Formal Reasoning about Capability Machines

Lau Skorstengaard

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Formal Reasoning about Capability Machines

A Dissertation
Presented to the Faculty of Science and Technology
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Lau Skorstengaard
Friday 30th August, 2019

Abstract

Today, computer security is often based on mitigations that make exploitation cumbersome or unlikely. In other words, mitigation techniques provide no security *guarantee*, and time and time again mitigations have been circumvented. To stop the ongoing exploit-mitigation arms-race, we need security enforcement mechanisms that can be proven impossible to circumvent. Unfortunately, mainstream computers have the necessary security primitives to support such enforcement mechanisms. Capability machines are computers that provide additional security primitives by replacing pointers with capabilities. Capabilities are unforgeable tokens of authority that must be used to access memory and other system resources.

In this dissertation, we present foundational research in the field of secure compilation that targets capability machines. In particular, we present novel and provably secure enforcement mechanisms for control-flow and encapsulation properties of high-level programming languages – properties that all programmers rely on. The security proof demonstrates for the first time how state-of-the-art reasoning techniques for high-level programming languages can be used to reason about capability safety and secure compilation at the very lowest level. We also contribute new proof techniques for secure compilation proofs.

In Chapter 1, we introduce the concept of capabilities, show various instances of capabilities throughout computer science, and present and motivate the contents of this dissertation.

In Chapter 2, we present a formalisation of a capability machine with local capabilities and a calling convention that provably enforces well-bracketed control-flow (WBCF) and local-state encapsulation (LSE). We also present a notion of capability safety in terms of a logical relation. Using the logical relation, we prove correctness of program examples that interact with unknown code and rely on non-trivial control-flow for correctness.

In Chapter 3, we present a formalisation of a capability machine with linear capabilities along with StkTokens: another calling convention provably enforcing WBCF and LSE. Via a novel proof technique called fully-abstract overlay semantics, we prove that StkTokens enforces these properties. The overlay semantics simplifies the full-abstraction proof significantly by retaining the language syntax and adding the desired property semantically.

Resumé

De fleste programmer er skrevet i højniveausprogrammeringssprog med abstraktioner, der gør, at programmører ikke behøver at bekymre sig om hardwaredetaljer. Computere kan dog ikke køre sådanne programmer, hvilket løses ved at oversætte dem til et lavniveausprogrammeringssprog, som computeren forstår. Umiddelbart lyder dette uskyldigt, men hvad sker der egentlig med højniveausabstraktionerne efter oversættelsen? Er de bibeholdt, og hvis ikke betyder det noget for sikkerheden? Svaret er, groft sagt, at abstraktionerne ikke er bibeholdt, og det giver anledning til sikkerhedsbrister. Eksempelvis er *buffer overflow* angreb og *return oriented-programming* angreb baseret på, at forventede højniveauabstraktioner ikke bibeholdes efter oversættelse.

Sikker oversættelse (i forskningsverdenen kendt som *secure compilation*) omhandler oversættelser, der bevarer sikkerhed efter oversættelse. Der er mange former for sikkerhed og ligeså mange former for sikker oversættelse. *Full abstraction* er en form for sikker oversættelse, der beviseligt bevarer abstraktioner - selv hvis programmet interagerer med ukendte ondsindede programmer. Dette er en meget stærk egenskab, som kun er opfyldt, hvis et oversat program håndhæve abstraktionerne. Der eksisterer i dag ikke nogen rigtig oversætter, som er *fully abstract*[1].

Et problem, i forhold til at lave en sikker oversættelse, er, at vores computere ikke har tilstrækkelige sikkerhedsprimitiver til at håndhæve højniveausabstraktionerne. Hvis vi ønsker at have sikre computere, så bliver vi nødt til at gå væk fra den traditionelle computer over til en computer med flere sikkerhedsprimitiver, men hvilken? Et bud kunne være *capability machines*, hvilket er computere, der udskifter pointers med *capabilities*. En *capability* er essentielt set en polet, der ikke kan blive forfalsket og som giver adgang til noget, eksempelvis at læse fra en del af hukommelsen.

I denne afhandling tager vi et stort skridt mod at lave en rigtig sikker oversætter ved at vise, at capability machines kan udgøre grundlaget for sikker oversættelse. Vi viser, hvordan en capability machines ekstra sikkerhedsprimitiver kan håndhæve en håndfuld højniveausabstraktioner. Vi giver tilmed matematiske beviser for, at abstraktionerne er håndhævet.

[1] Der eksisterer oversættelser mellem legetøjsprogrammeringsprog, der beviseligt opfylder egenskaben.

Acknowledgements

First and foremost, I want to thank my supervisor Lars Birkedal for his guidance throughout my PhD studies. I am especially grateful for Lars' words of encouragement when I was needlessly hard on my own work. I also want to extend a big thanks to Dominique Devriese for a long and fruitful collaboration as well as for hosting me at KU Leuven on a number of occasions. Although Dominique has not officially supervised me, he has provided plenty of advice and guidance.

I also want to extend my gratitude to the people that helped and encouraged me during my early studies. In particular, Jens Studsgaard deserves recognition for inspiring me to study computer science, and Olivier Danvy deserves thanks for guiding me during my bachelor and master studies and encouraging me to apply for the PhD studies.

Finally, a big thanks goes to my friends and family who have supported me during my studies. I thank you all for taking an interest in my PhD project by asking questions about it whether or not you believed the answer would make sense to you. No PhD student is an island; in my case that is all thanks to you.

Lau Skorstengaard,
Aarhus, Friday 30th August, 2019.

Contents

Part I

Overview

Chapter 1

Introduction

Capability machines are low-level machines with built-in fine-grained access control. Capabilities are unforgeable tokens of authority that replace pointers on a capability machine; every operation on a capability machine is subject to a dynamic capability authority check. Capability machines have been around for decades, but recently they have seen an increased interest with the increased focus on security on computers. Security issues are often subtle, so to ensure security the enforcement must be 100% watertight. The only way to be certain that security enforcement cannot be circumvented is to formally prove it. This dissertation shows that security properties can be enforced on capability machines in a provably secure way.

This dissertation is comprised of two papers that formalises two capability machines and show how they can provably enforce security properties of high-level languages. In this introduction, we provide intuition about the general notion of capabilities. The introduction is somewhat informal in its presentation whereas the papers present capability machines formally.

The remainder of the introduction consists of three sections: an introduction to capabilities, two ways to motivate the dissertation, and an outline of the dissertation. Capabilities exist as a concept on all abstraction layers in a computer. The introduction to capabilities, Section 1.1, contains a general capability characterisation that fits the concept of capability found on every layer. Section 1.1 also presents capabilities on different abstraction layers. Specifically, the section presents capability machines, operating system capabilities, and object capabilities. In Section 1.2, we motivate the dissertation from two angles. From one angle, the goal is to study the essences of the capabilities on capability machines through formalisation. The other angle is secure compilation. Capability machines provide security primitives that make them good candidates for targets of secure compilations. Section 1.3 contains an outline of the dissertation. It provides some background for the papers contained in the dissertation and outlines their contributions.

1.1 Capabilities

The notion of a capability was formally introduced by Dennis and Van Horn [32] in 1966. Since then, the term capability has been used in different areas of computer science making it a somewhat ambiguous term. Capabilities exist on many of the abstraction layers of a computer, and it means something different for each layer. However, we can isolate some common features of all kinds of capabilities. First of all, capabilities carry some authority that permits the holder to perform some operation. The kind of operation a capability authorises depends on the context the capabilities exist in. In some sense, the authority of a capability is a local property as a capability authority check only involves the capability and can be done independently of the rest of the system. In other words, a capability cannot put restrictions on the remainder of the system. For instance, a capability can give authority to access a file. However, a capability cannot enforce that *only* you have access to the file as that would be a restriction dependent on the rest of the system. This is also a practical matter as authority checks happen dynamically, and it would be an insurmountable task to check that no other capability in a system gives access to a file.

Generally speaking, capabilities are used to restrict authority, so it is important that capability integrity is maintained. To achieve this, capabilities cannot be fabricated which means that authority cannot be generated out of thin air. In other words, authority has to be bestowed as it cannot be made up. We do not give a precise definition of the notion of a capability, but considering the above common characteristics of capabilities, we give the following characterisation:

> A capability is an unforgeable token of authority.

To illustrate how well this characterisation fits the notion of capabilities, consider the following three places where capabilities exist. On the very lowest layer, the machine itself, capabilities can be used as memory access control. Machine level capabilities carry the authority to access memory, and they are the only way to access memory. Furthermore, capabilities are tracked on the machine either by reserving a bit in every word that tags it as a capability or in a separate tagging table. In either case, the tag is enforced by the machine; it cannot be directly manipulated. When capabilities are moved around in memory, the capability tag is dynamically updated by the machine. We often refer to such capabilities as memory capabilities. Memory capabilities fit very well with the capability characterisation; they give the authority to manipulate memory, and they cannot be forged due to hardware enforcement.

In the context of operating systems, capabilities exist as an alternative way of handling access control. When using capabilities, you keep track of the resources each user has access to (as opposed to access control lists

where for each resource (files, printers, etc.), you keep track of who may access it). In other words, the capability list keeps track of a users authority. Access control is handled by the operating system, and the operating system protects the capability lists, so they are only manipulated by privileged users. Again, the concept of a capability list fits the capability characterisation fairly well. The capability list contains a number of capabilities that each provide some authority over the system (e.g. memory access and hardware resources), and the operating system makes sure that the authority cannot be forged.

For programming languages, the notion of capabilities occur in conjunction with object references. Object-oriented programming languages have the notion of objects: a coupling of state, usually expressed as variables, and behaviour, defined in methods. Object instances are accessed through object references that cannot be circumvented. Object capabilities are essentially object references; in fact, it can be difficult to tell the difference. Proper object capabilities fit the capability characterisation as they provide authority to access the object and cannot be forged.

Capabilities have one crucial thing in common; they support the *principle of least privilege* (POLP) [77]. POLP says that a program should only have the amount of authority necessary to perform its purpose. When programs have too much authority, they tend to be vulnerable to subtle security issues. This is well illustrated by *The Confused Deputy* story [45], where a compiler has ambient access to a file system and there happen to be important billing information stored. A malicious user that does not have access to the sensitive parts of the file system asks the compiler to store the debug information where the billing information is stored. The operating system has no way to check the intention of a program, so the compiler is allowed to overwrite the file. This is an example of a privilege escalation attack. The compiler unwillingly uses its authority to act on the user's behalf escalating the user's privilege. The question is who is at fault here? Hardy [45] argues that no one is in fault; the issue is the fundamental way access control is handled in the system. If the system had used capabilities instead, then the compiler could have required the user to provide the debug location in terms of a capability. The malicious user does not have a capability for the sensitive part of the file system, in particular the billing information, so they cannot overwrite it. The Confused Deputy story illustrates an important shortcoming in conventional systems, namely that use of authority can happen unintentionally. In a capability system, all actions have to be intentional as you need to explicitly provide the authority for the action. For instance, if you want to write to a file, then it is not sufficient to specify the path of the file; you must specify the authority to write to the file, in terms of a capability. Conceptually, you show more intention to perform an operation when you specify the authority compared to when you rely on ambient authority. The intentional programming that comes with capabilities supports POLP very well. A program gets

access to the minimal set of capabilities necessary to perform its task, and it can explicitly use them limiting the risk of unintentional use of authority.

In the following subsections, we will describe the use of capabilities on different abstraction layers to make it clear what they look like in practice. Specifically, we will describe capability machines that use memory capabilities, operating systems that use capabilities for access control, and programming languages that uses object capabilities.

1.1.1 Capability Machines

Capability machines have been around for decades. Since Dennis and Van Horn [32] introduced capabilities, there have been many proposals for machines with native capabilities. For instance, early capability machines include The Chicago Magic Number Computer [103], the Plessey System 250 [39], the Cambridge CAP Computer [63], the IBM System/38 [46], and the Intel iAPX 432 [7]. They vary in a number of ways including how the capabilities are represented on the machine and how they can be used. For a technical presentation of these machines, we refer to Levy [60] that also describe some of the early capability based operating systems. The technical details of specific capability machines are beyond the scope of this dissertation. In the following, we will describe capability machines in broad terms focusing on the common aspects of capabilities. Further, we will relate this to the two capability machines used as inspiration for the formalisations of capability machines used in this dissertation, namely the M-Machine [29] and the CHERI processor [69].

In its essence, a capability machine is almost the same as a normal low-level machine with registers and memory. The main difference between the two is that capability machines replace pointers with capabilities. The capabilities represent authority to access part of memory (or some machine resource), and all machine operations are subject to a dynamic authority check that makes sure that valid capabilities for the operation are used.

A memory capability is represented by a range of addresses $[b, e]$ that it gives authority over, a set of permissions P that describe the operations that can be performed on the range of addresses, and a current address c that the capability works on. The actual representation of a capability on a machine may differ from the above. For instance, the range may be represented as a base address and a length, and the current address may be an offset from the base address. Picking a capability representation is also a question of trade-offs. If you want a representation that takes up fewer bits, then you may need to compromise on what is representable. For instance, one of the good properties of capabilities is the fine-grained access control they provide; access to memory ranges can be specified down to particular addresses. However, if you are willing to compromise on the granularity of memory ranges, then you can save a few bits and get a more compact capability representation.

For instance, you can leave out the least significant bit of the range authority representation if you require the range to be aligned in memory.

The instructions on a capability machine are, for the most part, the same as on a normal machine, but they need to be capability aware. That is, the machine instructions must be executed with capabilities that grants the authority to perform the operation. The machine checks that the supplied capability works within its range of authority and that it has the necessary permission to perform the operation. For instance, take a store instruction `store r1 r2`. The instruction takes the content of source register `r2` and stores it to the memory where the target register `r1` points to. In order for the store instruction to succeed, a capability with authority to store to memory must be provided. That is, register `r1` must contain a capability that points within its range of authority and with write permission. The store only succeeds if the authority check does. Capabilities cannot be modified arbitrarily like normal data. For instance, the range of authority cannot be increased. To support this, a capability machine must provide some way to manipulate capabilities. Generally speaking, modification of capabilities must be monotonically decreasing in authority. In other words, it must not be possible to gain authority that you did not already have. For instance, an instruction that changes the range of authority of a capability must make sure that the range of authority of the new capability is a subset of the old range of authority. Alternatively, a capability machine may have an instruction that sets the capability flag for a piece of data with the format of a capability. To protect capability integrity, a new capability constructed by setting the capability flag must still be derived from a capability. This means that a data-to-capability instruction must take two arguments: a piece of data and a capability that the new capability can be derived from.

Capabilities must be unforgeable to be useful. This means that it must be possible to distinguish capabilities from data in memory. There are different approaches to achieve this. For instance, early capability machines had disjoint memory for capabilities and data. This also meant that capabilities could not reside as part of data structures. If you think of capabilities as a replacement of pointers, then this is an issue as many data structures (e.g. linked lists) require pointers to be stored in memory. It is, however, not a complete deal breaker as the data structure could store a number that indicates what capability should be used. Another approach to ensure capability integrity is tagging. In this approach, all words have a tag that tells whether it is a capability. The tag is maintained and protected by the machine. When a capability is moved around in memory, the tag is automatically set, and modification of the tag is highly restricted. The tag can either be part of the capability representation, or it can be stored in an efficient, protected tagging table that keeps track of the capabilities. The in memory tagging is also a place open for compromise. If every address in memory can hold a capability, and the capability representation takes up multiple words, then every

word will need the tag. However, if the locations a capability can reside in are limited to a specific alignment, then we only need a tag on the addresses that may contain a capability. These are, again, trade offs that capability hardware developers have to consider.

Computers often run many programs at the same time, and often the programs do not trust each other. To protect itself, a program needs some form of encapsulation mechanism that protects the internals of a program from direct access. However, a program may not want to be completely isolated from the outside world, and it may want to be able to communicate with the outside world in some fashion. Today, the operating system takes care of process isolation which in part solves this issue. The caveat is that all interaction has to go through the operating system. Switching between contexts is fairly expensive due to the coarse-grained memory protection of today's computers. Further, a program may want to execute a program that it does not trust in the same process which the coarse-grained memory protection does not support. On capability machines, the memory protection is fine-grained and suitable for executing mutually distrusting programs, but only if the machine has a capability encapsulation mechanism. To illustrate the necessity, consider what would happen in the interaction of two distrusting programs A and B. Say A is running, and it has a capability for calling B. A does not trust B, so A wants to make sure that B cannot access its internal capabilities. The question is where should A store the capabilities so B cannot access them. If A stores its capabilities in the register file, then B can access them after it is called. A could store the capabilities in the memory it has access to, but then the question is how A restores the capabilities when control is returned to A. A will have to give B a capability that B can use to return to A, so A would have to make its internal capabilities accessible from the return capability. However, this would also make A's internal capabilities accessible to B as B gets the return capability. If the return capability could be encapsulated, so it only can be used to return to A without giving authority to access the underlying memory, then A's internal capabilities could be stored safely. Without encapsulation, A cannot have a capability for calling B as A could abuse that to read B's internal capabilities. Encapsulation is an important aspect of a capability machine that must be implemented in order to have a capability machine that is reasonably versatile. As we will see when we describe the M-Machine and the CHERI processor, there have been different takes on implementing encapsulation on capability machines. The M-Machine, for instance, implements encapsulation with a new capability permission.

The permission of a capability specifies what operations the capability grants authority to perform. The simplest kind of memory capability has the permissions we know from operating systems, i.e. read, write, and execute. However, capabilities come in many flavors sometimes defined by new permissions. For instance, the write permission may distinguish between write

permission for data and write permission for capabilities which gives much more fine grained control when handing out authority.

In the following sections, we will take a closer look at two capability machines, namely the M-Machine and the CHERI Processor. The two machines serve as inspiration for the capability machine formalisations found in this dissertation.

The M-Machine

The M-Machine [28, 29, 41] is a research multicomputer from the late 90's. It was used to experiment with architectural ideas which included capabilities. Carter et al. [24] describes the capability[1] memory addressing scheme used by the M-Machine. They motivate the usage of capabilities with context switches. Context switches protect the memory of a thread from being accessed by another thread. However, context switches are expensive, so they are best avoided. Capabilities offer an alternative to context switches as the memory can be protected by the capabilities.

The M-Machine uses a rather small capability representation that is 64 bits long with a 1 bit tag. The 64-bit length means that the capability representation fits within one word (assuming the capability tag is stored in a separate table). The capability representation uses 1 bit for a capability tag, 4 bits for permissions, 6 bits for the segment length, and 54 bits for addressing.

The capability permissions on the M-Machine feature standard memory capability permissions such as read, write, and execute. The permissions also include *enter* permission which is a simple yet effective way to get encapsulation. An enter capability can only be used in a jump operation. The jump turns the enter capability into an executable capability. All other operations on the capability, such as a load or an attempt to alter the address the capability points to, result in an error. The enter capability is used to setup entry points for programs forcing other programs to cross the security boundary to a program at certain points. When a program sets up a boundary with enter capabilities, it protects the internals of the program as the enter capability provides very limited authority.

The M-Machine uses a short capability representation at the cost of the granularity of the memory segments that can be represented. Specifically, the memory segment a capability governs must have a length that is a power of two. The length field of an M-machine capability specifies the exponent rather than the length. Further, to make room for addressing in the capability representation, the memory segments must be aligned in memory according to their length. The 54 bits addressing field consists of two elements: a segment indicator and an offset into the segment. The bit length of the two fields depends on the segment length. The offset takes up the base-2 logarithm of

[1] They call them *guarded pointers*.

the length of the segment, and the segment indicator takes up the rest. In other words, all addresses of a segment can be addressed, so the granularity tradeoff is that capabilities cannot address outside the segment it has authority over[2] and in terms of what segments can be specified.

The M-Machine has normal instructions such as move (`MOV`), load (`LD`), and (`JMP`) [29]. The instructions check the authority of the provided capabilities before they perform the operation. For instance, the load instruction generates an error value if the source register does not contain a capability or if the source register contains a capability with enter permission. Carter et al. [24] suggests that the M-Machine should have an instruction for restricting the permission of a capability (`RESTRICT`) and an instruction for reducing the length of the segment of a capability (`SUBSEG`). However, the M-Machine has neither of the two instructions as they are emulated using an instruction that can create capabilities from integers. An unrestricted integer to capability instruction breaks capability integrity, so it needs to be executed in privileged mode. Non-privileged software gets access to the `SUBSEG` and `RESTRICT` emulations through enter capabilities.

The CHERI Processor

CHERI [2] is an ongoing research project at the Computer Laboratory of the University of Cambridge. The goal is to construct a capability machine that could feasibly be used in real computers along with the necessary software to support it. The project recognises that it would be infeasible to scrap the many millions lines of existing programs to move to a new architecture, so they explore a hybrid capability system that supports existing programs that are not capability aware as well as programs that are. As the project explores both the hardware and software side, it consists of multiple sub-projects including the CHERI processor [69, 98], CheriBSD [100], and CHERI Clang/LLVM [3]. In the following, we sketch the CHERI processor.

Most existing programs have no concept of what a memory capability is which means that they will only use the instructions from a standard instruction set. Further, most programs do interact with the memory which, on a capability machine, means that programs must use memory capabilities to show that they have the authority to access the memory. However, programs compiled without capabilities in mind do not intentionally use capabilities which means that they will fail on memory accesses. In order to execute such programs, the CHERI processor must let legacy programs execute independently of the capability system. At the same time, the CHERI processor relies on the capabilities for the compartmentalisation that prevents programs from interfering with each other. The CHERI solution is

[2] Addressing outside a capability's range of authority may not seem important, but conceptually it is not a problem to point outside the segment as long as the memory is never addressed.

to let a program execute relatively to default capabilities. Specifically, the CHERI processor has a default data capability register which may contain a capability which all traditional loads and stores will be executed relatively to. Similarly, the program counter register is extended to a program-counter capability register that contains the capability that points to the executing program. This setup allows a capability aware operating system to set the capability registers before execution of a capability unaware program. The capability unaware program can execute as it usually would as long as it stays within the boundaries set by the default capabilities. In other words, the compartmentalisation enabled by the capabilities is still in force, and the program will never know that it executes on a capability machine unless it tries to escape its compartment. A program that attempts to escape its compartment is not well-behaved, so it is only reasonable that the program does not get to continue execution.

The CHERI processor supports a normal instruction set to support legacy programs, but it also needs to enable capability-aware programs to make full use of the capabilities. To this end, CHERI has a set of capability instructions in addition to the existing instruction. The capability instructions work directly on capabilities. For instance in addition to the normal load instruction, there is a capability load instruction `CL` which loads directly from the memory via a specified capability. The capability aware instructions also include instructions for modifying the capabilities (among others `CSetLen` for setting the length of a capability and `CSetOffset` for setting the capability offset) as well as projecting the fields of a capability (such as, `CGetLen` and `CGetOffset` that gets the length field and offset field respectively).

The capability representation in CHERI is 256-bit which allows the capabilities full granularity for 64-bit addressing. The representation uses 64 bits, respectively, for the base, length, and offset of the capability. 31 bits are used for permissions, 24 bits are used for the otype (we will explain the otype later), and 1 bit is used for a capability tag. This leaves 8 bits that are used for experimental features. The 256-bit capability representation is the ISA-level representation which means that it conceptually is the one the instructions use. However, the actual representation on the machine may be different. In fact, Woodruff et al. [101] presents new, compact capability representations for CHERI called CHERI Concentrate. CHERI Concentrate comes in several formats from a 128-bit representation down to an 18-bit representation.

The otype in the capability representation is part of the CHERI mechanism for capability encapsulation. The encapsulation mechanism seals capabilities, so they cannot be manipulated or used directly to access memory even when the unsealed version of the capability could. The sealing mechanism is inspired by object capabilities (described in more detail in Section 1.1.3). Namely, a sealed capability cannot be used on its own; it needs to be part of a pair. A pair consists of a code capability that corresponds to the methods of a class and the data capability that corresponds to the in-

stance specific data. The otype relates capabilities, so only capabilities with the same otype can be used together. More than two capabilities can have the same otype which conceptually corresponds to the fact that there can be multiple instances of the same class. The sealing mechanism only makes sense, if the ability to create sealed capabilities with a specific otype is restricted, otherwise one could just create a sealed capability with a specific otype in order to use another sealed capability in an unintended way. CHERI has a special `Permit_Seal` permission that allows a capability to be used for sealing with in the `CSeal` instruction. Specifically, a capability with the `Permit_Seal` set can be used to seal capabilities with the otypes within its range of authority. The `CSeal` instruction takes two capabilities: one that is going be sealed and one with the `Permit_Seal` permission. The capability is sealed with the otype that the `Permit_Seal`-capability currently points to. Sealed capabilities can be unsealed again with the `CUnseal` instruction. This instruction also requires a `Permit_Seal`-capability for the unsealing to be successful, so arbitrary sealed capabilities cannot be unsealed. The main way for unsealing capabilities is not unsealing; it is a call. A code and data capability pair corresponds to a method which can be called with the `CCall` instruction. The `CCall` (omitting some compartmentalisation details) pushes the current execution on a call stack, unseals the capability pair, and installs the two unsealed capabilities in appropriate registers. The execution continues guided by the code capability. The `CReturn` instruction returns from the call. In a recent version of CHERI [98], CHERI proposes to add a sealing mechanism similar to the enter capabilities of the M-Machine. Specifically, this sealing mechanism allows capabilities to be sealed with as "enter-sealed" without using a `Permit_Seal`-capability. The enter-like capability can be used in a normal jump which unseals the capability.

The extra bits in the capability representation leaves room for experimentation. For instance, CHERI has something called local capabilities [69] that provides a simple form of control-flow enforcement. The idea with local capabilities is to have a capability that can only be stored in a limited amount of places. Specifically, a local capability can only be stored via a capability with a special `Permit_Store_Local` permission. By being conservative when handing out `Permit_Store_Local` capabilities, the places local capabilities can be stored is severely limited. Further on CHERI, local capabilities cannot be passed in when the `CCall` instruction is executed which means that local capabilities are confined to the compartment they stem from. The stack capabilities on CHERI are local, so they cannot be accidentally leaked in a call. Non-local capabilities are called global, and they work the same way as without local capabilities.

In the above, we have described CHERI as though it is a processor. However, it is actually a protection model that describes how an existing architecture may be extended with capabilities. In fact, the CHERI project is implementing the CHERI protection model for multiple ISAs, and at the time

of writing CHERI-MIPS, CHERI-RISC-V, and CHERI-x86-64 are under development and at different stages of maturity.

1.1.2 Operating Systems and Capabilities

An operating system manages the hardware and software resources on a computer which includes making them available to services that need them. It also includes prohibiting access to resources for services without permission to use them. This is what we know as access control. Conceptually, an operating system keeps track of an access control matrix like the on in Figure 1.1. The access control matrix keeps track of what users or services has access to what resources and how they may access them. The access control matrix is usually fairly sparse, so it makes sense to only store a compressed version of it. One can either store the columns or the rows of the matrix. Storing the columns of the matrix corresponds to keeping track of who has access to each file. For instance in Figure 1.1 for File2, we would store that Edith and John has read and execute authority over the file. This approach is called *access-control list* (ACL) and is used by all major operating systems. The alternative is to store what each user is allowed to do, i.e. the rows of the access control matrix. For instance Figure 1.1 for Anne, we would store that they have read and write access to File1, read access to File4, and write access to File6. This approach is known as *capability lists* and is mostly used in operating systems designed with security in mind. In the end, the two approaches can represent the same access control matrices, so the main difference between them is what operations they support[3]. ACLs make it very easy to find out who has access to a resource as you simply look at the list for the resource. At the same time, it is difficult to find out what a user has access to as you need to go through the lists of all resources. Capability lists are in some sense dual to ACLs as capability lists make it easy to figure out what a user has access to but difficult to find out who has access to a given resource. In a system where authority is ambient, ACLs make it very easy to check whether an action is allowed as you simply check the ACLs of the resources involved in the action. However while ambient authority may seem convenient as you do not have to specify what authority you use, it is also the cause of security vulnerabilities as illustrated by The Confused Deputy Story [45]. Capability lists, on the other hand, are suited for the intentional approach where system calls take the authority as an argument. Specifically, a system that uses capability lists can require a user that performs a system call to specify the capability they want to use by its index in the capability list. The system can then easily check the specified capabilities against the desired operation. Even if the user has the authority to perform an operation, they can still opt to not use it.

[3] Miller et al. [66] argues that viewing capabilities as rows in the access control matrix is too naïve as it does not model how authority may change over time.

	File1	File2	File3	File4	File5	File6
Anne	RW			R		W
Edith	RW	RX	RW			W
John	R	RX				W

Figure 1.1: An access control matrix describes the authority of every user in a system. In a system that uses access control lists, each column is stored (the orange boxes). In a system that uses capability lists, each row is stored (the blue, rounded boxes).

On the surface, capability lists seem straight forward to implement, but when it comes to actual operating system implementations, there are many technical things to consider. For instance, how exactly are the capabilities stored, and what protects them from tampering? Another important question is how capabilities may propagate in the operating system? If capabilities can only be granted by a privileged user, then the benefits of a capability system is severely diminished. For instance if the compiler in The Confused Deputy Story was used in a capability system, then the user must be able to specify a capability for the compiler to use. In other words, users must be allowed to propagate capabilities at their own discretion which begs the question: how can capabilities be propagated? Can users give capabilities away and to whom? Can users take capabilities under the right circumstances?

The capabilities in an operating system can be modelled by a directed graph. The nodes of the graph are users and system resources, and the edges are the capabilities (annotated with permissions). Such a directed graph gives a momentary picture of the capabilities in the operating system, but the graph evolves over time as the capabilities propagate in the system. This means that a model of capabilities in an operating system must specify a set of rules that describe graph changes that correspond to the ways capabilities can propagate in the actual operating system. For instance, in the *take-grant model* [62] capability propagation follows four rules: grant, take, remove, and create. The *grant* rule allows a capability holder to give capabilities via its write capabilities. The *take* rule allows a user to take capabilities from a user it has a read capability for. The *remove* rule allows a capability to reduce the permissions of a capability (i.e., remove labels from an edge). If all permissions are removed, then the capability is removed. Finally, the *create* rule allows new subjects to be created. The creator receives a capability for the new subject with all relevant permissions. Formal models give an abstract specification of the capability propagation possibilities in an operating system. It also gives a model which security policies can be evaluated against. For instance, if a user in a system should never have a capability for a spe-

cific system resource, then this can be verified by checking whether the graph manipulation rules allow the user to get the capability or not.

Early operating systems with capabilities include CAL-TSS [58], Hydra [102], and StarOS [49] that are all described in Levy [60]. More recent operating systems with capabilities include CheriBSD [100], SAFE [25], seL4 [53], Capsicum [96] (inspiration for CHERI), Gnosis [43], and Eros [81].

seL4

seL4 [53] is a general purpose operating system that has been formally verified to be functionally correct [54]. The implementation of seL4 has been verified against its abstract specification in the theorem prover Isabelle/HOL. Despite being formally verified, seL4 still has many features expected of any reasonable operating system, e.g. virtual memory, threads, and inter process communication. Unlike standard operating systems, seL4 uses capabilities for access control. The seL4 capabilities are based on an extension of the take-grant model [38]. First of all, seL4 does not allow capabilities to be taken, so it has no take rule. Further, capabilities are immutable in seL4, so the remove rule can only remove capabilities entirely. The seL4 also has a revoke rule which allows a set of capabilities to be removed from the graph. This may seem redundant given the remove rule, but conceptually it captures the ability to remove all of the capabilities of a subsystem. In seL4 capabilities are created by casting an untyped capability to a new capability. In some sense, this designates an unused area in memory for a specific purpose which allows that part of memory to be used. This means that the create rule requires an untyped capability. The use of untyped capabilities means that all memory is specified by capabilities at all times (even though all memory may not be allocated).

The seL4 microkernel runs on multiple different platforms that all have in common that they are not capability machines. This means that the capability access control is entirely modelled and enforced in software. In order to prevent tampering, capabilities are stored in their own address space. The capabilities are stored in kernel objects called *Capability Nodes* (CNodes) that correspond to nodes in the capability graph model. A CNode is a list of capabilities (the edges in the graph model) which may point to other CNodes. All system calls in seL4 take capabilities as arguments to make sure that the requested operation is authorised. The capability arguments in a system call are specified by their index in the CNode. As the capabilities are stored in their own address space, the only way to manipulate capabilities is via CNode methods. Each thread in seL4 has a *capability space* (CSpace) which is a collection of CNodes that contain all the capabilities of the thread.

CheriBSD

CheriBSD [100] is an adaption of FreeBSD that adds support for the memory protection in CHERI. CHERI has native capabilities which CheriBSD relies on. In fact, CheriBSD is a capability aware FreeBSD in the sense that it knows about the CHERI capability hardware and makes sure to handle the capabilities appropriately when necessary (e.g. by setting up capabilities on thread creation and storing capabilities at context switches). The fact that CheriBSD runs on capability hardware means that the addition to FreeBSD to get CheriBSD are relatively small.

Capsicum

Capsicum [96] is an extension of UNIX that adds capability support to the existing operating system. Specifically, it allows the operating system to enter a capability mode where some of the normal UNIX properties are disabled. For instance, it severely restricts invocation of system calls and denies access to global namespaces. This can be used to easily sandbox applications. The capabilities in Capsicum are implemented as an extension to the existing file descriptors in UNIX, so it runs on normal hardware. With Capsicum, developers can gradually adopt the capability model. In contrast, operating systems that enforce use of capabilities everywhere prevent reuse and force developers to adapt existing applications to use capabilities before they can be used.

According to the CHERI homepage [2], Capsicum encountered a number of limitations in current CPU design that made compartmentalisation difficult. The CHERI project spawned as a direct consequence.

1.1.3 Object Capabilities

Object-oriented (OO) programming languages are commonplace today, and most programmers are familiar with them. A program written in an OO language puts abstractions on different parts of memory that specify what the memory represents. Further, it defines how the memory should be accessed which helps the programmer make sure that the abstraction is maintained. In an OO language, the programmer defines their memory abstractions in terms of classes. A class has fields that define the memory it needs and methods that define how the memory should be accessed. In order to use a class, the programmer instantiates its fields with actual data which results in an object of the class. Object references allow objects to know about each other and invoke methods of other objects. Without object-oriented programming, one has to keep track of what a section of memory is used for and make sure to always access it in a way that is consistent the use. In other words, an OO language lets the programmer worry about abstractions and concepts in their programs rather than how to layout memory and keep it consistent.

Object capabilities take the object paradigm one step further by adding a level of security to the design friendly object paradigm. Specifically, object references are replaced with object capabilities. Object references provide the means to access objects, but it is not part of the OO paradigm that object references are protected from being circumvented or forged, i.e. objects can be accessed without using an object reference. In a language with object capabilities, the object capability is not only the means to access an object, it also represents the authority to do so. The run-time system of an object capability language must enforce that objects are only accessible through an object capability and that object capabilities cannot be forged. For instance, it should not be possible to access the memory of an object directly, and it should not be possible to cast a memory address to an object capability. The difference between a reference and a capability may be subtle, but it is important for security.

Programs are usually not static, so it should be possible to propagate object capabilities. However, object capabilities should not be allowed to be arbitrarily propagated as it might undermine the security they provide. The programming language should only allow object capabilities to be propagated according to a set of simple rules that it can enforce. The objects and object capabilities are very well modelled by a graph which means that the rules for propagation can be defined in terms of graph mutation rules. Miller [67] suggests four rules for obtaining connectivity in the object capability model: initial condition, parenthood, endowment, and introduction. A program may be initialised with a predetermined set of capabilities which is covered by the initial condition. If an object creates another object, then the creator gets an object capability for the new object which is the parenthood rule. The creator of an object can also give some of its capabilities to a new object which is the endowment rule. Finally, the introduction rule allows objects to give object capabilities to an object that it has a capability for.

A programming language with object capabilities needs to enforce the capabilities. If the language is compiled to a capability machine such as CHERI, then there is native support for enforcement. On a CHERI processor, the sealing mechanism is designed specifically with object capabilities in mind. An object capability on CHERI consists of two sealed capabilities: a code and a data capability. Sealed code and data capabilities are coupled together by the otype. A code and data capability with the same otype can be used in an invocation which corresponds to a method call. The invocation unseals both capabilities placing the code capability in the program counter register and the data capability in a designated data capability register. The sealing mechanism makes sure that the data cannot be accessed in an unintended way protecting the abstraction in represents. Object capabilities can also be emulated on a machine with no native capabilities. In this case, the programming language's run-time system must enforce the object capabilities entirely in software. This may put certain restrictions on what

the language allows. For instance, it may only be possible to interact with programs compiled for the same run-time platform. Further, the system the run-time environment runs in must protect it from other programs running on the same machine. Neither of these requirements are unreasonable. In fact, Java has similar restrictions. The Java Virtual Machine (JVM) executes Java Byte Code, and we expect the operating system to make sure that other programs do not interfere with the JVM.

No major programming language has adopted object capabilities as a native feature of the language. Scala programmers can use the *ocaps* [5] library to work in an object capability model. There are also many smaller languages that use the object capability model, e.g., E [67], Joe-E [65], Caja [1], Pony [6, 27], and Newspeak [4, 22]. The object-capability model is not a goal in and of itself. Object-capability languages usually has object capabilities as a means to achieve some other security related goal of the language.

Joe-E

Java is an object-oriented programming language that falls short of being an object-capability language. For instance, programmers can circumvent access modifiers using the reflection library which means that fields private to a class are not necessarily inaccessible from outside the class. Another issue is the ambient authority of Java programs. Many library methods with side-effects on the outside world are implemented as static methods, so access cannot be restricted to them.

Joe-E [65] is an object-capability language implemented as a subset of Java. That is, Joe-E identifies the features of Java that breaks with the capability paradigm and removes them from Java. Java is widely adopted and known by many programmers. By making Joe-E a subset of Java, programmers that know the Java syntax need little introduction before they can start writing Joe-E programs. However, Joe-E is not meant to make existing Java code capability safe. Most existing Java programs would not be a Joe-E program because they are not written with capabilities in mind and make use of the features Joe-E prohibit.

Joe-E is implemented with a verifier that checks whether a program sticks to the Joe-E subset. This allows Joe-E to use the JVM for execution and rely on the formal semantics defined by Java. The verifier makes sure that the program only uses the capability safe features of Java. For instance, programs can only use a tamed version of the reflection API which does not allow the access modifiers to be circumvented. Joe-E also tames the standard library removing all ambient authority replacing it with capabilities. There are no default capabilities in Joe-E, so a program must be granted the capabilities it needs to perform side-effects on the outside world.

The advantage of Joe-E over Java is that Joe-E supports local safety reasoning better which makes it sufficient to look at a class in isolation when

arguing whether it is correct. In order to figure out whether a Java class is safe to use, it is necessary to consider the environment it is used in because another class may use reflection to break invariants on the class' fields. In Joe-E, it suffices to consider the class itself because all the unsafe features of Java are disabled. Further, Java does not support POLP (principle of least privilege) as a Java class has ambient authority. On the other hand, a Joe-E class has exactly the authority granted by the capabilities it has. This makes it easy to follow POLP and only give the program the authority it actually needs.

Caja

Most internet users probably trust the website of their local newspaper. However, even trusted websites may fetch content, like an ad, that they haven't authored and display it as part of their website. Such content may contain JavaScript which is executed when the page is loaded which in turn means that the browser may be executing malicious JavaScript code. Assuming that the browsers sandboxing is secure, the damage done by malicious JavaScript code is contained to the page, but the website may want to contain it further. This would require someway to limit what the fetched JavaScript has authority to do which is not necessarily an easy task and should arguably not be the responsibility of a website developer.

