ in Section 3.2.1, we see that $\mathsf{pc}' \in [\mathtt{st}, \mathtt{st} + \mathtt{len})$ must hold with $\mathsf{pcc}' = \langle \mathtt{1}, (\mathtt{false}, \mathsf{bin}(\mathtt{perms}), \mathtt{st}, \mathtt{len}, \mathtt{o}) \rangle$ and $\mathsf{permit\_execute} \in \mathtt{perms}$.

  – But we just argued that $\langle \mathsf{pcc}', cr', m' \rangle \preceq \langle c, C \rangle$ which gives us by definition 5.4 of the $\preceq$ relation that:

$$\forall cap \in \{\mathsf{pcc}'\} \cup \mathsf{Range}(cr').\ cap = \langle \mathtt{1}, (\mathtt{false}, \mathsf{bin}(\mathtt{perms}), \mathtt{st}, \mathtt{len}, \mathtt{o}) \rangle \wedge$$
$$\mathsf{permit\_execute} \in \mathtt{perms}$$
$$\Rightarrow \forall a' \in [\mathtt{st}, \mathtt{st} + \mathtt{len}).\ a' \in c.\mathit{Code}$$

  – So, in particular we can instantiate a' to pc', then we obtain $\mathsf{pc}' \in c.\mathit{Code}$.

  Case proved.

- **Case cstore**

  The informal observation for this case is that the tag bit of a stored memory word gets cleared; so any change in memory does not introduce a new valid capability.

  – We observe that $\mathsf{pcc}' = \mathsf{pcc}''$ and $cr' = cr''$.

  – So if we choose $c = c''$, then we have that conjuncts 9, 10, 11, 13, 14, 15, and 16 hold immediately from the induction hypothesis which states that $\langle \mathsf{pcc}'', cr'', m'' \rangle \preceq \langle c'', C \rangle$.

  – We next obtain the necessary precondition $m' = m''[\mathsf{content}(r''(\mathtt{rt})) \mapsto \langle \mathtt{0}, \mathtt{content\_s} \rangle]$ of the cstore rule, and we make case distinction on the possible instantiations of the memory address a which is universally quantified for conjuncts 1, 2, 5, 6, 7, and 8 as follows:

    - Case $a = \mathsf{content}(r''(\texttt{rt}))$ In this case conjuncts 1, 2, 3, 5, 6, 7, and 8 hold vacuously because of the mentioned precondition of the cstore rule which requires that $m'(a) = \langle \mathtt{0}, \_ \rangle$.
    - Case $a \neq \mathsf{content}(r''(\texttt{rt}))$ In this case, we observe that conjuncts 1, 2, 3, 5, 6, 7, and 8 follow immediately from the induction hypothesis because $m'(a) = m''(a)$.
  - So we have shown all the conjuncts of $\langle \mathsf{pcc}', cr', m' \rangle \preceq \langle c'', C \rangle$ to be true.
  - We next show that $\mathsf{pc}' \in c''.\mathit{Code}$. This holds for exactly the same reasons argued in the previous case. We avoid repetition.

  Case proved.

- **Case cmove**
  - We observe that $m' = m''$, from which it follows that conjuncts 1, 2, 3, 4, 5, 6, 7, and 8 follow immediately from the induction hypothesis.
  - We also observe that $\mathsf{pcc}' = \mathsf{pcc}''$.
  - We then obtain the necessary precondition $cr' = cr''[\texttt{cd} \mapsto cr''(\texttt{cb})]$ of the rule cmove.
  - By case distinction on the name of the register in the domain of cr' (let's denote it $\texttt{crname}$; i.e., let $\texttt{crname}$ range over the domain of cr' or cr'' depending on the context), we get the following two cases:
    - Case $\texttt{crname} = \texttt{cd}$
      Then conjuncts 9, 10, 11, 12 13, 14, 15, and 16 hold by instantiating the induction hypothesis about cr'' and pcc'' with $\texttt{crname} = \texttt{cb}$.
    - Case $\texttt{crname} \neq \texttt{cd}$
      Then conjuncts 9, 10, 11, 12 13, 14, 15, and 16 about pcc' and cr' follow immediately from the induction hypothesis about pcc'' and cr''.
  - So we have shown all the conjuncts of $\langle \mathsf{pcc}', cr', m' \rangle \preceq \langle c'', C \rangle$ to be true.
  - We next show that $\mathsf{pc}' \in c''.\mathit{Code}$. This holds for exactly the same reasons argued in the previous case. We avoid repetition.

  Case proved.

- **Case cloadcap**
  - We observe that $m' = m''$, from which it follows that conjuncts 1, 2, 3, 4, 5, 6, 7, and 8 follow immediately from the induction hypothesis.
  - We also observe that $\mathsf{pcc}' = \mathsf{pcc}''$.

  - We then obtain the necessary precondition $cr' = cr''[\texttt{cd} \mapsto m''(\mathsf{content}(r''(\texttt{rt})))]$ of the rule $\mathsf{cloadcap}$.
  - By case distinction on the name of the register in the domain of cr' (let's denote it `crname`; i.e., let `crname` range over the domain of cr' or cr'' depending on the context), we get the following two cases:
    * Case $\texttt{crname} = \texttt{cd}$
      Then conjuncts 9, 10, 11, 12 13, 14, 15, and 16 hold by instantiating the induction hypothesis about m'' with $\mathsf{a}' = \mathsf{content}(\texttt{r}''(\texttt{rt}))$.
    * Case $\texttt{crname} \neq \texttt{cd}$
      Then conjuncts 9, 10, 11, 12 13, 14, 15, and 16 about pcc' and cr' follow immediately from the induction hypothesis about pcc'' and cr''.
  - So we have shown all the conjuncts of $\langle \mathsf{pcc}', cr', m' \rangle \preceq \langle c'', C \rangle$ to be true.
  - We next show that $\mathsf{pc}' \in c''.Code$. This holds for exactly the same reasons argued in the previous case. We avoid repetition.

  Case proved.

- **Case cstorecap**
  - We observe that $\mathsf{pcc}' = \mathsf{pcc}''$ and $cr' = cr''$.
  - So if we choose $c = c''$, then we have that conjuncts 9, 10, 11, 12 13, 14, 15, and 16 hold immediately from the induction hypothesis which states that $\langle \mathsf{pcc}'', cr'', m'' \rangle \preceq \langle c'', C \rangle$.
  - We next obtain the necessary precondition
    $m' = m''[\mathsf{content}(r''(\texttt{rt})) \mapsto cr''(\texttt{cs})]$ of the $\mathsf{cstorecap}$ rule, and we make case distinction on the possible instantiations of the memory address a which is universally quantified for conjuncts 1, 2, 3, 5, 6, 7, and 8 as follows:
    * Case $a = \mathsf{content}(r''(\texttt{rt}))$ In this case conjuncts 1, 2, 3, 4, 5, 6, 7, and 8 follow from conjuncts 9, 10, 11, 12 13, 14, 15, and 16 respectively of the induction hypothesis about cr'' when instantiated with the value $cap = cr''(\texttt{cs})$.
    * Case $a \neq \mathsf{content}(r''(\texttt{rt}))$ In this case, we observe that conjuncts 1, 2, 3, 5, 6, 7, and 8 follow immediately from the induction hypothesis because $m'(a) = m''(a)$.
  - So we have shown all the conjuncts of $\langle \mathsf{pcc}', cr', m' \rangle \preceq \langle c'', C \rangle$ to be true.
  - We next show that $\mathsf{pc}' \in c''.Code$. This holds for exactly the same reasons argued in the previous case. We avoid repetition.

Case proved.

- **Case ccleartag**
  - We observe that $m' = m''$, from which it follows that conjuncts 1, 2, 3, 4, 5, 6, 7, and 8 follow immediately from the induction hypothesis.
  - We also observe that $\mathsf{pcc}' = \mathsf{pcc}''$.
  - We then obtain the necessary preconditions $cr''(\mathtt{cb}) = \langle \mathtt{tag\_b}, \mathtt{content\_b} \rangle$ and $cr' = cr''[\mathtt{cd} \mapsto \langle 0, \mathtt{content\_b} \rangle]$ of the rule $\mathsf{ccleartag}$.
  - By case distinction on the name of the register in the domain of cr' (let's denote it $\mathtt{crname}$; i.e., let $\mathtt{crname}$ range over the domain of cr' or cr'' depending on the context), we get the following two cases:
    * Case $\mathtt{crname} = \mathtt{cd}$
      Then conjuncts 9, 10, 11, 12 13, 14, 15, and 16 hold vacuously because the tag bit is 0.
    * Case $\mathtt{crname} \neq \mathtt{cd}$
      Then conjuncts 9, 10, 11, 12 13, 14, 15, and 16 about pcc' and cr' follow immediately from the induction hypothesis about pcc'' and cr''.
  - So we have shown all the conjuncts of $\langle \mathsf{pcc}', cr', m' \rangle \preceq \langle c'', C \rangle$ to be true.
  - We next show that $\mathsf{pc}' \in c''.Code$. This holds for exactly the same reasons argued in the previous case. We avoid repetition.

Case proved.

- **Case candperm**
  - We observe that $m' = m''$, from which it follows that conjuncts 1, 2, 3, 4, 5, 6, 7, and 8 follow immediately from the induction hypothesis.
  - We also observe that $\mathsf{pcc}' = \mathsf{pcc}''$.
  - We then obtain the necessary preconditions
    $cr''(cb) = \langle 1, (\mathtt{s\_b}, \mathsf{bin}(\mathtt{perms\_b}), \mathtt{addr\_b}, \mathtt{len\_b}, \mathtt{otype\_b}) \rangle$ and
    $cr' = cr''[\mathtt{cd} \mapsto \langle 1, (\mathtt{s\_b}, \mathsf{bin}(\mathtt{perms\_b}) \cap r''(\mathtt{rt}), \mathtt{addr\_b}, \mathtt{len\_b}, \mathtt{otype\_b}) \rangle]$ of the rule $\mathsf{candperm}$.
  - By case distinction on the name of the register in the domain of cr' (let's denote it $\mathtt{crname}$; i.e., let $\mathtt{crname}$ range over the domain of cr' or cr'' depending on the context), we get the following two cases:
    * Case $\mathtt{crname} = \mathtt{cd}$
      Let the expression $\mathsf{bin}(perms\_b) \cap r''(\mathtt{rt})$ be called $\mathsf{bin}(\mathtt{perms\_d})$. We observe that $p \in \mathtt{perms\_d} \Rightarrow p \in \mathtt{perms\_b}$.

Then, we observe that whenever the antecedent of any of the statements 9, 10, 11, 12 13, 14, 15, and 16 holds for $\texttt{crname} = \texttt{cd}$, then it must also hold about cr'' when instantiated with $\texttt{crname} = \texttt{cb}$. So, from the induction hypothesis, we know that the consequents hold, and that conjuncts 9, 10, 11, 12 13, 14, 15, and 16 thus hold for this case. Some of the conjuncts even hold vacuously because of the precondition of the rule candperm on the sealed bit having to be false.

  * Case $\texttt{crname} \neq \texttt{cd}$

    Then conjuncts 9, 10, 11, 12 13, 14, 15, and 16 about pcc' and cr' follow immediately from the induction hypothesis about pcc'' and cr''.

- So we have shown all the conjuncts of $\langle \mathsf{pcc}', cr', m' \rangle \preceq \langle c'', C \rangle$ to be true.
- We next show that $\mathsf{pc}' \in c''.\mathit{Code}$. This holds for exactly the same reasons argued in the previous case. We avoid repetition.

Case proved.

- **Case cincbase**

  - We observe that $m' = m''$, from which it follows that conjuncts 1, 2, 3, 4, 5, 6, 7, and 8 follow immediately from the induction hypothesis.
  - We also observe that $\mathsf{pcc}' = \mathsf{pcc}''$.
  - We then obtain the necessary preconditions $cr' = cr''[\texttt{cd} \mapsto \langle 1, (\texttt{s\_b}, \mathsf{bin}(\texttt{perms\_b}), \texttt{addr\_b} + r''(\texttt{rt}), \texttt{len\_b} - r''(\texttt{rt}), \texttt{otype\_b}) \rangle]$, and $r''(\texttt{rt}) \leq \texttt{len\_b}$ of the rule cincbase.

    From these, together with the fact that $r''(\texttt{rt})$ is non-negative which follows from the type information $\mathit{Content} = \mathbb{N}$, we conclude that $\texttt{addr\_b} \leq \texttt{addr\_d} \leq \texttt{addr\_d} + \texttt{len\_d} \leq \texttt{addr\_b} + \texttt{len\_b}$ where $cr'(\texttt{cd}) = \langle 1, (\texttt{s\_b}, \mathsf{bin}(\texttt{perms\_b}), \texttt{addr\_d}, \texttt{len\_d}, \_) \rangle$.
  - By case distinction on the name of the register in the domain of cr' (let's denote it $\texttt{crname}$; i.e., let $\texttt{crname}$ range over the domain of cr' or cr'' depending on the context), we get the following two cases:

    * Case $\texttt{crname} = \texttt{cd}$

      Then conjuncts 9, 10, 11, 12 13, 14, 15, and 16 hold by instantiating the induction hypothesis about cr'' and pcc'' with $\texttt{crname} = \texttt{cb}$, and by noticing that $a' \in [\texttt{addr\_d}, \texttt{addr\_d} + \texttt{len\_d})$ $\Rightarrow a' \in [\texttt{addr\_b}, \texttt{addr\_b} + \texttt{len\_b})$ which follows from the inequality concluded above. Some of the conjuncts even hold vacuously because of the precondition of the rule cincbase on the sealed bit having to be false

    - Case `crname` $\neq$ `cd`
      Then conjuncts 9, 10, 11, 12 13, 14, 15, and 16 about pcc' and cr' follow immediately from the induction hypothesis about pcc'' and cr''.