Caja [1] allows websites to instrument JavaScript code from an unknown origin with object capabilities before execution. This allows a website to limit the amount of objects in the DOM a piece of unknown JavaScript may access. Caja works something like this: First, the website specifies a security policy with the objects from the DOM that should be available. Then, Caja prepares the website by securing the objects specified by the security policy and making the rest unavailable. Next, the unknown JavaScript is retrieved from its source and given to the Caja compiler which instruments it with code that makes it respect the object capability model. Finally, the instrumented code can be executed on the website. Caja solves the issue limiting authority of a unknown content, so the website developer only has to come up with the correct security policy for their setting.

1.2 Capabilities in This Dissertation

The investigation of capability machines in this dissertation can be seen from different angles. In this section, we present two ways to motivate the work of this dissertation.

1.2.1 Formalising Capabilities

The typical informal conjecture about capability is that they make systems more secure (for some more or less formal notion of security). In order to test this conjecture, it is key to formalise capability systems and make it precise what secure means. For instance, the take-grant model is an example of a formal model of an operating system with capabilities. With this model, security is expressed in terms of security policies that the graph and all possible mutations of the graph must satisfy. Graph-based models of capabilities have been dominant as they naturally model the momentary structure of a capability system. Further, by defining rules for capability propagation the model also takes into account how the system may change over time. However, the graph-based model over-approximates the behaviour of the nodes in the graph because it assumes that all rules for capability propagation may be used on a node. This assumption is fine for operating systems where user behaviour is mostly unknown. However in the cases where behaviour is known, the behaviour can qualify a capability's authority further than the standard permissions. If we know for sure that a program never stores capabilities given to it, then we should not have to consider the case where it stores them. This is not captured by the graph-based reasoning approached, and it becomes unsatisfactory in settings where we need to rely on behaviour for security.

Devriese et al. [33] recognised the limitations of the graph-based approach and developed a model for a JavaScript-like language calculus λ_{JS} [44] with object capabilities. Their model is defined in terms of a logical relation, and it can be seen as a notion of capability safety as it expresses that programs have to respect certain invariants that captures program behaviour. They also show that the graph-based approach can be emulated by their logical relation.

As illustrated in this introduction, capabilities exist on many different abstraction layers, and we need to find models suitable for reasoning on each layer. This dissertation formalises memory capabilities and other types of capabilities on the machine level. Further, logical relations are presented that allow us to show non-trivial properties of programs running on a capability machine as well as properties of a calling convention. By formalising the capabilities and developing logical relations to reason about them, we also study the very essence of them as it must be captured by the logical relation.

1.2.2 Secure Compilation

The concept of a program can be viewed from multiple angles. To a computer a program is a series of instructions that it needs to follow. The instructions are detail intensive as it specifies what register, memory addresses and so on that are involved. On the other hand to a programmer, a program is the

product of expressing an idea or solving a problem. From the point of view of the programmer, a good programming language makes it easy to express the programmer's ideas. Modern programming languages achieve this by abstracting away registers and other hardware details, so programmers can express their ideas in terms of high-level abstractions. However, this means that there is a mismatch between the language a computer understands and the language programmers write programs in.

Compilation is one way to bridge the gap between high-level programming languages and low-level programming languages. A compiler translates a source program written in a high-level language with many abstractions to a detail-dense low-level program that the computer understands. In other words, programmers can write programs and execute their translation.

Most interesting programs do not execute in complete isolation; they interact with other programs running on the computer. However, some programs are malicious in nature and try to get well-meaning programs to reveal sensitive data or execute in an unintended way. At the surface level, a programmer may think that their programs are secure because of the abstractions in the high-level language. However, abstractions of high-level languages are seldom preserved in an enforceable way over compilation which means that programmers cannot rely on them for security.

Generally speaking, a secure compilation is a compilation that preserves some security property. In this dissertation, we consider a notion of secure compilation called *full-abstraction* [8]. A fully abstract compiler provably preserves (and reflects) the observational behaviour of a program in any possible context it may be executed in. This means that a fully abstract compiler must preserve the abstractions of the source program in the translated program. Full-abstraction has been studied extensively between high-level language [75]. In order for a compilation to be practically applicable, it must target a language that can be executed which most likely would be machine code. Further, a compilation that targets machine code needs to produce code that enforces the high-level abstractions of the source language. Mainstream computers do not seem to have the necessary security primitives to provably enforce most high-level language abstractions. In order to produce a real fully-abstract compiler, we need to consider what machine the compiler should target and what security primitives the machine needs to have to enforce the desired high-level abstractions.

There are other notions of secure compilation than full abstraction. For instance, Abate et al. [12] presents a number of alternatives. It is beyond the scope of this dissertation to discuss what notion of secure compilation we need.

Programmers often have an informal intuition about the language properties of their favourite language. For instance in a simple imperative programming language with functions and calls, one would think that a function call returns to the point immediately after the function. We call this

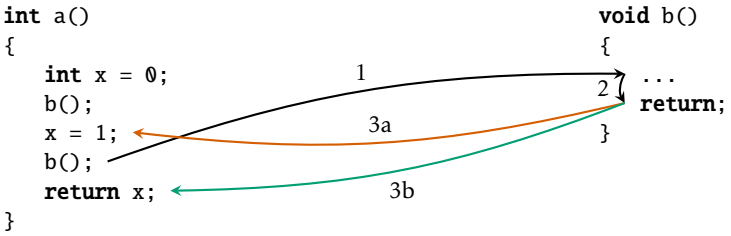


Figure 1.2: A well-bracketed control flow returns from a function call to the program point immediately after the function call. The arrows indicate two possible control flows in the C-like program. First function a calls function b (arrows 1 and 2). A well-bracketed call would return to the statement after the call (arrow 3b) whereas a non-well-bracketed call would return somewhere else, for instance after the first b call (arrow 3a). Local-state encapsulation means that function b cannot read function a's local variable x.

property well-bracketed control flow. We also expect that a function's local variables are inaccessible outside the context of the function. We call this property local-state encapsulation. The small C-like program in Figure 1.2 illustrates the two properties.

Most people's informal reasoning about their program's behaviour rely on well-bracketed control flow and local-state encapsulation. However, programs that interact with other programs not compiled from the same high-level programming language may not have the same abstractions and may thus ignore them. Further, a malicious handwritten low-level program may even deliberately try to break the properties. For some languages, unknown programs from the same language may be able to break abstractions we expect to hold. For instance, buffer overflow attacks or return oriented programming [80] have been used to alter the flow of a program. Compiled programs that interact with machine code of unknown origin have even less reason to believe that the high-level abstractions are respected. In order for a compiled program to actually rely on the high-level abstractions, it must be able to enforce them. Mitigation techniques are developed to preserve high-level abstractions, but often people come up with an attack that circumvents the mitigation. The problem is that modern computers do not supply security primitives that supports effective enforcement.

This dissertation shows that well-bracketed control flow and local-state encapsulation can be enforced on two different capability machines which supports capability machines as a plausible target for secure compilation. The dynamically checked capabilities are exactly what allows us to prove well-bracketed control flow and local-state encapsulation. In other words, capability machines have the necessary security primitives to enforce high-level language abstractions.

1.3 Dissertation Outline

This dissertation consists of two chapters, each based on a peer-reviewed paper. Both chapters are revised versions of the respective conference paper. The paper in Chapter 2 has been accepted for TOPLAS, and the paper in Chapter 3 is in preparation for a journal submission.

In this section, we give some background on how each of the papers came to be and outline the contents of them.

Familiarity with logical relations is assumed in both chapters. For readers new to logical relations, we recommend reading Skorstengaard [82].

1.3.1 Chapter 2: Reasoning about Capabilities

The chapter consists of the journal version of the conference paper Skorstengaard et al. [83].

Prior to this paper, Dominique Devriese and my supervisor, Lars Birkedal, had investigated reasoning techniques for object capabilities [33]. The paper had shown that a logical relation could be used to reason about object capabilities. Further, the logical relation could be used to state a new stronger notion of capability safety.

Observing the fact that capabilities exist on other abstraction layers, we decided to investigate whether a logical relation could be defined that would allow us to reason about capabilities on a capability machine. Our preliminary results were promising as we were able to define a simple model of an M-Machine-like capability machine with memory capabilities and an enter capability. While promising, we were not quite satisfied as the result seemed similar to Devriese et al. [33]. With inspiration from CHERI [69], we added local capabilities to our capability machine model and figured out how it changed the logical relation. With the addition of local capabilities, we decided to find example programs to prove correct that would show how good the logical relation was for reasoning. This exercise made it apparent how difficult and detail heavy it was to write even the simplest program. Moreover, in order to write programs that interacted with arbitrary machine code, we would have to come up with enforcement mechanisms that would protect the local state of the programs. Rather than making ad hoc enforcement, we decided to make a new calling convention that we believed would enforce well-bracketed control-flow and local-state encapsulation. We also defined a series of macros that made it easier to specify programs. With the macros we defined a series of example including a faithful translation of the Awkward Example [36, 76] as the flagship. Flaws in the calling convention were exposed as we tried to prove the correctness of the examples, so we modified it while we were proving the examples. In the end, we had a calling convention that allowed us to successfully prove the correctness of all the examples.

The result was a paper accepted at ESOP 2018 that presented the following:

- a formalisation of a simple yet somewhat realistic capability machine with local capabilities.
- a logical relation for reasoning about programs running on that capability machine.
- a calling convention that ensures well-bracketed control-flow and local-state encapsulation.
- a series of program examples that rely on the two properties along with proof of their correctness.

The paper was later extended to a journal version which has been accepted for ACM Transactions on Programming Languages and Systems (TOPLAS).

1.3.2 Chapter 3: StkTokens

The chapter consists of an extended version of a conference paper [86]. The extended version explains many of the details about the logical relation that were originally put in the technical report.

After the ESOP-paper, we evaluated the result and came up with two things that could be better. First, the revocation of local capabilities added what we believed to be substantial overhead to every call. This overhead would probably be too much in practical settings. Second, we had proven that the calling convention worked on a handful of non-trivial examples. However, as proof of the calling convention ensuring well-bracketed control-flow and local-state encapsulation, it lacked in generality.

Convinced that we could do better, we started a new project. The overhead we identified in the calling convention was inherent to the use of local capabilities, so we had to base the new calling convention on something else. Based on our experiences developing the ESOP calling convention, we knew that it was crucial to have a kind of revocable capability. In other words, we had to find something that could replace local capabilities. We ended up with linear capabilities (capabilities that cannot be duplicated). At the time we thought they were a novel discovery, but we later learned that they had been used on the CRASH/SAFE machine [25] and at the same time CHERI considered implementing a variation of them [98]. Nonetheless, using linear capabilities we came up with a new calling convention. For every call, the calling convention splits the stack in two, encapsulates one part, and gives the other part to the callee for them to use. The unencapsulated part of the stack also acts as a return token which the callee has to give back on return. This gave the calling convention the name StkTokens.

In order to prove that StkTokens enforces well-bracketed control flow and local-state encapsulation, we would first have to formally define the two properties. However, we were unable to come up with a good direct definition or find one in the literature. In the end, we started looking at full abstraction. After all, the two properties are abstractions in high-level languages, so what we think of as well-bracketed control flow and local state encapsulation can be defined in terms of the operational semantics of a programming language with a stack. In order to prove full abstraction for a compiler from a full high-level language to a capability machine, we would have to enforce all of the high-level abstraction. We decided to make a new capability machine with a built-in call stack that would otherwise be similar to the target machine. To begin with, the machine was not quite the same as it had variable length instructions and native call and return instructions. However, this was not a very promising setup as we could not come up with a back translation that we believed would work. The problem was that we did not know how to distinguish the translation of actual call and return instructions from instruction patterns that just happened to look like a call or return. In the end, we decided to keep the syntax of the target and source language the same and just change the operational semantics of the source machine to use a built-in stack when interpreting certain instruction patterns. This resulted in a trivial back-translation making the proofs easier. We called this an overlay semantics because it can be thought of as a mostly transparent overlay that keeps the machine the same except for a few cases where the overlay takes precedence.

StkTokens and the overlay semantics were developed in parallel. Sometimes we would change StkTokens because some details would be difficult to reason about. For instance to begin with, the capability machine had enter capabilities. Without going into details, enter capabilities only work if an activation record can be placed on the stack which in turn requires the stack to be executable. To make reasoning simpler, we wanted the machine to respect write-XOR-execute. However, this meant that the stack could not be writable and executable at the same time. Therefore, we replaced the enter capabilities with a CHERI-inspired sealing mechanism [69] which solved the problem.

The results were presented at POPL 2019. The paper included the following thing:

- a formalisation of a capability machine with linear capabilities and an overlay semantics for it that adds a built-in stack.
- the StkTokens calling convention that enforces well-bracketed control flow and local state encapsulation.
- a full abstraction theorem that formally proves the properties of StkTokens.

The version of the paper in this dissertation is in preparation for a journal submission.

1.4 Future Work

The ultimate goal of this line of work is to have a secure compiler from a real high-level language to a capability machine. This is, however, still far out in the future; the work in this dissertation is a small step on the path towards that goal. In the following we describe possible future work in this line of research.

The result in Chapter 3 is formulated in terms of full-abstraction. However, it is not given that full-abstraction is the notion of secure compilation we should strive for. Abate et al. [12] describe a hierarchy of trace-based secure compilation properties where full-abstraction is a relatively weak property. In future work, we could investigate whether the result in Chapter 3 can be lifted to stronger notion of secure compilation.

The proofs in the technical reports for this thesis (Skorstengaard et al. [84] and Skorstengaard et al. [85]) are very detail dense and possibly at the limit of what can reasonably be done by hand. The detail density seems to be inherent to proving properties of low-level machines as instruction languages are detail dense. In order to keep track of these details and make sure that everything is checked, the reasoning should be aided by a computer. In other words, the proofs should be mechanised. Further, many proof cases are similar or even trivial which suggest that they could be automated. Most proof assistants offer support for automation, so it would be natural to investigate possibilities for proof automation along the development of mechanised proofs. The worlds in the logical relations of this dissertation are constructed from scratch. The world construction techniques we use are so well-studied that they are built into the program logic frameworks Iris which even has tool support for working with the constructed worlds. As such, there is nothing special about the worlds in this dissertation, so it is plausible that the work could be done in Iris Coq [56] which would both mechanise the proofs and support easier world construction.

This dissertation presents two enforcement mechanisms for well-bracketed control-flow and local-state encapsulation. However, high-level programming languages have other abstractions, e.g. address-hiding references (references that do not expose the underlying memory address), that must also be enforced by a secure compiler (assuming a notion of secure compilation preserves and reflects abstractions). In future work, we want to investigate enforcement of other abstractions on capability machines. It is not given that capability machines provide the necessary kinds of capabilities for enforcing all high-level language abstraction. In these cases, this work would include coming up with new capabilities that on the one hand are strong enough to

enforce the property and on the other hand can be realistically implemented on real capability machines.

When we have more enforcement mechanisms, the next step is to make a proof-of-concept secure compiler from a simple high-level programming language with multiple realistic abstractions to a capability machine. The purpose of this line of work would be to investigate how one would go about proving a larger secure compiler. Specifically, we would investigate whether the fully-abstract overlay semantics (or an equivalent technique for a different notion of secure compilation) would support a somewhat simplified reasoning by considering each abstraction at some level of isolation.

All the above work would provide the necessary foundations to start the project of building a real secure compiler at the scale of CompCert [59].

Part II

Publications

Chapter 2

Reasoning About a Machine with Local Capabilities - Provably Safe Stack and Return Pointer Management

This is the journal version of the conference paper

> Lau Skorstengaard, Dominique Devriese, and Lars Birkedal.
>
> Reasoning about a machine with local capabilities.
>
> In *Programming Languages and Systems*. Springer International Publishing, 2018

This journal version has been accepted for Programming Languages and Systems (TOPLAS). The new contributions of the journal version are outlined in the introduction of the paper.

Abstract

Capability machines provide security guarantees at machine level which makes them an interesting target for secure compilation schemes that provably enforce properties such as control-flow correctness and encapsulation of local state. We provide a formalization of a representative capability machine with local capabilities and study a novel calling convention. We provide a logical relation that semantically captures the guarantees provided by the hardware (a form of capability safety) and use it to prove control-flow correctness and encapsulation of local state. The logical relation is not specific to our calling convention and can be used to reason about arbitrary programs.

2.1 Introduction

Compromising software security is often based on attacks that break programming language properties relied upon by software authors such as control-flow correctness, local-state encapsulation, etc. Commodity processors offer little support for defending against such attacks: they offer security primitives with only coarse-grained memory protection and limited compartmentalization scalability. As a result, defenses against attacks on control-flow correctness and local-state encapsulation are either limited to mitigation of only certain common forms of attacks (leading to an attack-defense arms race [90]) and/or rely on techniques like machine code rewriting [10, 94], machine code verification [68], virtual machines with a native stack [61] or randomization [42]. The latter techniques essentially emulate protection techniques on existing hardware at the cost of performance, system complexity, and/or security.

Capability machines are a type of processors that remediate these limitations with a better security model at the hardware level. They are based on old ideas [24, 32, 81] that have recently received renewed interest. In particular, the CHERI project has proposed new ideas and ways of tackling practical challenges like backwards compatibility and realistic OS support [69, 100]. Capability machines tag every word (in the register file and in memory) to enforce a strict separation between numbers and capabilities (a kind of pointers that carry authority). Memory capabilities carry the authority to read and/or write to a range of memory locations. There is also a form of *object capabilities*, which represent the authority to invoke a piece of code without exposing the code's encapsulated private state (e.g., the M-Machine's enter capabilities (described in Section 2.2) or CHERI's sealed code/data pairs).

Unlike commodity processors, capability machines lend themselves well to enforcing local-state encapsulation. Potentially, they will enable compilation schemes that enforce this property in an efficient but also 100% watertight way (ideally evidenced by a mathematical proof, guaranteeing that we do not end up in a new attack-defense arms race). However, a lot needs to happen before we get there. For example, it is far from trivial to devise a compilation scheme adapted to the details of a specific source language's notion of encapsulation (e.g., private member variables in OO languages often behave quite differently than private state in ML-like languages). And even if such a scheme were defined, a formal proof depends on a formalization of the encapsulation provided by the capability machine at hand.

A similar problem is the enforcement of control-flow correctness on capability machines. An interesting approach is taken in CheriBSD [69]: the standard contiguous C stack is split into a central, trusted stack and disjoint, private, per-compartment stacks. The trusted stack is managed by trusted call and return instructions. To prevent illegal use of stack references, the approach relies on *local capabilities*, a type of capabilities offered by CHERI to

temporarily relinquish authority, namely for the duration of a function invocation whereafter the capability can be revoked. However, details are scarce (how does it work precisely? what features are supported?) and a lot remains to be investigated (e.g., combining disjoint stacks with cross-domain function pointers seems like it will scale poorly to large numbers of components). Finally, there is no argument that the approach is watertight and it is not even clear what security property is targeted exactly.

In this paper, we make two main contributions: (1) an alternative calling convention that uses local capabilities to enforce stack frame encapsulation and well-bracketed control flow, and (2) perhaps more importantly, we adapt and apply the well-studied techniques of step-indexed Kripke logical relations for reasoning about code on a representative capability machine with local capabilities in general and correctness and security of the calling convention in particular. More specifically, we make the following contributions:

- We formalize a simple but representative capability machine featuring local capabilities and its operational semantics (Section 2.2).

- We define a novel calling convention enforcing control-flow correctness and encapsulation of stack frames (Section 2.3). It relies solely on local capabilities and does not require OS support (like a trusted stack or call/return instructions). It supports higher-order cross-component calls (e.g., cross-component function pointers) and can be efficient assuming only one additional piece of processor support (w.r.t. CHERI): an efficient instruction for clearing a range of memory.

- We present a novel step-indexed Kripke logical relation for reasoning about programs on the capability machine. It is an untyped logical relation, inspired by previous work on object capabilities [33]. We prove an analogue of the standard fundamental theorem of logical relations — to the best of our knowledge, our theorem is the most general and powerful formulation of the formal guarantees offered by a capability machine (a form of capability safety [33, 64]), including the specific guarantees offered for local capabilities. It is very general and not tied to our calling convention or a specific way of using the system's capabilities. We are the first to apply these techniques for reasoning about capability machines and we believe they will prove useful for many other purposes than our calling convention.

- We introduce two novel technical ideas in the unary, step-indexed Kripke logical relation used to formulate the above theorem: the use of a *single* orthogonal closure (rather than the earlier used biorthogonal closure) and a variant of Dreyer et al. [36]'s public and private future

worlds [36] to express the special nature of local capabilities. The logical relation and the fundamental theorem expressing capability safety are presented in Section 2.4.

- We demonstrate our results by applying them to challenging examples, specifically constructed to demonstrate local-state encapsulation and control-flow correctness guarantees in the presence of cross-component function pointers (Section 2.8). The examples demonstrate both the power of our formulation of capability safety and our calling convention.

This paper is an extension of the published conference paper Skorstengaard et al. [83]. We have made improvements to readability and completeness throughout the paper. For instance, we have added an introduction to Section Logical Relation that provides informal intuition about how the logical relation machinery comes into play on a capability machine. We highlight the following changes:

- We have added proof sketches for the Fundamental Theorem and for the correctness lemma for the awkward example (Lemma 2.8.4).
- We have added figures that illustrate central parts of the calling convention.
- A section about malloc has been added. Specifically, we provide the specification for the malloc used in the examples of the paper.
- A section about macro instructions with descriptions of all the macros and the implementation of `scall` has been added.
- We have added a section on reasoning about programs that run on a capability machine. This section explains how one reasons about common scenarios that arise in programs on a capability machine, and, in particular, how the logical relation is used. It also introduces a number of lemmas that prove recurring bits once and for all.
- We have expanded on the explanation of the awkward example.
- Details previously found only in the technical appendix [84] have been moved to the paper.

We have written a technical appendix [84] which contains additional details and proofs left out from this paper.

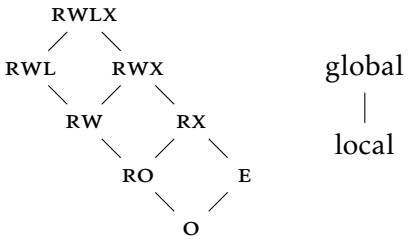


Figure 2.1: Permission and locality hierarchy.

2.2 A Capability Machine with Local Capabilities

In this paper, we work with a formalization of a capability machine with all the characteristics of real capability machines as well as local capabilities much like CHERI's. Otherwise, it is kept as simple as possible. It is inspired by both the M-Machine [24] and CHERI [69]. For simplicity, we assume an infinite address space and unbounded integers (see Section 2.9 for a discussion of these assumptions).

We define the syntax of our capability machine in Figure 2.2. We assume an infinite set of addresses Addr and define machine words as either integers or capabilities of the form $((perm, g), base, end, a)$. Such a capability represents the authority to execute permissions *perm* on the memory range $[base, end]$, together with a current address a and a locality tag g indicating whether the capability is global or local. On a capability machine, there is no notion of pointers other than capabilities, so we will use the terms interchangeably. The available permissions are null permission (O), readonly (RO), read/write (RW), read/execute (RX), and read/write/execute (RWX) permissions. Additionally, there are three special permissions: read/write-local (RWL), read/write-local/execute (RWLX) and enter (E), which we will explain below. The orderings of permissions and locality are displayed in Figure 2.1. The write permission is subsumed by the write-local permission, i.e. a write-local capability can be used to store both local and global capabilities to memory which is why write-local is above write in the permission hierarchy. We denote the pairwise ordering of permission and locality with $\sqsubseteq$.

We assume a finite set of register names RegName. We define register files *reg* and memories *ms* as functions mapping register names resp. addresses to words. The state of the entire machine is represented as a configuration that is either a running state $\Phi \in \text{ExecConf}$ containing a memory and a register file or a failed or halted state where the latter is paired with the final state of memory.

The machine's instruction set is rather basic. Instructions i include relatively standard jump (`jmp`), conditional jump (`jnz`), and move (`move`, copies words between registers) instructions. Also familiar are load and store instructions for reading from and writing to memory (`load` and `store`) and

arithmetic operators (`lt` (less than), `plus` and `minus`, operating only on numbers). There are three instructions for modifying capabilities: `lea` (modifies the current address), `restrict` (modifies the permission and local/global tag), and `subseg` (modifies the range of a capability). Importantly, these instructions take care that the resulting capability always carries less authority than the original (e.g. `restrict` will only weaken a permission according to the hierarchy in Figure 2.1). Finally, the instruction `isptr` tests whether a word is a capability or a number and instructions `getp`, `getl`, `getb`, `gete` and `geta` provide access to a capability's permissions, local/global tag, base, end and current address, respectively.

Figure 2.3 shows the step relation of the operational semantics. Essentially, a configuration Φ either decodes and executes the instruction pointed to by Φ.reg(pc) if it is an executable capability with its address in the valid range; otherwise it fails. If the configuration takes a step, it does so according to the interpretation of the instruction the program counter points to. The interpretation of a handful of instructions is displayed in the table in Figure 2.4 The table in the figure shows for instructions i the result of executing them in configuration Φ. The instructions `fail` and `halt` obviously fail and halt respectively. `move` simply modifies the register file as requested and updates the pc to the next instruction using the meta-function *updPc*.

The `load` instruction loads the contents of the requested memory location into a register, but only if the capability has appropriate authority (i.e. read

$$
\begin{array}{rrcl}
a \in & \text{Addr} & \stackrel{\text{def}}{=} & \mathbb{N} \\
perm \in & \text{Perm} & ::= & \text{O} \mid \text{RO} \mid \text{RW} \mid \text{RWL} \mid \text{RX} \mid \text{E} \mid \text{RWX} \mid \text{RWLX} \\
g \in & \text{Global} & ::= & \text{global} \mid \text{local} \\
c \in & \text{Cap} & \stackrel{\text{def}}{=} & \{((perm, g), b, e, a) \mid b, a \in \text{Addr}, e \in \text{Addr} \cup \{\infty\}\} \\
w \in & \text{Word} & \stackrel{\text{def}}{=} & \mathbb{Z} + \text{Cap} \\
r \in & \text{RegName} & ::= & \text{pc} \mid r_0 \mid r_1 \mid \ldots \\
reg \in & \text{Reg} & \stackrel{\text{def}}{=} & \text{RegName} \to \text{Word} \\
m \in & \text{Mem} & \stackrel{\text{def}}{=} & \text{Addr} \to \text{Word} \\
\Phi \in & \text{ExecConf} & \stackrel{\text{def}}{=} & \text{Reg} \times \text{Mem} \\
ms \in & \text{MemSeg} & \stackrel{\text{def}}{=} & \text{Addr} \rightharpoonup \text{Word} \\
 & \text{Conf} & \stackrel{\text{def}}{=} & \text{ExecConf} + \{failed\} + \{halted\} \times \text{Mem}
\end{array}
$$

$$
\begin{array}{rcl}
r & \in & \mathbb{Z} + \text{RegName} \\
i & ::= & \texttt{jmp}\ r \mid \texttt{jnz}\ r\ r \mid \texttt{move}\ r\ r \mid \texttt{load}\ r\ r \mid \texttt{store}\ r\ r \mid \texttt{lt}\ r\ r\ r \mid \texttt{plus}\ r\ r\ r \mid \\
 & & \texttt{minus}\ r\ r\ r \mid \texttt{lea}\ r\ r \mid \texttt{restrict}\ r\ r \mid \texttt{subseg}\ r\ r\ r \mid \texttt{isptr}\ r\ r \mid \texttt{getl}\ r\ r \mid \\
 & & \texttt{getp}\ r\ r \mid \texttt{getb}\ r\ r \mid \texttt{gete}\ r\ r \mid \texttt{geta}\ r\ r \mid \texttt{fail} \mid \texttt{halt}
\end{array}
$$

Figure 2.2: The syntax of our capability machine assembly language.

$$\Phi \rightarrow \begin{cases} [\![decode(n)]\!](\Phi) & \text{if } \Phi.\text{reg}(\text{pc}) = ((perm, g), b, e, a) \text{ and } b \leq a \leq e \\ & \quad \text{and } perm \in \{\text{RX}, \text{RWX}, \text{RWLX}\} \text{ and } \Phi.\text{mem}(a) = n \\ failed & \text{otherwise} \end{cases}$$

$$updPc(\Phi) = \begin{cases} \Phi[\text{reg.pc} \mapsto newPc] & \text{if } \Phi.\text{reg}(\text{pc}) = ((perm, g), b, e, a) \\ & \quad \text{and } newPc = ((perm, g), b, e, a + 1) \\ failed & \text{otherwise} \end{cases}$$

Figure 2.3: The operational semantics step relation and a function for taking care of updating the program counter in each step.

permission and an appropriate range). The `restrict` instruction updates a capability's permissions and global/local tag in the register file, but only if the new permissions are weaker than the original according to the permission hierarchy in Figure 2.1. The `subseg` instruction reduces the range of authority of a capability. In order to represent the unbounded address space, we use -42 to represent infinity.

The `jmp` instruction updates the program counter to a requested location, but it is complicated by the presence of *enter capabilities* after the M-Machine's [24]. Enter capabilities cannot be used to read, write or execute and their address and range cannot be modified. They can only be used to jump to. When that happens, their permission changes to RX. They can be used to represent a kind of closures: an opaque package containing a piece of code together with local encapsulated state. Such a package can be built as an enter capability $c = ((\text{E}, g), b, e, a)$ where the range $[b, a-1]$ contains local state (data or capabilities) and $[a, e]$ contains instructions. The package is opaque to an adversary holding c. When c is jumped to however, the instructions can start executing and have access to the local data through the updated version of c, now in the pc-register.

The instruction `lea` manipulates the current address of non enter-capabilities. It is fine for a capability to have a current address outside its range of authority as long as it is not used with an instruction that requires a specific capability. The instruction `geta` queries the current address of a capability and stores it in a register.

Finally, the `store` instruction updates the memory to the argument value if the capability has write authority for the specified location. However, the instruction is complicated by the presence of *local capabilities* modeled after the ones in the CHERI processor [69]. At a high-level, local capabilities are special in that they can only be kept in registers, i.e. they cannot be stored to memory. This means that local capabilities can be *temporarily* given to an

i	$[\![i]\!](\Phi)$	Conditions
`fail`	*failed*	
`halt`	$(halted, \Phi.\text{mem})$	
`move` r_1 r_2	$updPc(\Phi[\text{reg}.r_1 \mapsto w])$	$r_2 \in \text{Reg} \Rightarrow w = \Phi.\text{reg}(r_2)$ and $r_2 \in \mathbb{Z} \Rightarrow w = r_2$
`load` r_1 r_2	$updPc(\Phi[\text{reg}.r_1 \mapsto w])$	$\Phi.\text{reg}(r_2) = ((perm, g), b, e, a)$ and $w = \Phi.\text{mem}(a)$ and $b \leq a \leq e$ and $perm \in \{\text{RWX}, \text{RWLX}, \text{RX}, \text{RW}, \text{RWL}, \text{RO}\}$
`restrict` r_1 r_2	$updPc(\Phi[\text{reg}.r_1 \mapsto w])$	$\Phi.\text{reg}(r_2) = ((perm, g), b, e, a)$ and $(perm', g') = decodePermPair(\Phi.\text{reg}(r_2))$ and $(perm', g') \sqsubseteq (perm, g)$ and $w = ((perm', g'), b, e, a)$
`subseg` r_1 r_2 r_3	$updPc(\Phi[\text{reg}.r_1 \mapsto w])$	$\Phi.\text{reg}(r_1) = ((perm, g), b, e, a)$ and for $i \in \{2, 3\}$ $n_i = \Phi.\text{reg}(r_i)$ and $n_2 \in \mathbb{N}$ and $b \leq n_2$ and $n_3 \leq e$ where either $n_3 \in \mathbb{N}$ or ($n_3 = -42$ and $e = \infty$) and $perm \neq \text{E}$ and $w = ((perm, g), n_2, n_3, a)$
`jmp` r	$\Phi[\text{reg.pc} \mapsto newPc]$	if $\Phi.\text{reg}(r) = ((\text{E}, g), b, e, a)$, then $newPc = ((\text{RX}, g), b, e, a)$ otherwise $newPc = \Phi.\text{reg}(r)$
`lea` r_1 r_2	$updPc(\Phi[\text{reg}.r_1 \mapsto c])$	$\Phi.\text{reg}(r_1) = ((perm, g), b, e, a)$ and $n = \Phi.\text{reg}(r_2)$ and $n \in \mathbb{Z}$ and $perm \neq \text{E}$ and $c = ((perm, g), b, e, a + n)$
`geta` r_1 r_2	$updPc(\Phi[\text{reg}.r_1 \mapsto a])$	$\Phi.\text{reg}(r_2) = ((\_, \_), \_, \_, a)$
`store` r_1 r_2	$updPc(\Phi[\text{mem}.a \mapsto w])$	$\Phi.\text{reg}(r_1) = ((perm, g), b, e, a)$ and $perm \in \{\text{RWX}, \text{RWLX}, \text{RW}, \text{RWL}\}$ and $b \leq a \leq e$ and $w = \Phi.\text{reg}(r_2)$ and if $w = ((\_, \text{local}), \_, \_, \_)$, then $perm \in \{\text{RWLX}, \text{RWL}\}$
...		
_	*failed*	otherwise

Figure 2.4: An excerpt from the operational semantics.

adversary, for the duration of an invocation. If we make sure to clear the capability from the register file after control is passed back to us, the adversary is unable to store the capability. However, there is one exception to the rule above: local capabilities can be stored to memory for which we have a capability with write-local authority (i.e. permission RWL or RWLX). This is intended to accommodate a stack where register contents can be stored, including local capabilities. As long as all capabilities with write-local authority are themselves local and the stack is cleared after control is passed back by the adversary, we will see that this does not break the intended behavior of local capabilities.

We point out that our local capabilities capture only a part of the semantics of local capabilities in CHERI. Specifically in addition to the above, CHERI's default implementation of the CCall exception handler forbids local capabilities from being passed across module boundaries. Such a restriction fundamentally breaks our calling convention since we pass around local return pointers and stack capabilities. However, CHERI's CCall is not implemented in hardware but in software precisely to allow experimenting with alternative models like ours.

In order to have a reasonably realistic system, we use a simple model of linking where a program has access to a linking table that contains capabilities for other programs. We also assume malloc to be part of the trusted computing base satisfying a certain specification. Malloc and linking tables are described further in the next section. The specification of malloc is presented in Section 2.5 as it uses the semantic model we build in Section 2.4. For full details on the linking table, we refer to the technical appendix [84].

2.3 Stack and Return Pointer Management Using Local Capabilities

One of the contributions in this paper is a demonstration that local capabilities on a capability machine support a calling convention that enforces control-flow correctness in a way that is provably watertight, potentially efficient, does not rely on a trusted central stack manager, and supports higher-order interfaces to an adversary, where an adversary is just some unknown piece of code. In this section, we explain the high-level approach of this calling convention. We motivate each security measures with a situation we want to avoid (motivating each measure separately with a summary table at the end). After that, we define a number of reusable macro-instructions that can be used to conveniently apply the proposed convention in subsequent examples.

The basic idea of our approach is simple: we stick to a single, rather standard, C stack and register-passed stack and return pointers much like a standard C calling convention. However, to prevent various ways of misusing

this basic scheme, we put local capabilities to work and take a number of not-always-obvious safety measures. The safety measures are presented in terms of what *we* need to do to protect ourselves against an *adversary*, but this is only for presentation purposes as our code assumes no special status on the machine. In fact, an adversary can apply the same safety measures to protect themselves against us. In the following paragraphs, we will explain the issues to be considered in all the relevant situations: when (1) starting our program, (2) returning to the adversary, (3) invoking the adversary, (4) returning from the adversary, (5) invoking an adversary callback, and (6) having a callback invoked by the adversary.

Program start-up We assume that the language runtime initializes the memory as follows: a contiguous array of memory is reserved for the stack, for which we receive a stack pointer in the register r_{stk}. We stress that the stack is not built-in, but merely an abstraction we put on this piece of the memory. The stack pointer is local and has RWLX permission. Note that this means that we will be placing and executing instructions on the stack. Crucially, the stack is the only part of memory for which the runtime (including malloc, loading, linking) will ever provide RWLX or RWL capabilities. Additionally, our examples typically also assume some memory to store instructions or static data. Another part of memory (called the heap) is initially governed by malloc and at program start-up, no other code has capabilities for this memory. Malloc hands out RWX capabilities for allocated regions as requested (no RWLX or RWL permissions). For simplicity, we assume that memory allocated through malloc cannot be freed.[1]

Returning to the adversary Perhaps the simplest situation is returning to the adversary after they invoked our code (Figure 2.5a). In this case, we have received a return pointer from them, and we just need to jump to it as usual. An obvious security measure to take care of is properly clearing the non-return-value registers before we jump (since they may contain data or capabilities that the adversary should not get access to). Additionally, we may have used the stack for various purposes (register spilling, storing local state when invoking other functions etc.), so we also need to clear that data before returning to the adversary (Figure 2.5b and Figure 2.5c).

However, if we are returning from a function that has itself invoked adversary code (Figure 2.5d), then clearing the used part of the stack is not enough. The *unused* part of the stack may also contain data and capabilities, left there by the adversary, including local capabilities since the stack is write-local. As we will see later, we rely on the fact that the adversary cannot keep hold of local capabilities when they pass control to the trusted code and receive control back. In this case, the adversary could use the unused part

[1] In more realistic settings, reusing freed memory on a capability machine can be made safe by checking or enforcing that there are no dangling pointers to the freed memory, as implemented, for example, in CHERI-JNI [26].

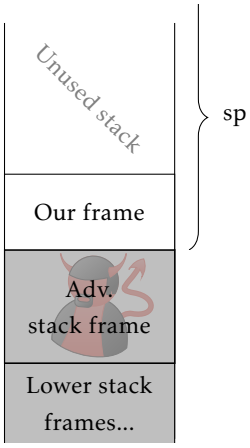
of the stack to store local pointers and load them from there after they get control back (Figure 2.5e). To prevent this, we need to clear (i.e. overwrite with zeros) the entire part of the stack that the adversary has had access to; not just the parts that we have used ourselves (Figure 2.5f). Since we may be talking about a large part of memory, this requirement is the most problematic aspect of our calling convention for performance (see the discussion in Section 2.9 for how this might be mitigated).

Invoking the adversary A slightly more complex case is invoking the adversary. As above, we clear all the non-argument registers, as well as the part of the stack that we are not using (because, as above, it may contain local capabilities from previously executed code that the adversary could exploit in the same way). We leave a copy of the stack pointer in r_{stk}, but only after we have used the subseg instruction to shrink its authority to the part that we are not using ourselves.

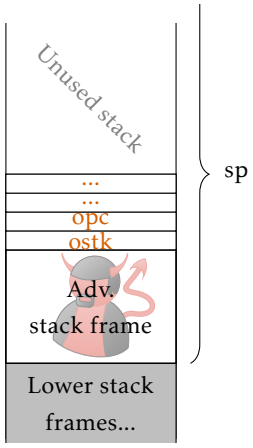
In one of the registers, we also provide a return pointer which must be a local capability. If it were global, the adversary would be able to store away the return pointer in a global data structure (i.e. there exists a global capability for it) and jump to it later in circumstances where this should not be possible. For example, they could store the return pointer, legally jump to it a first time, wait to be invoked again, and then jump to the old return pointer a second time instead of the new return pointer received for the second invocation. Similarly, they could store the return pointer, invoke a function in our code, wait for us to invoke them again, and then jump to the old return pointer rather than the new one received for the second invocation. By making the return pointer local, we prevent such attacks. The adversary can only store local capabilities with a write-local capability, and the only piece of memory governed by a write-local capability is the stack. Since the stack pointer itself is also local, it can also only be stored on the stack. There is no way for the adversary to recover either of these local capabilities because we clear the part of the stack that they had access to before we pass control back to them.

Note that storing stack pointers for use during future invocations would also be dangerous in itself, i.e. not just because it can be used to store return pointers. Imagine that the adversary stores their stack pointer (Figure 2.6a), invokes trusted code that uses part of the stack to store private data (Figure 2.6b) and then invokes the adversary again with a stack pointer restricted to exclude the part containing the private data (Figure 2.6c). If the adversary had a way of keeping hold of their old stack pointer, it could access the private data stored there by the trusted code and break local-state encapsulation.

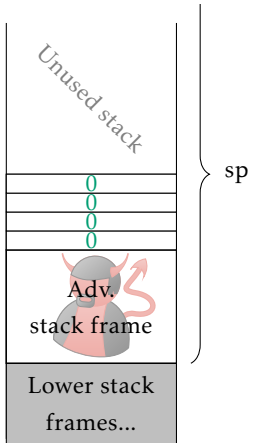
Returning from the adversary Return pointers must be passed as local capabilities. But what should their permissions be, what memory should they point to and what should that memory (the activation record) contain? Let us answer the last question first by considering what should happen



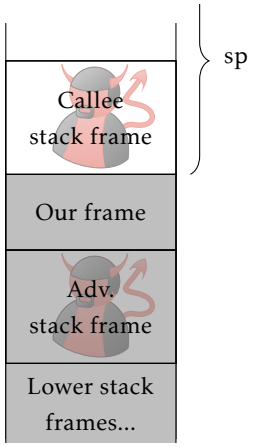
(a) Just before we return to the adversary. Our stack frame is the top-most, and we do not have access to the adversary stack frame.



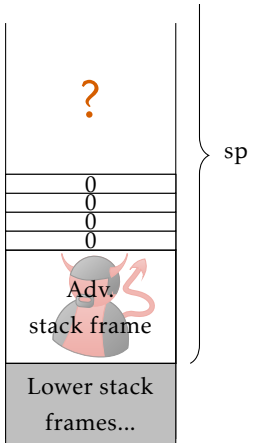
(b) Just after we returned to the adversary but did not take care to clear our local stack frame.



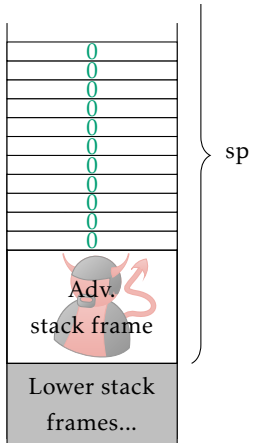
(c) Just after we returned to the adversary, and we did take care to clear our local stack frame.



(d) After an adversary has called us, and we have called the adversary.

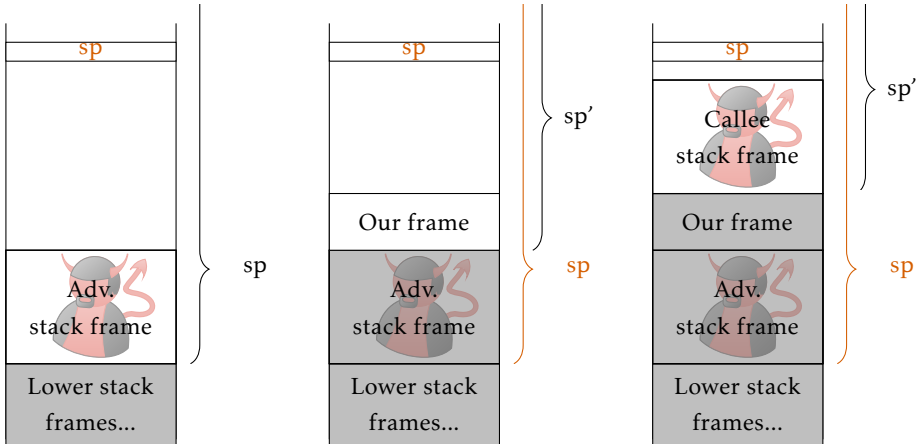


(e) After the adversary has returned to us, and we have returned from the first adversary call. At this point, the adversary has access to the local state from the nested adversary call.



(f) After the adversary has returned to us, and we have returned from the first adversary call after we took care to clear the unused part of the stack.

Figure 2.5: Depictions of the stack in relation to stack clearing. The greyed out areas are parts of the stack that are not accessible at the time. The sp capability is the stack pointer.



(a) The stack after the adversary has stored its stack pointer far up on the stack.

(b) The stack after the adversary has called some trusted code.

(c) The stack after the trusted code has called the adversary. The adversary's old stack pointer is still available on the stack.

Figure 2.6: Illustration of the situations related to stack clearing when invoking an adversary. The greyed out areas are parts of the stack that are not accessible at the time.

when the adversary jumps to a return pointer. In that case, the program counter should be restored to the instruction after the jump to the adversary, so the activation record should store this old program counter. Additionally, the stack pointer should also be restored to its original value. Since the adversary has a more restricted authority over the stack than the code making the call, we cannot hope to reconstruct the original stack pointer from the stack pointer owned by the adversary. Instead, it should be stored as part of the activation record.

Clearly, neither the old program counter nor the old stack pointer should be accessible by the adversary. In other words, the return pointer provided to the adversary must be a capability that they can jump to but not read from,

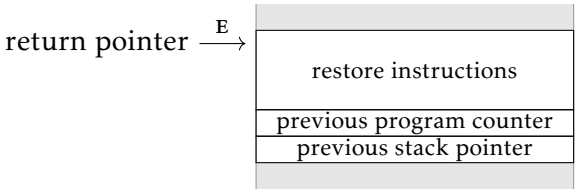


Figure 2.7: Structure of an activation record

i.e. an enter capability. To make this work, we construct the activation record as depicted in Figure 2.7. The E return pointer has authority over the entire activation record (containing the previous return and stack pointer) and its current address points to a number of restore instructions in the record. Upon invocation, the instructions in the activation record are executed and can load the old stack pointer and program counter back into the register file. As the return pointer is an enter pointer, the adversary cannot get hold of the activation record's contents. However after invocation, its permission is updated to RX, so the contents become available to the restore instructions.

The final question that remains is: where should we store this activation record? The attentive reader may already see that there is only one possibility: since the activation record contains the old stack pointer, which is local, the activation record can only be constructed in a part of memory where we have write-local access, i.e. on the stack. Note that this means we will be placing and executing instructions on the stack, i.e. it will not just contain code pointers and data. This means that our calling convention should be combined with protection against stack smashing attacks (i.e. buffer overflows on the stack overwriting activation records' contents). Luckily, the capability machine's fine-grained memory protection [100] should make it reasonably easy for a compiler to implement such protection by making sure that only appropriately bounded versions of the stack pointer are made available to source language code.

Invoking an adversary callback If we have a higher-order interface to the adversary, we may need to invoke an adversary callback. In this case, not so much changes with respect to the situation where we invoke static adversary code. The adversary can provide a callback as a capability for us to jump to, either an E-capability if they want to protect themselves from us or just an RX capability if they are not worried about that. However, there is one scenario that we need to prevent: if they construct the callback capability to point into the stack, it may contain local capabilities that they should not have access to upon invocation of the callback. As before, this includes return and stack pointers from previous stack frames that they may be trying to illegally use inside the callback.

To prevent this, we only accept callbacks from the adversary in the form of global capabilities, which we dynamically check before invoking them (and we fail otherwise). This should not be an overly strict requirement: our own callbacks do not contain local data themselves, so there should be no need for the adversary to construct callbacks on the stack.[2]

Having a callback invoked by the adversary The above leaves us with perhaps the hardest scenario: how to provide a callback to the adversary. The basic idea is that we allocate a block of memory using malloc that we

[2]Note that it does prevent a legitimate but non-essential scenario where the adversary wants to give us temporary access to a callback not allocated on the stack.

fill with the capabilities and data that the callback needs, as well as some prelude instructions that load the data into registers and jumps to the right code. Note that this implies that no local capabilities can be stored as part of a closure. We can then provide the adversary with an enter-capability covering the allocated block and pointing to the contained prelude instructions. However, the question that remains in this setup is: from where do we get a stack pointer when the callback is invoked?

Our answer is that the adversary should provide it to us; just as we provide them with a stack pointer when we invoke their code. However, it is important that we do not just accept any capability as a stack pointer but check that it is safe to use. Specifically, we check that it is indeed an RWLX capability. Without this check, an adversary could potentially get control over our local stack frame during a subsequent callback by passing us a local RWX capability to a global data structure instead of a proper stack pointer and a global callback for our callback to invoke. If our local state contains no local capabilities, then, otherwise following our calling convention, the callback would not fail and the adversary could use a stored capability for the global data structure to access our local state. To prevent this from happening, we need to make sure the stack capability carries RWLX authority since the system wide assumption then tells us that the adversary cannot have global capabilities to our local stack.

Calling convention With the security measures introduced and motivated, let us summarize our proposed calling convention:

At program start-up A local RWLX stack pointer resides in register r_{stk}. No global write-local capabilities.

Before returning to the adversary Clear non-return-value registers. Clear the part of the stack we had access to (not just the part we used).

Before invoking the adversary Push activation record to the stack. Create return pointer as local E-capability to the instructions in the record. Restrict the stack capability to the unused part and clear it. Clear non-argument registers.

Before invoking an adversary callback Make sure callback is global.

When invoked by an adversary Make sure received stack pointer has permission RWLX.

Modularity The calling convention ensures well-bracketed calls and local-state encapsulation for the caller but not the callee. In the above presentation to make it easy to distinguish between the parties involved, we present the callee as some adversarial code that we do not trust. In reality, the callee could be well-behaved and wish to ensure well-bracketed calls and local-state encapsulation as well. The calling convention puts no restriction on

the callee that the caller itself does not follow, so by following the calling convention, the callee can also obtain those guarantees. In other words, the calling convention is modular and scales to scenarios with multiple distrusting parties invoking each other.

2.4 Logical Relation

Now that we have defined our calling convention, how can we be sure that it works? More concretely, suppose that we have a program that uses the convention in its interaction with untrusted adversary code. Can we formally prove the program's correctness if it relies on well-bracketed control flow and private state encapsulation for the interaction with the adversary? Clearly, such a proof should depend on a formal expression of the guarantees provided by the capability machine, including the specific guarantees for local capabilities.

In this section, we construct such a formalization. We make use of some well-studied and powerful (but non-trivial) machinery from the literature. Specifically, we employ a unary step-indexed Kripke logical relation with recursive worlds and some additional special characteristics of our own. Step-indexing, Kripke logical relations, and recursive worlds are techniques that may be familiar from lambda calculus settings, but it may not be clear to the reader how they apply in this more low-level assembly language. Therefore, in the next section, we do not immediately dive into the details, but first we try to provide some informal intuition about how all of this machinery comes into play in our setting.

Note: even though the calling convention is the main application in this paper, the logical relation we construct is very general and can be seen as a formulation of capability safety; hence it should be regarded as an independent contribution.

2.4.1 Formalizing the guarantees of the capability machine

What differentiates a capability machine from a more standard assembly language is that we can bound the authority of an executing block of code based solely on the capabilities it has access to. Specifically, it does not matter which instructions are actually executed, i.e. the bound also applies to untrusted adversary code that has not been inspected or modified in any way.

Worlds But what does a "bound on the authority" of an executing block of code mean? In our setting, there are no externally observable side-effects; the only primitive authority that code may hold is authority over memory. As such, the authority bounds we consider are related to memory, but in a form that is more fine-grained than standard read/write authority: a piece

of code's authority can be bounded by arbitrary memory invariants that it is required to respect. Specifically, we will define worlds $W \in \text{World}$, which describe a set of memory invariants, and our results will express authority bounds on code as *safety with respect to such a world*, i.e. the fact that the code respects the invariants registered in the world.

Safe values Suppose we have a world W expressing that the memory must contain value 42 at address 0, may contain arbitrary values at addresses 50-60, a RW capability for address 0 at address 73, and an integer at address 100 that may only increase over time[3]. Our main theorem will state that if the current register file only contains safe words (numbers or capabilities which preserve the invariants in W under any interaction), then an execution will necessarily also preserve the memory invariants (irrespective of the instructions being executed).

To make this more precise, we need to define the set $\mathcal{V}(W) \in \mathcal{P}(\text{Word})$ of words that are safe w.r.t. W. Essentially, the set should only include words that preserve W's invariants under any interaction, but should otherwise be as liberal as possible. Numbers are clearly always safe as they cannot be used to break invariants. Whether a capability is safe depends on the authority that it carries. In the above-described world, a read capability for address 0 is safe as it can only be used to read the value 42, which is itself safe. However, a write capability for address 0 is not safe: it can be used to overwrite the memory at that address with a value other than 42 breaking the invariant for that address.

Step-indexing More generally, we want to define that a read capability for memory range $[b, e]$ is safe if the world guarantees that the words at those addresses are themselves safe. However, this definition is cyclic: suppose the world guarantees that the memory at address a will contain a read capability for address a? Then the definition says that a read capability for address a is safe if and only if the same read capability for address a is safe. This form of cyclic reasoning is related to similar challenges in languages with recursive types or higher-order ML-style references, and a standard solution is to use step-indexing [16]: essentially, the cycle is broken by defining safety up to a certain number of interaction steps. All words will be considered safe up to 0 steps (since if there is no interaction, nothing unsafe can happen), and, for example, a read capability will be safe up to n steps if the world guarantees that the words at the corresponding addresses are safe up to $n-1$ steps. We can then prove that the above read capability for address a is safe up to any number of steps.

[3] Indeed, we will allow a notion of *evolvable* invariants, aka *protocols*, that can express such a temporal property.

Future worlds Worlds are defined as a set of invariants on the memory, but what if we allocate fresh memory through malloc? We may want to establish new invariants for this freshly allocated memory and be sure that the adversary will also respect those (if we don't provide them with capabilities through which the new invariants can be broken). To accommodate this, we allow worlds to evolve, for example by adding additional invariants for freshly allocated memory. Formally, we define valid ways for a world W to evolve into a new world W' through a future-world relation $W' \sqsupseteq W$ and we ensure that the set of safe words in world W must remain safe in any future world W'. Defining safety w.r.t. a notion of evolvable worlds makes our logical relation into a *Kripke* logical relation [76].

Invariants and Recursive Worlds Worlds group a set of memory invariants, but how are they actually defined formally? We represent each invariant by a region $r \in \text{Region}$. We will see later that regions contain state machines to support a notion of evolvable invariants. Every state of the state machine contains a predicate H that defines the valid memory segments. Unfortunately, it is not enough to just take $H \in \mathcal{P}(\text{MemSeg})$, because sometimes the invariant may itself be world-dependent. For example, we may want to express invariants like "the memory at address 50 contains a value that is safe in the current world". As explained, worlds may evolve, and the set of safe values may grow in future worlds. Therefore, we need to index H over worlds, i.e., take $H \in \text{World} \to \mathcal{P}(\text{MemSeg})$. The result is worlds containing regions with world-indexed predicates, i.e., the set of worlds must be recursively defined. We will see how such a recursive definition can be accommodated using techniques from the literature (essentially an advanced application of step-indexing).

Local capabilities When we invoke an untrusted piece of code and provide it with certain global capabilities, it may have stored those capabilities in memory. In this case, we will only be able to reinvoke the code if we can guarantee that those values are still valid. Formally, worlds represent the invariants that global capabilities' safety relies on and the reinvocation is only safe in future worlds where the invariants are respected.

However, if we provide the adversary with local capabilities in that first invocation, then the situation is a bit different. The adversary has no way to store these local capabilities, so if we make sure that there are also no old local capabilities in the register file for the second invocation (including the capability being invoked), then the adversary cannot use them any more.[4]

[4] We ignore write-local capabilities in this discussion. If the adversary does have access to write-local capabilities in the first and second invocation, then the memory they address must be cleared entirely before the second invocation in order for the reinvocation to remain safe.

Therefore, we can allow the second invocation to happen in any *private* future worlds ($W' \sqsupseteq^{priv} W$) in which safe global capabilities remain safe but local capabilities do not. This private future world relation is more liberal than the standard public one ($W' \sqsupseteq^{pub} W$, in which *all* safe capabilities remain safe). Concretely, worlds may contain *temporary* regions representing invariants that only local capabilities may rely on for their safety and that may be revoked (disabled) in private future worlds.

Interestingly, this idea is a variant of a notion of public/private future worlds that has been previously used in the literature (see Section 2.9). However, temporary regions are new in our setting and there is an interesting interplay with the recursiveness of the worlds: for a temporary region, the predicate $H \in \text{World} \to \mathcal{P}(\text{MemSeg})$ (which defines the safe memory segments in the current world) is only required to be monotone w.r.t. public future worlds (i.e. safe memory segments remain safe in public future worlds). On the other hand for permanent regions, the world-indexed predicate must be monotone w.r.t. private future worlds. As a consequence, the memory for a permanent region may not contain local capabilities (as their safety would be broken in private future worlds), which in turn implies that only local capabilities may have write-local permission (a general sanity requirement when using local capabilities[5])[6].

2.4.2 Worlds

A world is a semantic model of the memory that carves out memories with a particular shape from all the possible memories. In our correctness proofs, worlds allow us to rely on the memory having a particular shape, but sometimes we will also have to guarantee that the memory has a certain shape. Essentially, a world is a collection of invariants. The memory satisfies the world when part of the memory satisfy each of the invariants.

Worlds are represented as a finite map from region names, modeled as natural numbers, to regions that each correspond to an invariant on part of the memory. We have three types of regions: *permanent*, *temporary*, and *revoked*. Each permanent and temporary region contains a state transition system with public and private transitions that describe how the invariant is allowed to change over time. In other words, they are protocols for the region's memory. Protocols imposed by permanent regions stay in place indefinitely. Any capability, local or global, can depend on these protocols. Protocols imposed by temporary regions can be revoked in private future worlds. Doing this may break the safety of local capabilities but not global

[5] In fact, local capabilities become useless as soon as the adversary has access to a single global, write-local capability.

[6] For explanation purposes, this discussion ignores certain ways to allow for local capabilities in a permanent region, for example, by not requiring that they are valid or requiring that they are local versions of valid global capabilities.

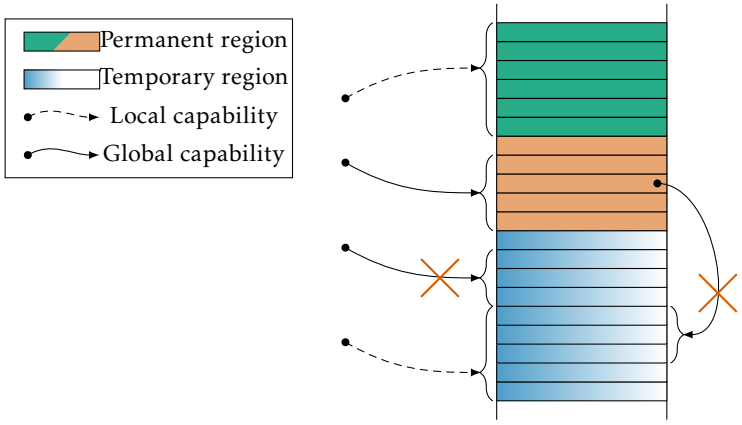


Figure 2.8: The relation between local/global capabilities and temporary/permanent regions. The colored fields are regions governing parts of memory. Global capabilities cannot depend on temporary regions.

ones. This means that local capabilities can safely depend on the protocols imposed by temporary regions, but global capabilities cannot since a global capability may outlive a temporary region that is revoked. This is illustrated in Figure 2.8.

We need the future world relation to be extensional, so we do not actually remove a revoked temporary region from the world, but we turn it into a special revoked region that exists for this purpose. Such a revoked region contains no state transition system and puts no requirements on the memory. It simply serves as a mask for a revoked temporary region. Masking a region like this goes back to earlier work of Ahmed [14] and was also used by Thamsborg and Birkedal [91].

Regions are used to define safe memory segments, but this set may itself be world-dependent. In other words, our worlds are defined recursively. Recursive worlds are common in Kripke models and the following theorem uses the method of Birkedal and Bizjak [18], Birkedal et al. [20] for constructing them. The formulation of the lemma is technical, so we recommend that non-expert readers ignore the technicalities and accept that there exists a set of worlds Wor and two relations $\sqsupseteq^{priv}$ and $\sqsupseteq^{pub}$ satisfying the (recursive) equations in the theorem (where the $\blacktriangleright$ operator can be safely ignored[7]). The solution Wor is some c.o.f.e. that we do not know much about that is, essentially, isomorphic to the our worlds. The isomorphism ξ allows us to move

[7]The interested reader can find a brief coverage of c.o.f.e.'s and $\blacktriangleright$ in Appendix 2.A.2.

between the solution and the worlds we can work with and respects the future world relations.

Theorem 2.4.1. *There exists a c.o.f.e. (complete ordered family of equivalences)* Wor *and preorders* $\sqsupseteq^{priv}$ *and* $\sqsupseteq^{pub}$ *such that* $(\text{Wor}, \sqsupseteq^{priv})$ *and* $(\text{Wor}, \sqsupseteq^{pub})$ *are preordered c.o.f.e.'s, and there exists an isomorphism* ξ *such that*

$$\begin{aligned}
\xi &: \text{Wor} \cong \blacktriangleright(\mathbb{N} \xrightarrow{fin} \text{Region}) \\
\text{Region} &= \{\text{revoked}\} \uplus \\
&\quad \{\text{temp}\} \times \text{RState} \times \text{Rels} \times (\text{RState} \to (\text{Wor} \xrightarrow[\sqsupseteq^{pub}]{mon,\ ne} \text{UPred}(\text{MemSeg}))) \uplus \\
&\quad \{\text{perm}\} \times \text{RState} \times \text{Rels} \times (\text{RState} \to (\text{Wor} \xrightarrow[\sqsupseteq^{priv}]{mon,\ ne} \text{UPred}(\text{MemSeg})))
\end{aligned}$$

and for $W, W' \in \text{Wor}$. $\quad W' \sqsupseteq^{priv} W \Leftrightarrow \xi(W') \sqsupseteq^{priv} \xi(W)$ $\quad W' \sqsupseteq^{pub} W \Leftrightarrow \xi(W') \sqsupseteq^{pub} \xi(W)$ $\quad \lozenge$

In the above theorem, RState × Rels corresponds to the aforementioned state transition system where Rels contains pairs of relations corresponding to the public and private transitions, and RState is a set of world states that we assume to at least contain the states we use in this paper. The last part of the temporary and permanent regions is a state interpretation function that determines what memory segments the region permits in each state of the state transition system. The different monotonicity requirements in the two interpretation functions reflect how permanent regions rely only on permanent protocols whereas temporary regions can rely on both temporary and permanent protocols. UPred(MemSeg) is the set of step-indexed, downwards closed predicates on memory segments: $\text{UPred}(\text{MemSeg}) = \{A \subseteq \mathbb{N} \times \text{MemSeg} \mid \forall (n, ms) \in A. \forall m \leq n. (m, ms) \in A\}$.

With the recursive domain equation solved, we could take Wor as our notion of worlds, but it is technically more convenient to work with the following definition instead:

$$\text{World} = \mathbb{N} \xrightarrow{fin} \text{Region}$$

Future Worlds

The future world relations model how memory may evolve over time. We have a *public* and a *private* future world relation that, respectively, model the memory changes any capability can rely on and the memory changes only local capabilities can rely on. As local capabilities fall within the category of all capabilities, the public future relation is subsumed in the private future relation.

The public future world $W' \sqsupseteq^{pub} W$ requires that $\text{dom}(W') \supseteq \text{dom}(W)$ and $\forall r \in \text{dom}(W). W'(r) \sqsupseteq^{pub} W(r)$. That is in a public future world, new regions

may have been allocated, and existing regions may have evolved according to the public future region relation (defined below). The private future world relation $W' \sqsupseteq^{priv} W$ is defined similarly, using a private future region relation. The public future region relation is the simplest. It satisfies the following properties:

$$\frac{(s,s') \in \phi_{pub}}{(v,s',\phi_{pub},\phi,H) \sqsupseteq^{pub} (v,s,\phi_{pub},\phi,H)} \qquad \frac{(\text{temp},s,\phi_{pub},\phi,H) \in \text{Region}}{(\text{temp},s,\phi_{pub},\phi,H) \sqsupseteq^{pub} \text{revoked}}$$

$$\frac{}{\text{revoked} \sqsupseteq^{pub} \text{revoked}}$$

In public future worlds, both temporary and permanent regions are only allowed to transition according to the public part of their transition system. Additionally, revoked regions must either remain revoked or be replaced by a temporary region. This means that the public future world relations allows us to reinstate a region that has been revoked earlier. The private future region relation satisfies:

$$\frac{(s,s') \in \phi}{(v,s',\phi_{pub},\phi,H) \sqsupseteq^{priv} (v,s,\phi_{pub},\phi,H)} \qquad \frac{r \in \text{Region}}{r \sqsupseteq^{priv} (\text{temp},s,\phi_{pub},\phi,H)}$$

$$\frac{r \in \text{Region}}{r \sqsupseteq^{priv} \text{revoked}}$$

Here, revocation of temporary regions is allowed. In fact, temporary regions can be replaced by an arbitrary region not just the special revoked. Conversely, revoked regions may also be replaced by any other region. On the other hand, permanent regions cannot be masked away. They are only allowed to transition according to the private part of the transition system.

Rather than deleting regions in future worlds, we follow the approach in **? ?**] and use masks to signal which regions are active. This approach implies that the future world relation is a preorder and hence we can use the method in **? ?**] to solve the recursive world equation.

World Satisfaction

A memory satisfies a world, written $ms :_n W$, if it can be partitioned into disjoint parts such that each part is accepted by an active (permanent or temporary) region. Revoked regions are not taken into account as their memory protocols are no longer in effect.

$$ms :_n W \text{ iff } \begin{cases} \exists P : active(W) \to \text{MemSeg}.\, ms = \biguplus_{r \in active(W)} P(r) \text{ and} \\ \quad \forall r \in active(W). \\ \qquad \exists H, s.\, W(r) = (\_, s, \_, \_, H) \text{ and } (n, P(r)) \in H(s)(\xi^{-1}(W)) \end{cases}$$

2.4.3 Logical Relation

The logical relation defines semantically when values, program counters, and configurations are capability safe. The logical relation is defined in Figure 2.9 and Figure 2.10, and we provide some explanations in the following paragraphs. For space reasons, we omit some definitions and explain them only verbally, but precise definitions can be found in the appendix. The logical relation is recursively defined, so we encourage first time readers to just read the section in its entirety and do a second read afterwards.

First, the *observation relation* $\mathcal{O}$ defines what configurations we consider safe. A configuration is safe with respect to a world, when the execution of said configuration does not break the memory protocols of the world. Roughly speaking, this means that when the execution of a configuration halts, then there is a private future world that the resulting memory satisfies. Notice that failing is considered safe behavior. In fact, the machine often resorts to failing when an unauthorized access is attempted such as loading from a capability without read permission. This is similar to Devriese et al. [33]'s logical relation for an untyped language but unlike typical logical relations for typed languages that require programs to not fail.

The *register-file relation* $\mathcal{R}$ defines safe register-files as those that contain safe words (i.e. words in the $\mathcal{V}$-relation defined below) in all registers but the pc-register. The *expression relation* $\mathcal{E}$ defines what words are safe to use as a program counter. A word is safe to use as a program counter when it can be used to form a safe configuration by plugging it into the pc-register of a safe register file (i.e. a register file in $\mathcal{R}$) and pairing it with a memory satisfying the world. Note that integers and non-executable capabilities (e.g. RO and E capabilities) are considered safe program counters because when they are plugged into a register file and paired with a memory, the execution will immediately fail, which is safe.

The *value relation* $\mathcal{V}$ defines when words are safe. We make the value relation as liberal as possible by defining it based on the principle "what is the most we can allow an adversary to use a capability for without breaking the memory protocols."

Non-capability data is always safe because it provides no authority. Capabilities give the authority to manipulate memory and potentially break memory protocols, so they need to satisfy certain conditions to be safe. In Figure 2.10, we define such a condition for each kind of permission a capability can have.

Capabilities with read permission cannot directly break memory protocols because they cannot make changes to the memory. However, a read capability could be used to read a capability that can break memory protocols. For this reason, a capability with read permissions is only safe when it can be used only to read safe capabilities. This is captured by the *readCond*-condition. More precisely, the condition requires the memory addressed by

$$\mathcal{O} : \text{World} \xrightarrow{ne} \text{UPred}(\text{Reg} \times \text{MemSeg})$$

$$\mathcal{O}(W) \stackrel{def}{=} \left\{ (n,(reg,ms)) \;\middle|\; \begin{array}{l} \forall ms_f, mem', i \leq n. \\ (reg, ms \uplus ms_f) \rightarrow_i (halted, mem') \Rightarrow \\ \quad \exists W' \sqsupseteq^{priv} W, ms_r, ms'. \\ \quad\quad mem' = ms' \uplus ms_r \uplus ms_f \text{ and } ms' :_{n-i} W' \end{array} \right\}$$

$$\mathcal{R} : \text{World} \xrightarrow[\sqsupseteq^{pub}]{mon,\, ne} \text{UPred}(\text{Reg})$$

$$\mathcal{R}(W) \stackrel{def}{=} \{(n, reg) \mid \forall r \in \text{RegName} \setminus \{\text{pc}\}.\, (n, reg(r)) \in \mathcal{V}(W)\}$$

$$\mathcal{E} : \text{World} \xrightarrow{ne} \text{UPred}(\text{Word})$$

$$\mathcal{E}(W) \stackrel{def}{=} \left\{ (n, pc) \;\middle|\; \begin{array}{r} \forall n' \leq n, (n', reg) \in \mathcal{R}(W), ms :_{n'} W. \\ (n', (reg[\text{pc} \mapsto pc], ms)) \in \mathcal{O}(W) \end{array} \right\}$$

$$\mathcal{V} : \text{World} \xrightarrow[\sqsupseteq^{pub}]{mon,\, ne} \text{UPred}(\text{Word})$$

$$\begin{array}{l} \mathcal{V}(W) \stackrel{def}{=} \{(n,i) \mid i \in \mathbb{Z}\} \cup \\ \quad \{(n, ((\textsc{o}, g), b, e, a))\} \cup \\ \quad \{(n, ((\textsc{ro}, g), b, e, a)) \mid (n, (b, e)) \in readCond(g)(W)\} \cup \\ \quad \left\{ (n, ((\textsc{rw}, g), b, e, a)) \;\middle|\; \begin{array}{l} (n,(b,e)) \in readCond(g)(W) \text{ and} \\ (n,(b,e)) \in writeCond(i^{nwl}, g)(W) \end{array} \right\} \cup \\ \quad \left\{ (n, ((\textsc{rwl}, g), b, e, a)) \;\middle|\; \begin{array}{l} (n,(b,e)) \in readCond(g)(W) \text{ and} \\ (n,(b,e)) \in writeCond(i^{pwl}, g)(W) \end{array} \right\} \cup \\ \quad \left\{ (n, ((\textsc{rx}, g), b, e, a)) \;\middle|\; \begin{array}{l} (n,(b,e)) \in readCond(g)(W) \text{ and} \\ (n,(\{\textsc{rx}\},b,e)) \in execCond(g)(W) \end{array} \right\} \cup \\ \quad \{(n, ((\textsc{e}, g), b, e, a)) \mid (n, (b, e, a)) \in enterCond(g)(W)\} \cup \\ \quad \left\{ (n, ((\textsc{rwx}, g), b, e, a)) \;\middle|\; \begin{array}{l} (n,(b,e)) \in readCond(g)(W) \text{ and} \\ (n,(b,e)) \in writeCond(i^{nwl}, g)(W) \text{ and} \\ (n,(\{\textsc{rwx}, \textsc{rx}\},b,e)) \in execCond(g)(W) \end{array} \right\} \cup \\ \quad \left\{ (n, ((\textsc{rwlx}, g), b, e, a)) \;\middle|\; \begin{array}{l} (n,(b,e)) \in readCond(g)(W) \text{ and} \\ (n,(b,e)) \in writeCond(i^{pwl}, g)(W) \text{ and} \\ (n,(\{\textsc{rwlx}, \textsc{rwx}, \textsc{rx}\},b,e)) \in execCond(g)(W) \end{array} \right\} \end{array}$$

Figure 2.9: The logical relation.

$$readCond(g)(W) = \left\{ (n,(b,e)) \;\middle|\; \begin{array}{l} \exists r \in localityReg(g,W). \\ \exists [b',e'] \sqsupseteq [b,e].\, W(r) \overset{n}{\subsetneq} \iota^{pwl}_{b',e'} \end{array} \right\}$$

$$writeCond(\iota,g)(W) = \left\{ (n,(b,e)) \;\middle|\; \begin{array}{l} \exists r \in localityReg(g,W). \\ W(r) \text{ is address-stratified and} \\ \exists [b',e'] \sqsupseteq [b,e].\, W(r) \overset{n-1}{\supseteq} \iota_{b',e'} \end{array} \right\}$$

$$execCond(g)(W) = \left\{ (n,(P,b,e)) \;\middle|\; \begin{array}{l} \forall n' < n, W' \sqsupseteq W. \\ \forall a \in [b',e'] \subseteq [b,e], perm \in P. \\ (n',((perm,g),b',e',a)) \in \mathcal{E}(W') \end{array} \right\}$$

$$enterCond(g)(W) = \left\{ (n,(b,e,a)) \;\middle|\; \begin{array}{l} \forall n' < n. \forall W' \sqsupseteq W. \\ (n',((\textsc{rx},g),b,e,a)) \in \mathcal{E}(W') \end{array} \right\}$$

$$\text{where } g = \text{local} \Rightarrow \sqsupseteq = \sqsupseteq^{pub} \text{ and } g = \text{global} \Rightarrow \sqsupseteq = \sqsupseteq^{priv}$$

Figure 2.10: Permission-based conditions

the capability to actually be governed by some region $W(r)$. This region should at least require all the memories accepted by the state interpretation function to contain safe words but stricter requirements may be imposed on the memories. This is expressed in terms of an upper bound on the possibly permitted memories defined in terms of a so called "standard region". In this case, it is a permit-write-local standard region $\iota^{pwl}_{[b,e]}$ (defined in Figure 2.11). The permit-write-local standard region only accepts memories with address range $[b,e]$. More importantly, the permit-write-local standard region's state interpretation function only accepts memory segments that only contain safe words. The relation between the actual region in the world and the upper bound is $W(r) \overset{n}{\subsetneq} \iota^{pwl}_{[b,e]}$ (defined in Appendix 2.A.1). Essentially, the relation requires the two regions state transition systems to be the same. Further in the current state of region $W(r)$ and in any world, the state interpretation function should only allow memories that are also allowed in the standard region.

The locality of capabilities play a role in whether or not they are safe. Generally speaking, global capabilities should only depend on permanent regions. This is expressed in *readCond* with $localityReg(g,W)$ which projects out all the regions the capability can depend on based on its locality. Specifically, if the capability is local, then all active regions (non-revoked) are projected. On the other hand, if the capability is global, then only the permanent regions are projected.

Capabilities with write permission can be used to write to memory, so to define when a capability with write permission is safe we ask ourselves

$$
\begin{aligned}
\iota_A^{pwl}, \iota_A^{nwl} &: \text{Region} \\
\iota_A^{pwl} &\stackrel{def}{=} (\text{temp}, 1, =, =, H_A^{pwl}) \\
\iota_A^{nwl} &\stackrel{def}{=} (\text{temp}, 1, =, =, H_A^{nwl}) \\
\iota_A^{nwl,p} &\stackrel{def}{=} (\text{perm}, 1, =, =, H_A^{nwl}) \\
H_A^{pwl} &: \text{RState} \to (\text{Wor} \xrightarrow[\sqsupseteq^{pub}]{mon,\ ne} \text{UPred}(\text{MemSeg})) \\
H_A^{pwl}(s)(\hat{W}) &\stackrel{def}{=} \left\{ (n, ms) \middle| \begin{array}{l} \text{dom}(ms) = A \text{ and} \\ \forall a \in A. (n-1, ms(a)) \in \mathcal{V}(\xi(\hat{W})) \end{array} \right\} \cup \{(0, ms)\} \\
H_A^{nwl} &: \text{RState} \to (\text{Wor} \xrightarrow[\sqsupseteq^{priv}]{mon,\ ne} \text{UPred}(\text{MemSeg})) \\
H_A^{nwl}(s)(\hat{W}) &\stackrel{def}{=} \left\{ (n, ms) \middle| \begin{array}{l} \text{dom}(ms) = A \text{ and} \\ \forall a \in A. \\ \quad ms(a) \text{ is non-local and} \\ \quad (n-1, ms(a)) \in \mathcal{V}(\xi(\hat{W})) \end{array} \right\} \cup \{(0, ms)\}
\end{aligned}
$$

Figure 2.11: The standard permit write-local and no write-local regions.

what an adversary should be allowed to do with it. An adversary should at least be able to write safe words to memory. Safe words cannot be used to break memory invariants, so we will permit any safe word to be written. A capability with write permission[8] only allows you to write to memory not read from it, so we can allow anything to be written to memory as long as it cannot be read back and used to break memory protocols. For this reason, the condition on capabilities with write-permission is defined as a lower bound on what can be written where the lower bound is "any safe word".

The condition on write capabilities is complicated by the fact that we have two flavours of write permission: write and write-local. A write-local capability can be used to write both local and global capabilities, so it is (at least) allowed to write any safe capabilities. Write capabilities, on the other hand, cannot write local capabilities, so we set the lower-bound as non-local safe words.

The requirements on write-capabilities are captured by the *writeCond*-condition, and it is defined in a similar fashion to the *readCond*-condition. The regions the write-capability may depend on are projected from the world with the *localityReg*-function, and $\overset{n-1}{\supseteq}$ is used to relate the actual region to the lower bound. We need a different lower-bound depending on whether

[8] On the capability machine we consider, write permission always comes with read permission, so we could have made a stricter condition for write capabilities.

the permission of the capability is write or write-local, so *writeCond* is parameterized with the region it actually uses. If the capability has write-local permission, then we use the permit-write-local standard region. If capability only has write permission, then we use a no-write-local standard region $\iota^{nwl}_{[b,e]}$ (defined in Figure 2.11). In addition to the requirements of a permit-write-local standard region, the no-write-local standard region requires all words in the accepted memory segments to be non-local.

Finally, there is a technical requirement that the region must be *address-stratified*. Intuitively, this means that if a region accepts two memory segments, then it must also accept every memory segment "in between" in the sense that it should be possible to come from one memory segment to the other. Our capability machine can only update one memory address at a time, so it should be accepted to construct a memory that is on its way to become the one we want to accept. Specifically, if two memory segments are accepted by a region, then the region must also accept every memory segment where each address contains a value from one of the two accepted memory segments.

Due to the permission combinations on the capability machine we consider, a *writeCond* always comes with a *readCond* which creates a somewhat tight bound for the possible regions that make a capability safe. Further, we could have made the *writeCond* based on the fact that a writer permission always comes with a read permission, but we opted to make the logical relation as general as possible, so it can be reused in a setting with a richer permission hierarchy.

The conditions *enterCond* and *execCond* are very similar. Both require that the capability can be safely jumped to. However, executable capabilities can be updated to point anywhere in their range, so they must be safe as a program counter (in the $\mathcal{E}$-relation) no matter the current address. The range of an executable capability can also be reduced, so they must also be safe as program counter no matter what their range of authority is reduced to. In contrast, enter capabilities are opaque and can only be used to jump to the address they point to. This is why *enterCond* depends on the current address of the capability unlike for other types of capabilities. They also change permission when jumped to, so we require them to be safe as a program counter after the permission is changed to RX. Because the capabilities are not necessarily invoked immediately, this must be true in any future world, but it depends on the capability's locality which future worlds we consider. If it is global, then we require safety as a program counter in *private* future worlds (where temporary regions may be revoked). For local capabilities, it suffices to be safe in *public* future worlds where temporary regions are still present.