  - So we have shown all the conjuncts of $\langle \mathsf{pcc}', cr', m' \rangle \preceq \langle c'', C \rangle$ to be true.
  - We next show that $\mathsf{pc}' \in c''.Code$. This holds for exactly the same reasons argued in the previous case. We avoid repetition.

Case proved.

- **Case csetlen**

  - We observe that $m' = m''$, from which it follows that conjuncts 1, 2, 3, 4, 5, 6, 7, and 8 follow immediately from the induction hypothesis.
  - We also observe that $\mathsf{pcc}' = \mathsf{pcc}''$.
  - This case is similar to **Case cincbase**. The main observation is that we can conclude the same proposition $\mathtt{addr\_b} \leq \mathtt{addr\_d} \leq \mathtt{addr\_d} + \mathtt{len\_d} \leq \mathtt{addr\_b} + \mathtt{len\_b}$ from the preconditions:

    (i) $cr' = cr''[\mathtt{cd} \mapsto \langle \mathtt{1}, (\mathtt{s\_b}, \mathsf{bin}(\mathtt{perms\_b}), \mathtt{addr\_b}, r''(\mathtt{rt}), \mathtt{otype\_b})\rangle]$, and

    (ii) $r''(\mathtt{rt}) \leq \mathtt{len\_b}$
    of the rule csetlen and from the fact that:

    (iii) $r''(\mathtt{rt})$ is non-negative which follows from the type information $Content = \mathbb{N}$.

  Case can hence be proved with the rest of the argument being identical to cincbase. We avoid repetition.

- **Case cseal**

  The intuition is that this case should hold because whatever an unsealed capability is allowed to do is a subset of what the same sealed capability is allowed to do.

  - We observe that $m' = m''$, from which it follows that conjuncts 1, 2, 3, 4, 5, 6, 7, and 8 follow immediately from the induction hypothesis.
  - We also observe that $\mathsf{pcc}' = \mathsf{pcc}''$.
  - We then obtain the necessary preconditions

    (i) $cr''(\mathtt{ct}) = \langle \mathtt{1}, (\mathtt{s\_t}, \mathsf{bin}(\mathtt{perms\_t}), \mathtt{addr\_t}, \mathtt{len\_t}, \mathtt{otype\_t})\rangle$,

    (ii) $cr''(\mathtt{cs}) = \langle \mathtt{1}, (\mathtt{s\_s}, \mathsf{bin}(\mathtt{perms\_s}), \mathtt{addr\_s}, \mathtt{len\_s}, \mathtt{otype\_s})\rangle$,

    (iii) $\mathtt{otype\_d} = \mathtt{addr\_t}$

    (iv) $cr' = cr''[\mathtt{cd} \mapsto$
    $\langle 1, (\mathtt{true}, \mathsf{bin}(\mathtt{perms\_s}), \mathtt{addr\_s}, \mathtt{len\_s}, \mathtt{otype\_d})\rangle]$, and

(v) $cr'' \vdash \mathsf{permits\_seal}(\mathtt{cs}, \mathtt{ct})$

of the rule cseal.

- By case distinction on the name of the register in the domain of cr' (let's denote it `crname`; i.e., let `crname` range over the domain of cr' or cr'' depending on the context), we get the following two cases:

  * Case `crname` = `cd`

    We first show that conjunct 12 holds.

    If the conjunct holds vacuously of cr'' on register `cs`, then it will continue to hold vacuously in cr' of register `cd`.

    Else, we do case distinction on the permission that is included in `perms_s`. The cases are not necessarily mutually exclusive, but they are exhaustive.

    Case permit_execute $\in$ `perms_s`

    Then we know that since sealing succeeded, namely, `s_s` = `false`, then conjunct 12 follows from conjunct 9 of the induction hypothesis.

    Case permit_store $\in$ `perms_s`

    Then we know that since sealing succeeded, namely, `s_s` = `false`, then conjunct 12 follows from the truth of disjunct $a' \in c^*.\mathit{Data}$ of the consequent of conjunct 13 of the induction hypothesis.

    Case permit_load $\in$ `perms_s`

    Then we know that since sealing succeeded, namely, `s_s` = `false`, then conjunct 12 follows from the truth of disjuncts $a' \in c^*.\mathit{Data} \vee a' \in c^*.\mathit{Code}$ of the consequent of conjunct 14 of the induction hypothesis.

    Case permit_store_capability $\in$ `perms_s`

    Follows from conjunct 15.

    Case permit_load_capability $\in$ `perms_s`

    Follows from conjunct 16.

    These cases are exhaustive for the case when conjunct 12 holds non-vacuously, so conjunct 12 holds.

    We have that conjunct 10 now follows from conjunct 9 of the induction hypothesis (i.e., by obtaining the consequent $a' \in c''.\mathit{Code}$ of 9 as the fulfilling disjunct of the consequent of 10).

    We have that conjuncts 9, 11, 13, 14, 15, and 16 follow directly from the corresponding conjuncts of the induction hypothesis about cr'' on register `cs` because the induction hypothesis holds for both values of the "sealed" bit.

    So all of conjuncts 9, 10, 11, 12, 13, 14, 15, and 16 hold.

  * Case `crname` $\neq$ `cd`

Then conjuncts 9, 10, 11, 12, 13, 14, 15, and 16 about pcc' and cr' follow immediately from the induction hypothesis about pcc'' and cr''.

– So we have shown all the conjuncts of $\langle \mathsf{pcc}', cr', m' \rangle \preceq \langle c'', C \rangle$ to be true.

– We next show that $\mathsf{pc}' \in c''.Code$. This holds for exactly the same reasons argued in the previous case. We avoid repetition.

Case proved.

- **Case $\mathsf{cunseal}$**

  – We observe that $m' = m''$, from which it follows that conjuncts 1, 2, 3, 4, 5, 6, 7, and 8 follow immediately from the induction hypothesis.

  – We also observe that $\mathsf{pcc}' = \mathsf{pcc}''$.

  – We then obtain the necessary preconditions

  (i) $cr'' \vdash \mathsf{permits\_unseal}(\mathtt{cs}, \mathtt{ct})$

  (ii) $cr' = cr''[\mathtt{cd} \mapsto \mathsf{clear\_otype}(\mathsf{remove\_non\_ephemeral}(\mathsf{unsealed}(cr''(\mathtt{cs})), cr''(\mathtt{ct})))]$

  of the rule $\mathsf{cunseal}$.

  – By case distinction on the name of the register in the domain of cr' (let's denote it `crname`; i.e., let `crname` range over the domain of cr' or cr'' depending on the context), we get the following two cases:

    ∗ Case `crname` $= \mathtt{cd}$

    We have that conjuncts 10, 12 now hold vacuously.

    We have that conjuncts 11, 13, 14, 15, and 16 follow directly from the corresponding conjuncts of the induction hypothesis about cr'' on register `cs` because the induction hypothesis holds for both values of the "sealed" bit.

    We now show that conjunct 9 holds.

    We conclude from precondition (i) that we obtained above that $\mathsf{permit\_seal} \in \mathtt{perms\_t}$ and, hence, from conjunct 11 of the induction hypothesis about cr'' for register `ct` that $\mathtt{addr\_t} \in c''.Data \ \cup \ c''.Code$ and also from precondition (i) that $\mathtt{otype\_s} = \mathtt{addr\_t}$ (where we denote $cr''(cs)$ by $(\mathtt{s\_s}, \mathsf{bin}(\mathtt{perms\_s}), \mathtt{addr\_s}, \mathtt{len\_s}, \mathtt{otype\_s})$).

    We need to show that conjunct 9 holds. In the case when it does not hold vacuously, then we use the notation

    $cr'(\mathtt{cd}) = \langle 1, (\mathtt{false}, \mathsf{bin}(\mathtt{perms}), \mathtt{st}, \mathtt{len}, \mathtt{o}) \rangle$ and we distinguish two cases from conjunct 10 of the induction hypothesis about $cr''(\mathtt{cs})$.

    Either the disjunct about $a' \in c''.Code$ holds or the disjunct:

    $\exists c \in C. \forall a' \in [\mathtt{st}, \mathtt{st} + \mathtt{len}).$

    $a' \in c''.J \wedge a' \in c.Code \wedge o \in c.Code \cup c.Data$ **holds.**

We claim that the latter case is impossible to hold because then the unsealing would not have succeeded. (The proof: We have $o \in c''.\mathit{Data} \cup c''.\mathit{Code}$ which is required for the unsealing, but the disjunct implies that $o \in c.\mathit{Data} \cup c.\mathit{Code}$. But we know that $c \neq c''$ from the constraint that $c''.J$ are **external** jump addresses and we know that $c \cap c'' = \emptyset$ from $\mathsf{valid}(C)$. So $o \in c.\mathit{Data} \cup c.\mathit{Code}$ cannot hold and the disjunct must be false.)
In particular, we conclude that in conjunct 10 of the induction hypothesis about $cr''(\texttt{cs})$, $a' \in c''.\mathit{Code}$ **must hold**, which suffices to conclude that conjunct 9 holds for $cr'(\texttt{cd})$.

  * Case $\texttt{crname} \neq \texttt{cd}$
    Then conjuncts 9, 10, 11, 12, 13, 14, 15, and 16 about pcc' and cr' follow immediately from the induction hypothesis about pcc'' and cr''.

- So we have shown all the conjuncts of $\langle \mathsf{pcc}', cr', m' \rangle \preceq \langle c'', C \rangle$ to be true.
- We next show that $\mathsf{pc}' \in c''.\mathit{Code}$. This holds for exactly the same reasons argued in the previous case. We avoid repetition.

Case proved.

- **Case cjr, cjrcond-true**
  - We observe that $m' = m''$, from which it follows that conjuncts 1, 2, 3, 4, 5, 6, 7, and 8 follow immediately from the induction hypothesis.
  - We also observe that $cr' = cr''$, from which it follows that conjuncts 10, 11, 12, 13, 14, 15, and 16 about cr' follow immediately from the induction hypothesis about cr''.
  - We now show that conjunct 9 holds.

    We observe that since $cr' = cr''$, then the conjunct holds for $cap \in \mathsf{Range}(cr')$ as an immediate result of the induction hypothesis that it holds for $cap \in \mathsf{Range}(cr'')$.

    So it remains to show that if $\mathsf{pcc}' = \langle 1, (\texttt{false}, \mathsf{bin}(\texttt{perms}), \texttt{st}, \texttt{len}, \texttt{o}) \rangle$ and $\mathsf{permit\_execute} \in \texttt{perms}$, then $\forall a' \in [\texttt{st}, \texttt{st} + \texttt{len}).\ a' \in c''.\mathit{Code}$.

    To show that, we obtain the necessary precondition $cr'' \vdash \mathsf{permits\_execute}(\texttt{cb})$ of the rules cjr and $\mathsf{cjrcond\text{-}true}$, from which (by looking at the preconditions of the rule $\mathsf{permits\_execute}$) we obtain the antecedent of conjunct 9 of the induction hypothesis on cr'' for register $\texttt{cb}$. So, we get the consequent $\forall a' \in [\texttt{st}, \texttt{st} + \texttt{len}).\ a' \in c''.\mathit{Code}$ for $cr''(\texttt{cb}) = \langle 1, (\texttt{false}, \mathsf{bin}(\texttt{perms}), \texttt{st}, \texttt{len}, \texttt{o}) \rangle$.

    Now from the precondition $\mathsf{pcc}' = cr(\texttt{cb})$ of the rules cjr and $\mathsf{cjrcond\text{-}true}$, we can hence use the consequent $\forall a' \in [\texttt{st}, \texttt{st} + \texttt{len}).\ a' \in c''.\mathit{Code}$ to show that conjunct 9 holds about pcc'.

- We now need to show that $\mathsf{pc}' \in c'.Code$ for some c' such that $\langle \mathsf{pcc}', cr', m' \rangle \preceq \langle c', C \rangle$. Above we have already shown that $\langle \mathsf{pcc}', cr', m' \rangle \preceq \langle c'', C \rangle$, so $c' = c''$. So we need to show that $\mathsf{pc}' \in c''.Code$.

  This again holds for the same reason as in all the previous cases, which is explained in the very first case of this proof.

Case proved.

- **Case ccall**

  - *Here we use the claim:* ***if two valid sealed capabilities have the same*** `otype`***, then their bounds lie in the same compartment.*** *The claim can be proved by conjuncts 4 and 12 of the induction hypothesis, and from validity of the compartments set –*$\mathsf{valid}(C)$*– which implies pairwise disjointness of the compartments. Thus, we know that any one* `otype` *value makes all the capabilities that share this value have bounds that lie in the address space of exactly one compartment.*
  - We observe that $m' = m''$, from which it follows that conjuncts 1, 2, 3, 4, 5, 6, 7, and 8 follow immediately from the induction hypothesis.
  - We then obtain the necessary precondition $cr'' \vdash \mathsf{callable}(\mathtt{cc}, \mathtt{cd})$ of the rule ccall. From the definition of the rule $\mathsf{callable}$, we obtain the necessary preconditions $cr''(cc) = \langle \mathtt{1}, (\mathtt{s\_c}, \mathsf{bin}(\mathtt{perms\_c}), \mathtt{addr\_c}, \mathtt{len\_c}, \mathtt{otype\_c}) \rangle$, $\mathtt{s\_c} = \mathtt{true}$ and $\{\mathsf{permit\_execute}\} \subseteq \mathtt{perms\_c}$, from the three of which we immediately know that conjunct 10 of the induction hypothesis about cr'' must hold non-vacuously of register $\mathtt{cc}$.
  - Then, we distinguish two possible cases. (Informally, they are cases that witness when the active compartment changes or when it does not.) We claim that the two cases are mutually exclusive. (The proof simply follows from the definition of a valid compartment set which requires disjointness of compartments c^* and c, so o cannot be an address belonging to the address spaces of both compartments at the same time. And we know they are different compartments in the first place because of the restriction on $c^*.J$ being only external jump targets.)