In the technical appendix, we prove that safety of all values is preserved in public future worlds, and that safety of global values is also preserved in

private future worlds:

Lemma 2.4.2 (Double monotonicity of value relation)**.** *For worlds W and W', $n \in \mathbb{N}$, word w, permission perm, and addresses b, e, a, the following holds*

- *If $W' \sqsupseteq^{pub} W$ and $(n,w) \in \mathcal{V}(W)$, then $(n,w) \in \mathcal{V}(W')$.*
- *If $W' \sqsupseteq^{priv} W$ and $(n,w) \in \mathcal{V}(W)$ and $w = ((perm, \text{global}), b, e, a)$ (i.e. w is a global capability), then $(n,w) \in \mathcal{V}(W')$.*

◊

In Section 2.3, we require capabilities with write-local permission to be local. This indicates that our logical relation should imply the same, namely that capabilities with write-local permission are local.

Lemma 2.4.3 (Stack capabilities are local)**.** *For memory segment ms, $n \in \mathbb{N}$, world W, permission perm, locality g, and addresses b, e, and a, if $ms :_n W$ and $(n, ((perm, g), b, e, a)) \in \mathcal{V}(W)$ and $b \leq e$ and $perm \in \{\text{RWLX}, \text{RWL}\}$, then $g = \text{local}$.*
◊

In our definition of worlds, nothing prevents a world from having regions that are overlapping. This may seem like an issue with the $\mathcal{V}$-relation as it allows different permission-based requirements to be satisfied by different regions. In practice, it is not an issue as we will always have a memory satisfaction assumption which doubles as a well-formedness condition on the world as it prevents the regions from overlapping. The memory satisfaction assumption in Lemma 2.4.3 is there to ensure a well-formed world.

2.4.4 Capability Machine Safety

With the logical relation defined, we can now state the fundamental theorem of our logical relation: a strong theorem that formalizes the guarantees offered by the capability machine. Essentially, it says a capability that only grants safe authority is capability safe as a program counter.

Theorem 2.4.4 (Fundamental Theorem)**.** *If one of the following holds:*

- $perm = \text{RX}$ *and* $(n,(b,e)) \in readCond(g)(W)$
- $perm = \text{RWX}$ *and* $(n,(b,e)) \in readCond(g)(W)$ *and* $(n,(b,e)) \in writeCond(\iota^{nwl}, g)(W)$
- $perm = \text{RWLX}$ *and* $(n,(b,e)) \in readCond(g)(W)$ *and* $(n,(b,e)) \in writeCond(\iota^{pwl}, g)(W)$,

then $(n, ((perm, g), b, e, a)) \in \mathcal{E}(W)$ ◊

Proof sketch. Induction over n. By definition of $\mathcal{E}(W)$, show

$$(n', (reg[\text{pc} \mapsto ((perm, g), b, e, a)], ms)) \in \mathcal{O}(W)$$

assuming $n' \leq n$, $(n', reg) \in \mathcal{R}(W)$, and $ms :_{n'} W$. By definition of $\mathcal{O}$ let ms_f, mem', and $i \leq n'$ be given and for $\Phi = (reg[\text{pc} \mapsto ((perm, g), b, e, a)], ms \uplus ms_f)$ assume $\Phi \rightarrow_i (halted, mem')$ and show there exists $W' \sqsupseteq^{priv} W$ that part of mem' satisfies. First observe that $i \neq 0$ as Φ is a non-halted configuration, so Φ takes at least one step, i.e. $\Phi \rightarrow \Phi' \rightarrow_{i-1} (halted, mem')$.

The rest of the proof considers the different possibilities for the step $\Phi \rightarrow \Phi'$, i.e. it considers each of the instructions that could have been executed. For each of these cases, we argue that (1) Φ' is consistent with the world in the sense that the register-file and memory still respect the world and that (2) the rest of the execution respects the world. Depending on where the pc in Φ' comes from, the second result is proven in one of two ways. If the step to Φ' was a jump, then the new pc is one of the safe values in Φ's registers, and the value relation is used to argue that the jump is safe. On the other hand, if the pc was just incremented, then we can apply the induction hypothesis.

In order to argue (1), we look at the possible states for Φ according to the operational semantics. If we consider the memory in Φ', then it either (a) remains unchanged or (b) one address has been updated and the register-file contains an appropriate capability for writing. The latter occurs when the executed instruction is `store`. Otherwise we are in the former case. In case (a), the memory hasn't changed, so we conclude that the memory still respects the world simply by downwards closure of memory satisfaction. In case (b), we use an auxiliary lemma that uses the safety of the write capability used by the `store` instruction to show that the updated memory satisfies the world. To show that the updated register-file is safe, we consider the changes made to it by all instructions in separate lemmas and show that they all preserve safety of the register file.

The complete proof can be found in the technical appendix [84]. □

The permission-based conditions of Theorem 2.4.4 make sure that the capability only provides safe authority in which case the capability must be in the $\mathcal{E}$ relation, i.e. it can safely be used as a program counter in an otherwise safe register-file.

The Fundamental Theorem can be understood as a general expression of the guarantees offered by the capability machine which is an instance of a general property called capability safety [33, 64]. The theorem says that an arbitrary capability $((perm, g), b, e, a)$ is safe as a program counter without making any assumptions about what program it points to (the only assumptions we have are about the read or write authority that it carries). As such, the theorem expresses the capability safety of the machine which guarantees that *any* instruction is safe to execute and will not be able to go beyond the authority of the values it has access to. We demonstrate this in Section 2.8

where Theorem 2.4.4 is used to reason about capabilities that point to arbitrary instructions. The relation between Theorem 2.4.4 and local-state encapsulation and control-flow correctness will also be shown by example in Section 2.8 as the examples depend on these properties for correctness.

2.5 Malloc

In the examples presented in Section 2.8, we will assume the existence of a trusted malloc routine, so that both the trusted code and the adversary are able to allocate new memory. Malloc is considered part of the trusted computing base as mentioned in Section 2.2. This is unavoidable: if we cannot trust malloc, then we cannot use the memory it allocates as we have no idea whether it is aliased by some untrusted program.

Our semantic model is not specific to a particular implementation of malloc, so rather than providing the implementation we provide a malloc specification. The specification expresses standard expectations about the behaviour of malloc, but making the specification realistic requires some of the technical machinery from the logical relation. As such, this section is a bit technical and can safely be skipped on first read. Definition 2.5.1 is the malloc specification. Following the definition, we provide an informal description of the definition with references to each of the items in the definition.

Definition 2.5.1 (Malloc Specification)**.** c_{malloc} satisfies the specification for malloc iff the following conditions hold:

1. $c_{malloc} = ((\textsc{e}, \text{global}), \_, \_, \_)$

2. There exists a $\iota_{malloc,0}$ such that

   a) $\iota_{malloc,0}.v = \text{perm}$

   b) For all $\iota' \sqsupseteq^{priv} \iota_{malloc,0}, W, i$ with $W(i) = \iota'$, we have that

   $$\iota'.H(\iota'.s)(\xi^{-1}(W)) = \iota'.H(\iota'.s)(\xi^{-1}([i \mapsto W(i)]))$$

   c) For all $\Phi \in \text{ExecConf}$, $ms_{footprint}, ms_{frame} \in \text{MemSeg}$, i, n, $size \in \mathbb{N}$,

$w_{ret} \in \text{Word}$, $\iota_{malloc} \sqsupseteq^{priv} \iota_{malloc,0}$, we have that

$$
\begin{array}{l}
\text{If } \Phi.\text{mem} = ms_{footprint} \uplus ms_{frame} \wedge ms_{footprint} :_n [i \mapsto \iota_{malloc}] \wedge \\
\quad \Phi.\text{reg}(r_1) = size \wedge size \geq 0 \wedge \Phi.\text{reg}(r_0) = w_{ret} \wedge \\
\quad \Phi.\text{reg}(\text{pc}) = updPcPerm(c_{malloc}) \\
\text{Then,} \\
\quad \exists \Phi' \in \text{ExecConf}.\, \exists ms'_{footprint}, ms_{alloc} \in \text{MemSeg}. \\
\quad\quad \exists j \in \mathbb{N}.\, j > 0 \wedge \exists b', e' \in \text{Addr}.\, \exists \iota'_{malloc} \in \text{Region}. \\
\quad\quad\quad \Phi \rightarrow_j \Phi' \wedge \\
\quad\quad\quad \Phi'.\text{mem} = ms'_{footprint} \uplus ms_{alloc} \uplus ms_{frame} \wedge \\
\quad\quad\quad \iota'_{malloc} \sqsupseteq^{pub} \iota_{malloc} \wedge \\
\quad\quad\quad ms'_{footprint} :_{n-j} [i \mapsto \iota'_{malloc}] \wedge \\
\quad\quad\quad \text{dom}(ms_{alloc}) = [b', e'] \wedge \forall a \in [b', e'].\, ms_{alloc}(a) = 0 \wedge \\
\quad\quad\quad \Phi'.\text{reg} = \Phi.\text{reg}[\text{pc} \mapsto updPcPerm(w_{ret})] \\
\quad\quad\quad\quad\quad\quad [r_1 \mapsto ((\text{RWX}, \text{global}), b', e', b')] \\
\quad\quad\quad\quad\quad\quad [r_{t1}, r_{t2}, r_{t3} \mapsto 0, 0, 0] \wedge \\
\quad\quad\quad size - 1 = e' - b'
\end{array}
$$

d) For all $\Phi \in \text{ExecConf}$,

$$
\begin{array}{l}
\text{If } (\Phi.\text{reg}(r_1) \notin \mathbb{Z} \vee \Phi.\text{reg}(r_1) < 0) \wedge \Phi.\text{reg}(\text{pc}) = updPcPerm(c_{malloc}) \\
\text{Then } \exists j \in \mathbb{N}.\, \Phi \rightarrow_j failed
\end{array}
$$

♦

We require a global capability for invoking malloc (because if the capability were local, then a program with access to malloc would have to give up this access when invoking untrusted code, 1). The capability is assumed to have enter permission (1), so malloc can protect its internal state even when the capability is shared with untrusted code.

In addition to these syntactic requirements, we also specify standard behavioural expectations of malloc. Intuitively, we require that when malloc is invoked with a length argument, then it will return a capability for a fresh piece of memory of that size. It should be fresh in the sense that malloc has not already allocated any of that memory and will not do so in the future. Also, when invoked with a nonsensical length argument such as a negative integer or a capability, malloc should simply fail. However, formulating these requirements is harder than one might expect. The problem is that a realistic implementation of malloc needs to rely on internal state (the free memory is part of malloc's internal state) that changes after every invocation and relies on invariants on that state. We can only expect that malloc behaves according to its specification if its internal state satisfies the current state of the invariants in an executing system. We express this in terms of the semantic model from Section 2.4.

To allow malloc implementations to rely on internal state and invariants for that state, we assume an initial region for malloc (2). The region is assumed to be permanent (since safety of the global malloc capability will depend on its presence, 2a). Furthermore, we want to express that malloc does not depend on any other memory than its own internal state. This is expressed by a restriction on the malloc region's state interpretation function which (as explained in Section 2.4.2) defines what memory segments it permits in a given world. We require that the accepted memory segments only depend on the presence of the malloc region itself, i.e. in any world the same memory segments are accepted if we remove all regions except the malloc region. This property should continue to hold throughout execution, so it must hold true for any private future region of the initial malloc region (2b).

The malloc specification also dictates what malloc should do when invoked in a memory with its internal state valid according to the malloc region (some future evolution of the initial malloc region). If malloc is invoked with an invalid length argument (that is, a negative integer or a capability), then we simply require malloc to fail (2d). This part of the specification does not actually rely on the malloc region: for simplicity, we assume malloc does not need its internal state to check the argument. When malloc is invoked with a valid length (2c), then it should return a fresh memory segment of the correct length. This segment is required to come from the footprint of malloc, i.e. the memory owned by the malloc region before the call. After malloc returns, we require the malloc region to have evolved (according to the public future world relation) to a new state where the new footprint is disjoint from the allocated memory. This implies that future invocations of malloc can never return previously-allocated memory.

For convenience, we also require that malloc returns the non-argument registers (except registers for malloc internal computations) of the register-file unchanged after the call. This allows the caller to keep private capabilities in the register file, without having to protect them by storing them in a private stack frame.

As described previously, the specification of malloc ensures that malloc has no capabilities pointing *out* of malloc. It does not, however, say anything about capabilities that point *in* to malloc. If we want to be able to trust malloc, we obviously cannot allow arbitrary capabilities to point in to it. We have chosen to keep the malloc specification simple and let this assumption be in the lemmas that use malloc. It is sufficient for these lemmas to require that there are no outside capabilities for malloc in the initial configuration as capabilities cannot appear out of thin air, and the malloc specification makes sure that capabilities are not leaked.

Malloc should not just be available to trusted programs but also to possibly malicious programs. This is safe as it follows from the specification that the malloc capability is always safe in a world with the malloc region:

Lemma 2.5.2 (Malloc is safe to pass to adversary)**.** *For all c_{malloc} that satisfies the specification for malloc with region $\iota_{malloc,0}$, if $W(r) \sqsupseteq^{priv} \iota_{malloc,0}$, then $(n, c_{malloc}) \in \mathcal{V}(W)$ for all n.* ◊

The reason we allow trusted code and adversary to invoke malloc is just to make our work more realistic, but we are otherwise not interested in its details. As such, we do not give a malloc implementation. We are, however, confident that it is possible to make an implementation of malloc that satisfies the malloc specification in Definition 2.5.1. There are in fact two simplifications in our system that makes things easier: First, we do not consider deallocation of memory which means that the data structure malloc uses to keep track of free memory does not have to handle reclaimed memory. Second, the malloc specification does not permit malloc to run out of memory and thus refuse allocation. This is possible on our simple capability machine because it has an infinite address space. An initial capability with an infinite range of authority would of course need to be part of malloc, but it could also double as the data structure that keeps track of free memory.

2.6 Reusable macro instructions

With the calling convention and logical relation defined, we would like to show its usefulness by proving the correctness of a series of examples that rely on well-bracketedness and local-state encapsulation and use the calling convention to enforce these properties. However, the programs that run on our capability machine are assembly programs. Program examples that would be small in a high-level language become big and unintelligible at this low level. Thus to make our program examples intelligible, we introduce a series of low-level abstractions in the form of macros. We define a number of reusable macros capturing the calling convention as well as other conveniences. The macros that utilize the stack assume that it is available in register r_{stk}.

The macro `scall` $r(\overline{r_{args}}, \overline{r_{priv}})$ captures the parts of the calling convention related to actually transferring control to adversarial code. Specifically, it pushes the contents of the private registers, $\overline{r_{priv}}$, to the stack and pushes the "restoration" code to the stack. The restoration code is executed as the first thing upon return; it restores the stack pointer and the old program counter. After the restoration code is pushed to the stack, `scall` $r(\overline{r_{args}}, \overline{r_{priv}})$ adjusts the pc to point to the first instruction after the jump and pushes it to the stack. The `scall`macro constructs a protected return pointer from a stack pointer by adjusting it to point to the first instruction of the return code and encapsulating it by restricting its permission to E. Next, the `scall` macro reduces the range of authority of the stack pointer to the unused part of the stack and clears it (as discussed in Section 2.3). Finally, the `scall` macro clears non-argument registers and jumps to r. Upon return after the restora-

tion code on the stack has been executed, the `scall` macro pops the restoration code from the stack and restores the private state by popping it from the stack to the appropriate registers. Figure 2.12 displays the implementation of `scall` and the restoration code used by `scall`. The implementation of `scall` uses some of the macros we present next.

The macro `mclear` r clears all the memory addresses that the capability in register r has authority over. It is used by `scall` to clear the unused part of the stack before control is transferred. It should also be used to clear the stack before returning to adversarial code. Similarly, the macro `rclear` R clears all the registers in the set R. It is also used by `scall`, and it should also be used before returning to adversarial code.

The macros `prepstack` r and `reqglob` r are the last macros related to the calling convention. The former, `prepstack` should be used when one receives a stack from an unknown source. The `prepstack` macro first ensures the stack capability has permission read/write-local/execute. Then, it adjusts the stack capability, so it follows the convention for the stack[9]. The other macro, `reqglob`, ensures that the capability in register r is global. This macro should be used on callbacks received from unknown sources to ensure that they cannot be derived from the stack pointer.

The remaining macros are not directly related to the calling convention, but they help making the examples in Section 2.8 intelligible. The macros `push` r and `pop` r respectively add and remove elements from the stack. The macro `fetch` r *name* fetches the capability related to *name* from the linking table and stores it in register r. The macro `malloc` r n invokes malloc with size argument n. The `malloc` macro assumes that a capability for malloc resides in the linking table and is basically a fetch of the malloc capability followed by a setup of a return pointer. Finally, the macro `crtcls` $\overline{(x_i, r_i)}$ r allocates a closure where r points to the closure's code and a new environment is allocated (using malloc) and the contents of $\overline{r_i}$ are stored in the environment. In the code referred to by r, an implicit load from the environment happens when an instruction refers to x_i.

The Appendix contains the implementation of all the macros used in `scall`. The technical appendix [84] contains more detailed descriptions of all the macros as well as all implementations. We stress that the macros correspond to series of instructions as seen in Figure 2.12; the macros are introduced for intelligibility. The examples of Section 2.8, the program examples are stated using the macros, but the proofs work on the expanded macros.

[9]The stack capability should always point to the topmost word on the stack. A stack received from an unknown source can be treated as empty, so the stack capability should point just outside its range of authority.

```
1  // Push the private registers to the stack.
2    push r_priv,1
3    ...
4    push r_priv,n
5  // Push the restoration code to the stack.
6    push encode(i1)
7    push encode(i2)
8    push encode(i3)
9    push encode(i4)
10 // Push the old pc to the stack.
11   move r_t1 pc
12   lea r_t1 off
13   push r_t1
14 // Push the stack pointer to the stack.
15   push r_stk
16 // Set up the protected return pointer.
17   move r_0 r_stk
18   lea r_0 off_rec
19   restrict r_0 encodePermPair((local, E))
20 // Restrict the stack capability to the unused part of the stack.
21   geta r_t1 r_stk
22   plus r_t1 r_t1 1
23   getb r_t2 r_stk
24   subseg r_stk r_t1 r_t2
25 // Clear the unused part of the stack.
26   mclear r_stk
27 // Clear non-argument registers.
28   rclear R
29   jmp r
30 return:
31 // Pop the restore code.
32   pop r_t1
33   pop r_t1
34   pop r_t1
35   pop r_t1
36 // Pop the private state into approriate registers.
37   pop r_priv,1
38   ...
39   pop r_priv,n
```

i_1 = **move** r_t1 pc
i_2 = **lea** r_t1 off_{stk}
i_3 = **load** r_stk r_t1
i_4 = **pop** pc

The restoration code used in scall.

Figure 2.12: Implementation of scall $r(r_{args,1}, \ldots, r_{args,m}, r_{priv,1}, \ldots, r_{priv,n})$. The restoration code is presented in the top right corner. The variable off_{ret} is the offset to the label return, and $off_{rec} = -5$ which is the offset to the first instruction of the activation record. The set R = RegName \ {pc, r_stk, r_0, $r, r_{args,1}, \ldots, r_{args,m}$}. The variable $off_{stk} = 5$ which is the offset to the address where the old stack pointer is stored on the stack.

2.7 Reasoning about programs on a capability machine

There are many details to get right when programming in assembly. These details carry over to proofs about assembly programs, so many of the proofs in Section 2.8 about example programs are a bit cumbersome. It is especially annoying when the same line of reasoning is applied in multiple places. To mitigate this, we have defines a number of lemmas that capture common reasoning patterns in these proofs. In this section, we present the most central lemmas used to reason about assembly programs.

Our capability machine only allows the final memory of an execution to be observed, so naturally the correctness lemmas we prove in Section 2.8 are statements about the memory in the halted configuration. In order to prove a property about part of the final memory, we create a world with a region the ensures the desired property and show that the initial configuration is in the $\mathcal{O}$-relation w.r.t. that world. The region that ensures the property must be permanent, so it is not revoked during execution. The $\mathcal{O}$-relation says that if the initial configuration halts successfully, then the memory in the halting state must still respect a private future world of the initial world. It is, however, bothersome and error prone to try to argue about the entire execution in one go. Instead we want to reason modularly in the sense that we only want to reason about parts of the execution at a time. To this end, we prove an anti-reduction lemma that essentially says that if we can show that an initial configuration Φ steps to a configuration Φ' and Φ' is in the $\mathcal{O}$-relation, then Φ must also be in the observation relation.

Lemma 2.7.1 (Anti-reduction for $\mathcal{O}$)**.**

$$
\begin{aligned}
&\forall n, n', i, reg, reg', ms, ms', ms_r, W, W'.\\
&\quad n' \geq n - i \wedge W' \sqsupseteq^{priv} W \wedge\\
&\quad (\forall ms_f.\, (reg, ms \uplus ms_r \uplus ms_f) \rightarrow_i (reg', ms' \uplus ms_r \uplus ms_f)) \wedge\\
&\quad (n', (reg', ms')) \in \mathcal{O}(W')\\
&\qquad \Rightarrow (n, (reg, ms \uplus ms_r)) \in \mathcal{O}(W)
\end{aligned}
$$

$\Diamond$

We use the anti-reduction lemma when we reason about the execution of known code. When we want to reason about unknown code, the anti-reduction lemma does not apply because we do not know what instructions are being executed. This is where the FTLR (Theorem 2.4.4) comes into play. As a reminder, the FTLR says that if a capability only has access to safe values with respect to a world, then it is safe to use the capability for execution in the same world. The unknown code we consider will be assumed to only have access to safe values which typically means it has access to a linking

table with a malloc capability and otherwise consists of instructions. These assumptions allow us to use the FTLR to reason about the unknown code as malloc is a safe value (cf. Lemma 2.5.2) and instructions are integers; integers are always safe values. It is important to remember that safety is relatively to the semantic model of a memory and not the actual memory itself. For this reason, the assumptions we make on unknown code must be expressed as a region in a world.

In order to argue that a specific configuration is safe, we need to argue that the configuration is safe with respect to the world. That is, we need to argue that the memory satisfies the world and the register-file is in the $\mathcal{R}$-relation. For instance in the case where the untrusted code assumes control first, we will use the FTLR to show that the capability for the unknown code can be used for execution. The unknown code will get access to some known, trusted code through a capability in the initial register-file, so we will have to argue that the known code is safe. We argue about the known code with Lemma 2.7.1.

Another common pattern in proofs about programs on our capability machine concerns the use of `scall`. The following lemma captures the commonalities of reasoning about programs using `scall`. For instance, setting up the local stack frame and constructing stack pointers and protected return pointers for the callee always amount to the same line of reasoning which is captured by the lemma. From another point of view, it can be seen as a specification for `scall`.

Lemma 2.7.2 (`scall` works)**.** *For all* $n \in \mathbb{N}$, $ms, ms_{stk}, ms_{unused}, ms_f \in \text{MemSeg}$, $W \in \text{World}$, $reg \in \text{Reg}$, $r, \overline{r_{arg}}, \overline{r_{priv}} \in \text{RegName}$, *and* $c_{next} \in \text{Cap}$, *if*

1. $ms :_n revokeTemp(W)$
2. $\text{dom}(ms_f) \cap (\text{dom}(ms_{stk} \uplus ms_{unused} \uplus ms)) = \emptyset$
3. (reg, ms) *is looking at* `scall` $r(\overline{r_{arg}}, \overline{r_{priv}})$ *followed by* c_{next}
4. *reg points to stack with* ms_{stk} *used and* ms_{unused} *unused*
5. Hyp-Callee *For all* $ms_{rec}, ms'_{unused}, ms'' \in \text{MemSeg}$, $W' \in \text{World}$, $c_{ret}, c_{stk} \in \text{Cap}$, *and* $reg' \in \text{Reg}$, *if*
   - $\text{dom}(ms_{unused}) = \text{dom}(ms_{rec} \uplus ms'_{unused})$
   - $W' = revokeTemp(W)[\iota^{sta}(\text{temp}, ms_{stk} \uplus ms_{rec} \uplus ms_f), \iota^{pwl}(\text{dom}(ms'_{unused}))]$[10]
   - $ms'' :_{n-1} W'$
   - *reg' points to stack with* $\emptyset$ *used and* ms'_{unused} *unused*

[10] We use the update-notation without a region name to indicate that it should be a fresh region name.

- $reg' = reg_0[\text{pc} \mapsto updPcPerm(reg(r)), \overline{r_{arg}} \mapsto reg(\overline{r_{arg}}), r_0 \mapsto c_{ret}, r_{stk} \mapsto c_{stk}, r \mapsto reg(r)]$
- $(n-1, c_{ret}) \in \mathcal{V}(W')$
- $(n-1, c_{stk}) \in \mathcal{V}(W')$

then we have $(n-1, (reg', ms'')) \in \mathcal{O}(W')$

6. Hyp-Cont *For all* $n' \in \mathbb{N}$, $W'' \in \text{World}$, ms'', ms''_{unused}, *and* $reg' \in \text{Reg}$, *if*

- $n' \leq n-2$
- $W'' \sqsupseteq^{pub} revokeTemp(W)$
- $ms'' :_{n'} revokeTemp(W'')$
- *for all* r, *we have:*

$$reg'(r) \begin{cases} = c_{next} & \text{if } r = \text{pc} \\ = reg(r) & \text{if } r \in \overline{r_{priv}} \\ \in \mathcal{V}(revokeTemp(W'')) & \text{if } reg'(r) \text{ is a global capability and} \\ r \notin \{\text{pc}, \overline{r_{priv}}, r_{stk}\} & \end{cases}$$

- reg' *points to stack with* ms_{stk} *used and* ms''_{unused} *unused*

then we have $\left(n', (reg', ms'' \uplus ms_f \uplus ms_{stk} \uplus ms''_{unused})\right) \in \mathcal{O}(W'')$

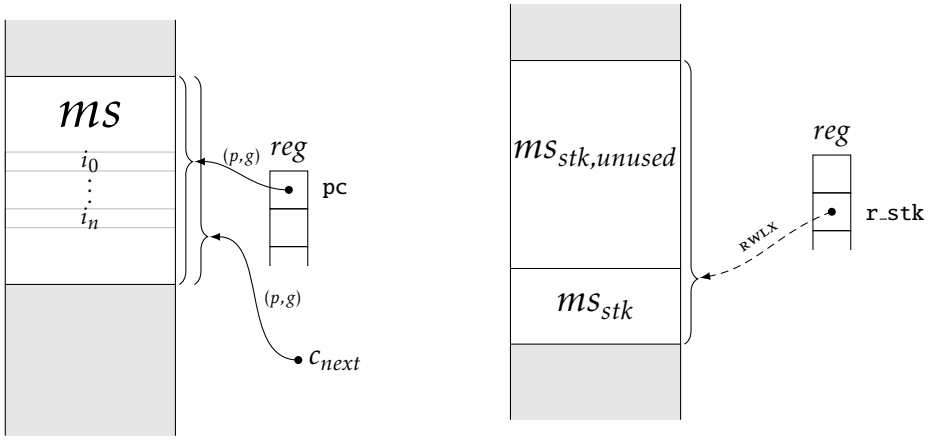
Then

- $\left(n, (reg, ms \uplus ms_f \uplus ms_{stk} \uplus ms_{unused})\right) \in \mathcal{O}(W)$

◊

The `scall` lemma is stated in terms of a number of auxiliary definitions found in the appendix. The region $\iota^{sta}(g, ms)$ is a static region with locality g. It is static in the sense that it only accepts the memory segment ms (Appendix 2.A.7). The function $revokeTemp : \text{World} \to \text{World}$ yields the input world but with all the temporary regions replaced with revoked regions (Appendix 2.A.1). The definition of (reg, ms) *is looking at* $[i_0 \dots i_n]$ *followed by* c_{next} is visualized in Figure 2.13a; it requires the capability in pc of reg to point to the first address of the instructions $[i_0 \dots i_n]$ in ms and c_{next} to point to the address immediately after the instructions. The definition of *reg points to stack with* ms_{stk} *used and* $ms_{stk,unused}$ *unused* is visualized in Figure 2.13b; it requires ms_{stk} and $ms_{stk,unused}$ to be adjacent and continuous memory segments with the local RWLX-capability in `r_stk` of reg governing the two memory segments and pointing to the top most address of ms_{stk} (Appendix 2.A.5).

Roughly, the `scall` lemma (Lemma 2.7.2) states that an invocation of scall is safe if `scall` is executed in a reasonable state (1-3), the callee is safe to execute (5), and returning to the code after the scall is safe (6).



(a) Visualization of "(reg, ms) is looking at $[i_0 \dots i_n]$ followed by c_{next}". reg(pc) points to i_0 and c_{next} points to the address after i_n

(b) Visualization of "reg points to stack with ms_{stk} used and $ms_{stk,unused}$ unused". $reg(r_{stk})$ is a local RWLX-capability that points to the top most address of ms_{stk}.

Figure 2.13

In the common case, `scall` is used to reason about untrusted code that we only have general assumptions about. As explained previously, we use the FTLR (Theorem 2.4.4) to argue about unknown code; this is no exception. That is, when we argue that assumption 5 in the `scall` lemma is satisfied, we use the FTLR along with the assumptions we have (e.g. linking table, malloc capability etc.). In Hyp-Callee, we assume the memory is safe, so it suffices to show that the register-file contains safe values, which amounts to showing that the arguments in the call are safe.

By using Lemma 2.7.2 to reason about `scall`, the proofs become agnostic to the actual implementation of `scall`. In other words, should we change the implementation of `scall`, then we just need to prove that Lemma 2.7.2 holds for the new implementation in order to get that all our results still hold true.

The `malloc` macro is also common in our examples, so we prove Lemma 2.7.3 to help reason about it. The structure of the `malloc` lemma is close to that of the `scall` lemma. It also has requirements on the configuration just before `malloc` is executed (1-8) as well as requirements on the execution afterwards (9). It does not have any requirements on the callee as the `malloc` specification defines its behavior.

Lemma 2.7.3 (`malloc` works)**.** *If*

1. (reg, ms) *is looking at* $\texttt{malloc}\ r\ k$ *followed by* c_{next}
2. $k \geq 0$
3. (reg, ms) *links malloc as* k *to* c_{malloc}
4. c_{malloc} *satisfies the malloc specification with* $\iota_{malloc,0}$
5. $W \sqsupseteq^{priv} [i \mapsto \iota_{malloc,0}]$
6. $ms :_n W$
7. $ms = ms' \uplus ms_{footprint}$
8. $ms_{footprint} :_n [i \mapsto W(i)]$
9. Hyp-Cont *If*
   - $n' \leq n - 1$
   - $\iota_{malloc} \sqsupseteq^{pub} W(i)$
   - $ms'_{footprint} \uplus ms' :_{n'} W[i \mapsto \iota_{malloc}]$
   - $ms'_{footprint} :_{n'} [i \mapsto \iota_{malloc}]$

$$reg'(r') = \begin{cases} c_{next} & r' = \text{pc} \\ ((\text{RWX}, \text{global}), b, e, a) & r' = r \\ reg(r) & r' \notin \text{RegName}_t \cup \{\text{pc}, r, r_1\} \end{cases}$$

   - $e - b = k - 1$
   - $\text{dom}(ms_{alloc}) = [b, e]$
   - $\forall a \in [b, e].\, ms_{alloc}(a) = 0$

Then we have $\left(n', (reg', ms' \uplus ms'_{footprint} \uplus ms_{alloc})\right) \in \mathcal{O}(W[\iota_{malloc}])$

Then

$$(n, (reg, ms)) \in \mathcal{O}(W)$$

◊

In the technical appendix [84], we also define a lemma for the macro used to create closures, `crtcls`.

```
1 f1: push 1                    1 f2: malloc r_l 1
2     fetch r_1 adv             2     store r_l 1
3     scall r_1([],[])          3     fetch r_1 adv
4     pop r_1                   4     call r_1([],[r_l])
5     assert r_1 1              5     assert r_l 1
6     halt                      6     halt
7                               7
```

Figure 2.14: Two example programs that rely on local-state encapsulation. `f1` uses our stack-based calling convention. `f2` does not rely on a stack.

2.8 Examples

In this section, we demonstrate how our formalization of capability safety allows us to prove local-state encapsulation and control-flow correctness properties for challenging program examples. The security measures of Section 2.3 are deployed to ensure these properties. Since we are dealing with assembly language, there are many details to the formal treatment. Therefore, we omit some details in the lemma statements. The examples may look deceivingly short, but it is because we use the macro instructions described in Section 2.6. The examples would be unintelligible without the macros as each macro expands to multiple basic instructions.

2.8.1 Encapsulation of Local State

The programs `f1` and `f2` in Figure 2.14 demonstrate the capability machine's encapsulation of local state. They are very similar: both store some local state, call an untrusted piece of code (*adv*), and test whether the local state is unchanged after the call. They differ in the way they do this. Program `f1` uses our stack-based calling convention (captured by `scall`) to call the adversary, so it can use the available stack to store its local state. On the other hand, `f2` uses malloc to allocate memory for its local state and uses `call` a calling convention based on heap allocated activation records (described in Appendix 2.A.4) to invoke the adversarial code.

For both programs, we prove that if they are linked with an adversary, *adv*, allowed to allocate memory but has no other capabilities, then the assertion will never fail during execution (see Lemmas 2.8.1 and 2.8.2 below). The two examples also illustrate the versatility of the logical relation. The logical relation is not specific to any calling convention, so we can use it to reason about both programs even though they use different calling conventions.

In order to formulate results about `f1` and `f2`, we need a way to observe whether the assertion fails. To this end, we assume they have access to a flag (an address in memory). If the assertion fails, then the flag is set to 1 and execution halts.

Lemma 2.8.1 (`f1` is correct)**.** *Let*

$$\begin{aligned} c_{adv} &\stackrel{def}{=} ((\text{E}, \text{global}), \dots) & c_{stk} &\stackrel{def}{=} ((\text{RWLX}, \text{local}), \dots) \\ c_{f1} &\stackrel{def}{=} ((\text{RWX}, \text{global}), \dots) & c_{link} &\stackrel{def}{=} ((\text{RO}, \text{global}), \dots) \\ c_{malloc} &\stackrel{def}{=} ((\text{E}, \text{global}), \dots) & reg &\in \text{Reg} \\ m &\stackrel{def}{=} ms_{f1} \uplus ms_{flag} \uplus ms_{link} \uplus ms_{adv} \uplus ms_{malloc} \uplus ms_{stk} \uplus ms_{frame} \end{aligned}$$

where each of the capabilities have an appropriate range of authority and pointer[11]*. Furthermore*

- c_{malloc} *satisfies the specification for malloc with* $\iota_{malloc,0}$
- $ms_{malloc} :_n [0 \mapsto \iota_{malloc,0}]$
- ms_{f1} *contains* c_{link}, c_{flag} *and the code of* `f1`
- $ms_{flag}(flag) = 0$
- ms_{link} *contains* c_{adv} *and* c_{malloc}
- ms_{adv} *contains* c_{link} *and otherwise only instructions.*

If $(reg[\text{pc} \mapsto c_{f1}][r_{stk} \mapsto c_{stk}], m) \rightarrow^* (halted, m')$, *then* $m'(flag) = 0$ ◊

To prove Lemma 2.8.1, it suffices to show that the start configuration is safe (in the $\mathcal{O}$ relation) for a world with a permanent region that requires the assertion flag to be 0. By Lemma 2.7.1, it suffices to show that the configuration is safe after some reduction steps. We then use the `scall` lemma (Lemma 2.7.2) by which it suffices to show that (1) the configuration that `scall` will jump to is safe and (2) the configuration just after `scall` is done cleaning up is safe. We use the Fundamental Theorem to reason about the unknown adversarial code as described in Section 2.7, but notice that the adversary capability is an enter capability, which the Fundamental Theorem says nothing about. Luckily, the enter capability has RX-permission after the jump at which point the Fundamental Theorem applies.

We have a similar lemma for `f2`:

Lemma 2.8.2 (`f2` is correct)**.** *Making similar assumptions about capabilities and linking as in Lemma 2.8.1 but assuming no stack pointer and assuming* c_{f2} *points to* `f2`, *if* $(reg[\text{pc} \mapsto c_{f2}], m) \rightarrow^* (halted, m')$, *then* $m'(flag) = 0$. ◊

[11] These assumptions are kept intentionally vague for brevity. Full statements are in the Appendix.

```
1 f3: push 1
2     fetch r1 adv
3     scall r1([],[r1])
4     pop r2
5     assert r2 1
6     push 2
7     scall r1([],[])
8     halt
```

```
1 g1:  malloc r2 1
2      store r2 0
3      move pc r3
4      lea r3 off
5      crtcls [(x,r2)] r3
6      rclear RegName \ {pc,r0,r1}
7      jmp r0
8 f4:  reqglob r1
9      prepstk rstk
10     store x 0
11     scall r1([],[r0,r1,renv])
12     store x 1
13     scall r1([],[r0,renv])
14     load r1 x
15     assert r1 1
16     mclear rstk
17     rclear RegName \ {r0,pc}
18     jmp r0
```

Figure 2.15: Two programs that rely on well-bracketedness of `scall`s to function correctly. The variable *off* is the offset to `f4`.

2.8.2 Well-Bracketed Control-Flow

The stack-based calling convention `scall` ensures well-bracketed control-flow. This is illustrated by program examples `f3` and `g1` in Figure 2.15.

The program `f3` has two calls to an adversary. In order for the assertion on line 5 to succeed, the calls must be well-bracketed. If the adversary were able to store the return pointer from the first call and invoke it in the second call, then `f3` would have 2 on top of its stack and the assertion would fail. However, the security measures in Section 2.3 prevent this attack. Specifically, the return pointer is local, so it can only be stored on the stack. However, the part of the stack that is accessible to the adversary is cleared before the second invocation preventing attempts to store the return pointer. In fact, the following lemma shows that there are also no other attacks that can break well-bracketedness of this example, i.e. the assertion never fails. It is similar to the two previous lemmas:

Lemma 2.8.3. *Making similar assumptions about capabilities and linking as in Lemma 2.8.1 and assuming c_{f3} points to `f3` if* $(reg[\text{pc} \mapsto c_{f3}][r_{stk} \mapsto c_{stk}], m) \rightarrow^* (halted, m')$, *then* $m'(flag) = 0$. ◊

The final example, `g1` with `f4`, is a faithful translation of a tricky example known from the literature (known as the awkward example) [36, 76]. For comparison, we show a version of the original example in Figure 2.16, highlighting the code locations corresponding to `g1` and `f4`.

```
1 fun _ =>
2 g1: let x = ref 0 in
3     fun adv =>
4 f4:   x := 0;
5       adv();
6       x := 1;
7       adv();
8       assert(x == 1)
```

Figure 2.16: The original awkward example from Dreyer et al. [36], Pitts and Stark [76], in ML notation.

Let us first look at this ML program. At the top-level, it is a lambda function that can be invoked by the context. When it is invoked, it allocates a fresh mutable variable x of integer type that initially contains the value 0. Next, the function returns a second closure (let's call this cl_x) that can be invoked by the context whenever it chooses. cl_x itself takes a callback function adv that it will invoke twice after setting x to 0 and 1 respectively. After the second invocation of adv, cl_x will verify that x is still set to 1.

The assertion on line 8 of Figure 2.16 should never fail. From the code, this seems natural since adv does not have access to x. Therefore after the second invocation on line 7, x should still be in the state it was before that invocation. However, adv could have access to cl_x and could invoke cl_x again within the second invocation of adv. Then cl_x would set x to 0 again (on line 5) and reinvoke another callback. If that second callback were somehow able to skip its own caller and return directly to the caller of adv, it would end up on line 8 of Figure 2.16 with x set to 1, causing the assertion to fail. Without going into details, such an attack is perfectly possible if adv has access to a call/cc primitive (or equivalent). In other words, if we are able to prove that the assertion will never fail, this attack, and other similar attacks, on well-bracketed control flow are adequately prevented.