    We note that we are considering case distinction over the consequent of conjunct 10 of the induction hypothesis about cr'' on register $\mathtt{cc}$:

    * **Case disjunct** $\forall a' \in [\mathtt{addr\_c}, \mathtt{addr\_c} + \mathtt{len\_c})$.
      $a' \in c''.Code \land o \in c''.Code \cup c''.Data$ **holds**

      Then we choose $c' = c''$, and we show that $\langle \mathsf{pcc}', cr', m' \rangle \preceq \langle c'', C \rangle$

      We have already shown that the conjuncts about m' hold.

We obtain the necessary preconditions $cd' = \mathsf{unsealed}(cr''(\texttt{cd}))$ and $cr' = \{\texttt{cdd} \mapsto \texttt{cd}'\}$ of the rule ccall.

We now see that conjuncts 11, 13, 14, 15, and 16 follow immediately from the induction hypothesis, conjuncts 10, and 12 hold vacuously, conjunct 9 follows immediately from both conjunct 10 of the induction hypothesis and the necessary precondition $\texttt{pcc}' = cc'$ of the rule ccall.

Case proved.

* **Case disjunct** $\exists c \in C. \forall a' \in [\texttt{st}, \texttt{st} + \texttt{len}).$

  $a' \in c''.J \wedge a' \in c.\mathit{Code} \wedge o \in c.\mathit{Code} \cup c.\mathit{Data}$ **holds**

  Then we obtain such c, and we choose $c' = c$ then show that $\langle \texttt{pcc}', cr', m' \rangle \preceq \langle c', C \rangle$.

  Again, we obtain the necessary preconditions $cd' = \mathsf{unsealed}(cr''(\texttt{cd}))$ and $cr' = \{\texttt{cdd} \mapsto \texttt{cd}'\}$ of the rule ccall.

  Here we see, by the necessary disjointness of c' and c'' and the necessary precondition by conjunct 12 of the induction hypothesis which must have held non-vacuously of cr'' on register $\texttt{cd}$ that conjuncts 15, 16 and 11 must have held vacuously in the induction hypothesis, and so they continue to hold vacuously for cr' about the sole value cd'.

  We also observe that conjuncts 10, and 12 hold vacuously as above.

  We now use the claim of equality of $\texttt{otype}$ for both $\texttt{cc}$ and $\texttt{cd}$ and conclude that conjunct 9 holds of cr' for the value cd' based on conjunct 10 of the induction hypothesis, and it holds for $\texttt{pcc}'$ for the same reasons as in the previous case together with 10 of the induction hypothesis and observing the true disjunct that we are currently considering.

  We then see that conjuncts 13 and 14 about the load and store follow for cr' on value cd' by the necessity of the truth of the disjuncts about external store addresses and load addresses, respectively. This necessity follows again from disjointness of compartments which makes the complementary disjuncts impossible. Now conjuncts 13 and 14 about loads and stores are guaranteed to hold (in particular, the bounds are guaranteed to be in $c.\mathit{Data}$) because by the definition of a valid compartments set, we have for c'', $(c''.L \cup c''.S) \subseteq \bigcup\limits_{c_i \in C} c_i.\mathit{Data}$.

  All conjuncts are proved. Case is proved.

- **Case allocate** This case is vacuous because the theorem considers only transitions for which $\mathsf{next\_free}$ remains fixed.

All cases covered. $\square$

Theorem 5.6. *(Execution that preserves $\preceq$ respects external jumps)*

If C is a valid set of compartments ($\mathsf{valid}(C)$), $c_1 \in C$, $c_2 \in C$, $c_1 \neq c_2$

$\langle \mathsf{pcc}, cr, m\rangle \preceq \langle c_1, C\rangle$,

$\mathsf{pc} \in c_1.Code$,

$\langle m, r, cr, \mathsf{pc},\ \mathsf{pcc},\ \mathsf{next\_free}\rangle \to \langle m', r', cr', \mathsf{pc}',\ \mathsf{pcc}',\ \mathsf{next\_free}\rangle$,

$\langle \mathsf{pcc}', cr', m'\rangle \preceq \langle c_2, C\rangle$,

then

$\mathsf{pc}' \in c_1.J$.

Proof. The theorem holds vacuously for every case except ccall where we observe that, as mentioned in the proof of this case in the previous theorem that the disjunct $\exists c \in C. \forall a' \in [\mathtt{st}, \mathtt{st} + \mathtt{len})$.

$a' \in c''.J \wedge a' \in c.Code \wedge o \in c.Code \cup c.Data$ holds. Then the conclusion follows immediately by observing the rule executable in the precondition of the rule legal-transition and obtaining the precondition $\mathsf{pcc}' = cc'$ of the rule ccall. □

Chapter 6

Future Work

In this chapter, we lay out a plan for possible future directions towards building a secure compiler and for formalizing more security goals achievable by the CHERI formal model.

6.1 Memory Safety using CHERI

Memory safety is defined in [9, Section 7] as both spatial and temporal safety.

Spatial memory safety is definable according to the given source programming language. If we consider unsafe languages like C which can witness buffer overflow vulnerabilities, then, if we think of capabilities as pointers [11], then it is easy to see that the design of capabilities in such a way that they can provide byte-level granularity of protection allows the compiler to provide guarantees for mitigation of serious risks potentially caused by these vulnerabilities.

Temporal memory safety is the guarantee that memory access and deallocation operations take place on only currently allocated memory. This means that vulnerabilities like Use-After-Free [51] or Double-Free are violations to temporal memory safety.

We illustrate (informally) how temporal safety can be easily achieved using capabilities, especially with two main techniques: one is to think of pointers as capabilities, and the second is to make use of ephemeral (local) capabilities.

The operation of freeing a pointer can be implemented by deleting all permissions of the corresponding capability.

Let us consider the "Use-After-Free" flaw. The goal is to have defined behavior (in our formal model, namely going stuck, or aborting the program) instead of undefined behavior when use-after-free occurs.