Our low-level version of the awkward example in Figure 2.15, consists of two parts, g1 and f4, corresponding to the code locations marked in Figure 2.16. The program g1 corresponds to the top-level closure in the ML code. It is a closure generator that generates closures with a mutable variable x set to 0 in its environment and f4 as the program (note that we omit some calling convention security measures because the stack is not used in the closure generator).

The program f4 expects one argument, the callback adv, in r1. f4 sets x to 0 and invokes adv. When it returns, f4 sets x to 1 and calls adv again. When it returns the second time, f4 asserts x is still 1 and returns.

This example is more complicated than the previous ones because it involves a closure invoked by the adversary and an adversary callback invoked by us. As explained in Section 2.3, we need to check that (1) the stack pointer that the closure receives from the adversary has write-local permission and (2) the adversary callback is global.

The attack that we explained above is actually more natural at this low-level. The callback adv now gets a return capability. As explained, it could invoke cl_x again, during its second invocation, with a second callback function adv′ for cl_x to invoke. If adv had some way of passing its return capability to adv′ (by storing it on the stack or in the heap, or hide it in an unused

register), then the assertion could be made to fail. However, our calling convention prevents any of this from happening as we prove in the following lemma.

Lemma 2.8.4. *Let*

$$c_{adv} \overset{def}{=} ((\text{RWX}, \text{global}), \dots) \quad c_{g1} \overset{def}{=} ((\text{E}, \text{global}), \dots)$$

and otherwise make assumptions about capabilities and linking similar to Lemma 2.8.1. Then if $(reg_0[\text{pc} \mapsto c_{adv}][r_{stk} \mapsto c_{stk}][r_1 \mapsto c_{g1}], m) \rightarrow^* (halted, m')$*, then* $m'(flag) = 0$. ◊

Proof sketch. Define a world W_1 with the following regions

- A malloc region, $\iota_{malloc,0}$.
- A permanent region for the linking table that only accepts ms_{link} and requires everything to be in $\mathcal{V}$.
- A stack region, $\iota^{pwl}_{b_{stk},e_{stk}}$.
- An adversary region, $\iota^{nwl,p}_{b_{adv},e_{adv}}$
- A permanent static region for the flag and `g1`, i.e. a region that only accepts ms_{flag} and ms_{g1}.

If we can show

$$\left(\begin{array}{l} reg_0[\text{pc} \mapsto c_{adv}, r_{stk} \mapsto c_{stk}, r_1 \mapsto c_{g1}], \\ ms_{malloc} \uplus ms_{link} \uplus ms_{stk} \uplus ms_{adv} \uplus ms_{flag} \uplus ms_{g1} \end{array} \right) \in \mathcal{O}(W_1), \tag{2.1}$$

[12]then the $\mathcal{O}$-relation ensures that a successfully halting configuration terminates in a memory that respects a private future world of W which in particular means that it respects the permanent static region that governs the assertion flag.

As described in Section 2.7, we use the FTLR (Theorem 2.4.4) to reason about unknown code, so we use it to reason about c_{adv}. With the standard region $\iota^{nwl,p}$ chosen for the adversary, it is easy to show that the *readCond* and *writeCond* holds for c_{adv} which gives us $c_{adv} \in \mathcal{E}(W)$ by the FTLR. In order to get (2.1), we need to show that (a) the memory satisfies the world and (b) the register file is in the $\mathcal{R}(W)$. We have defined the world, so there (almost) is a one-to-one correspondence between regions in the world and memory segments in the initial configuration, so (a) easily follows. In order to show (b), we need to show $c_{stk} \in \mathcal{V}(W)$ and $c_{g1} \in \mathcal{V}(W)$. The former follows easily from the choice of a ι^{pwl}-region for the stack in W. In order to argue the latter,

[12]We ignore step indexes in this proof sketch.

we basically have to argue that c_{g1}, the closure generator, respects the world W. This amounts to showing that the closures generated by `g1` also respect the invariants of W. We reason about the local variable x in the closure in the same way Dreyer et al. [36] does. We ignore x in the remainder of this proof sketch to focus on the parts of the proof related to the setting of the capability machine.

The capability for the generated closure is a global enter capability that we call c_{f4}. The remainder of the proof amounts to showing that it is safe to return c_{f4} to the adversary, i.e. $c_{f4} \in \mathcal{V}(W')$ where W' is W with relevant regions added after executing `g1`. The adversary can use c_{f4} whenever, so all we may assume about the configuration that c_{f4} is invoked in is that the register-file reg and memory ms satisfies a world W_1 that is a private future world of W', i.e. $reg \in \mathcal{R}(W_1)$ and $ms : W_1$. We have to show that the invocation respects W_1 which corresponds to showing $(reg[\text{pc} \mapsto updPcPerm(c_{f4})], ms) \in \mathcal{O}(W_1)$. When c_{f4} is invoked, we know exactly what instructions are executed up until the `scall`, namely it is ensured that the callback is global and the stack pointer has read/write-local/execute permission, and x is set to 0. Because we know part of the execution, we can apply the anti-reduction lemma (Lemma 2.7.1).

The next part in the execution is an `scall`, so, according to Section 2.7, we should apply the `scall` lemma (Lemma 2.7.2). For the sake of presentation, we here sketch some of the things the `scall` lemma actually takes care of. Based on the stack pointer's permission, we know by Lemma 2.4.3 that the capability is local (because it is a system wide assumption that there are no global read/write-local/execute capabilities *cf.* Section 2.3) which means that the region that governs it must be temporary. This allows us to construct a world $W_2 \sqsupseteq^{priv} W_1$ where the stack region of W_1 has been revoked. In W_2, two new regions govern the stack. One of the two new regions governs the caller stack frame, i.e. the part of the stack that contains the contents of private registers and stack recovery code. The region that takes care of this is a temporary static region which ensures that our local stack frame remains unchanged during the execution of the callback. The other new region takes care of the unused part of the stack which we are going to let the callback use for its execution. This is taken care of by a standard region ι^{pwl}-region which allows the callback to store any safe value on the stack. The callback is global, so it is safe to invoke it in a private future world of W_1 (even though the code is unknown, we do not need to use the FTLR because we assume that the arguments for the invocation of c_{f4} are safe). Notice that had the callback been local, this would not be the case, but it would also not be safe to invoke as it might be a stack pointer as discussed in Section 2.3. Note also that arguing that the memory satisfies W_2 when we invoke the callback only works because we cleared the stack entirely (including the unused part) before the invocation. Otherwise, it might contain local values that are only known to be safe in W_1, but for which we do not know that they will remain

safe in the private future world W_2. This part corresponds to arguing about Hyp-Callee in the `scall` lemma.

We need to argue that it is safe to give the return pointer that `scall` constructs to the adversary which corresponds to reasoning about the remainder of the execution of `f4` (corresponding to the Hyp-Cont case of the `scall` lemma). The return pointer is a local capability, so we may assume that it is invoked in some configuration that satisfies $W_3 \sqsupseteq^{pub} W_2$. This means that none of the temporary regions have been revoked, so the regions we replaced the old stack region with are still present in W_3. We know exactly how the execution proceeds upon return (the recovery code is executed, and x is set to 1). We use the anti-reduction lemma to reason about the execution until the second `scall`.

For the `scall` itself we apply the `scall` lemma. As the private stack has changed, we need to replace the two regions that govern the stack. This means that the second invocation of the callback takes place in a world $W_4 \sqsupseteq^{priv} W_3$. The reasoning about `scall` is similar to the first callback invocation. The callback capability is still safe because it is global which basically covers the Hyp-Callee case. For the Hyp-Cont case, we get a world $W_5 \sqsupseteq^{pub} W_4$ in which we need to argue that the remainder of the execution is safe. At this point, we can use the anti-reduction lemma one final time to get to the point where we jump to the return pointer.

If we can argue that it is safe to jump to the return pointer that we got initially from the adversary, then the proof is done. We have made no checks on the return pointer, so we have no idea whether it is local or global[13]. W.l.o.g. assume the return pointer is local. This means that it is safe in public future worlds of W_1 (remember that it came from *reg* and $reg \in \mathcal{R}(W_1)$). Hence we need to construct a world W_6 with all the temporary and revoked regions of W_1 (this corresponds to restoring the invariants the adversary relies on for its safe execution). Further, for W_6 to be a public future world, for all these regions, W_6 must use the same region names as W_1. As we used the anti-reduction lemma to get to this point, W_6 must also be constructed to be a private future world of W_5. Before we construct W_6, let us consider where each of the worlds we have seen so far came from: As illustrated in Figure 2.17, we constructed W_2 and W_4. These are the only private future worlds we have considered so far. This means that we know exactly what changes they made. In particular, we did not mask any of the temporary or revoked regions in W_1 with a permanent region. This means that in W_6, we can reinstate every temporary or revoked region of W_1. In the private future worlds we constructed, we only took transitions that are public relative to W_1. The worlds we were given, W_3 and W_5, may have taken arbitrary public transitions, but this is no problem with respect to the public future world

[13] We may assume that it is a capability that is executable when jumped to since otherwise the execution fails which is considered safe.

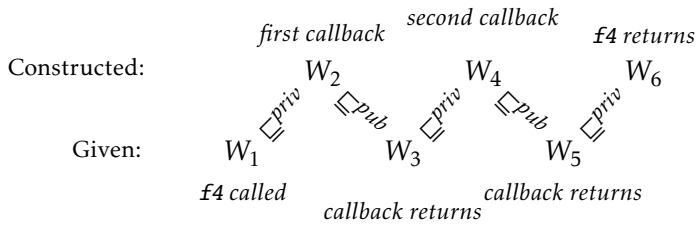


Figure 2.17: Illustration of the worlds in the proof of Lemma 2.8.4. In the proof, the top row of worlds are constructed by us, while the bottom row of worlds are given. W_6 is constructed such that $W_6 \sqsupseteq^{priv} W_5$ and $W_6 \sqsupseteq^{pub} W_1$.

relation. In W_5 there were regions to handle the stack. These regions need to be revoked as the W_1 stack regions replace them. The non-stack regions in W_5 that are not present in W_1 are also added in W_6, which the public future world relation permits as it is extensional. All in all, it is possible to construct W_6, so it is both a public future world of W_1 and a private world of W_5 which means that it is indeed safe to return to the adversary.

For the sake of presentation, we have omitted many details and made several simplifications in the above proof. The complete proof can be found in the technical appendix [84]. □

2.9 Discussion

Calling convention

Formulating control flow correctness While we claim that our calling convention enforces control-flow correctness, we do not prove a general theorem that shows this because it is not clear what such a theorem should look like. Formulations in terms of a control-flow graph, like the one by Abadi et al. [10], creates a view with all capabilities that may be available at some point in time. Hence control-flow-graph based formulations lead one to consider capabilities that are actually not available at a given point in time. Example `g1` relies on a more fine-grained view of the control flow, in particular when returning from the higher-order callbacks. In fact, our examples show that our logical relation implies a stronger form of control-flow correctness than control-flow-graph based formulations, although this is not made very explicit. In later work, Skorstengaard et al. [86] provide a more explicit and useful definition of control-flow correctness and local state encapsulation. The idea of their definition is to define a variant of a simple capability machine with a built-in stack as well as call and return instructions. The call and return instructions use the built-in stack which gives well-bracketed control and local state encapsulation by construction. They prove full abstraction [8] for a compilation from the capability machine with the built-in stack to one

without which means that the programs running on the capability machine without a built-in stack behave as though there was a built-in stack. Their full abstraction proof uses a logical relation, but it differs in a number of ways from the one we present. Their logical relation is a binary cross language logical relation tailored specifically to prove the full abstraction result. This means that it could not be used to prove program correctness results similar to ours. Their calling convention is based on something called linear capabilities rather than local capabilities. Linear capabilities offer a different kind of limited revocation.

Performance and the requirement for stack clearing The additional security measures of the calling convention described in Section 2.3 impose an overhead compared to a calling convention without security guarantees. However, most require only a few atomic checks or register clearings on boundary crossings between trusted code and adversary, which should produce an acceptable performance overhead. The only exception is the stack clearing requirement that we have in two situations: when returning to the adversary and when invoking an adversary callback. As we have explained, we need to clear all of the stack that we are not using ourselves not just the part that we have actually used. In other words on every boundary cross between trusted code and adversary code, a potentially large region of memory must be cleared.

First, contrary to what we explained before, we actually suspect that this overhead can be avoided when returning to the adversary. In that case, we now think it would suffice to clear only the (much smaller) part of the stack used by the trusted code itself. To understand this, it is useful to take another look at the illustrations in freffig:ret-adv2 related to this case. If we do not clear the stack upon return (as illustrated in Figure 2.5f), then the adversary might have used that stack to store local capabilities they received in a previous invocation (see Figure 2.5e). In other words, the stack clearing is necessary for revoking those local capabilities: the stack pointer and return pointer for the invocation illustrated in Figure 2.5e. While this approach is safe, we now suspect that we could do without the revocation in this case. The stack pointer which the adversary was given access to in the stack frame depicted in Figure 2.5d only carries authority that the adversary has access to in the higher stack frame anyway. Similarly, the return pointer is merely an entry pointer pointing into the trusted code's stack frame, and this is also a capability that the adversary we return to could have constructed themselves from their stack pointer.

This improvement of our calling convention is a recent insight and not yet reflected in our examples and our proofs. We do believe the proofs could be updated to accomodate for this change, but it would require some non-trivial changes. Consider the awkward example proof, for example, we would have to ensure that the world W_6, depicted in Figure 2.17 would not just be a private future world of W_5, but a public one (in addition to being a public future

world of W_1). This would allow us to do without the stack clearing, but it would entail some changes to the regions we use, and an extra argument that the old adversary's stack and return pointer remain valid after clearing the trusted code's stack frame and relaxing the invariant that used to ensure it could not be modified. While this change is a clear improvement, we do not actually believe that it fundamentally changes the efficiency characteristics of the approach: the cost is halved, but remains asymptotically the same.

A second important remark we want to make is that the need for stack clearing in our calling convention is an instance of a general caveat when using CHERI's local capabilities as a restricted form of capability revocation. Consider how our use of local capabilities can be interpreted as temporarily delegating the stack and return capability to callees and then revoking the granted authority after the callee returns. From this perspective, local capabilities are a general feature enabling this temporary delegation of authority for the duration of an invocation and this is also how they are described by the authors [69]. However, our requirement for stack clearing on boundary crossings is also general. Revoking authority that was granted temporarily using local capabilities requires clearing all memory for which the invokee had write-local authority (or at least erasing all local capabilities from that memory). Without micro-architectural support for efficiently clearing large ranges of memory, local capabilities can only be used for revocation in scenarios where the duration of a revocation is unimportant or the adversary only has write-local access to small amounts of memory.

CheriBSD's use of local capabilities in CCall does not actually involve a form of revocation. CheriBSD's model involves a trusted stack manager that gives every compartment access to its own private stack using a local, write-local capability [69]. The locality of the stack capability allows the trusted stack manager to prevent compartments from leaking their stack pointer in a boundary crossing, but those capabilities are never actually revoked. In fact, a compartment can easily store away such local capabilities in its own private stack and recover them there during future invocations.

Since local capabilities seem intended to provide a restricted form of revocation, perhaps capability machines like CHERI should consider to provide special support for this requirement. Ideally, such support would take the form of a highly-optimized instruction for erasing a large block of memory. Recent work suggests that such a feature could perhaps be added to processors like CHERI, using a special hardware cache that tracks whether or not a memory location contains zero [48].

Modularity It is important that our calling convention is modular, i.e. we do not assume that our code is specially privileged w.r.t. the adversary, and they can apply the same measures to protect themselves from us as we do to protect ourselves from them. More concretely, the requirements we have on callbacks and return pointers received from the adversary are also satisfied by callbacks and return pointers that we pass to them. For example,

our return pointers are local capabilities because they must point to memory where we can store the old stack pointer, but the adversary's return pointers are also allowed to be local. Adversary callbacks are required to be global, but the callbacks we construct are allocated on the heap and also global.

Arguments and local capabilities Local capabilities are a central part of the calling convention as they are used to construct stack and return pointers. The use of local capabilities for the calling convention unfortunately limits the extent to which local capabilities can be used for other things. Say we are using the calling convention and receive a local capability other than the stack and return pointer, then we need to be careful if we want to use it because it may be an alias to the stack pointer. That is, if we first push something to the stack and then write to the local capability, then we may be (tricked into) overwriting our own local state. The logical relation helps by telling us what we need to ascertain or check in such scenarios to guarantee safety and preserve our invariants, but such checks may be costly and it is not clear to us whether there are practical scenarios where this might be realistic.

We also need to be careful when we receive a capability from an adversary that we want to pass on to a different (instance of the) adversary. It turns out that the logical relation again tells us when this is safe. Namely, the logical relation says that we can only pass on arguments that are safe in the world we invoke the adversary in. For instance, when we receive a stack pointer from an adversary, then we may at some point want to pass on part of this stack pointer to, say, a callback. In order to do so, we need to make sure the stack pointer is safe which means that, if we have revoked temporary invariants, the stack must not directly or indirectly allow access to local values that we cannot guarantee safety of. When received from an adversary, we have to consider the contents of the stack unsafe, so before we pass it on, we have to clear it, or perform a dynamic safety analysis of the stack contents and anything it points to. Clearing everything is not always desirable and a dynamic safety analysis is hard to get right and potentially expensive.

In summary, the use of local capabilities for other things than stack and return pointers is likely only possible in very specific scenarios when using our calling convention. While this is unfortunate, it is not unheard of that processors have built-in constructs that are exclusively used for handling control flow, such as, for example, the call and return instructions that exist in some instruction sets.

Single stack A single stack is a good choice for the simple capability machine presented here because it works well with higher-order functions. An alternative to a single stack would be to have a separate stack per component. The trouble with this approach is that, with multiple stacks and local stack pointers, it is not clear how components would retrieve their stack pointer upon invocation without compromising safety. A safe approach could be to have stack pointers stored by a central, trusted stack management component, but it is not clear how that could scale to large numbers of separate

components. Handling large numbers of components is a requirement if we want to use capability machines to enforce encapsulation of, for example, every object in an object-oriented program or every closure in a functional program.

Capability machine formalization

Simplifications Our capability machine formalization assumes unbounded integers and an infinite address space. Further, it has a much simpler instruction set than that of a full ISA. By making these simplifications, we avoid tedious details but end up with a less realistic machine. However, the intent of this work is to gain ground in the formal work necessary to prove properties about low-level assembly programs on a capability machine not to apply it to a full fledged capability machine.

We do not believe that any of our simplifications are beyond reason and expanding the result seems plausible. If we added bounded integers, we would have to change the operational semantics to take over and under flows into account. This could be achieved by changing the `plus` and `minus` instruction to use modulo arithmetic. If we added a bounded address space, then we would have to take memory out of bounds into account. This would require changes to the memory operations (`store` and `load`) and the step relation of the operational semantics. The malloc specification would also need to be updated as it would have to signal failure when it runs out of memory. Finally, expanding the instruction set to a realistic ISA would add a handful of instructions that we would need to reason about in the proofs. All in all, it would add a lot of tedious work to expand this result to a realistic machine. The amount of details in the proofs is already at the threshold for what should be done with pen and paper, so expanding this work to a realistic machine would require mechanized proofs that can take care of the tedious details.

Reasoning about capability machine programs

Limitations The logical relation defined in Section 2.4 allows us to reason about capability machine programs. A limitation w.r.t. previous work is that the logical relation is tailored towards proofs of a specific class of properties.

Imagine, for example, that we invoke a block of adversary code in such a way that it only ever receives capabilities within a specific range of memory. After the code returns, we may try to prove that any capabilities passed back to us in the registers are still confined to that range of memory. The property talks about the specific implementation of a higher-order value rather than its behavior, like the invariants that are required/preserved when we use it.

Such properties are hard to prove in our model. For the example, it would be easy to conclude that the returned values are in the value relation (see Figure 2.9). This gives us a lot of behavioral information, like conditions under which the capabilities are safe to use and invariants that will be preserved

when we do, but it does not tell us much about the range of authority of the capability. As a very concrete example, capabilities with permission o are always in the value relation irrespective of their range of authority. Behaviorally, this makes perfect sense; o-capabilities cannot be used for anything.

For our purposes, this restriction is unproblematic since we are only interested in proving behavioral properties (e.g., an assertion will never fail). In other situations, however, we may be interested in proving properties like the ones that are often considered in object capability literature: confinement, no authority amplification etc. Although such properties are more restrictive and tough to use for reasoning, Devriese et al. [33] have demonstrated how a logical relation like ours can be adapted to also support them by quantifying the logical relation over a custom interpretation of effectful computations and the type of references. We expect their solution can be readily adapted to our setting modulo some details (like the fact that we do not just have read-write capabilities but also others).

Logical relation

Single orthogonal closure The definitions of $\mathcal{E}$ and $\mathcal{V}$ in Figure 2.9 apply a single orthogonal closure, a new variant of an existing pattern called biorthogonality. Biorthogonality is a pattern for defining logical relations [57, 76] in terms of an observation relation of safe configurations (like we do). The idea is to define safe evaluation contexts as the set of contexts that produce safe observations when plugged with safe values and define safe terms as the set of terms that can be plugged into safe evaluation contexts to produce safe observations. This is an alternative to more direct definitions where safe terms are defined as terms that evaluate to safe values. An advantage of biorthogonality is that it scales better to languages with control effects like call/cc. Our definitions can be seen as a variant of biorthogonality; we take only a single orthogonal closure. We do not define safe evaluation contexts but immediately define safe terms as those that produce safe observations when plugged with safe values. This is natural because we model arbitrary assembly code that does not necessarily respect a particular calling convention. Return pointers are in principle values like all others and there is no reason to treat them specially in the logical relation.

Interestingly, Hur and Dreyer [47] also use a step-indexed, Kripke logical relation for an assembly language (for reasoning about correct compilation from ML to assembly). However, because they only model non-adversarial code that treats return pointers according to a particular calling convention, they can use standard biorthogonality rather than a single orthogonal closure like us.

Public/private future worlds A novel aspect of our logical relation is how we model the temporary, revokable nature of local capabilities using public/private future worlds. The main insight is that this special nature generalizes that of the syntactically-enforced unstorable status of evaluation con-

texts in lambda calculi without control effects (of which well-bracketed control flow is a consequence). To reason about code that relies on this (particularly, the original awkward example), Dreyer et al. [36] (DNB) formally capture the special status of evaluation contexts using Kripke worlds with public and private future world relations. Essentially, they allow relatedness of evaluation contexts to be monotone with respect to a weaker future world relation (public) than relatedness of values, formalizing the idea that it is safe to make temporary internal state modifications (private world transitions, which invalidate the continuation, but not other values) while an expression is performing internal steps, as long as the code returns to a stable state (i.e. transitions to a public future world of the original) before returning. We generalize this idea to reason about local capabilities: validity of local capabilities is allowed to be monotone with respect to a weaker future-world relation than other values, which we can exploit to distinguish between state changes that are always safe (public future worlds) and changes that are only valid if we clear all local capabilities (private future worlds). Our future world relations are similar to DNB's (for example, our proof of the awkward example uses exactly the same state transition system), but they turn up in an entirely different place in the logical relation: rather than using public future worlds for the special syntactic category of evaluation contexts, they are used in the value relation depending on the locality of the capability at hand. Additionally, our worlds are a bit more complex because, to allow local memory capabilities and write-local capabilities, they can contain (revokable) temporary regions that are only monotonous w.r.t. public future worlds, while DNB's worlds are entirely permanent.

Local capabilities in high-level languages We point out that local capabilities are quite similar to a feature proposed for the high-level language Scala: Osvald et al. [72]'s second-class or local values. They are a kind of values that can be provided to other code for immediate use without allowing them to be stored in a closure or reference for later use. We believe reasoning about such values will require techniques similar to what we provide for local capabilities.

Why use a logical relation and not a simpler proof technique? The concept of a capability exists on different levels of abstraction on computers, so capabilities have been studied and safety properties proven about them in other contexts than assembly languages. Traditionally, logical relations have not been used for safety property proofs, so why do we need one here? To answer this question let's compare this work to Sewell et al. [79] where they prove *integrity* and *authority confinement* for the seL4 Microkernel. The security properties they show are relative to a security policy that specifies the upperbound of capabilities a subject in the system can posses. The security policy is expressed as a collection of possible capabilities in the system, e.g. *subject a can have a write capability for memory B*. However, their policy are, in a sense, binary: subject A is either allowed to write to memory B or not.

If writing is allowed, then any value can be written, as long as that value is itself legal. **?**]'s results are not parametrised by a (possibly more restrictive) policy that subject A should observe on memory B. Our approach is much more fine-grained and allows us to define, for example, a policy that subject A can write to memory B, but only if the value is an even number. Because our invariants are indexed by worlds themselves, we can even define a policy that capabilities written to memory B must respect certain other invariants themselves. It is exactly the expressive invariants that enable us to prove the awkward example. We place a protocol on the memory where variable `x` is stored that says that it sometimes can be either 0 or 1 and at other times it has to be 1.

Capability safety is not just about the permission a capability carries, it is also about how the capability is used. Say for instance, a program has a write capability that will only be used to write even numbers. If this is the only write capability for that part of memory, then we should be able to rely on that part of memory only containing even numbers. In order to rely on this in our proofs, our notion of capability safety must be able to take the meaning of the program into account and express the memory invariant.

2.10 Related Work

In this section, we summarize how our work relates to previous work. We do not repeat the work we discussed in Section 2.9.

Capability machines originate with Dennis and Van Horn [32] and we refer to Levy [60] and N. M. Watson et al. [69] for an overview of previous work. The capability machine formalized in Section 2.2 is a simple but representative model modeled mainly after the M-Machine [24] (the enter pointers resemble the M-Machine's) and CHERI [69, 100] (the memory and local capabilities resemble CHERI's). The latter is a recent and relatively mature capability machine. CHERI combines capabilities with a virtual memory approach in the interest of backwards compatibility and gradual adoption. As discussed, our local capabilities can cross module boundaries contrary to what is enforced by CHERI's default CCall implementation.

Plenty of other papers enforce well-bracketed control flow at a low level but most are restricted to preventing particular types of attacks and enforce only partial correctness of control flow. This includes particularly the line of work on *control-flow integrity* [10]. This line of work uses a quite different attacker model than us. They assume an attacker that is unable to execute code but can overwrite arbitrary data at any time during execution (to model buffer overflows). By checking the address of every indirect jump and using memory access control to prevent overwriting code, this work enforces what they call control-flow integrity formalized as the property that every jump will follow a legal path in the control-flow graph. As discussed in Sec-

tion 2.9, such a property ignores temporal properties and seems hard to use for reasoning.

More closely related to our work are papers that use a trusted stack manager and some form of memory isolation to enforce control-flow correctness as part of a secure compilation result [50, 74]. Our work differs from theirs in that we use a different form of low-level security primitive (a capability machine with local capabilities rather than a machine with a primitive notion of compartments). Further, we do not use a trusted stack manager but a decentralized calling convention based on local capabilities. Also, both prove a secure compilation result from a high-level language which clearly implies a general form of control-flow correctness while we define a logical relation that can be used to reason about specific programs that rely on well-bracketed control flow.

Our logical relation is a unary, step-indexed Kripke logical relation with recursive worlds [14, 16, 20, 76], closely related to the one used by Devriese et al. [33] to formulate capability safety in a high-level JavaScript-like lambda calculus. Our Fundamental Theorem is similar to theirs and expresses capability safety of the capability machine. Because we are not interested in externally observable side-effects (like console output or memory access traces), we do not require their notion of effect parametricity. Our logical relation uses several ideas from previous work like Kripke worlds with regions containing state transition systems [13], public/private future worlds [36] (see Section 2.9 for a discussion), and biorthogonality [17, 47, 76].

Swasey et al. [87] have recently developed a *logic*, OCPL, for verification of object capability patterns. The logic is based on Iris [51, 52, 55], a state of the art higher-order concurrent separation logic, and is formalized in Coq building on the Iris Proof Mode for Coq [56]. OCPL gives a more abstract and modular way of proving capability safety for a lambda-calculus (with concurrency) compared to the earlier work by Devriese et al. [33]. In the future, we would like to develop a new program logic for reasoning about capability safety for our capability machine model. In fact, we think the lemmas in Section 2.7 are suggestive of the style of results that could be captured in such a logic. We think Iris would also be a natural starting point for such an endeavour since Iris is a framework that can be instantiated with different programming languages. OCPL was able to leverage existing Iris specifications for a high-level programming language; for our capability machine model, however, it would be necessary to devise new kinds of specifications for our low-level programs with unstructured control-flow. It is likely that we could get inspiration from earlier work on logics for assembly programming languages, such as XCAP [71]. Building a logic around the semantic model presented here would remove some of the tedious repetitive proof details making it more realistic to prove properties about larger more realistic programs.

If we want to scale this approach even further, we would like to reason at

a higher level of abstraction namely at the level of a high-level language. That is, we would rather construct a program logic for a high-level programming language, so we can reason about the programs in the language we actually write them in and at the same time get guarantees about the compiled program. To achieve this, we would have to construct a secure compilation [75] that preserves the security abstraction of the high-level language.

El-Korashy also defined a formal model of a capability machine, namely CHERI, and uses it to prove a compartmentalization result [37] (not implying control-flow correctness). He also adapts control-flow integrity (see above) to the machine and shows soundness, seemingly without relying on capabilities.

Acknowledgements

This research was supported in part by the ModuRes Sapere Aude Advanced Grant from The Danish Council for Independent Research for the Natural Sciences (FNU). Support for an STSM was received from COST Action EUTypes (CA15123). Dominique Devriese holds a Postdoctoral fellowship from the Research Foundation Flanders (FWO).

2.A Appendix

In this appendix, we give more precise formulations of lemmas that were mentioned in the paper, and the most important supporting definitions and lemmas. The goal is to provide details that can help to understand in more detail what we discuss in the paper. Full details and proofs are not given here, but for those we refer to the technical appendix [84].

2.A.1 Logical relation

n-subset simulation

$$\frac{(s, \phi_{pub}, \phi) = (s', \phi'_{pub}, \phi') \qquad \forall \hat{W}.\, H\,(s)\,(\hat{W}) \overset{n}{\subseteq} H'\,(s')\,(\hat{W})}{(v, s, \phi_{pub}, \phi, H) \overset{n}{\lesssim} (v', s', \phi'_{pub}, \phi', H')}$$

where $\overset{n}{\subseteq}$ is defined as follows: define erasure for step-indexed sets as

$$\lfloor A \rfloor_n = \{(m, a) \in A \mid m < n\}$$

and define $\overset{n}{\subseteq}$ as

$$A \not\subseteq B \;\; \texttt{iff} \;\; \lfloor A \rfloor_n \subseteq \lfloor B \rfloor_n$$

Transition system relations

$$\text{Rels} = \left\{ (\phi_{pub}, \phi) \in \mathcal{P}(\text{RState}^2) \times \mathcal{P}(\text{RState}^2) \,\middle|\, \begin{array}{l} \phi_{pub}, \phi \text{ are reflexive and} \\ \text{transitive and } \phi_{pub} \subseteq \phi \end{array} \right\}$$

Erasure

$$\lfloor W \rfloor_S \overset{def}{=} \lambda r. \begin{cases} W(r) & W(r).v \in S \\ \bot & \text{otherwise} \end{cases}$$

Active region projection

$$active : \text{World} \to 2^{\text{RegionName}}$$
$$active(W) \overset{def}{=} \text{dom}(\lfloor W \rfloor_{\{\text{perm,temp}\}})$$

Revoke temporary regions in a world

$$revokeTemp : \text{World} \to \text{World}$$
$$revokeTemp(W) \overset{def}{=} \lambda r. \begin{cases} \text{revoked} & \text{if } W(r) = (\text{temp}, s, \phi_{pub}, \phi, H) \\ W(r) & \text{otherwise} \end{cases}$$

Projection of regions based on locality

$$localityReg(g, W) \stackrel{def}{=} \begin{cases} \mathrm{dom}(\lfloor W \rfloor_{\{\mathrm{perm,temp}\}}) & \text{if } g = \text{local} \\ \mathrm{dom}(\lfloor W \rfloor_{\{\mathrm{perm}\}}) & \text{if } g = \text{global} \end{cases}$$

Address stratification

$$\begin{aligned} &\iota = (v, s, \phi_{pub}, \phi, H) \text{ is address-stratified iff} \\ &\quad \forall s', \hat{W}, n, ms, ms'. \\ &\qquad (n, ms), (n, ms') \in H\ s'\ \hat{W} \Rightarrow \\ &\qquad\quad \mathrm{dom}(ms) = \mathrm{dom}(ms') \wedge \\ &\quad \forall a \in \mathrm{dom}(ms).\, (n, ms[a \mapsto ms'(a)]) \in H\ s'\ \hat{W} \end{aligned}$$

2.A.2 Complete ordered family of equivalences (c.o.f.e)

This is an excerpt from Birkedal and Bizjak [18] about c.o.f.e.'s.

Definition 2.A.1 (o.f.e.)**.** An *ordered family of equivalence* (o.f.e) is a set and a family of equivalences $\left(X, \left(\stackrel{n}{=}\right)_{n=0}^{\infty}\right)$ that satisfy the following properties:

- $\stackrel{0}{=}$ is the total relation on X
- $\forall n. \forall x, y \in X. x \stackrel{n+1}{=} y \Rightarrow x \stackrel{n}{=} y$
- $\forall x, y \in X. (\forall n. x \stackrel{n}{=} y) \Rightarrow x = y$

We say that an o.f.e. $\left(X, \left(\stackrel{n}{=}\right)_{n=0}^{\infty}\right)$ is *inhabited* if there exists an element $x \in X$. ⧫

If you are familiar with metric spaces observe that o.f.e.'s are but a different presentation of bisected 1-bounded ultrametric spaces.

Definition 2.A.2 (Cauchy sequences and limits)**.** Let $\left(X, \left(\stackrel{n}{=}\right)_{n=0}^{\infty}\right)$ be an o.f.e. and $\{x_n\}_{n=0}^{\infty}$ be a sequence of elements of X. Then $\{x_n\}_{n=0}^{\infty}$ is a *Cauchy sequence* if

$$\forall k \in \mathbb{N}, \exists j \in \mathbb{N}, \forall n \geq j, x_j \stackrel{k}{=} x_n$$

or in words, the elements of the chain get arbitrarily close.

An element $x \in X$ is the *limit* of the sequence $\{x_n\}_{n=0}^{\infty}$ if

$$\forall k \in \mathbb{N}, \exists j \in \mathbb{N}, \forall n \geq j, x \stackrel{k}{=} x_n.$$

A sequence may or may not have a limit. If it has we say that the sequence *converges*. The limit is necessarily unique in this case and we write $\lim_{n\to\infty} x_n$ for it. ⧫

Definition 2.A.3 (c.o.f.e.)**.** A *complete* ordered family of equivalences (c.o.f.e) is an o.f.e $\left(X, \left(\overset{n}{=}\right)_{n=0}^{\infty}\right)$ where all Cauchy sequences have a limit. ⧫

Definition 2.A.4. Let $\left(X, \left(\overset{n}{=}_X\right)_{n=0}^{\infty}\right)$ and $\left(Y, \left(\overset{n}{=}_Y\right)_{n=0}^{\infty}\right)$ be two ordered families of equivalences and f a function from the set X to the set Y. The function f is

- *non-expansive* if for any $x, x' \in X$, and any $n \in \mathbb{N}$,

$$x \overset{n}{=}_X x' \implies f(x) \overset{n}{=}_Y f(x')$$

- *contractive* if for any $x, x' \in X$, and any $n \in \mathbb{N}$,

$$x \overset{n}{=}_X x' \implies f(x) \overset{n+1}{=}_Y f(x')$$

⧫

Theorem 2.A.5 (Banach's fixed point theorem)**.** *Let $\left(X, \left(\overset{n}{=}\right)_{n=0}^{\infty}\right)$ be a an inhabited c.o.f.e. and $f : X \to X$ a contractive function. Then f has a unique fixed point.* ◊

Definition 2.A.6 (The category $\mathcal{U}$)**.** The category $\mathcal{U}$ of complete ordered families of equivalences has as objects complete ordered families of equivalences and as morphisms non-expansive functions. ⧫

Definition 2.A.7. The functor $\blacktriangleright$ is a functor on $\mathcal{U}$ defined as

$$\blacktriangleright\left(X, \left(\overset{n}{=}\right)_{n=0}^{\infty}\right) = \left(X, \left(\overset{n}{\equiv}\right)_{n=0}^{\infty}\right)$$
$$\blacktriangleright(f) = f$$

where $\overset{0}{\equiv}$ is the total relation and $x \overset{n+1}{\equiv} x'$ iff $x \overset{n}{=} x'$ ⧫

From now on, we often use the underlying set X to denote a (complete) o.f.e. $\left(X, \left(\overset{n}{=}\right)_{n=0}^{\infty}\right)$, leaving the family of equivalence relations implicit.

Definition 2.A.8. A functor $F : \mathcal{U}^{\mathrm{op}} \times \mathcal{U} \to \mathcal{U}$ is *locally non-expansive* if for all objects X, X', Y, and Y' in $\mathcal{U}$ and $f, f' \in \mathcal{U}(X, X')$ and $g, g' \in \mathcal{U}(Y', Y)$ we have

$$f \overset{n}{=} f' \wedge g \overset{n}{=} g' \implies F(f,g) \overset{n}{=} F(f',g').$$

It is *locally contractive* if the stronger implication

$$f \overset{n}{=} f' \wedge g \overset{n}{=} g' \implies F(f,g) \overset{n+1}{=} F(f',g').$$

holds. Note that the equalities are equalities on function spaces. ⧫

Proposition 2.A.9. *If F is a locally non-expansive functor then $\blacktriangleright \circ F$ and $F \circ (\blacktriangleright^{op} \times \blacktriangleright)$ are locally contractive. Here, the functor $F \circ (\blacktriangleright^{op} \times \blacktriangleright)$ works as*

$$(F \circ (\blacktriangleright^{op} \times \blacktriangleright))(X, Y) = F(\blacktriangleright^{op}(X), \blacktriangleright(Y))$$

on objects and analogously on morphisms and $\blacktriangleright^{op} : \mathcal{U}^{op} \to \mathcal{U}^{op}$ is just $\blacktriangleright$ working on $\mathcal{U}^{op}$ (i.e., its definition is the same). ◊

Definition 2.A.10. A fixed point of a locally contractive functor F is an object $X \in \mathcal{U}$, such that $F(X, X) \cong X$. ⧫

The following is America and Rutten's fixed point theorem [15].

Theorem 2.A.11. *Every locally contractive functor F such that $F(1,1)$ is inhabited has a unique fixed point. The fixed point is unique among inhabited c.o.f.e.'s. If in addition $F(\emptyset, \emptyset)$ is inhabited then the fixed point of F is unique.* ◊

In Birkedal et al. [19] one can find a category-theoretic generalization, which shows how to obtain fixed points of locally contractive funtors on categories enriched in $\mathcal{U}$, in particular on the category of preordered c.o.f.e.'s. A preodered c.o.f.e. is a c.o.f.e. equipped with a preorder that is closed under taking limits of converging sequences. The formulation in *loc. cit.* also applies to solve mutually recursive domain equations on preordered c.o.f.e.'s; see Bizjak [21] for an explicit statement. That is the solution theorem we use to prove Theorem 2.4.1.

2.A.3 Load instruction sufficiency lemma

Lemma 2.A.12 (Conditions for store instruction are sufficient)**.** *If*

- $ms = ms' \uplus ms_f$
- $ms' :_n W$
- $((perm, g), b, e, a) = c$
- $(n, c) \in \mathcal{V}(W)$
- $writeAllowed(perm)$
- $withinBounds(c)$
- $(n, w) \in \mathcal{V}(W)$
- *if* $w = ((\_, \text{local}), \_, \_, \_)$, *then* $perm \in \{\text{RWLX}, \text{RWL}\}$

then $a \in \text{dom}(ms')$ *(i.e.* $ms[a \mapsto w] = ms'[a \mapsto w] \uplus ms_f$*) and* $ms'[a \mapsto w] :_n W$ ◊

2.A.4 Macros

Implementation of the macros used in `scall`. Implementations of the macros not presented here can be found in the technical appendix [84].

`push` r

```
1   lea r_stk 1
2   store r_stk r
```

`pop` r

```
1   load r r_stk
2   minus r_t1 0 1
3   lea r_stk r_t1
```

`rclear` $r_1,\ldots,r_n$

```
1   move r_1 0
2   move r_2 0
3   ...
4   move r_n 0
```

`mclear` r

```
1   move r_t r
2   getb r_t1 r_t
3   geta r_t2 r_t
4   minus r_t2 r_t1 r_t2
5   lea r_t r_t2
6   gete r_t2
7   minus r_t1 r_t2 r_t1
8   plus r_t1 r_t1 1
9   move r_t2 pc
10  lea r_t2 off_continue
11  move r_t3 pc
12  lea r_t3 off_iter
13 iter:
14  jnz r_t2 r_t1
15  lea pc off_varend
16 continue:
17  store r_t 0
18  lea r_t 1
19  minus r_t1 r_t1 1
20  jmp r_t3
21 end:
22  move r_t 0
23  move r_t1 0
24  move r_t2 0
25  move r_t3 0
```

Where $off_{continue}$, off_{iter} and off_{end} are the offsets to the labels `continue`, `iter` and `end`, respectively.

`call` $r(\bar{r}_{args}, \bar{r}_{priv})$

The `call` macro constitutes a calling convention based on heap allocated activation records. This alternative to `scall` is included to illustrate that the

logical relation can be used to reason about other calling conventions. In the following, $\bar{r}_{args}$ and $\bar{r}_{priv}$ are lists of registers. An overview of this call:

- Set up activation record
- Create local enter capability for activation (protected return pointer)
- Clear unused registers
- Jump
- Upon return: Run activation code
  - Restore private registers
  - Jump to return capability

```
1   malloc r_t size
2 // store private state in activation record
3   store r_t r_priv,1
4   lea r_t 1
5   store r_t r_priv,2
6   lea r_t 1
7   ...
8   lea r_t 1
9   store r_t r_priv,n
10  lea r_t 1
11 // store old pc
12  move r_t1 pc
13  lea r_t1 off_end
14  store r_t r_t1
15  lea r_t1 1
16 // store activation record
17  store r_t encode(i_1)
18  lea r_t1 1
19  ...
20  lea r_t1 1
21  store r_t encode(i_m)
22  lea r_t1 k
23  restrict r_t1 encodePermPair((local,e))
24  move r_0 r_t1
25 // Clear unused registers
26  rclear R // R = RegisterName - {r,pc,r_0,r_args}
27  jmp r
28 end:
```

Where $\bar{r_{priv}} = r_{priv,1}, \ldots, r_{priv,n}$, $size$ is the size of the activation record, off_{end} is the offset to the end label, and k is $m-1$, i.e. the offset to the first instruction of the activation code.
The activation record. The instructions correspond to $i_1, \ldots, i_m$ in the above.

```
1   move r_t pc
2   getb r_t1 r_t
```

```
 3   geta r_t2 r_t
 4   minus r_t1 r_t1 r_t2
 5 // load private state
 6   lea r_t r_t1
 7   load r_priv,1 r_t
 8   lea r_t 1
 9   load r_priv,2 r_t
10   lea r_t 1
11   ...
12   lea r_t 1
13   load r_priv,n r_t
14   lea r_t 1
15 // load old pc
16   load pc r_t
```

2.A.5 Reasoning about programs definitions

Definition 2.A.13. We say that (reg, ms) is looking at $[i_0, \cdots, i_n]$ followed by c_{next} iff

- $reg(\text{pc}) = ((p, g), b, e, a)$
- $p = \text{RWX}$, $p = \text{RX}$, or $p = \text{RWLX}$
- $a + n \leq e$, $b \leq a \leq e$
- $ms(a + 0, \cdots, a + n) = [i_0, \cdots, i_n]$
- $c_{next} = ((p, g), b, e, a + n + 1)$

♦

Definition 2.A.14. We say that "(reg, ms) links key as j to c" iff

- $reg(pc) = ((perm, g), b, e, a)$
- $ms(b) = ((\_, \_), b_{link}, \_, \_)$
- $ms(b_{link} + j) = c$

♦

Definition 2.A.15. We say that reg points to stack with ms_{stk} used and ms_{unused} unused iff

- $reg(r_{stk}) = ((\text{RWLX}, \text{local}), b_{stk}, e_{stk}, a_{stk})$
- $\text{dom}(ms_{unused}) = [a_{stk} + 1, \cdots, e_{stk}]$
- $\text{dom}(ms_{stk}) = [b_{stk}, \cdots, a_{stk}]$
- $b_{stk} - 1 \leq a_{stk}$

♦

2.A.6 Example correctness lemmas

Lemma 2.A.16 (Correctness lemma for `f1`, copy of Lemma 2.8.1).
For all $n \in \mathbb{N}$ let

$$
\begin{aligned}
c_{adv} &\stackrel{def}{=} ((\text{E}, \text{global}), b_{adv}, e_{adv}, b_{adv} + \mathit{offsetLinkFlag}) \\
c_{f1} &\stackrel{def}{=} ((\text{RWX}, \text{global}), \text{f}\,1 - \mathit{offsetLinkFlag}, \text{1f}, \text{f}\,1) \\
c_{malloc} &\stackrel{def}{=} ((\text{E}, \text{global}), b_{malloc}, e_{malloc}, b_{malloc} + \mathit{offsetLinkFlag}) \\
m &\stackrel{def}{=} ms_{f1} \uplus ms_{flag} \uplus ms_{link} \uplus ms_{adv} \uplus ms_{malloc} \uplus ms_{frame}
\end{aligned}
$$

and

- *c_{malloc} satisfies the specification for malloc and $\iota_{malloc,0}$ is the region from the specification.*

where

$$
\begin{aligned}
&\text{dom}(ms_{f1}) = [\text{f}\,1 - \mathit{offsetLinkFlag}, \text{1f}] \\
&\text{dom}(ms_{flag}) = [\mathit{flag}, \mathit{flag}] \\
&\text{dom}(ms_{link}) = [\mathit{link}, \mathit{link} + 1] \\
&\text{dom}(ms_{adv}) = [b_{adv}, e_{adv}] \\
&ms_{malloc} :_n [0 \mapsto \iota_{malloc,0}]
\end{aligned}
$$

and

- *$ms_{f1}(\text{f}\,1 - \mathit{offsetLinkFlag}) = ((\text{RO}, \text{global}), \mathit{link}, \mathit{link}+1, \mathit{link})$, $ms_{f1}(\text{f}\,1 - \mathit{offsetLinkFlag} + 1) = ((\text{RW}, \text{global}), \mathit{flag}, \mathit{flag}, \mathit{flag})$, the rest of ms_{f1} contains the code of $f\,1$.*
- *$ms_{flag} = [\mathit{flag} \mapsto 0]$*
- *$ms_{link} = [\mathit{link} \mapsto c_{malloc}, \mathit{link} + 1 \mapsto c_{adv}]$*
- *ms_{adv} contains a global read-only capability for ms_{link} on its first address. The remaining cells of the memory segment only contain instructions.*

if

$$(reg[\text{pc} \mapsto c_{f1}], m) \rightarrow_n (\mathit{halted}, m'),$$

then

$$m'(\mathit{flag}) = 0$$

◊

Lemma 2.A.17 (Correctness lemma for `f2`, detailed version of Lemma 2.8.2)**.** *let*

$$
\begin{aligned}
c_{adv} &\stackrel{def}{=} ((\textsc{e}, \text{global}), b_{adv}, e_{adv}, b_{adv} + \mathit{offsetLinkFlag}) \\
c_{f2} &\stackrel{def}{=} ((\textsc{rwx}, \text{global}), \mathtt{f2} - \mathit{offsetLinkFlag}, 2\mathtt{f}, \mathtt{f2}) \\
c_{malloc} &\stackrel{def}{=} ((\textsc{e}, \text{global}), b_{malloc}, e_{malloc}, b_{malloc} + \mathit{offsetLinkFlag}) \\
c_{stk} &\stackrel{def}{=} ((\textsc{rwlx}, \text{local}), b_{stk}, e_{stk}, b_{stk} - 1) \\
c_{link} &\stackrel{def}{=} ((\textsc{ro}, \text{global}), \mathit{link}, \mathit{link} + 1, \mathit{link}) \\
\mathit{reg} &\in \text{Reg} \\
m &\stackrel{def}{=} ms_{f2} \uplus ms_{flag} \uplus ms_{link} \uplus ms_{adv} \uplus ms_{malloc} \uplus ms_{stk} \uplus ms_{frame}
\end{aligned}
$$

and

- c_{malloc} *satisfies the specification for malloc and* $\iota_{malloc,0}$ *is the region from the specification.*

where

$$
\begin{aligned}
&\text{dom}(ms_{f2}) = [\mathtt{f2} - \mathit{offsetLinkFlag}, 2\mathtt{f}] \\
&\text{dom}(ms_{flag}) = [\mathit{flag}, \mathit{flag}] \\
&\text{dom}(ms_{link}) = [\mathit{link}, \mathit{link} + 1] \\
&\text{dom}(ms_{stk}) = [b_{stk}, e_{stk}] \\
&\text{dom}(ms_{adv}) = [b_{adv}, e_{adv}] \\
&ms_{malloc} :_n [0 \mapsto \iota_{malloc,0}] \qquad \textit{for all } n \in \mathbb{N}
\end{aligned}
$$

and

- $ms_{f2}(\mathtt{f2}{-}\mathit{offsetLinkFlag}) = ((\textsc{ro}, \text{global}), \mathit{link}, \mathit{link}{+}1, \mathit{link})$, $ms_{f2}(\mathtt{f2}{-}\mathit{offsetLinkFlag}{+}1) = ((\textsc{rw}, \text{global}), \mathit{flag}, \mathit{flag}, \mathit{flag})$, *the rest of* ms_{f2} *contains the code of* $f2$.
- $ms_{flag} = [\mathit{flag} \mapsto 0]$
- $ms_{link} = [\mathit{link} \mapsto c_{malloc}, \mathit{link} + 1 \mapsto c_{adv}]$
- $ms_{adv}(b_{adv}) = c_{link}$ *and* $\forall a \in [b_{adv} + 1, e].\, ms_{adv}(a) \in \mathbb{Z}$

if

$$(\mathit{reg}[\text{pc} \mapsto c_{f2}][r_{stk} \mapsto c_{stk}], m) \rightarrow_n (\mathit{halted}, m'),$$

then

$$m'(\mathit{flag}) = 0$$

◊

Lemma 2.A.18 (Correctness lemma for f3, detailed version of Lemma 2.8.3)**.**
For all $n \in \mathbb{N}$ *let*

$$
\begin{aligned}
c_{adv} &\stackrel{def}{=} ((\text{E}, \text{global}), b_{adv}, e_{adv}, b_{adv} + \mathit{offsetLinkFlag}) \\
c_{f3} &\stackrel{def}{=} ((\text{RWX}, \text{global}), \mathtt{f3} - \mathit{offsetLinkFlag}, \mathtt{3f}, \mathtt{f3}) \\
c_{stk} &\stackrel{def}{=} ((\text{RWLX}, \text{local}), b_{stk}, e_{stk}, b_{stk} - 1) \\
c_{malloc} &\stackrel{def}{=} ((\text{E}, \text{global}), b_{malloc}, e_{malloc}, b_{malloc} + \mathit{offsetLinkFlag}) \\
c_{link} &\stackrel{def}{=} ((\text{RO}, \text{global}), \mathit{link}, \mathit{link} + 1, \mathit{link}) \\
\mathit{reg} &\in \text{Reg} \\
m &\stackrel{def}{=} ms_{f3} \uplus ms_{flag} \uplus ms_{link} \uplus ms_{adv} \uplus ms_{malloc} \uplus ms_{stk} \uplus ms_{frame}
\end{aligned}
$$

and

- c_{malloc} *satisfies the specification for malloc.*

where

$$
\begin{aligned}
&\text{dom}(ms_{f3}) = [\mathtt{f3} - \mathit{offsetLinkFlag}, \mathtt{3f}] \\
&\text{dom}(ms_{flag}) = [\mathit{flag}, \mathit{flag}] \\
&\text{dom}(ms_{link}) = [\mathit{link}, \mathit{link} + 1] \\
&\text{dom}(ms_{stk}) = [b_{stk}, e_{stk}] \\
&\text{dom}(ms_{adv}) = [b_{adv}, e_{adv}] \\
&ms_{malloc} :_n [0 \mapsto \iota_{malloc,0}]
\end{aligned}
$$

and

- $ms_{f3}(\mathtt{f3}{-}\mathit{offsetLinkFlag}) = ((\text{RO}, \text{global}), \mathit{link}, \mathit{link}{+}1, \mathit{link})$, $ms_{f3}(\mathtt{f3}{-}\mathit{offsetLinkFlag}{+}1) = ((\text{RW}, \text{global}), \mathit{flag}, \mathit{flag}, \mathit{flag})$, *the rest of* ms_{f3} *contains the code of* $f3$.
- $ms_{flag} = [\mathit{flag} \mapsto 0]$
- $ms_{link} = [\mathit{link} \mapsto c_{malloc}, \mathit{link} + 1 \mapsto c_{adv}]$
- $ms_{adv}(b_{adv}) = c_{link}$ *and all other addresses of* ms_{adv} *contain instructions.*

if

$$(\mathit{reg}[\text{pc} \mapsto c_{f3}][r_{stk} \mapsto c_{stk}], m) \rightarrow_n (\mathit{halted}, m'),$$

then

$$m'(\mathit{flag}) = 0$$

◊

Lemma 2.A.19 (Correctness of $g1$, detailed version of Lemma 2.8.4)**.** *For all* $n \in \mathbb{N}$ *let*

$$
\begin{aligned}
c_{adv} &\stackrel{def}{=} ((\textsc{rwx}, \text{global}), b_{adv}, e_{adv}, b_{adv} + \mathit{offsetLinkFlag}) \\
c_{g1} &\stackrel{def}{=} ((\textsc{e}, \text{global}), \mathtt{g1} - \mathit{offsetLinkFlag}, \mathtt{4f}, \mathtt{g1}) \\
c_{stk} &\stackrel{def}{=} ((\textsc{rwlx}, \text{local}), b_{stk}, e_{stk}, b_{stk} - 1) \\
c_{malloc} &\stackrel{def}{=} ((\textsc{e}, \text{global}), b_{malloc}, e_{malloc}, b_{malloc} + \mathit{offsetLinkFlag}) \\
c_{link} &\stackrel{def}{=} ((\textsc{ro}, \text{global}), \mathit{link}, \mathit{link}, \mathit{link}) \\
m &\stackrel{def}{=} ms_{g1} \uplus ms_{flag} \uplus ms_{link} \uplus ms_{adv} \uplus ms_{malloc} \uplus ms_{stk} \uplus ms_{frame}
\end{aligned}
$$

where

- c_{malloc} *satisfies the specification for malloc with* $\iota_{malloc,0}$

$$
\begin{aligned}
&\mathrm{dom}(ms_{g1}) = [\mathtt{g1} - \mathit{offsetLinkFlag}, \mathtt{4f}] \\
&\mathrm{dom}(ms_{flag}) = [\mathit{flag}, \mathit{flag}] \\
&\mathrm{dom}(ms_{link}) = [\mathit{link}, \mathit{link}] \\
&\mathrm{dom}(ms_{stk}) = [b_{stk}, e_{stk}] \\
&\mathrm{dom}(ms_{adv}) = [b_{adv}, e_{adv}] \\
&ms_{malloc} :_n [0 \mapsto \iota_{malloc,0}]
\end{aligned}
$$

and

- $ms_{g1}(\mathtt{g1} - \mathit{offsetLinkFlag}) = ((\textsc{ro}, \text{global}), \mathit{link}, \mathit{link}, \mathit{link})$, $ms_{g1}(\mathtt{g1} - \mathit{offsetLinkFlag} + 1) = ((\textsc{rw}, \text{global}), \mathit{flag}, \mathit{flag}, \mathit{flag})$, *the rest of* ms_{g1} *contains the code of* $g1$ *immediately followed by the code of* $f4$*.*
- $ms_{flag} = [\mathit{flag} \mapsto 0]$
- $ms_{link} = [\mathit{link} \mapsto c_{malloc}]$
- $ms_{adv}(b_{adv}) = c_{link}$ *and all other addresses of* ms_{adv} *contain instructions.*
- $\forall a \in \mathrm{dom}(ms_{stk}).\, ms_{stk}(a) = 0$

if

$$(reg_0[\mathrm{pc} \mapsto c_{adv}][r_{stk} \mapsto c_{stk}][r_1 \mapsto c_{g1}], m) \rightarrow_n (\mathit{halted}, m'),$$

then

$$m'(\mathit{flag}) = 0$$

◊

Figure 2.18: Illustration of transition system in ι_x. The dashed line is the private transition.

2.A.7 Awkward example

The region for variable x The region ι_x, is the region omitted from the proof sketch for the awkward example. Figure 2.18 illustrates the transition system of ι_x.

Definition 2.A.20.

$$\begin{aligned}
\iota_x &= (perm, 0, \phi_{pub}, \phi, H_x) \\
\phi_{pub} &= \{(0,1)\}^* \\
\phi &= (1,0) \cup \phi_{pub} \\
H_x\ s\ \hat{W} &= \{(n, ms) \mid ms(x) = s \wedge n > 0\} \cup \{(0, ms)\}
\end{aligned}$$

♦

Static region This static region only requires that the memory segment is the given one. As it does not require safety, capabilities for this region cannot be gives to adversarial code.

$$\begin{aligned}
\iota^{sta}(v, ms) &= (v, 1, =, =, H^{sta}\ ms) \\
H^{sta}\ ms\ s\ \hat{W} &= \{(n, ms) \mid n > 0\} \cup \{(0, ms') \mid ms' \in \text{Mem}\}
\end{aligned}$$

Static safe region Static region that also requires safety. It is safe to give adversarial code read capabilities for this region.

$$\begin{aligned}
\iota^{sta,u}(v, ms) &= (v, 1, =, =, H^{sta,u}\ ms) \\
H^{sta,u}\ ms\ s\ \hat{W} &= \left\{ (n, ms') \;\middle|\; \begin{array}{l} ms' = ms \wedge \\ \forall a \in \text{dom}(ms). \\ \quad ms(a) \text{ is non-local} \wedge \\ \quad (n-1, ms(a)) \in \mathcal{V}(\xi(\hat{W})) \end{array} \right\} \cup \{(0, ms') \mid ms' \in \text{Mem}\}
\end{aligned}$$

Chapter 3

StkTokens: Enforcing Well-Bracketed Control Flow and Stack Encapsulation Using Linear Capabilities

This is an extended version of the published conference paper

> Lau Skorstengaard, Dominique Devriese, and Lars Birkedal.
>
> StkTokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities.
>
> *Proceedings of the ACM on Programming Languages*, 3, 2019

The new contributions of this extended version are outlined in the introduction.

Abstract

We propose and study StkTokens: a new calling convention that provably enforces well-bracketed control flow and local state encapsulation on a capability machine. The calling convention is based on linear capabilities: a type of capabilities that are prevented from being duplicated by the hardware. In addition to designing and formalizing this new calling convention, we also contribute a new way to formalize and prove that it effectively enforces well-bracketed control flow and local state encapsulation using what we call a fully abstract overlay semantics.

3.1 Introduction

Secure compilation is an active topic of research (e.g. [11, 34, 50, 70, 73, 75]), but a real secure compiler is yet to be built. Secure compilers preserve

source-language (security-relevant) properties even when the compiled code interacts with arbitrary target-language components. Generally, properties that hold in the source language but not in the target language need to be somehow enforced by the compiler. Two properties that hold in many high-level source languages are well-bracketed control flow and encapsulation of local state, but they are not enforced after compilation to assembly.

Well-bracketed control flow (WBCF) expresses that invoked functions must either return to their callers, invoke other functions themselves or diverge, and generally holds in programming languages that do not offer a primitive form of continuations. At the assembly level, this property does not hold immediately. Invoked functions get direct access to return pointers that they are supposed to jump to a single time at the end of their execution. There is, however, no guarantee that untrusted assembly code respects this intended usage. In particular, a function may invoke return pointers from other stack frames than its own: either frames higher in the call stack or ones that no longer exist as they have already returned.

Local state encapsulation (LSE) is the guarantee that when a function invokes another function, its local variables (saved on its stack frame) will not be read or modified until the invoked function returns. At the assembly level, this property also does not hold immediately. The calling function's local variables are stored on the stack during the invocation, and functions are not supposed to touch stack frames other than their own. However, untrusted assembly code is free to ignore this requirement and read or overwrite the local state of other stack frames.

To enforce these properties, target language security primitives are needed that can be used to prevent untrusted code from misbehaving without imposing too much overhead on well-behaved code. The virtual-memory based security primitives on commodity processors do not seem sufficiently fine-grained to efficiently support this. More suitable security primitives are offered by a type of CPUs known as capability machines [60, 69]. These processors use tagged memory to enforce a strict separation between integers and *capabilities*: pointers that carry authority. Capabilities come in different flavours. Memory capabilities allow reading from and writing to a block of memory. Additionally, capability machines offer some form of *object capabilities* that represent low-level encapsulated closures, i.e. a piece of code coupled with private state that it gains access to upon invocation. The concrete mechanics of object capabilities vary between different capability machines. For example, on a recent capability machine called CHERI they take the form of pairs of capabilities that represent the code and data parts of the closure. Each of the two capabilities are sealed with a common seal which make them opaque. The hardware transparently unseals the pair upon invocation [95, 97].

To enforce WBCF and LSE on a capability machine, there are essentially two approaches. The first is to use separate stacks for mutually distrust-

ing components, and a central, trusted stack manager that mediates cross-component invocations. This idea has been applied in CheriBSD (an operating system built on CHERI) [97], but it is not without downsides. First, it requires reserving separate stack space for all components, which scales poorly to large amounts of components. Also, in the presence of higher-order values (e.g., function pointers, objects), the stack manager needs to be able to decide which component a higher-order value belongs to in order to provide it the right stack pointer upon invocation. It is not clear how it can do this efficiently in the presence of large amounts of components. Finally, this approach does not allow passing stack references between components.

A more scalable approach retains a single stack shared between components. Enforcing WBCF and LSE in this approach requires a way to temporarily provide stack and return capabilities to an untrusted component and to revoke them after it returns. While capability revocation is expensive in general, some capability machines offer restricted forms of revocation that can be implemented efficiently. For example, CHERI offers a form of *local* capabilities that can only be stored in registers or on the stack but not in other parts of memory. Skorstengaard et al. [83] have demonstrated that by making the stack and return pointer local, and by introducing a number of security checks and measures, the two properties can be guaranteed. In fact, a similar system was envisioned in earlier work on CHERI [99]. However, a problem with this approach is that revoking the local stack and return capabilities on every security boundary crossing requires clearingt the entire unused part of the stack, an operation that may be prohibitively expensive.

In this work, we propose and study StkTokens: an alternative calling convention that enforces WBCF and LSE with a single shared stack. Instead of CHERI's local capabilities, it builds on *linear* capabilities; a new form of capabilities that has not previously been described in the published literature, although related ideas have been described by Szabo [88, 89, "scarce objects"] and in technical documents. Concurrently with our work, Watson et al. have developed a (more realistic) design for linear capabilities in CHERI that is detailed in the latest CHERI ISA reference [98]. The hardware prevents these capabilities from being duplicated. We propose to make stack and return pointers linear and require components to hand them out in cross-component invocations and to give them back on return. The non-duplicability of linear capabilities together with some security checks allow us to guarantee WBCF and LSE without large overhead on boundary crossings and in particular without the need for clearing large blocks of memory.

A second contribution of this work is the way in which we formulate these two properties. Although the terms "well-bracketed control flow" and "local state encapsulation" sound precise, it is actually far from clear what they mean, and how best to formalize them. Existing formulations are either partial and not suitable for reasoning [10] or lack evidence of generality [83]. We propose a new formulation using a technique we call *fully abstract overlay*

semantics. It starts from the premise that security results for a calling convention should be reusable as part of a larger proof of a secure compiler. To this end, we define a second operational semantics for our target language with a native well-bracketed call stack and primitive ways to do calls and returns. This well-behaved semantics guarantees WBCF and LSE natively for components using our calling convention. As such, these components can be sure that they will only ever interact with other well-behaved components that respect our desired properties. To express security of our calling convention, we then show that considering the same components in the original semantics does not give adversaries additional ways to interact with them. More formally, we show that mapping a component in the well-behaved semantics to the same component in the original semantics is fully abstract [9], i.e. components are indistinguishable to arbitrary adversaries in the well-behaved language iff they are indistinguishable to arbitrary adversaries in the original language.

Compared to Skorstengaard et al. [83] that prove LSE and WBCF for a handful of examples, this approach expresses what it means to enforce the desirable properties in a general way and makes it clear that we can support a very general class of programs. Additionally, formulating security of a calling convention in this way makes it potentially reusable in a larger security proof of a full compiler. The idea is that such a compiler could be proven fully abstract with respect to the well-behaved semantics of the target language, so that the proof could rely on native well-bracketedness and local stack frame encapsulation. Such an independent result could then be composed with ours to obtain security of the compiler targeting the real target language, by transitivity of full abstraction.

In this paper, we make the following contributions:

- We present LCM: A formalization of a simple CHERI-like capability machine with linear capabilities (Section 3.2).
- We present a new calling convention StkTokens that provably guarantees LSE and WBCF on LCM (Section 3.3).
- We present a new way to formalize these guarantees based on a novel technique called *fully-abstract overlay semantics* and we prove LSE and WBCF claims. This includes:
  - oLCM: an overlay semantics for LCM with built-in LSE and WBCF (Section 3.4)
  - proving full-abstraction for the embedding of oLCM into LCM (Section 3.5) by
  - using and defining a cross-language, step-indexed, Kripke logical relation with recursive worlds (Section 3.5).

This text is an extended version of a paper that was presented at POPL 2019 [86]. Compared to the earlier text, this version adds and explains many of the details that were left out in the conference version due to space restrictions. The added details include a better motivation of sealing (Section 3.2.1), well-formedness components (Section 3.4.2), and reasonable components (Section 3.4.3). The section on proving full-abstraction (Section 3.5) has been rewritten with details and explanations about the formal method used for the full-abstraction proof. This includes a detailed description and motivation of the world (Section 3.5.1) and logical relation (Section 3.5.2). This paper is accompanied by a technical report [85] with the elided details and proof.

3.2 A Capability Machine with Sealing and Linear Capabilities

In this section, we introduce a simple but representative capability machine with linear capabilities, that we call LCM (Linear Capability Machine). LCM is mainly inspired by CHERI [69] with linear capabilities as the main addition. For simplicity, LCM assumes an infinite address space and unbounded integers.

The concept of a capability is the cornerstone of any capability machine. In its simplest form, a capability is a permission and a range of authority. The permission dictates the operations the capability can be used for, and the range of authority specifies the range of memory it can act upon. The capabilities on LCM are of the form $((perm, lin), base, end, addr)$ (defined in Figure 3.2 with the rest of the syntax of LCM). Here $perm$ is the permission, and $[base, end]$ is the range of authority. The available permissions are read-write-execute (RWX), read-write (RW), read-execute (RX), read-only (R), and null-permission (0) ordered by $\leq$ as illustrated in Figure 3.1. In addition to the permission and range, capabilities also have a current address $addr$ and a linearity lin. The linearity is either normal for traditional capabilities or linear for linear ones. A linear capability is a capability that cannot be duplicated. This is enforced dynamically on the capability machine, so when a linear capability is moved between registers or memory, the source is cleared. The non-duplicability of linear capabilities means that a linear capability cannot become aliased if it wasn't to begin with.

Any reasonable capability machine needs a way to set up boundaries between security domains with different authorities. It also must have a way to cross these boundaries such that (1) the security domain we move from can encapsulate and later regain its authority and (2) the security domain we move to regains all of its authority. On LCM

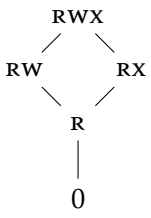


Figure 3.1: Permission hierarchy

we have CHERI-like sealed capabilities to achieve this [69, 95]. A sealed capability sealed(σ,sc) is a pair of a seal σ and a capability sc. A sealed capability makes an underlying capability opaque which means that the underlying capability cannot be changed or used for the operations it normally gives permission to. In other words, the authority the underlying capability represents is encapsulated under the seal. In order to seal a sealable with a seal σ, it is necessary to have the authority to do so. The permission to make sealed capabilities is represented by a sets of seals seal($\sigma_{base}, \sigma_{end}, \sigma_{current}$). A set of seals is a capability that represents the authority to seal sealables with seals in the range $[\sigma_{base}, \sigma_{end}]$. In spirit of memory capabilities, a set of seals has a current seal $\sigma_{current}$ that is selected for use in the next seal operation. As we will see later, sealed capabilities can be unsealed with an `xjmp`, an operation that operates on a pair of capabilities sealed with the same seal. The instruction will be explained in more detail below, but essentially, it unseals the pair of capabilities, transfers control to one of them (the code part of the pair) and makes the other one (the data part of the pair) available to the invoked code. The combination of sealed capabilities and `xjmp` gives (1) and (2).

Words on LCM are capabilities and data (represented by integers $\mathbb{Z}$). We assume a finite set of register names RegName containing at least the registers pc, r_{rdata}, r_{rcode}, r_{stk}, r_{data}, r_{t1}, and r_{t2}. We define register files as functions from register names to words. Complete memories map all addresses to words and memory segments map some addresses to words (i.e. partial functions). LCM has two terminated configurations halted and failed that respectively signify a successful execution and an execution where something went wrong, e.g., an out-of-bounds memory access. An executable configuration is a register file and memory pair.

LCM's instruction set is somewhat basic with the instructions one expects on most low-level machines as well as capability-related instructions. The standard instructions are: unconditional and conditional jump (`jmp` and `jnz`), copy between registers (`move`), instructions that load from memory and store to memory (`load` and `store`), and arithmetic operations (`plus`, `minus`, and `lt`). The simplest of the capability instructions inspect the properties of capabilities: type (`gettype`), linearity (`getl`), range (`getb` and `gete`), current address or seal (`geta`) or permission (`getp`). The current address (or seal) of a capability (or set of seals) can be shifted by an offset (`cca`) or set to the base address (`seta2b`). The `restrict` instruction reduces the permission of a capability according to the permission order $\leq$. Generally speaking, a capability machine needs an instruction for reducing the range of authority of a capability. Because LCM has linear capabilities, the instruction for this splits the capability in two rather than reducing the range of authority (`split`). The reverse is possible using `splice`. Sealables can be sealed using `cseal` and pairs of sealed capabilities can be unsealed by crossing security bound-

$$
\begin{array}{rrcl}
\mathrm{a}, \mathrm{base} \in & \mathrm{Addr} & \stackrel{def}{=} & \mathbb{N} \\
\mathrm{end} \in & \mathrm{Addr} \cup \{\infty\} & & \\
\sigma_{\mathrm{base}}, \sigma \in & \mathrm{Seal} & \stackrel{def}{=} & \mathbb{N} \\
\sigma_{\mathrm{end}} \in & \mathrm{Seal} \cup \{\infty\} & & \\
\mathrm{perm} \in & \mathrm{Perm} & ::= & \textsc{rwx} \mid \textsc{rx} \mid \textsc{rw} \mid \textsc{r} \mid 0 \\
& \mathrm{l} & ::= & \mathrm{linear} \mid \mathrm{normal} \\
sc \in & \mathrm{Sealables} & ::= & ((\mathrm{perm}, \mathrm{l}), \mathrm{base}, \mathrm{end}, \mathrm{a}) \mid \mathrm{seal}(\sigma_{\mathrm{base}}, \sigma_{\mathrm{end}}, \sigma) \\
c \in & \mathrm{Cap} & ::= & \mathrm{Sealables} \mid \mathrm{sealed}(\sigma, \mathrm{sc}) \\
w \in & \mathrm{Word} & \stackrel{def}{=} & \mathbb{Z} \uplus \mathrm{Cap} \\
r \in & \mathrm{RegName} & ::= & \mathrm{pc} \mid \mathrm{r}_{\mathrm{rdata}} \mid \mathrm{r}_{\mathrm{rcode}} \mid \mathrm{r}_{\mathrm{stk}} \mid \mathrm{r}_{\mathrm{data}} \mid \mathrm{r}_{\mathrm{t1}} \mid \mathrm{r}_{\mathrm{t2}} \mid \ldots \\
reg \in & \mathrm{RegFile} & \stackrel{def}{=} & \mathrm{RegName} \to \mathrm{Word} \\
mem \in & \mathrm{Memory} & \stackrel{def}{=} & \mathrm{Addr} \to \mathrm{Word} \\
ms \in & \mathrm{MemSeg} & \stackrel{def}{=} & \mathrm{Addr} \rightharpoonup \mathrm{Word} \\
\Phi \in & \mathrm{ExecConf} & \stackrel{def}{=} & \mathrm{Memory} \times \mathrm{RegFile} \\
& \mathrm{Conf} & \stackrel{def}{=} & \mathrm{ExecConf} \cup \{\mathrm{failed}\} \cup \{\mathrm{halted}\}
\end{array}
$$

$$
\begin{array}{rcl}
r & \in & \mathrm{RegisterName} \\
rn & ::= & r \mid \mathbb{N} \\
\mathrm{Instr} & ::= & \texttt{jmp}\ r \mid \texttt{jnz}\ r\ rn \mid \texttt{move}\ r\ rn \mid \texttt{load}\ r\ r \mid \texttt{store}\ r\ r \mid \texttt{plus}\ r\ rn\ rn \mid \texttt{minus}\ r\ rn\ rn \mid \\
& & \texttt{lt}\ r\ rn\ rn \mid \texttt{gettype}\ r\ r \mid \texttt{getp}\ r\ r \mid \texttt{getl}\ r\ r \mid \texttt{getb}\ r\ r \mid \texttt{gete}\ r\ r \mid \texttt{geta}\ r\ r \mid \\
& & \texttt{cca}\ r\ nrn \mid \texttt{seta2b}\ r \mid \texttt{restrict}\ r\ rn \mid \texttt{cseal}\ r\ r \mid \texttt{xjmp}\ r\ r \mid \texttt{split}\ r\ r\ r\ rn \mid \\
& & \texttt{splice}\ r\ r\ r \mid \texttt{fail} \mid \texttt{halt}
\end{array}
$$

Figure 3.2: The syntax of our capability machine with seals and linear capabilities.

$$
\frac{\Phi(\mathrm{pc}) = ((p, \_), b, e, a) \qquad b \le a \le e \qquad p \in \{\textsc{rwx}, \textsc{rx}\}}{\Phi \to [\![decode(\Phi.mem(a))]\!](\Phi)} \qquad \frac{\forall \Phi' \neq \mathrm{failed}.\ \Phi \not\twoheadrightarrow \Phi'}{\Phi \to \mathrm{failed}}
$$

$$
updPc(\Phi) = \begin{cases} \Phi[reg.\mathrm{pc} \mapsto w] & \Phi(\mathrm{pc}) = ((p,l),b,e,a) \wedge w = ((p,l),b,e,a+1) \\ \Phi & \text{otherwise} \end{cases}
$$

$$
linClear(w) = \begin{cases} 0 & isLinear(w) \\ w & \text{otherwise} \end{cases}
$$

$$
xjmpRes(c_1, c_2, \Phi) = \begin{cases} \Phi[reg.\mathrm{pc} \mapsto c_1][reg.\mathrm{r}_{\mathrm{data}} \mapsto c_2] & nonExec(c_2) \\ \mathrm{failed} & \text{otherwise} \end{cases}
$$

Figure 3.3: Functions used in operational semantics of LCM.

$i \in$ Instr	$[\![i]\!](\Phi)$	Conditions
`halt`	halted	
`fail`	failed	
`move` r rn	$updPc(\Phi[reg.rn \mapsto w_2]$ $[reg.r \mapsto w_1])$	$rn \in$ RegName and $w_1 = \Phi(rn)$ and $w_2 = linClear(\Phi(rn))$
`load` r_1 r_2	$updPc(\Phi[reg.r_1 \mapsto w_1]$ $[mem.a \mapsto w_a])$	$\Phi(r_2) = ((p, \_), b, e, a)$ and $b \leq a \leq e$ and $p \in \{\text{RWX}, \text{RW}, \text{RX}, \text{R}\}$ and $w_1 = \Phi.mem(a)$ and $isLinear(w_1) \Rightarrow p \in \{\text{RWX}, \text{RW}\}$ and $w_a = linClear(w_1)$
`store` r_1 r_2	$updPc(\Phi[reg.r_2 \mapsto w_2]$ $[mem.a \mapsto \Phi(r_2)])$	$\Phi(r_1) = ((p, \_), b, e, a)$ and $p \in \{\text{RWX}, \text{RW}\}$ and $b \leq a \leq e$ and $w_2 = linClear(\Phi(r_2))$
`geta` r_1 r_2	$updPc(\Phi[reg.r_1 \mapsto w])$	If $\Phi(r_2) = ((\_, \_), \_, \_, a)$ or $\Phi(r_2) = \text{seal}(\_, \_, a)$, then $w = a$ and otherwise $w = -1$
`cca` r rn	$updPc(\Phi[reg.r \mapsto w])$	$\Phi(rn) = n \in \mathbb{Z}$ and either $\Phi(r) = ((p, l), b, e, a)$ or $\Phi(r) = (\sigma_b, \sigma_e, \sigma)$ and $w = ((p, l), b, e, a + n)$ or $w = (\sigma_b, \sigma_e, \sigma + n)$, respectively
`jmp` r	$\Phi[reg.r \mapsto w]$ $[reg.\text{pc} \mapsto \Phi(r)]$	$w = linClear(\Phi(r))$
`xjmp` r_1 r_2	Φ'	$\Phi(r_1) = \text{sealed}(\sigma, c_1)$ and $\Phi(r_2) = \text{sealed}(\sigma, c_2)$ and $w_1 = linClear(c_1)$ and $w_2 = linClear(c_2)$ and $\Phi' = xjmpRes(c_1, c_2, \Phi[reg.r_1, r_2 \mapsto w_1, w_2])$
`split` r_1 r_2 r_3 rn	$updPc(\Phi[reg.r_3 \mapsto w]$ $[reg.r_1 \mapsto c_1]$ $[reg.r_2 \mapsto c_2])$	$\Phi(r_3) = ((p, l), b, e, a)$ and $\Phi(rn) = n \in \mathbb{N}$ and $b \leq n < e$ and $c_1 = ((p, l), b, n, a)$ and $c_2 = ((p, l), n + 1, e, a)$ and $w = linClear(\Phi(r_3))$
`splice` r_1 r_2 r_3	$updPc(\Phi[reg.r_2 \mapsto w_2]$ $[reg.r_3 \mapsto w_3]$ $[reg.r_1 \mapsto c])$	$\Phi(r_2) = ((p, l), b, n, \_)$ and $\Phi(r_3) = ((p, l), n + 1, e, a)$ and $b \leq n < e$ and $c = ((p, l), b, e, a)$ and $w_2, w_3 = linClear(\Phi(r_2), \Phi(r_3))$
`cseal` r_1 r_2	$updPc(\Phi[reg.r_1 \mapsto sc])$	$\Phi(r_1) \in$ Sealables and $\Phi(r_2) = \text{seal}(\sigma_b, \sigma_e, \sigma)$ and $\sigma_b \leq \sigma \leq \sigma_e$ and $sc = \text{sealed}(\sigma, \Phi(r_1))$
...		
_	failed	otherwise

Figure 3.4: An excerpt of the operational semantics of LCM

aries (`xjmp`, see below). Finally, LCM has instructions to signal whether an execution was successful or not (`halt` and `fail`).