There are two cases:

1. The illegitimate use and the free are in the **same** function, e.g.:

```
char* ptr = (char*)malloc (SIZE);
if (err) {
  abrt = 1;
  free(ptr);
}
...
if (abrt) {
  logError("operation aborted before commit", ptr);
}
```

   [51]

   Here the “free” operation will already revoke the capability, so the attempt to “logError” in the freed “ptr” will always lead to the defined behavior of getting stuck (or in a more practically-suited formal model, to a known exception).

2. The illegitimate use and the free are in **different** functions.

```
void A() {
  ptr = allocate(SIZE);
  B(ptr);
  free(ptr);
}

void B(p) {
  if (good_times()) {
    mem[idx] = p;
  }

  if (evil_times()) {
    old_ptr = mem[idx];
    use_irresponsibly(old_ptr);
  }
}
```

   In this case, assuming that storing the pointer in memory copies the value of the pointer (i.e., copies (aliases) the underlying capability), so we end up with a copy of the capability lying in memory, which is not “freed” (i.e., still has its permissions set).

   So, a potentially undesired scenario is that A calls B; in `good_times()`, B stores the pointer and returns. Then A in turn frees the pointer. Next, some function E calls B; in `evil_times()`, B loads the pointer that A had passed to it earlier, and B now has access to potentially freed memory.

   Now, if A has a way to annotate `ptr` as “`__ephemeral__ ptr = allocate(SIZE);`”, then B cannot store it (i.e., “`mem[idx] = p;`” will raise an exception, or get stuck).

Spatial safety (e.g., preventing out-of-bounds access) is less complicated, except that providing an example of support for memory-safe operations in a C-like semantics possibly requires some extra assumptions and changes to the C abstract machine as proposed in [52], which is beyond the scope of this section, as we intend to provide just a proof of concept. But it is worth noting that the notion of having capabilities on memory regions with granularity of one memory word suffices (if the high-level abstract semantics enforces the declaration of bounds) to guarantee spatial safety.

Abstract machine for temporal memory safety

We suggest an abstract representation of memory allocation and deallocation operations that is similar to the ideas presented in MemSafe [53]. An abstract machine for temporal memory safety models a set of addresses A in which each address is tagged with a timestamp. The machine state also keeps track of a possibly unbounded set of free memory addresses, F. Arbitrary silent transitions that are not memory access operations do not alter the machine state and are allowed. A transition that takes one of the free memory addresses $f \in F$, associates it with a fresh timestamp τ_{fresh}, and adds it to the set A, (i.e., $F' := F - \{f\}$, and $A' := A \cup (f, \tau_{fresh})$ as effects), is the only allowed memory allocation transition. Any other memory access operation (read/write/execute) should use addresses exclusively from A. The argument to the access operation is a pair of address a and timestamp τ. In order for the operation to proceed, the pair has to exist in A. A free operation should also use timestamped-addresses exclusively from A, and it returns them back to the set F (i.e., freeing (a, τ) has the effect $A' = A - \{(a, \tau)\}, F' = F \cup \{a\}$).

One way, then, to reason about memory safety is to ensure that compilation produces machine code whose legal transitions can satisfy a simulation relation with the legal transitions of the abstract machine described above.

6.2 Caller-Callee Authentication

In architectures like "Protected Software Module Architectures" [54], the need often arises for having a trusted way provided by the hardware of identifying the module that uses a specific function or API. In other words, an authenticated mechanism for function calls is required. This can be useful on some embedded architectures where arbitrary untrusted modules can run on the same embedded device leading to the need for some authentication of which module uses some API, so that accountability and integrity goals can be achieved on part of the outputs of the whole device which are potentially triggered by the mentioned use of APIs by arbitrarily different and potentially untrusted modules [55].

In this section we illustrate informally a simple way by means of which a CHERI-based machine can achieve such an authentication mechanism.

We assume that a machine that will make use of such authentication mechanism will have some trusted procedure that allocates memory and capabilities to the different modules or compartments. And it is a rather reasonable assumption to think of these modules or compartments to be known to each other if they are to interact with each other correctly. We assume that a static memory map is known to the programmers of each module. This is a reasonable assumption to make about the software development of some embedded devices. Note that what we mean by the notion of being known is not the cryptographic sense of having a key-exchange phase, but rather it is just a phase of the software development process through which it suffices that programmers of each "callee/API" adapt their code to the number of legitimate callers that it (the callee/API) expects. Note that an initialization phase of the callee program is thought of as responsible for distributing all and only the relevant capabilities to the corresponding modules of each expected caller. It does that according to the agreed upon memory map, and is enabled to do that by means of having been given the necessary capabilities during the initialization phase of the whole operating system or startup procedure.

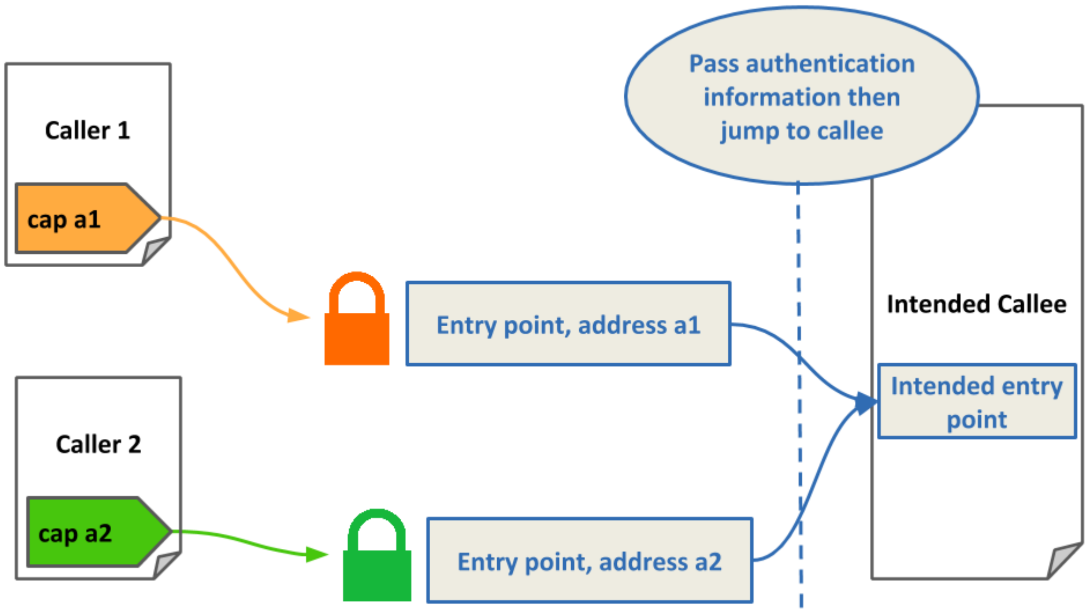
Caller Authentication by Unforgeability of Capabilities and Compartmentalization

If we assume that compartmentalization is implemented correctly, and if we assume that compartments are not allowed to share capabilities with each other except through the trusted initialization phase that creates the memory compartments based on a statically known memory map, then, callers of an API can be authenticated by means of giving each caller a capability on a distinguished address. This address will be available for execution for only this caller. (It is guaranteed that no other caller gets a capability on this memory address.) Figure 6.1 illustrates the idea of having an added layer of indirection to which a call is sent by two distinguished callers of the same API to two different addresses, from which authentication information is collected and then the call is redirected to the API common entry point.