The operational semantics of LCM is displayed in Figure 3.3 and 3.4 with the instruction interpretations in Figure 3.4 and auxiliary functions in Figure 3.3. The operational semantics is defined in terms of a step relation that executes the next instruction in an executable configuration Φ which results in a new executable configuration or one of the two terminated configurations. The executed instruction is determined by the capability in the pc register, i.e. $\Phi(\text{pc})$ (we write $\Phi(r)$ to mean $\Phi.reg(r)$). In order for the machine to take a step, the capability in the pc must have a permission that allows execution, and the current address of the capability must be within the capability's range of authority. If both conditions are satisfied, then the word pointed to by the capability is decoded to an instruction which is interpreted relative to Φ. The interpretations of some of the instructions are displayed in Figure 3.4. In order to step through a program in memory, most of the interpretations use the function *updPc* which simply updates the capability in the pc to point to the next memory address. The instructions that stop execution or change the flow of execution do not use *updPc*. For instance, the `halt` and `fail` instructions are simply interpreted as the halted and failed configurations, respectively, and they do not use *updPc*.

The `move` instruction simply moves a word from one register to another. It is, however, complicated slightly by the presence of the non-duplicable linear capabilities. When a linear capability is moved, the source register must be cleared to prevent duplication of the capability. This is done uniformly in the semantics using the function *linClear* that returns 0 for linear capabilities and is the identity for all other words. When a word w is transferred on the machine, then the source of w is overwritten with $linClear(w)$ which clears the source if w was linear and leaves it unchanged otherwise. In the case of `move`, the source register rn is overwritten with $linClear(\Phi(rn))$.

The `store` and `load` instructions are fairly standard. They require a capability with permission to either write or read depending on the operation, they check that the capability points within the range of authority. Linear capabilities introduce one extra complication for `load` as it needs to clear the loaded memory address when it contains a linear capability in order to not duplicate the capability. In this case, we require that the memory capability used for loading also has write-permission.

The instruction `geta` projects the current address (or seal) from a capability (or set of seals), and returns -1 for data and sealed capabilities. `cca` (change current address) changes the current address or seal of a capability or set of seals, respectively, by a given offset. Note that this instruction does not need to use *linClear* like the previous ones, because it modifies the capability in-place, i.e. the source register is also the target register. The `jmp` instruction is a simple jump that just sets register pc.

The operational side of the sealing in LCM consists of two instructions:

`cseal` for sealing a capability and `xjmp` for unsealing a pair of capabilities. Given a sealable *sc* and a set of seals where the current seal σ is within the range of available seals, the `cseal` instruction seals *sc* with σ. Apart from dealing with linearity, `xjmp` takes a pair of sealed capabilities, unseals them, and puts one in the pc register and the other in the r_{data} register, but only if they are sealed with the same seal and the data capability (the one placed in r_{data}) is non-executable. A pair of sealed capabilities can be seen as a closure where the code capability (the capability placed in pc) is the program and the data capability is the local environment. Because of the opacity of sealed capability, the creator of the closure can be sure that execution will start where the code capability points and only in an environment with the related data, i.e. sealed with the same seal. This makes `xjmp` the mechanism on LCM that transfers control between security domains. Opaque sealed capabilities encapsulate a security domain's local state and authority, and they only become accessible again when control is transferred to the security domain. Some care should be taken for sealing because reusing the same seal for multiple closures makes it possible to jump to the code of one closure with the environment of another. LCM does not have an instruction for unsealing capabilities directly, but it can be (partially) simulated using `xjmp`.

Instructions for reducing the authority of capabilities are commonplace on capability machines as they allow us to limit what a capability can do before it is passed away. For normal capabilities, reduction of authority can be done without actually giving up any authority by duplicating the capability first. With linear capabilities authority cannot be preserved in this fashion as they are non-duplicable. In order to make a lossless reduction of the range of authority, LCM provides special hardware support in the form of `split` and `splice`. The `split` instruction takes a capability with range of authority $[base, end]$ and an address n and creates two new capabilities, with $[base, n]$ and $[n+1, end]$ as ranges of authority. Everything else, i.e. permission, linearity and current address, is copied without change to the new capabilities. With `split`, we can reduce the range of authority of a linear capability without losing any authority as we retain it in the second capability. The `splice` instruction essentially does the inverse of `split`. Given two capabilities with adjacent ranges of authority and the same permissions and linearity, `splice` splices them together into one capability. The two instructions work in the same way for seal sets. We do not provide special support for lossless reduction of capability permissions, but this could probably be achieved with more fine-grained permissions. This would also allow linear capabilities to have aliases, but only by linear capabilities with disjoint permissions.

The interpretation of the remaining instructions are displayed in Appendix 3.A.1. The instructions `getb`, `gete`, `getl`, and `getp` all project information about capabilities. The `getb` and `gete` instructions, respectively, project the base and end address of the range of authority. The linearity of a permission is projected with `getl`, and finally the permission is pro-

jected with `getp`. The instructions `getb` and `gete` also work on sets of seals. The instruction `gettype` returns an integer representation of the type of a word which allows programs to be defensive in the sense that they can check whether a word is a capability before they use it. LCM also has arithmetic instructions `plus`, `minus`, and `lt`. The latter instruction compares two numbers and writes 1 or 0 to a target register depending on whether one number is less than the other. The instruction `seta2b` sets the current address (or seal) of a capability (or set of seals) to the base address of the range of authority (or range of seals). This instruction makes it easy to work relatively to the base address of a capability. This instruction is not strictly necessary as it can be emulated with other instructions. Finally, we have the `restrict` instruction which restricts the permission of a capability according to the $\leq$ relation.

3.2.1 The purpose of sealing

To motivate the necessity of an encapsulation mechanism like sealing, consider the following example. We, a trusted piece of code, want to transfer control to code that we distrust, and we want to give them the means to return to us. That is, we need to give them a return capability. If we did not have an encapsulation mechanism, our only option would be to give them an executable capability for the address we want them to return to. The untrusted code could use the return capability as intended, but it could also manipulate and make it point to a different address of our code. Jumping to such a capability would cause an the program to execute in a way we did not intend for it. Further in order to be able to retrieve our capabilities from before transferring control, we would have to store the capabilities somewhere accessible from the return capability. However, the untrusted code would have access to all this through the return capability because it gives the same authority to us as the untrusted code. This is why, any reasonable capability machine must have an encapsulation mechanism to allow programs to make boundaries between security domains.

In the example, we could seal the return capability to establish a boundary. Specifically, it would prevent the untrusted code from changing the target of the return capability forcing them to return to the point we specified. It would also prevent the untrusted code from reading our capabilities. All in all, this means that we can transfer control to untrusted code without giving up our capabilities or handing them over.

LCM does not have a direct unsealing instruction, but it is still possible to emulate a limited unsealing mechanism. Say you have a sealed non-executable word $\text{sealed}(\sigma, w)$ as well as a set of seals $\text{seal}(\sigma_b, \sigma_e, \sigma)$ that contains σ, i.e. $\sigma \in [\sigma_b, \sigma_e]$. Now assume part of the code would like to have access to w in r_{data}. Take a capability for this code, possibly by adjusting the pc, and seal it with σ. Now you have the data part and the code part of a sealed capability pair, so you can `xjmp` to it which unseals the data capability

and puts it in r_{data} for your code to use. Sealed executable capabilities cannot be unsealed in the same way because `xjmp` fails if the data capability is executable. However, given a sealed executable capability sealed(σ, c) and a set of seals that contains σ, we can still construct the data part of the sealed capability pair. This means that we can execute the code the capability points to together with data that it was not intended to be used with.

Sealing is meant for encapsulation, but it relies on seals being kept private as illustrated by the unsealing emulation. For this reason, it is important that the system is initialised such that each component has access to unique seals. We return to this in Section 3.4.2.

3.2.2 Decoding and encoding functions

The operational semantics of the capability machine uses the function *decode* to decode instructions. We also assume a function *encode* to make it easy to specify programs in terms of instructions. Rather than specifying a decode function and an encode function, we assume that they are given with certain properties. The *decode* : Word $\to$ Instr should be surjective and injective for all non-`fail` instructions. Further, it should decode all capabilities as the `fail` instruction, i.e.

$$\forall c \in \text{Cap}.\, decode(c) = \texttt{fail}$$

The *encode* : Instr $\to \mathbb{Z}$ function should be injective, and it should be defined so *decode* is its left inverse, i.e.

$$\forall i \in \text{Instr}.\, decode(encode(i)) = i$$

These assumptions are sufficient to construct program examples and run them on the machine. The *encode* function allows us to specify the program "abstractly" in terms of instructions rather than machine words. As the *decode* function is the left inverse of *encode*, we can even execute the program in the operational semantics without ever worrying about what the actual encoding is.

The machine also assumes decode and encode functions for permissions *decodePerm* : Perm $\to \mathbb{Z}$ and *encodePerm* : $\mathbb{Z} \to$ Perm. We assume the *decodePerm* function to be the left inverse of *encodePerm* and surjective. For *encodePerm*, we assume it is surjective and that it does not encode anything to the `getp` error value -1, i.e.

$$\forall p \in \text{Perm}.\, encodePerm(p) \neq -1$$

For linearity we assume similar functions.

Finally in the interpretation of `gettype`, the machine uses an encode function for word types *encodeType*. This function encodes each kind of word as an integer. This is very much like the previous functions. It encodes each kind of word differently and all words of the same kind to the same integer.

3.2.3 Components, linking, programs, and contexts

The executable configuration describes the machine state, but it does not make it clear what components run on the machine and how they interact with each other. To clarify this, we introduce notions of components and programs from which we construct executable configurations. A component (defined in Figure 3.5) is basically a program with entry points in the form of imports that need to be linked. It has exports that can satisfy the imports of other components. A base component $comp_0$ consists of a code memory segment, a data memory segment, a list of imported symbols, a list of exported symbols, two lists specifying the available seals[1], and a set of all the linear addresses (addresses governed by a linear capability). The import list specifies where in memory imports should be placed, and imports are matched to exports via their symbols. The exports are words each associated with a symbol. A component is either a library component (without a main entry point) or an incomplete program with a main in the form of a pair of sealed capabilities. The latter can be seen as a program that still needs to be linked with libraries. Components are combined into new components by linking them together, as long as only one is an incomplete program with a main. Two components can be linked when their memories, seals, and linear addresses are disjoint. They are combined by taking the union of each of their constituents. For every import that is satisfied by an export of the other component, the data memory is updated to have the exported word on the imported address. The satisfied imports are removed from the import list in the resulting linked component and the exports are updated to be the exports of the two components.

We can now define the notion of a program as well as a context.

Definition 3.2.1 (Programs and Contexts)**.** A *program* is a component

$$(comp_0, c_{\text{main},c}, c_{\text{main},d})$$

with an empty import list. A *context* for a component *comp* is a component *comp'* such that $comp \bowtie comp'$ is a program. ♦

How a program is initialised to create an executable configuration, is discussed in Section 3.4. Some simplifications have been made in this presentation of LCM. See Skorstengaard et al. [85] for details.

3.3 Linear Stack and Return Capabilities

In this section, we introduce our calling convention StkTokens that ensures LSE and WBCF. We will gradually explain each of the security measures StkTokens takes and motivate them with the attacks they prevent.

[1] We will return to the seals in Section 3.4.

$$
\begin{array}{rl}
s & \in \text{Symbol} \qquad \text{import} ::= a \hookleftarrow s \qquad \text{export} ::= s \mapsto w \\
comp_0 & ::= (ms_{\text{code}}, ms_{\text{data}}, \overline{\text{import}}, \overline{\text{export}}, \overline{\sigma_{\text{ret}}}, \overline{\sigma_{\text{clos}}}, A_{\text{linear}}) \\
comp & ::= comp_0 \mid (comp_0, c_{\text{main},c}, c_{\text{main},d})
\end{array}
$$

$$
\frac{\begin{array}{c}
comp_0 = (ms_{\text{code},1}, ms_{\text{data},1}, \overline{import_1}, \overline{export_1}, \overline{\sigma_{\text{ret},1}}, \overline{\sigma_{\text{clos},1}}, A_{\text{linear},1}) \\
comp_0' = (ms_{\text{code},2}, ms_{\text{data},2}, \overline{import_2}, \overline{export_2}, \overline{\sigma_{\text{ret},2}}, \overline{\sigma_{\text{clos},2}}, A_{\text{linear},2}) \\
comp_0'' = (ms_{\text{code},3}, ms_{\text{data},3}, \overline{import_3}, \overline{export_3}, \overline{\sigma_{\text{ret},3}}, \overline{\sigma_{\text{clos},3}}, A_{\text{linear},3}) \\
ms_{\text{code},3} = ms_{\text{code},1} \uplus ms_{\text{code},2} \\
ms_{\text{data},3} = (ms_{\text{data},1} \uplus ms_{\text{data},2}) \left[a \mapsto w \;\middle|\; \begin{array}{l} (a \hookleftarrow s) \in (\overline{import_1} \cup \overline{import_2}), \\ (s \mapsto w) \in \overline{export_3} \end{array} \right] \\
\overline{export_3} = \overline{export_1} \cup \overline{export_2} \\
\overline{import_3} = \{a \hookleftarrow s \in (\overline{import_1} \cup \overline{import_2}) \mid s \mapsto \_ \notin \overline{export_3}\} \\
\overline{\sigma_{\text{ret},3}} = \overline{\sigma_{\text{ret},1}} \uplus \overline{\sigma_{\text{ret},2}} \\
\overline{\sigma_{\text{clos},3}} = \overline{\sigma_{\text{clos},1}} \uplus \overline{\sigma_{\text{clos},2}} \qquad A_{\text{linear},3} = A_{\text{linear},1} \uplus A_{\text{linear},2} \\
\text{dom}(ms_{\text{code},3}) \# \text{dom}(ms_{\text{data},3}) \qquad \overline{\sigma_{\text{ret},3}} \# \overline{\sigma_{\text{clos},3}}
\end{array}}{comp_0'' = comp_0 \bowtie comp_0'}
$$

$$
\frac{comp_0'' = comp_0 \bowtie comp_0'}{(comp_0'', c_{\text{main},c}, c_{\text{main},d}) = comp_0 \bowtie (comp_0', c_{\text{main},c}, c_{\text{main},d})}
$$

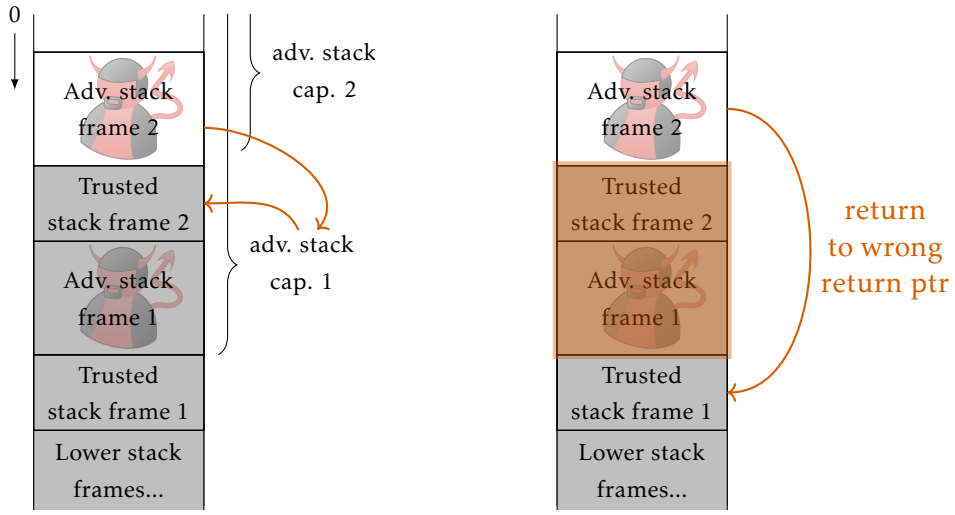
$$
\frac{comp_0'' = comp_0 \bowtie comp_0'}{(comp_0'', c_{\text{main},c}, c_{\text{main},d}) = (comp_0, c_{\text{main},c}, c_{\text{main},d}) \bowtie comp_0'}
$$

Figure 3.5: Components and linking of components.

StkTokens is based on a traditional single stack, shared between all components. To explain the technique, let us first consider how we might already add extra protection to stack and return pointers on a capability machine. First, we replace stack pointers with stack capabilities. When a new stack frame is created, the caller provisions it with a stack capability, restricted to the appropriate range, i.e. it does not cover the caller's stack frame. Return pointers, on the other hand, are replaced by a pair of sealed return capabilities, as we already explained in Section 3.2.1. They form an opaque closure that the callee can only jump to, and the caller's data becomes available to the caller's return code.

While the above adds extra protection, it is not sufficient to enforce WBCF and LSE. Untrusted callees receive a stack capability and a return pair that they are supposed to use for the call. However, a malicious callee (which we will refer to as an adversary[2]) can store the provided capabilities on the heap

[2]See Section 3.4.2 for more details on our attacker model.



(a) An adversary uses a previous stack frame's stack pointer.

(b) An adversary jumps to a previous stack frame's stack pointer.

Figure 3.6: Possible ways to abuse stack and return capabilities.

in order to use them later. Figure 3.6 illustrates two examples of this. In both examples our component and an adversarial component have been taking turns calling each other, so the stack now contains four stack frames alternating between ours and theirs. The figure on the left (Figure 3.6a) illustrates how we try to ensure LSE by restricting the stack capability to the unused part before every call to the adversary. However, restricting the stack capability does not help when we, in the first call, give access to the part of the stack where our second stack frame will reside as nothing prevents the adversary from duplicating and storing the stack pointer. Generally speaking, we have no reason to ever trust a stack capability received from an untrusted component as that stack capability may have been duplicated and stored for later use. In the figure on the right (Figure 3.6b), we have given the adversary two pairs of sealed return capabilities, one in each of the two calls to the adversarial component. The adversary stores the pair of sealed return capabilities from the first call in order to use it in the second call where they are not allowed. The figure illustrates how the adversarial code uses the return pair from the first call to return from the second call and thus break WBCF.

As the examples illustrate, this naive use of standard memory and object capabilities does not provide sufficient guarantees to enforce LSE and WBCF. The problem is essentially that the stack and return pointers that a callee receives from a caller remain in effect after their intended lifetime: either when the callee has already returned or when they have themselves invoked

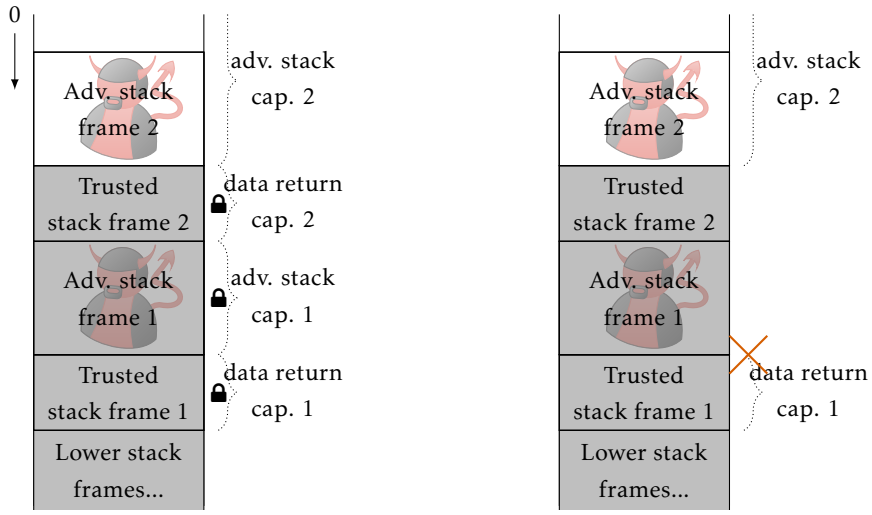
other code. Linear capabilities offer a form of revocation[3] that can be used to prevent this from happening.

The linear capabilities are put to use by requiring the stack capability to be linear. On call, the caller splits the stack capability in two: one capability for their local stack frame and another one for the unused part of the stack. The local stack frame capability is sealed and used as the data part of the sealed return pair. The capability for the remainder of the stack is given to the callee. Because the stack capability is linear, the caller knows that the capability for their local stack frame cannot have an alias. This means that an adversary would need the stack capability produced by the caller in order to access their local data. The caller gives this capability to the adversary only in a sealed form, rendering it opaque and unusable. This is illustrated in Figure 3.7a and prevents the issue illustrated in Figure 3.6a.

In a traditional calling convention with a single stack, the stack serves as a call stack keeping track of the order calls were made in and thus in which order they should be returned to. A caller pushes a stack frame to the stack on call and a callee pops a stack frame from the stack upon return. However without any enforcement, there is nothing to prevent a callee from returning from an arbitrary call on the call stack. This is exactly what the adversary does in Figure 3.6b when they skip two stack frames. In the presence of adversarial code, we need some mechanism to enforce that the order of the call stack is kept. One way to enforce this would be to hand out a token on call that can only be used when the caller's stack frame is on top of the call stack. The callee would have to provide this token on return to prove that it is allowed to return to the caller, and on return the token would be taken back by the caller to prevent it from being spent multiple times. As it turns out, the stack capability for the unused part of the stack can be used as such a token in the following way: On return the callee has to give back the stack capability they were given on invocation. When the caller receives a stack capability back on return, they need to check that this token is actually spendable, i.e. check whether their stack frame is on top of the call stack or not. They do this by attempting to restore the stack capability from before the call by splicing the return token with the stack capability for the local stack frame which at this point has been unsealed again. If the splice is successful, then the caller knows that the two capabilities are adjacent. On the other hand, if the splice fails, then they are alerted to the fact that their stack frame may not be the topmost. StkTokens uses this approach; and as illustrated in Figure 3.7b, it prevents the issue in Figure 3.6b as the adversary does not return a spendable token when they return.

In order for a call to have a presence on the call stack, its stack frame must be non-empty. We cannot allow empty stack frames on the call stack,

[3] Revocation in the sense that if we hand out a linear capability and later get it back, then the receiver no longer has it or a copy of it as it is non-duplicable.



(a) The non-duplicable linear stack capability for the trusted code's stack frame and the opacity of sealed capabilities ensures LSE.

(b) The trusted caller fails to splice the stack capability returned by the adversary with the capability for the trusted caller's local stack frame.

Figure 3.7: Abuse of stack and return capabilities prevention.

because then it would be impossible to tell whether the topmost non-empty stack frame has an empty stack frame on top of it. Non-empty stack frames come naturally in traditional C-like calling convention as they keep track of old stack pointers and old program counters on the stack, but in StkTokens these things are part of the return pair which means that a caller with no local data may only need an empty stack frame. This means that a caller using StkTokens needs to take care that their stack frame is non-empty in order to reserve their spot in the return order. There is also a more practical reason for a StkTokens caller to make sure their stack frame is non-empty: They need a bit of the stack capability in order to perform the splice that verifies the validity of the return token.

At this point, the caller checks that the return token is adjacent to the stack capability for the caller's local stack frame and they have the means to do so. However, this still does not ensure that the caller's stack frame is on top of the call stack. The issue is that stack frames may not be tightly packed leaving space between stack frames in memory. An adversarial callee may even intentionally leave a bit of space in memory above the caller's stack frame, so that they can later return out of order by returning the bit of the return token for the bit of memory left above the caller's stack frame. This is illustrated in Figure 3.8: In Figure 3.8a, a trusted caller has called an adversarial callee. The adversary calls the trusted code back, but first they split the

return token in two and store on the heap the part for the memory adjacent to the trusted caller's call frame (Figure 3.8b). The trusted caller calls the adversary back using the precautions we have described so far (Figure 3.8c). At this point (Figure 3.8b), the adversary has access to a partial return token adjacent to the trusted caller's first stack frame which allows the adversary to return from this call breaking WBCF.

For the caller to be sure that there are no hidden stack frames above its own, they need to make sure that the return token is exactly the same as the one they passed to the callee. In StkTokens, the base address of the stack capability is fixed as a compile-time constant (Note: the stack grows downwards, so the base address of the stack capability is the top-most address of the stack). The caller verifies the validity of the return token by checking whether the base address of a returned token corresponds to this fixed base address, which was the base address for the return token they gave to the callee. In the scenario we just sketched, the caller would be alerted to the attempt to break WBCF when the base address check of the return token fails in Figure 3.8d.

In StkTokens, the stack memory is only referenced by a single linear stack capability at the start of execution. Because of this, the return token can be verified simply by checking its base address and splicing it with the caller's stack frame. There is no need to check linearity because only linear capabilities to this memory exist.

The return pointer in the StkTokens scheme is a pair of sealed capabilities where the code part of the pair is the old program counter, and the data part is the stack capability for the local stack frame of the caller. Both of the capabilities in the pair are sealed with the same seal. All call points need to be associated with a unique seal (a return seal) that is only used for the return capabilities for that particular call point. The return seal is what associates the stack frame on the call stack with a specific call point in a program, so if we allowed return seals to be reused, it would be possible to return to a different call point than the one that gave rise to the stack frame, breaking WBCF. For similar reasons, we cannot allow return seals to be used to seal closures. Return seals should never be leaked to adversarial code as this would allow them to unseal the local stack frame of a caller breaking LSE. This goes for direct leaks (leaving a seal in a register or writing it to adversarial memory), as well as indirect leaks (leaking a capability for reading, either directly or indirectly, a return seal from memory).

We have sometimes phrased the description of the StkTokens calling scheme in terms of "them vs us". This may have created the impression of an asymmetric calling convention that places a special status on trusted components allowing them to protect themselves against adversaries. However, StkTokens is a modular calling scheme: no restriction is put on adversarial components that we do not expect trusted components to meet. Specifically, we are going to assume that both trusted and adversarial components are

initially syntactically well-formed (described in more detail in Section 3.4.2) which basically just restrict adversarial components to not break machine guarantees initially (e.g. no aliases for linear capabilities or access to seals of other components). This means that any component can ensure WBCF and LSE by employing StkTokens.

To summarise, StkTokens consists of the following measures:

1. Check the base address of the stack capability before and after calls.
2. Make sure that local stack frames are non-empty.
3. Create token and data return capability on call: split the stack capability in two to get a stack capability for your local stack frame and a stack capability for the unused part of the stack. The former is sealed and used for the data part of the return pair.
4. Create code return capability on call: Seal the old program counter capability.
5. Reasonable use of seals: Return seals are only used to seal old program counter capabilities, every return seal is only used for one call site, and they are not leaked.

Item 1-4 are captured by the code in Figure 3.9 , except for checking stack base before calls. We do not include this check because it only needs to happen once between two calls, so that the check after a call suffices if the stack base is not changed subsequently.

3.4 Formulating Security with a Fully Abstract Overlay Semantics

As mentioned, the StkTokens calling convention guarantees well-bracketed control flow (WBCF) and local state encapsulation (LSE). However, before we can prove these properties, we need to know how to even formulate them. Although the properties are intuitively clear and sound precise, formalizing them is actually far from obvious.

Ideally, we would like to define the properties in a way that is

1. *intuitive*
2. *useful for reasoning:* we should be able to use WBCF and LSE when reasoning about correctness and security of programs using StkTokens.
3. *reusable in secure compiler chains:* for compilers using StkTokens, one should be able to rely on WBCF and LSE when proving correctness and

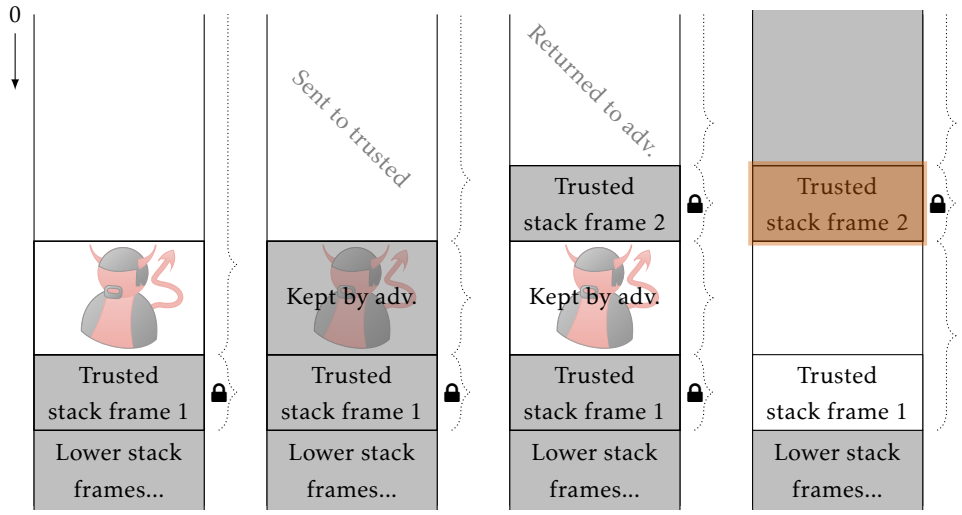


Figure 3.8: Partial return token used to return out of order.

security of other compiler passes and then compose such results with ours to obtain results about the full compiler.

4. *arguably "complete"*: the formalization should arguably capture the entire meaning of WBCF and LSE and should arguably be applicable to any reasonable program.

5. *potentially scalable*: although dynamic code generation and multi-threading are currently out of scope, the formalization should, at least potentially, extend to such settings.

Previous formalisations in the literature are formulated in terms of a static control flow graph [e.g., 10]. While these are intuitively appealing (1), it is not clear how they can be used to reason about programs (2) or other compiler passes (3), they lack temporal safety guarantees (4) and do not scale (5) to settings with dynamic code generation (where a static control flow graph cannot be defined). Skorstengaard et al. [83] provide a logical relation that can be used to reason about programs using their calling convention (2,3), but it is not intuitive (1), there is no argument for completeness (4), and it is unclear whether it will scale to more complex features (5).

We contribute a new way to formalise the properties using a novel approach we call fully abstract overlay semantics. The idea is to define a second operational semantics for programs in our target language. This second semantics uses a different abstract machine and different run-time values, but it executes in lock-step with the original semantics and there is a very close correspondence between the state of both machines.

```
// Ensure non-empty stack.
 1 : move r_t1 42
 2 : store r_stk r_t1
 3 : cca r_stk (−1)
// Split stack in local stack frame and unused.
 4 : geta r_t1 r_stk
 5 : split r_stk r_rdata r_stk r_t1
// Load the call seal.
 6 : move r_t1 pc
 7 : cca r_t1 (off_pc − 5)
 8 : load r_t1 r_t1
 9 : cca r_t1 off_σ
// Seal the local stack frame.
10 : cseal r_rdata r_t1
// Construct code return pointer.
11 : move r_rcode pc
12 : cca r_rcode 5
13 : cseal r_rcode r_t1
// Clear tmp registers and jump.
14 : move r_t1 0
15 : xjmp r_1 r_2
// The following is the return code.
// Returned stk. ptr. must have base stk_base.
16 : getb r_t1 r_stk
17 : minus r_t1 r_t1 stk_base
18 : move r_t2 pc
19 : cca r_t2 5
20 : jnz r_t2 r_t1
21 : r_t2 1
22 : jmp r_t2
23 : fail
// Splice with capability for local stack frame.
24 : splice r_stk r_stk r_data
// Pop 42 from the stack
25 : cca r_stk 1
// Clear tmp register
26 : move r_t2 0
```

Figure 3.9: The instructions for a $\texttt{call}^{off_{pc},off_\sigma}\ r_1\ r_2$ with off_{pc} the offset from line 1 of the call to the set of seals it uses and off_σ the offset in the set of seals to the call seal. stk_base is the globally agreed on stack base. There are some magic numbers in the code: line 1: 42, garbage data to ensure a non-empty stack. Line 7: −5, offset from line 6 (where pc was copied into r_{t1}) to line 1. Line 12: 5, offset to the return address. Line 19: 5, offset to fail. Line 21: offset to address after fail.

The main difference between the two semantics, is that the new one satisfies LSE and WBCF by construction: the abstract machine comes with a built-in stack, inactive stack frames are unaddressable and well-bracketed control flow is built-in to the abstract machine. Important run-time values like return capabilities and stack pointers are represented by special syntactic tokens that interact with the abstract machine's stack, but during execution, there remains a close, structural correspondence to the actual regular capabilities that they represent. For example, stack capabilities in the overlay semantics correspond directly to linear capabilities in the underlying semantics, and they have authority over the part of memory that the overlay views as the stack. The new run-time values in the overlay semantics affect the definition of the encoding function *encodeType*. All the new values correspond to concrete capabilities on the LCM machine which the encoding function must respect. For instance, the encoding of a stack pointer in the overlay semantics should be the same as the encoding of a linear capability on the LCM machine.

The fact that StkTokens enforces LSE and WBCF is then formulated as a theorem about the function that maps components in the well-behaved overlay semantics to the underlying components in the regular semantics. The theorem states that this function constitutes a fully abstract compiler, a well-known property from the field of secure compilation [9]. Intuitively, the theorem states that if a trusted component interacts with (potentially malicious) components in the regular semantics, then these components have no more expressive power than components which the trusted component interacts with in the well-behaved overlay semantics. In other words, they cannot do anything that doesn't correspond to something that a well-behaved component, respecting LSE and WBCF, can also do. More formally, our full-abstraction result states that two trusted components are indistinguishable to arbitrary other components in the regular semantics iff they are indistinguishable to arbitrary other components in the overlay semantics.

Our formal results are complicated by the fact that they only hold on a sane initial configuration of the system and for well-behaved components that respect the basic rules of the calling convention. For example, the system should be set up such that seals used by components for constructing return pointers are not shared with other components. We envision distributing seals as a job for the linker, so this means our results depend on the linker to do this properly. As another example, a seal used to construct a return pointer can be reused but only to construct return pointers for the same return point. Different seals must be used for different return points. Such seals should also never be passed to other components. These requirements are easy to satisfy: components should request sufficient seals from the linker, use a different one for every place in the code where they make a call to another component, and make sure to clear them from registers before every call. The general pattern is that StkTokens only protects components that do not shoot themselves in the foot by violating a few basic rules. In this section, we define a well-formedness judgement for the syntactic requirements on components as well as a reasonability condition that semantically disallows components to do certain unsafe things. Well-formedness is a requirement for all components (trusted and untrusted), but the reasonability requirement only applies to trusted components, i.e. those components for which we provide LSE and WBCF guarantees.

3.4.1 Overlay Semantics

The overlay semantics oLCM for LCM views part of the memory as a built-in stack (Figure 3.10). To this end, it adds a call stack and a free stack memory to the executable configurations of LCM. The call stack is a list with all the stack frames that are currently inaccessible because they belong to previous calls. Every stack frame contains encapsulated stack memory as well as the program point that execution is supposed to return to. The free stack memory is

$$\begin{aligned}
&\text{Sealables} ::= \text{Sealables} \mid \text{stack-ptr}(\text{perm}, \text{base}, \text{end}, \text{a}) \mid \\
&\qquad\qquad \text{ret-ptr-data}(\text{base}, \text{end}) \mid \text{ret-ptr-code}(\text{base}, \text{end}, \text{a}) \\
&\text{StackFrame} \stackrel{def}{=} \text{Addr} \times \text{MemSeg} \qquad\qquad \text{Stack} \stackrel{def}{=} \text{StackFrame}^* \\
&\text{ExecConf} \stackrel{def}{=} \text{Memory} \times \text{RegFile} \times \text{Stack} \times \text{MemSeg} \\
&\text{Instr} ::= \text{Instr} \mid \texttt{call}^{off_{\text{pc}}, off_{\sigma}}\ r\ r \qquad\qquad off_{\text{pc}}, off_{\sigma} \in \mathbb{N}
\end{aligned}$$

Figure 3.10: The syntax of oLCM. oLCM extends LCM by adding stack pointers, return pointers, and a built-in stack. Everything specific to the overlay semantics is written in blue.

the active part of the stack that has not been claimed by a call and thus can be used at this point of time. In order to distinguish capabilities for the stack from the capabilities for the rest of the memory, oLCM adds stack pointers. A stack pointer has a permission, range of authority, and current address, just like capabilities on LCM, but they are always linear. The final syntactic constructs added by oLCM are the code and data return pointers. The data return pointer corresponds to some stack pointer (which in turn corresponds to a linear capability), and the code return pointer corresponds to some capability with read-execute permission. Syntactically, the return capabilities contain just enough information to reconstruct what they correspond to on the underlying machine. On oLCM, return pointers are generated by calls from the capabilities they correspond to on LCM, and they are turned back to the capabilities they correspond to upon return.

The opaque nature of the return pointers is reflected in the interpretation of the instructions common to both LCM and oLCM as oLCM does not add special interpretation for them in non-`xjmp` instructions. Stack pointers, on the other hand, need to behave just like capabilities, so oLCM adds new cases for them in the semantics, e.g. `cca` can now also change the current address of a stack pointer as displayed in Figure 3.11. Similarly, `load` and `store` work on the free part of the stack when provided with a stack pointer. A store attempted with a stack capability that points to an address outside the free stack results in the failed configuration because that action is inconsistent with the view the overlay semantics has on the underlying machine. In other words, there should only be stack pointers for the stack memory.

As discussed earlier, our formal results only provide guarantees for components that respect the calling convention. Untrusted components are not assumed to do so. To formalize this distinction, oLCM has a set of trusted addresses T_A. Only instructions at these addresses can be interpreted as the oLCM native call and push frames to the call stack which guarantees LSE and WBCF. The constant T_A is a parameter of the oLCM step relation. Similarly, StkTokens assumes a fixed base address of the stack memory, that is

also passed around as such a parameter, for use in the native semantics of calls.

Apart from the step relation of LCM, oLCM has one overlay step that takes precedence over the others. This step is shown in Figure 3.11, and it is different from the others in the sense that it interprets a sequence of instructions rather than one. The sequence of instructions have to correspond to a call, i.e. the instructions in Figure 3.9 ($\texttt{call}_i^{off_{\text{pc}}, off_\sigma}\ r_1\ r_2$ corresponds to the i'th instruction in the figure and call_len is always 26, i.e. the number of instructions). Calls are only executed when the well-behaved component executes, so the addresses where the call resides must be in T_A, and the executing capability must have the authority to execute the call.

The interpretation of $\texttt{call}^{off_{\text{pc}}, off_\sigma}\ r_1\ r_2$ is also shown in Figure 3.11 and essentially does the following: The registers r_1 and r_2 are expected to contain a code-data pair sealed with the same seal and the unsealed values are invoked by placing them in the pc and r_{data} registers, respectively. The current active stack and the stack capability are split into the local stack frame of the caller and the rest. `call` also constructs a return capability c_{opc} and its address opc, pointing after the call instructions. The local stack frame and return address are pushed onto the stack, and the local stack capability and return capability are converted into a pair of sealed return capabilities. The return capabilities are sealed with the seal designated for the call.

The return capabilities, ret-ptr-code and ret-ptr-data are sealed and can only be used using the `xjmp` instruction, to perform a return. When this happens, the topmost call stack frame (opc, ms_{local}) is popped from the call stack. In order for the return to succeed, the return address in the code return pointer must match opc, and the range of addresses in the data return pointer must match the domain of the local stack. If the return succeeds, the stack pointer is reconstructed, and the local stack becomes part of the active stack again.

oLCM supports tail calls. A tail call is a call from a caller that is done executing, and thus doesn't need to be returned to or preserve local state. This means that a tail call should not reserve a slot in the return order by pushing a stack frame on the call stack, i.e. it should not use the built-in call. To perform a tail call, the caller simply transfers control to the callee using `xjmp`. The tail-callee should return to the caller's caller, so the caller leaves the return pair they received for the callee to use.