6.3 Secure Compilation

As mentioned in the introduction of this thesis, some vulnerabilities in computer systems arise because of programmers' false (i.e., over-optimistic) expectations about the guarantees that are given by the abstractions of a programming language [2].

FIGURE 6.1: On the left, caller 1 is given a sealed execute capability on address $a1$, and caller 2 is given a sealed execute capability on address $a2$. Compartmentalization guarantees that they cannot share capabilities with each other. Capability unforgeability guarantees that whenever address a_i is executed, then caller i was the origin of the call. This authentication information can, thus, be passed to the intended callee and execution can proceed with the intended API entry point.



Thus, the notion of designing a compiler to be secure has been captured by the notion of full abstraction. A fully-abstract translation from a source language to a target language preserves and reflects contextual observational equivalence of source programs [14, 2].

One challenge to achieving full abstraction is the inherent increased expressiveness of target programs and contexts compared to source programs and contexts. This loss of abstraction (or increase in expressiveness) can be exploited by target-level attackers.

Thus, hardware protection mechanisms like the ones offered by CHERI and formalized in this thesis can be helpful for preserving programming language abstractions. The philosophy of designing CHERI in the first place put into consideration lack of trust in compiler correctness of code that is outside the trusted code base (TCB) [11]. So, the security guarantees that CHERI intends to offer are indeed guarantees against machine-code-level attackers that could execute arbitrary instructions.

Previous work [56, 13, 57, 12] on fully-abstract compilation made use of hardware support features like Protected Module Architectures [58], and Micropolicies [9] to achieve security guarantees that are necessary for fully-abstract compilation.

We expect that a formal model of CHERI like the one we reasoned about in this thesis will have the potential to make it easy to exploit the security guarantees offered by the CHERI architecture for building a compiler that satisfies a notion of memory compartmentalization that is more suited for an object-oriented Java-like programming language. Proving, then, that such a compiler is secure in the sense of being fully abstract could potentially utilize the formalization that we established in this thesis.

One potential starting point towards defining a compiler and proving its security is to start with a simple Java-like language like the one in [13]. This has the benefit of avoiding all the dispute [52] over what standard a compiler should follow if a language like C is to be considered as a source language. After all, the whole idea of guaranteeing full abstraction as an approach to secure compilation is to preserve the guarantees that are given to the programmer by the source language. So, a choice of a source language with strong guarantees is a good match for such a kind of work. What full abstraction would additionally achieve, then, with the help of an architecture like CHERI is the preservation of such guarantees even in a stronger attack model than the ones assumed by current compilers.

Bibliography

[1] W. Jimenez, A. Mammar, and A. Cavalli, "Software vulnerabilities, prevention and detection methods: A review1," *Security in Model-Driven Architecture*, p. 6, 2009.

[2] F. Piessens, D. Devriese, J. T. Mühlberg, and R. Strackx, "Security guarantees for the execution infrastructure of software applications," ser. IEEE SecDev'16, 2016, private communication - To appear in IEEE SecDev 2016.

[3] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," *SIGARCH Comput. Archit. News*, vol. 32, no. 5, pp. 85–96, Oct. 2004. [Online]. Available: http://doi.acm.org/10.1145/1037947.1024404

[4] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The cheri capability model: Revisiting risc in an age of risk," *SIGARCH Comput. Archit. News*, vol. 42, no. 3, pp. 457–468, Jun. 2014. [Online]. Available: http://doi.acm.org/10.1145/2678373.2665740

[5] A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach, "A verified information-flow architecture," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14. New York, NY, USA: ACM, 2014, pp. 165–178. [Online]. Available: http://doi.acm.org/10.1145/2535838.2535839

[6] "Introduction to Intel Memory Protection Extensions," [Online; accessed 07-September-2016]. [Online]. Available: https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions

[7] J. Devietti, C. Blundell, M. M. Martin, and S. Zdancewic, "Hardbound: architectural support for spatial safety of the c programming language," in *ACM SIGARCH Computer Architecture News*, vol. 36, no. 1. ACM, 2008, pp. 103–114.

[8] U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, Jr., B. C. Pierce, and A. DeHon, "Architectural support for software-defined metadata processing," *SIGARCH Comput. Archit. News*, vol. 43, no. 1, pp. 487–502, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2786763.2694383

[9] A. A. De Amorim, M. Dénes, N. Giannarakis, C. Hritcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach, "Micro-policies: Formally verified, tag-based security monitors," in *Security and Privacy (SP), 2015 IEEE Symposium on.* IEEE, 2015, pp. 813–830.

[10] U. Erlingsson and F. B. Schneider, "Sasi enforcement of security policies: A retrospective," in *Proceedings of the 1999 Workshop on New Security Paradigms*, ser. NSPW '99. New York, NY, USA: ACM, 2000, pp. 87–95. [Online]. Available: http://doi.acm.org/10.1145/335169.335201

[11] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, J. Anderson, D. Chisnall, B. Davis, A. Joannou, B. Laurie, S. W. Moore, S. J. Murdoch, R. Norton, S. Son, and H. Xia, "Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 5)," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-891, Jun. 2016. [Online]. Available: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-891.pdf

[12] M. Patrignani, "The tome of secure compilation: Fully abstract compilation to protected modules architectures," 2015.

[13] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens, "Secure compilation to protected module architectures," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 37, no. 2, p. 6, 2015.

[14] M. Abadi, "Protection in programming-language translations," in *International Colloquium on Automata, Languages, and Programming.* Springer, 1998, pp. 868–883.

[15] J. C. Mitchell, "On abstraction and the expressive power of programming languages," *Science of Computer Programming*, vol. 21, no. 2, pp. 141–163, 1993.

[16] "Rigorous Engineering of Mainstream Systems," [Online; accessed 06-September-2016]. [Online]. Available: https://www.cl.cam.ac.uk/~pes20/rems/

[17] A. Fox, *Directions in ISA Specification.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 338–344. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32347-8_23

[18] ——, *Improved Tool Support for Machine-Code Decompilation in HOL4.* Cham: Springer International Publishing, 2015, pp. 187–202. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-22102-1_12

[19] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security.* ACM, 2005, pp. 340–353.

[20] ——, "A theory of secure control flow," in *Formal Methods and Software Engineering.* Springer, 2005, pp. 111–124.

[21] ——, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, pp. 4:1–4:40, Nov. 2009. [Online]. Available: http://doi.acm.org/10.1145/1609956.1609960

[22] H. M. Levy, *Capability-Based Computer Systems.* Newton, MA, USA: Butterworth-Heinemann, 1984.

[23] R. S. Fabry, "Capability-based addressing," *Commun. ACM*, vol. 17, no. 7, pp. 403–412, Jul. 1974. [Online]. Available: http://doi.acm.org/10.1145/361011.361070

[24] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee, “The m-machine multicomputer,” *International Journal of Parallel Programming*, vol. 25, no. 3, pp. 183–212, 1997.

[25] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie *et al.*, “Cheri: A hybrid capability-system architecture for scalable software compartmentalization,” in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 20–37.

[26] Y. Juglaret, C. Hritcu, A. A. de Amorim, and B. C. Pierce, “Beyond full abstraction: Formalizing the security guarantees of low-level compartmentalization,” *CoRR*, vol. abs/1602.04503, 2016. [Online]. Available: http://arxiv.org/abs/1602.04503

[27] R. M. Norton, “Hardware support for compartmentalisation,” University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-887, May 2016. [Online]. Available: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-887.pdf

[28] B. W. Lampson, “Protection,” *SIGOPS Oper. Syst. Rev.*, vol. 8, no. 1, pp. 18–24, Jan. 1974. [Online]. Available: http://doi.acm.org/10.1145/775265.775268

[29] J. H. Saltzer and M. D. Schroeder, “The protection of information in computer systems,” *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[30] R. N. Watson, J. Anderson, B. Laurie, and K. Kennaway, “Capsicum: Practical capabilities for unix.”

[31] J. S. Shapiro, J. M. Smith, and D. J. Farber, “Eros: A fast capability system,” *SIGOPS Oper. Syst. Rev.*, vol. 33, no. 5, pp. 170–185, Dec. 1999. [Online]. Available: http://doi.acm.org/10.1145/319344.319163

[32] R. J. Feiertag and P. G. Neumann, “The foundations of a provably secure operating system (psos).”

[33] N. P. Carter, S. W. Keckler, and W. J. Dally, “Hardware support for fast capability-based addressing,” in *ACM SIGPLAN Notices*, vol. 29, no. 11. ACM, 1994, pp. 319–327.

[34] J. Heinrich, *MIPS R4000 Microprocessor User’s manual*, 1994.

[35] D. E. Bell and L. J. LaPadula, “Secure computer systems: Mathematical foundations,” DTIC Document, Tech. Rep., 1973.

[36] M. Miller, K.-P. Yee, J. Shapiro, and C. Inc, “Capability myths demolished,” Tech. Rep., 2003.

[37] D. Volpano, C. Irvine, and G. Smith, “A sound type system for secure flow analysis,” *Journal of computer security*, vol. 4, no. 2-3, pp. 167–187, 1996.

[38] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits, “Fully abstract compilation to javascript,” *SIGPLAN Not.*, vol. 48, no. 1, pp. 371–384, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2480359.2429114

[39] J. D. Woodruff, “CHERI: A RISC capability machine for practical memory safety,” University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-858, Jul. 2014. [Online]. Available: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-858.pdf

[40] F. Chow, S. Correll, M. Himelstein, E. Killian, and L. Weber, "How many addressing modes are enough?" *SIGPLAN Not.*, vol. 22, no. 10, pp. 117–121, Oct. 1987. [Online]. Available: http://doi.acm.org/10.1145/36205.36193

[41] P. Wadler, "A taste of linear logic," in *International Symposium on Mathematical Foundations of Computer Science*. Springer, 1993, pp. 185–210.

[42] S. Maffeis, J. C. Mitchell, and A. Taly, "Object capabilities and isolation of untrusted web applications," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 125–140.

[43] D. Wagner and R. Dean, "Intrusion detection via static analysis," in *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*. IEEE, 2001, pp. 156–168.

[44] J. Wilander and M. Kamkar, "A comparison of publicly available tools for dynamic buffer overflow prevention." in *NDSS*, vol. 3, 2003, pp. 149–162.

[45] G. Morrisett, D. Walker, K. Crary, and N. Glew, "From system f to typed assembly language," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 21, no. 3, pp. 527–568, 1999.

[46] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni, "A syntactic approach to foundational proof-carrying code," *Journal of Automated Reasoning*, vol. 31, no. 3-4, pp. 191–229, 2003.

[47] M. Budiu, U. Erlingsson, and M. Abadi, "Architectural support for software-based protection," in *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ser. ASID '06. New York, NY, USA: ACM, 2006, pp. 42–51. [Online]. Available: http://doi.acm.org/10.1145/1181309.1181316

[48] J. H. Morris, Jr., "Protection in programming languages," *Commun. ACM*, vol. 16, no. 1, pp. 15–21, Jan. 1973. [Online]. Available: http://doi.acm.org/10.1145/361932.361937

[49] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '93. New York, NY, USA: ACM, 1993, pp. 203–216. [Online]. Available: http://doi.acm.org/10.1145/168619.168635

[50] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *Communications of the ACM*, vol. 9, no. 3, pp. 143–155, 1966.

[51] "CWE - CWE-416: Use After Free (2.9)," [Online; accessed 04-July-2016]. [Online]. Available: https://cwe.mitre.org/data/definitions/416.html

[52] D. Chisnall, C. Rothwell, R. N. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. G. Neumann, "Beyond the pdp-11: Architectural support for a memory-safe c abstract machine," in *ACM SIGPLAN Notices*, vol. 50, no. 4. ACM, 2015, pp. 117–130.

[53] M. S. Simpson and R. K. Barua, "Memsafe: ensuring the spatial and temporal memory safety of c at runtime," *Software: Practice and Experience*, vol. 43, no. 1, pp. 93–128, 2013.

[54] R. Strackx, J. Noorman, I. Verbauwhede, B. Preneel, and F. Piessens, "Protected software module architectures," in *ISSE 2013 Securing Electronic Business Processes*. Springer, 2013, pp. 241–251.

[55] J. Van Bulck, J. Noorman, J. T. Mühlberg, and F. Piessens, “Towards availability and real-time guarantees for protected module architectures,” in *Companion Proceedings of the 15th International Conference on Modularity*, ser. MODULARITY Companion 2016. New York, NY, USA: ACM, 2016, pp. 146–151. [Online]. Available: http://doi.acm.org/10.1145/2892664.2892693

[56] M. Patrignani and D. Clarke, “Fully abstract trace semantics for protected module architectures,” *Computer Languages, Systems & Structures*, vol. 42, pp. 22–45, 2015.

[57] Y. Juglaret and C. Hritcu, “Secure compilation using micro-policies,” 2015.

[58] R. Strackx, J. Noorman, I. Verbauwhede, B. Preneel, and F. Piessens, “Protected software module architectures,” in *ISSE 2013 Securing Electronic Business Processes*. Springer, 2013, pp. 241–251.