It is important to observe that the operational semantics of oLCM natively guarantee WBCF (well-bracketed control flow) and (local stack encapsulation) for calls made by trusted components. By inspecting the operational semantics of oLCM, we can see that it never allow reads or writes to inactive stack frames on the call stack. The built-in call for trusted code pushes the local stack frame to the inactive part of the stack, together with the return address. Such frames can be reactivated by `xjmp`ing to a return capability

pair, but only for the topmost stack frame and if the return address corresponds to the one stored in the call stack. In other words, WBCF and LSE are natively enforced in this semantics.

3.4.2 Well-Formed Components

The components introduced in Section 3.2.3 are pretty much unconstrained. For instance, a component can have multiple linear capabilities for the same piece of memory, and there are no restrictions on seals. In a real system, the operating system and linker would make sure everything is setup correctly. For instance, they would not allocate multiple linear capabilities for the same memory, and they would ensure sane seal allocation.

The component notion is used for both trusted and adversarial components. We expect the linker to link the two programs together making sure that all the system invariants are respected. We expect trusted components to be the product of a known compiler that uses StkTokens which means that the code must have a certain shape. The proper linker behaviour and the syntactic expectations we have of trusted component are captured in a syntactic well-formedness judgement which we explain in this section.

The well-formedness judgement imposes a quite rigid structure on well-formed components: a component's code memory may contain only data and sets of seals. The data memory may contain only data, memory capabilities to the component's data memory (respecting linearity) or sealed memory capabilities (to the component's data memory, sealed with closure seals). This makes a clear separation between code and data memory as neither can contain capabilities for the other. The separation of the two kinds of memory means that data structures in data memory can be shared without risking unintentionally leaking a return seals that were indirectly accessible through the data structure. Informally, a well-formed component respects Write-XOR-Execute which means that all capabilities for code memory at most has read-execute permission and all capabilities for data memory at most has read-write permission. A component's exports should be sealed code or data capabilities, sealed with a closure seal.

As discussed before, we only provide guarantees (LSE and WBCF) for components respecting certain semantic rules (reasonable use of seals). Components that do not satisfy these requirements are still allowed in oLCM, but (as we will see below) function calls in such components will not behave according to the special oLCM semantics that guarantees LSE and WBCF. To distinguish trusted components (which satisfy reasonability requirements) from adversarial components (which don't), we make use of a set of trusted addresses T_A. Any component's code memory must fall entirely within or outside of T_A, so every component is either trusted or adversarial. Some well-formedness requirements are specific for adversarial and trusted components. Well-formed adversarial components cannot have return seals. On

$$\frac{\begin{array}{c}\Phi(\mathrm{pc}) = ((p,\_),b,e,a) \\ [a,a+\mathrm{call\_len}-1] \subseteq T_A \qquad [a,a+\mathrm{call\_len}-1] \subseteq [b,e] \qquad p \in \{\textsc{rwx},\textsc{rx}\} \\ \Phi.mem(a,\ldots,\mathrm{a}+\mathrm{call\_len}-1) = \mathtt{call}_0^{off_{\mathrm{pc}},off_\sigma}\ r_1\ r_2 \cdots \mathtt{call}_{\mathrm{call\_len}-1}^{off_{\mathrm{pc}},off_\sigma}\ r_1\ r_2\end{array}}{\Phi \rightarrow^{T_A,\mathrm{stk\_base}} [\![\mathtt{call}^{off_{\mathrm{pc}},off_\sigma}\ r_1\ r_2]\!](\Phi)}$$

$i \in$ Instr	$[\![i]\!](\Phi)$	Conditions
`halt`	halted	
... (the operational semantics of LCM)		
`store` r_1 r_2	$updPc(\Phi[reg.r_2 \mapsto w_2]$ $[ms_{stk}.a \mapsto \Phi(r_2)])$	$\Phi(r_1) = \text{stack-ptr}(p,b,e,a)$ and $p \in \{\textsc{rwx},\textsc{rw}\}$ and $b \le a \le e$ and $w_2 = linClear(\Phi(r_2))$ and $a \in \mathrm{dom}(ms_{stk})$
`cca` r rn	$updPc(\Phi[reg.r \mapsto w])$	$\Phi(rn) = n \in \mathbb{Z}$ and $\Phi(r) = \text{stack-ptr}(p,b,e,a)$ and $w = \text{stack-ptr}(p,b,e,a+n)$
$\mathtt{call}^{off_{\mathrm{pc}},off_\sigma}\ r_1\ r_2$	$xjmpRes(c_1,c_2,$ $\left(\begin{array}{l}\Phi[reg.r_1,r_2 \mapsto w_1,w_2] \\ \quad [reg.\mathrm{r_{rcode}} \mapsto s_c] \\ \quad [reg.\mathrm{r_{rdata}} \mapsto s_d] \\ \quad [reg.\mathrm{r_{stk}} \mapsto c_{stk}] \\ \quad [ms_{stk} \mapsto ms_{stk,rest}] \\ \quad [stk \mapsto stk']\end{array}\right))$	$ms_{stk,local}, c_{local}, ms_{stk,rest}, c_{stk} =$ $splitStack(\Phi.reg(\mathrm{r_{stk}}), \Phi.ms_{stk})$ and $opc, c_{opc} = setupOpc(\Phi.reg(\mathrm{pc}))$ and $stk' = (opc, ms_{stk,local}) :: \Phi.stk$ and $\sigma =$ $getCallSeal($ $\Phi.reg(\mathrm{pc}), \Phi.mem, off_{\mathrm{pc}}, off_\sigma)$ and $s_c, s_d = sealReturnPair(\sigma, c_{opc}, c_{local})$ and $w_1, w_2 = linClear(\Phi.reg(r_1, r_2))$ and $\Phi.reg(r_1, r_2) =$ $\text{sealed}(\sigma', c_1), \text{sealed}(\sigma', c_2)$
...		
_	failed	otherwise

$$xjmpRes(c_1,c_2,\Phi) = \begin{cases} \Phi[reg.\mathrm{pc} \mapsto c_1] \\ \quad [reg.\mathrm{r_{data}} \mapsto c_2] & nonExec(c_2) \text{ and } c_1 \neq \text{ret-ptr-code}(\_) \text{ and } c_2 \neq \text{ret-ptr-data}(\_) \\ \Phi[reg.\mathrm{pc} \mapsto c_{opc}] \\ \quad [reg.\mathrm{r_{stk}} \mapsto c_{stk}] \\ \quad [reg.\mathrm{r_{data}} \mapsto 0] \\ \quad [stk \mapsto stk'] \\ \quad [ms_{stk} \mapsto ms_{stk} \uplus ms_{local}] & (opc, ms_{local}) :: stk' = \Phi.stk \wedge c_1 = \text{ret-ptr-code}(b,e,opc) \wedge c_2 = \text{ret-ptr-data}(a_{stk}, e_{stk}) \wedge \mathrm{dom}(ms_{local}) = [a_{stk}, e_{stk}] \wedge c_{stk} = reconstructStackPointer(\Phi.reg(\mathrm{r_{stk}}), c_2) \wedge c_{opc} = ((\textsc{rx}, \text{normal}), b, e, opc) \\ \text{failed} & \text{otherwise} \end{cases}$$

Figure 3.11: An excerpt of the operational semantics of oLCM (some details omitted). Auxiliary definitions are found in Figure 3.12.

$$splitStack(\text{stack-ptr}(\textsc{rw}, b_{stk}, e_{stk}, a_{stk}), ms_{stk}) = ms_{stk,local}, c_{local\_data}, ms_{stk,unused}, c_{stk} \textit{ iff } \begin{cases} b_{stk} < a_{stk} \leq e_{stk} \\ ms_{stk,local} = ms_{stk}|_{[a_{stk},e_{stk}]}[a_{stk} \mapsto 42] \\ ms_{stk,unused} = ms_{stk}|_{[b_{stk},a_{stk}-1]} \\ c_{stk} = \text{stack-ptr}(\textsc{rw}, b_{stk}, a_{stk}-1, a_{stk}-1) \\ c_{local\_data} = \text{ret-ptr-data}(a_{stk}, e_{stk}) \end{cases}$$

$$setupOpc(((\_,\_), b, e, a)) = opc, c_{opc} \textit{ iff } \begin{cases} opc = a + \text{call\_len} \wedge \\ c_{opc} = \text{ret-ptr-code}(b, e, opc) \wedge \end{cases}$$

$$getCallSeal(c_{\text{pc}}, mem, off_{\text{pc}}, off_{\sigma}) = \sigma \textit{ iff } \begin{cases} c_{\text{pc}} = ((\_,\_), b, e, a) \wedge b \leq a + off_{\text{pc}} \leq e \wedge \\ mem(a + off_{\text{pc}}) = \text{seal}(\sigma_b, \sigma_e, \sigma_a) \wedge \sigma_b \leq \sigma \leq \sigma_e \wedge \\ \sigma = \sigma_a + off_{\sigma} \end{cases}$$

$$sealReturnPair(\sigma, c_{opc}, c_{local}) = \text{sealed}(\sigma, c_{opc}), \text{sealed}(\sigma, c_{local})$$

$$reconstructStackPointer(\text{stack-ptr}(\textsc{rw}, \text{stk\_base}, a_{stk}-1, \_), \text{ret-ptr-data}(a_{stk}, e_{stk})) = \text{stack-ptr}(\textsc{rw}, \text{stk\_base}, e_{stk}, a_{stk}) \textit{ iff } \text{stk\_base} \leq a_{stk}$$

Figure 3.12: Auxiliary definitions used in the operational semantics of oLCM.

the other hand, well-formed trusted components can have return seals, but must make return seals available to all its calls in a way that is consistent with StkTokens. This means that each return seal is only used for one call.

These requirements are formally expressed by the judgement $T_A \vdash comp$ specifies, i.e. it defines when components satisfy the initial syntactic requirements necessary to be able to rely on unique linear capabilities, component unique seals, etc. The judgement is defined in Figure 3.13 with auxiliary judgements in Figure 3.14.

The $T_A \vdash comp$ judgement has two rules: Main and Base. The Main rule requires the component to have an entry point in terms of a main pair from the exports. Further, the base component should satisfy the well-formedness judgement. The Base judgement ensures sure that the component has a bit of structure. Roughly speaking, the Base rule requires the following:

- *Code and data memory are disjoint.*
- *Code memory is padded with zeros (ms_{pad}).* This prevents code memories from being spliced together. If the capabilities for two code memories can be spliced together, then the execution of one code memory can continue into the other creating an unintended control-flow. Further,

$$\frac{\begin{array}{c}\mathrm{dom}(ms_{\mathrm{code}}) = [b,e] \qquad [b-1,e+1] \# \mathrm{dom}(ms_{\mathrm{data}}) \\ ms_{\mathrm{pad}} = [b-1 \mapsto 0] \uplus [e+1 \mapsto 0] \qquad \exists A_{\mathrm{own}} : \mathrm{dom}(ms_{\mathrm{data}}) \to \mathcal{P}(\mathrm{dom}(ms_{\mathrm{data}})) \\ \mathrm{dom}(ms_{\mathrm{data}}) = A_{\mathrm{non-linear}} \uplus A_{\mathrm{linear}} \\ A_{\mathrm{linear}} = \biguplus_{a \in \mathrm{dom}(ms_{\mathrm{data}})} A_{\mathrm{own}}(a) \qquad \overline{export} = \overline{s_{\mathrm{export}} \mapsto w_{\mathrm{export}}} \\ \overline{import} = \overline{a_{\mathrm{import}} \leftarrow\!\shortmid\, s_{\mathrm{import}}} \qquad \{\overline{a_{\mathrm{import}}}\} \subseteq \mathrm{dom}(ms_{\mathrm{data}}) \\ \overline{s_{\mathrm{import}}} \# \overline{s_{\mathrm{export}}} \qquad (\emptyset \neq \mathrm{dom}(ms_{\mathrm{code}}) \subseteq T_A) \vee (\mathrm{dom}(ms_{\mathrm{code}}) \# T_A \wedge \overline{\sigma_{\mathrm{ret}}} = \emptyset) \\ \mathrm{dom}(ms_{\mathrm{data}}) \# T_A \qquad \overline{\sigma_{\mathrm{ret}}}, \overline{\sigma_{\mathrm{clos}}}, T_A \vdash_{\mathrm{comp-code}} ms_{\mathrm{code}} \\ \forall a \in \mathrm{dom}(ms_{\mathrm{data}}).\, \mathrm{dom}(ms_{\mathrm{code}}), A_{\mathrm{own}}(a), A_{\mathrm{non-linear}}, \overline{\sigma_{\mathrm{clos}}} \vdash_{\mathrm{comp-word}} ms_{\mathrm{data}}(a) \\ \overline{\mathrm{dom}(ms_{\mathrm{code}}), A_{\mathrm{non-linear}}, \overline{\sigma_{\mathrm{ret}}}, \overline{\sigma_{\mathrm{clos}}} \vdash_{\mathrm{comp-export}} w_{\mathrm{export}}}\end{array}}{T_A \vdash (ms_{\mathrm{code}} \uplus ms_{\mathrm{pad}}, ms_{\mathrm{data}}, \overline{import}, \overline{export}, \overline{\sigma_{\mathrm{ret}}}, \overline{\sigma_{\mathrm{clos}}}, A_{\mathrm{linear}})}\ \textsc{Base}$$

$$\frac{\begin{array}{c}comp_0 = (ms_{\mathrm{code}}, ms_{\mathrm{data}}, \overline{import}, \overline{export}, \overline{\sigma_{\mathrm{ret}}}, \overline{\sigma_{\mathrm{clos}}}, A_{\mathrm{linear}}) \\ T_A \vdash comp_0 \qquad (\_ \mapsto c_{\mathrm{main},c}), (\_ \mapsto c_{\mathrm{main},d}) \in \overline{export}\end{array}}{T_A \vdash (comp_0, c_{\mathrm{main},c}, c_{\mathrm{main},d})}\ \textsc{Main}$$

Figure 3.13: Well-formedness judgement.

the possible control-flows of a component suddenly depends on the memory locations of all the components[4].

- *All memory can either be addressed by any number of non-linear capabilities or at most one linear capability.* The data address space is split into $A_{non-linear}$ and A_{linear} that can be addressed by non-linear capabilities and linear capabilities, respectively. The judgement ensure uniqueness of linear capabilities by letting each address of the data memory take sole ownership of addresses in A_{linear}.
- *All import addresses are part of the data memory.*
- *Import and export symbols are disjoint.*
- *One of the following two are true*
  - *The code address space is disjoint from the trusted address space T_A and there are no return seals.* In this case, the component contains untrusted code. We are interested in WBCF and LSE from the perspective of the trusted code, so we do not let untrusted memory have return seals[5].

[4] Alternatively to this syntactic condition on components, we could require trusted components to never splice executable capabilities of unknown origin.

[5] Untrusted code still has access to seals that could be used to protect calls.

  - *The code address space is part of the trusted address space T_A.* In this case, the component contains trusted code. We do not impose requirements on the return seals here (the auxiliary judgements will impose restrictions on them), so the trusted component can have the return seals necessary for its calls.

- *The data address space is disjoint from the trusted address space T_A.* Data memory is not executable, so the trusted addresses never include the data memory addresses.

- *The code memory satisfies the component-code judgement (Figure 3.14a).*

- *Each word in the data memory satisfies the component-word judgement (Figure 3.14b).* This must be with respect to the linear addresses assigned to this address.

- *All the exports satisfy the components-export judgement (Figure 3.14c).*

Figure 3.14b defines the judgement $\vdash_{\text{comp-word}}$ which specifies the words that are well-formed in a component's data memory. The judgement is defined by three rules: W-Data, W-Capability, and W-Sealed-Capability. The W-Data rule says all data, i.e. integers, are well-formed words. The W-Capability rule specifies that all well-formed capabilities in data memory must at most have read and write permission. Further, if the capability is linear, then the range of authority must be within the linear address space owned by this address. Finally, if it is non-linear, then it must be in the non-linear address space. The W-Sealed-Capability specifies that well-formed sealed capabilities in data memory are sealed with a closure seal and whatever is sealed is itself a well-formed word. Because of the last requirement, the data memory cannot initially contain sealed code capabilities. However, a component is free to place sealed data capabilities in memory during execution.

Figure 3.14c defines the well-formed exports $\vdash_{\text{comp-export}}$. E-Word says that any well-formed word for data memory is well-formed as an export. In particular, this means that sealed capabilities for the data memory are acceptable exports which allows components to export the data part of a sealed capability pair. The E-Sealed-Code rule allows the code part of a sealed capability pair to be exported. In particular, it allows a capability with read-execute permission and its range of authority within the component's code address space to be exported when it has been sealed with a closure seal. Together, these two rules allow components to export closures.

Figure 3.14a defines the well-formed code memories $\vdash_{\text{comp-code}}$. The well-formed code memories are defined by two judgements. One judgement considers the entire memory $\overline{\sigma_{\text{ret}}}, \overline{\sigma_{\text{clos}}}, T_A \vdash_{\text{comp-code}} ms_{\text{code}}$ while the other considers the contents of each code memory address $\overline{\sigma_{\text{ret}}}, \overline{\sigma_{\text{ret,owned}}}, \overline{\sigma_{\text{clos}}}, T_A \vdash_{\text{comp-code}}$

$$\frac{ms_{\text{code}}(a) = \text{seal}(\sigma_b, \sigma_e, \sigma_b) \qquad [\sigma_b, \sigma_e] = (\overline{\sigma_{\text{ret}}} \cup \overline{\sigma_{\text{clos}}})}{\overline{\sigma_{\text{ret}}}, \overline{\sigma_{\text{ret,owned}}}, \overline{\sigma_{\text{clos}}}, T_A \vdash_{\text{comp-code}} ms_{\text{code}}, a}\ \text{C-Seals}$$

$$\frac{\begin{array}{c} ms_{\text{code}}(a) \in \mathbb{Z} \\ \left(\begin{array}{l} [a \cdots a + \text{call\_len} - 1] \subseteq T_A \wedge \\ ms_{\text{code}}([a \cdots a + \text{call\_len} - 1]) = \texttt{call}^{off_{\text{pc}}, off_\sigma}_{0..\text{call\_len}-1}\ r_1\ r_2 \end{array}\right) \Rightarrow \\ (ms_{\text{code}}(a + off_{\text{pc}}) = \text{seal}(\sigma_b, \sigma_e, \sigma_b) \wedge \sigma_b + off_\sigma \in \overline{\sigma_{\text{ret,owned}}}) \end{array}}{\overline{\sigma_{\text{ret}}}, \overline{\sigma_{\text{ret,owned}}}, \overline{\sigma_{\text{clos}}}, T_A \vdash_{\text{comp-code}} ms_{\text{code}}, a}\ \text{C-Instr}$$

$$\frac{\begin{array}{c} ms_{\text{code}} \text{ has no hidden calls} \\ \overline{\sigma_{\text{ret}}} \,\#\, \overline{\sigma_{\text{clos}}} \qquad \exists d_\sigma : \text{dom}(ms_{\text{code}}) \to \mathcal{P}(\text{Seal}).\ \overline{\sigma_{\text{ret}}} = \biguplus_{a \in \text{dom}(ms_{\text{code}})} d_\sigma(a) \wedge \\ \forall a \in \text{dom}(ms_{\text{code}}).\ \overline{\sigma_{\text{ret}}}, d_\sigma(a), \overline{\sigma_{\text{clos}}}, T_A \vdash_{\text{comp-code}} ms_{\text{code}}, a \\ \exists a.\ ms_{\text{code}}(a) = \text{seal}(\sigma_b, \sigma_e, \_) \wedge [\sigma_b, \sigma_e] \neq \emptyset \end{array}}{\overline{\sigma_{\text{ret}}}, \overline{\sigma_{\text{clos}}}, T_A \vdash_{\text{comp-code}} ms_{\text{code}}}\ \text{C-Mem}$$

(a) Code well-formedness.

$$\frac{z \in \mathbb{Z}}{A_{\text{code}}, A_{\text{own}}, A_{\text{non-linear}}, \overline{\sigma_{\text{clos}}} \vdash_{\text{comp-word}} z}\ \text{W-Data}$$

$$\frac{\begin{array}{c} \text{perm} \sqsubseteq \text{RW} \qquad l = \text{linear} \Rightarrow \emptyset \subset [b, e] \subseteq A_{\text{own}} \\ l = \text{normal} \Rightarrow [b, e] \subseteq A_{\text{non-linear}} \end{array}}{A_{\text{code}}, A_{\text{own}}, A_{\text{non-linear}}, \overline{\sigma_{\text{clos}}} \vdash_{\text{comp-word}} ((p, l), b, e, a)}\ \text{W-Capability}$$

$$\frac{A_{\text{code}}, A_{\text{own}}, A_{\text{non-linear}}, \overline{\sigma_{\text{clos}}} \vdash_{\text{comp-word}} sc \qquad \sigma \in \overline{\sigma_{\text{clos}}}}{A_{\text{code}}, A_{\text{own}}, A_{\text{non-linear}}, \overline{\sigma_{\text{clos}}} \vdash_{\text{comp-word}} \text{sealed}(\sigma, sc)}\ \text{W-Sealed-Capability}$$

(b) Word well-formedness.

$$\frac{[b, e] \subseteq A_{\text{code}} \qquad \sigma \in \overline{\sigma_{\text{clos}}}}{A_{\text{code}}, A_{\text{non-linear}}, \overline{\sigma_{\text{clos}}} \vdash_{\text{comp-export}} s \mapsto \text{sealed}(\sigma, ((\text{RX}, \text{normal}), b, e, a))}\ \text{E-Sealed-Code}$$

$$\frac{A_{\text{code}}, \emptyset, A_{\text{non-linear}}, \overline{\sigma_{\text{clos}}} \vdash_{\text{comp-word}} w}{A_{\text{code}}, A_{\text{non-linear}}, \overline{\sigma_{\text{clos}}} \vdash_{\text{comp-export}} s \mapsto w}\ \text{E-Word}$$

(c) Export well-formedness

Figure 3.14: Well-formedness judgement for code, words, and export. call_len it the length of the call code.

ms_{code}, a. Unlike data memory and exports, we cannot say whether code memory is well-formed by considering each word in isolation. The well-formedness of code memory depends on whether seals for calls are located in the place we expect them to be located. The judgement $\overline{\sigma_{\text{ret}}}, \overline{\sigma_{\text{ret,owned}}}, \overline{\sigma_{\text{clos}}}, T_A \vdash_{\text{comp-code}} ms_{\text{code}}, a$ has two rules. The C-Seal judgement requires address a in ms_{code} to contain a set of seals that is a subset of the closure seals and the return seals. C-Instr requires address a to contain an integer, which may be interpreted as instruction. If address a is the first address of a series of instructions that may be interpreted as a call, and all the addresses are within the trusted address space, then there must be a set of seals available at the address the call in question expects it and that set of seals must contain the return seal that corresponds to the return seal of this call. The return seal in question must be in the $\overline{\sigma_{\text{ret,owned}}}$ to ensure that no other call uses it. In order to ensure that return seals are at most used by one call, the $\overline{\sigma_{\text{ret}}}, \overline{\sigma_{\text{clos}}}, T_A \vdash_{\text{comp-code}} ms_{\text{code}}$ judgement takes all the available return seals and partitions them over code addresses. This means that no other call can use the same seal. Further, return seals cannot be used to seal non-return capabilities, so the C-Mem judgement requires the set of return seals to be disjoint from the set of closure seals. To limit the amount of corner cases we have to consider, we require the code memory to have no *hidden calls*.

Definition 3.4.1 (No hidden calls)**.** We say that a memory segment ms_{code} has no hidden calls iff

$$
\begin{aligned}
&\forall a \in \text{dom}(ms_{\text{code}}).\\
&\quad \forall i \in [0, \text{call\_len} - 1]\\
&\qquad ms_{\text{code}}(a+i) = \texttt{call}_{i}^{off_{\text{pc}}, off_{\sigma}}\ r_1\ r_2 \Rightarrow\\
&\qquad\quad (\text{dom}(ms_{\text{code}}) \supseteq [a-i, a+\text{call\_len}-i-1] \wedge\\
&\qquad\qquad ms_{\text{code}}([a-i, a+\text{call\_len}-i-1]) = \texttt{call}_{0..\text{call\_len}-1}^{off_{\text{pc}}, off_{\sigma}}\ r_1\ r_2) \vee\\
&\qquad\quad \exists j \in [a-i, a+\text{call\_len}-i-1] \cap \text{dom}(ms_{\text{code}}).\ ms_{\text{code}}(j) \neq \texttt{call}_{j-a-i}^{off_{\text{pc}}, off_{\sigma}}\ r_1\ r_2
\end{aligned}
$$

♦

Definition 3.4.1 says that if code memory contains an instruction that may be part of a call, then either it is a part of a call or there is some other instruction that witnesses the fact that it is not part of a call. Finally, the C-Mem judgement requires that the code at least have one non-empty set of seals available.

3.4.3 Reasonable Components

The static guarantees given by $T_A \vdash comp$ makes sure that components initially don't undermine the security measures needed for StkTokens, but it does not prevent a component from doing something silly during execution

that undermines StkTokens. In order for StkTokens to provide guarantees for a component, we expect it to not shoot itself in the foot and perform certain necessary checks not captured by the call code (Figure 3.9). More precisely, we expect four things of a reasonable component:

(1) It checks the stack base address before performing a call. As explained in Section 3.3, we do not include this check in the call code as it often would be redundant.

(2) It uses the return seals only for calls and the closure seals in an appropriate way which means that they should only be used to seal executable capabilities for code that behaves reasonably or non-executable things that do not undermine the security mechanisms StkTokens relies on.

(3) It does not leak return and closure seals or means to retrieve them. This means that sets of seals with return or closure seals cannot be left in registers when transferring control to another module. There are also indirect ways to leak seals such as leaking a capability for code memory or leaking a capability for code memory sealed with an unknown seal.

(4) It does not store return and closure seals or means to get them. By disallowing this, we make sure that data memory always can be safely shared as it does not contain seals or means to get them to begin with.

We capture these properties in 4 definitions. Definition 3.4.2 defines the reasonable words which means that they cannot be used to leak seals directly or indirectly. Definition 3.4.3 defines the reasonable pc's which means that if it is plugged into a configuration with a register file filled with reasonable words, then the configuration behaves reasonably. Definition 3.4.4 defines the reasonable configurations which captures the four informal behavioural properties. Finally, definition 3.4.5 lifts the notion of reasonability to components.

Reasonable words

To provide any guarantees, StkTokens rely on the program to no leak return seals in any way. But what does it mean to "leak" a return seal? It means that sets of seals that contain return seals as well as any means to obtain such sets cannot be leaked. The following definition makes this more precise.

Definition 3.4.2 (Reasonable word)**.** Take a set of trusted addresses T_A and sets of return and closure seals $\overline{\sigma_{\text{glob\_ret}}}$ and $\overline{\sigma_{\text{clos}}}$. We define that a word w is reasonable up to n steps in memory ms and free stack ms_{stk} if $n = 0$ or the following implications hold.

- If $w = \text{seal}(\sigma_b, \sigma_e, \_)$, then $[\sigma_b, \sigma_e] \# (\overline{\sigma_{\text{glob\_ret}}} \cup \overline{\sigma_{\text{glob\_clos}}})$

- If $w = ((p, \_), b, e, \_)$, then $[b, e] \# T_A$
- If $w = \text{sealed}(\sigma, sc)$ and $\sigma \notin (\overline{\sigma_{\text{glob\_ret}}} \cup \overline{\sigma_{\text{glob\_clos}}})$ then sc is reasonable up to $n-1$ steps.
- If $w = ((p, \_), b, e, \_)$ and $p \in \mathit{readAllowed}$ and $n > 0$, then $ms(a)$ is reasonable up to $n-1$ steps for all $a \in ([b, e] \setminus T_A)$
- If $w = \text{stack-ptr}(p, b, e, \_)$ and $p \in \mathit{readAllowed}$ and $n > 0$, then $ms_{stk}(a)$ is reasonable up to $n-1$ steps for all $a \in [b, e]$

♦

According to the definition, sets of seals that contain return or closure seals are not reasonable. StkTokens rely on return seals to be unique in order to work, but it does not rely on closure seals. However, it defeats the purpose of the closure seals if they are leaked to malicious code as it allows the malicious code to fabricate data capabilities for the code capabilities or code capabilities for the data capabilities which effectually allows malicious code to unseal the data capability. Further, it makes reasoning about leakage of return seals difficult because the closures have to be well-behaved no matter what data they are executed with.

Whether a capability is reasonable depends on what it gives access to. This is why the word reasonability is defined relative to the memory and stack. For instance, if a read capability gives access to a piece of memory that contains a set of seals with a return seal, then it would not be reasonable. This is why stack pointers and memory capabilities with read permission must only give access to memory with reasonable words. The definition is cyclic, but the step-index ensures that it is well-founded.

Reasonable pc and configuration

In order to define desired behaviour, we need to specify what may happen on a machine during execution. An execution steps between executable configurations, so we need to define when a configuration is reasonable. While the step relation is defined over configurations, it is the pc that decides what instruction is executed. Therefore, it only seems natural to also define when a pc is reasonable. Definition 3.4.3 roughly says that a pc is reasonable when you can plug it into a configuration where all the words in the register file are reasonable and the result is a reasonable configuration.

Definition 3.4.3 (Reasonable pc)**.** We say that an executable capability

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

behaves reasonably up to n steps if for any Φ such that

- $\Phi.reg(\text{pc}) = c$
- $\Phi.reg(r)$ is reasonable up to n steps in memory $\Phi.mem$ and free stack $\Phi.ms_{stk}$ for all $r \neq \text{pc}$
- $\Phi.mem$, $\Phi.ms_{stk}$ and $\Phi.stk$ are all disjoint

We have that Φ is reasonable up to n steps. ♦

With Definition 3.4.2 and 3.4.3 in place, we are all set to define when a configuration is reasonable.

Definition 3.4.4 (Reasonable configuration)**.** We say that an execution configuration Φ is reasonable up to n steps with $(T_A, \text{stk\_base}, \overline{\sigma_{\text{glob\_ret}}}, \overline{\sigma_{\text{glob\_clos}}})$ iff for $n' \leq n$:

1. *Guarantee stack base address before call...*
   If Φ points to $\texttt{call}^{off_{\text{pc}}, off_{\sigma}}\ r_1\ r_2$ in T_A for some r_1 and r_2, then all of the following hold:
   - $\Phi(r_{stk}) = \text{stack-ptr}(\_, \text{stk\_base}, \_, \_)$
   - $r_1 \neq \text{r}_{\text{t1}}$
   - $n' = 0$ or $\Phi(\text{pc}) + \text{call\_len}$ behaves reasonably up to $n' - 1$ steps (Definition 3.4.3)
2. *Use return seals only for calls, use closure seals appropriately...*
   If Φ points to $\texttt{cseal}\ r_1\ r_2$ in T_A and $\Phi(r_2) = \text{seal}(\sigma_b, \sigma_e, \sigma)$, then one of the following holds:
   - Φ is inside $\texttt{call}^{off_{\text{pc}}, off_{\sigma}}\ r_1'\ r_2'$ and $\sigma \in \overline{\sigma_{\text{glob\_ret}}}$
   - $\sigma \in \overline{\sigma_{\text{glob\_clos}}}$ and one of the following holds:
     - *executable*$(\Phi(r_1))$ and $n' = 0$ or $\Phi(r_1)$ behaves reasonably up to $n' - 1$ steps (Definition 3.4.3).
     - *nonExec*$(\Phi(r_1))$ and $n' = 0$ or $\Phi(r_1)$ is reasonable up to $n' - 1$ steps in memory $\Phi.ms$ and free stack $\Phi.ms_{stk}$ (Definition 3.4.2).
3. *Don't store private stuff...*
   If Φ points to $\texttt{store}\ r_1\ r_2$ in T_A, then $n' = 0$ or $\Phi.reg(r_2)$ is reasonable in memory $\Phi.mem$ up to $n' - 1$ steps.
4. *Don't leak private stuff...*
   If $\Phi \rightarrow^{T_A, \text{stk\_base}} \Phi'$, then one of the following holds:
   - All of the following hold:
     - $\Phi'.reg(\text{pc}) = ((p, l), b, e, a')$ and $\Phi.reg(\text{pc}) = ((p, l), b, e, a)$
     - Φ does not point to $\texttt{xjmp}\ r_1\ r_2$ for some r_1 and r_2

  - Φ does not point to $\texttt{call}^{off_{\text{pc}},off_{\sigma}}\ r_1\ r_2$ for some r_1 and r_2, off_{pc}, off_{σ}
  - $n' = 0$ or Φ' is reasonable up to $n' - 1$ steps
- All of the following hold:
  - Φ points to $\texttt{call}^{off_{\text{pc}},off_{\sigma}}\ r_1\ r_2$ for some r_1 and r_2
  - $n' = 0$ or $\Phi.reg(r)$ is reasonable in memory $\Phi.mem$ and free stack $\Phi.ms_{stk}$ up to $n' - 1$ steps for all $r \neq \text{pc}$
- All of the following hold:
  - Φ points to $\texttt{xjmp}\ r_1\ r_2$ for some r_1 and r_2
  - $n' = 0$ or $\Phi.reg(r)$ is reasonable in memory $\Phi.mem$ and free stack $\Phi.ms_{stk}$ up to $n' - 1$ steps for all $r \neq \text{pc}$

♦

The four items in Definition 3.4.4 correspond to the four informal items from the introduction of this section.

Item 3 and Item 4 make sure that return seals are not leaked. Item 3 says that only reasonable words can be stored to memory. This means that sets of return seals or execute capabilities cannot be stored to memory by a reasonable component. Intuitively, a well-formed and reasonable component has its seals available in the code memory, so it can always retrieve them from the code memory. In other words, it is not necessary to store them in the data memory. Further, by making sure that seals are not stored to memory, we can allow capabilities for data memory to be passed away if there is a need for that (for instance to have a shared buffer). Item 4 makes sure that seals are not leaked when transferring control to another component (i.e. on security boundary crossings). With the component setup, there are two ways to transfer control: xjmp and call. In both cases, we require that all of the argument registers contain reasonable words. An execution configuration may need to do other operations than calling other code, and seals should no be leaked at any point during execution. For this reason, Item 4 also says that if the next step is not a call and a jump, then the next execution configuration should also be reasonable.

Item 1 makes sure that the stack has the correct base before a call. In order to not have to reason about unreasonably generated code, we also add the requirement $r_1 \neq \text{r}_{\text{t1}}$ before calls. If we allowed $r_1 = \text{r}_{\text{t1}}$, then the call would be sure to fail as the first instruction of a call moves 42 to r_{t1}. Finally, this promises that the code after the call will behave reasonably.

A well-formed component makes sure that a return seal is uniquely available to every call. This is, however, not sufficient as it does not ensure that other parts of a program don't use the return seals. We do not want to specify what non-call code should look like, so we just require it to not use the call seals. This is what Item 2 ensures. It says that if the configuration points to a

seal-instruction, then either the instruction is part of a call and uses a return seal or the instruction seals part of a closure and uses a closure seal. In order to construct a closure, one must seal a code capability and a data capability. If this is an executable capability, then it should be reasonable as a pc. On the other hand, if it is not an executable capability, then this must be the data part of the pair, so it should just be a reasonable word. Definition 3.4.4 is cyclic through Definition 3.4.3, so both definitions are step-indexed to break the cycle. If

Reasonable component

A reasonable component has the informal behavioural properties from the introduction. Reasonability is captured by the previous definitions. These definitions are lifted to components by the following definition.

Definition 3.4.5 (Reasonable component)**.** We say that a component

$$(ms_{\text{code}}, ms_{\text{data}}, \overline{import}, \overline{export}, \overline{\sigma_{\text{ret}}}, \overline{\sigma_{\text{clos}}}, A_{\text{linear}})$$

is reasonable if the following hold: For all $(s \mapsto \text{sealed}(\sigma, sc)) \in \overline{c_{\text{export}}}$, with $executable(sc)$, we have that sc behaves reasonably up to any number of steps n.

We say that a component $(comp_0, c_{\text{main},c}, c_{\text{main},d})$ is reasonable if $comp_0$ is reasonable. ♦

In our result, we assume that adversarial components are well-formed, but not necessarily reasonable. The well-formedness assumption ensures that the trusted component can rely on basic security guarantees provided by the capability machine. For instance, if we did not require linearity to be respected initially, then adversarial code could start with an alias for the stack capability. The adversary is not assumed to be reasonable as we do not expect them to obey the calling convention in any way. Can adversarial code call into trusted components? The answer to that question is yes but not with LSE and WBCF guarantees. Formally, adversarial code can contain the instructions that constitute a call. However, for untrusted code, oLCM will not execute those instructions as a "native call" but execute the individual instructions separately. The callee then executes in the same stack frame as the caller, so WBCF and LSE do not follow (for that call).

We will assume trusted components, for which WBCF and LSE are guaranteed, to be both well-formed and reasonable.

3.4.4 Full Abstraction

All that is left before we state the full-abstraction theorem is to define how components are combined with contexts and executed, so that we can define contextual equivalence.

$$\frac{\begin{array}{c} c_{\text{main},c}, c_{\text{main},d} = \text{sealed}(\sigma, c'_{\text{main},c}, c'_{\text{main},d}) \qquad nonExec(c'_{\text{main},d}) \\ reg(\text{pc}, \text{r}_{\text{data}}) = c'_{\text{main},c}, c'_{\text{main},d} \qquad reg(\text{r}_{\text{stk}}) = \text{stack-ptr}(\text{RW}, b_{stk}, e_{stk}, e_{stk}) \\ reg(\text{r}_{\text{stk}}) = ((\text{RW}, \text{linear}), b_{stk}, e_{stk}, e_{stk}) \qquad reg(\text{RegName} \setminus \{\text{pc}, \text{r}_{\text{data}}, \text{r}_{\text{stk}}\}) = 0 \\ range(ms_{stk}) = \{0\} \qquad mem = ms_{\text{code}} \uplus ms_{\text{data}} \uplus ms_{stk} \\ [b_{stk}, e_{stk}] = \text{dom}(ms_{stk}) \# (\text{dom}(ms_{\text{code}}) \cup \text{dom}(ms_{\text{data}})) \qquad \overline{import} = \emptyset \end{array}}{((ms_{\text{code}}, ms_{\text{data}}, \overline{import}, \overline{export}, \overline{\sigma_{\text{ret}}}, \overline{\sigma_{\text{clos}}}, A_{\text{linear}}), c_{\text{main},c}, c_{\text{main},d}) \rightsquigarrow (mem, reg, \emptyset, ms_{stk})}$$

Figure 3.15: The judgement $prog \rightsquigarrow \Phi$, which defines the initial execution configuration Φ for executing a program $prog$.

Given a program $comp$, the judgement $comp \rightsquigarrow \Phi$ in Figure 3.15 defines an initial execution configuration that can be executed. It works almost the same on LCM (conditions in red) and oLCM (conditions in blue). On both machines a stack containing all zeroes is added, as part of the regular memory on LCM and as the free stack on oLCM. On oLCM, the initial stack is empty as no calls have been made. The component needs access to the stack, so a stack pointer is added to the register file in r_{stk}. On LCM this is just a linear read-write capability, but on oLCM it is the representation of a stack pointer. The entry point of the program is specified by main, so the two capabilities are unsealed (they must have the same seal) and placed in the pc and r_{data} registers. Other registers are set to zero.

Contextual equivalence roughly says that two components behave the same no matter what context we plug them into.

Definition 3.4.6 (Plugging a component into a context)**.** When $comp'$ is a context for component $comp$ and $comp' \bowtie comp \rightsquigarrow \Phi$, then we write $comp'[comp]$ for the execution configuration Φ. ♦

Definition 3.4.7 (LCM and oLCM contextual equivalence)**.**

On oLCM , we define that $comp_1 \approx_{\text{ctx}} comp_2$ iff

$$\forall \mathscr{C}. \emptyset \vdash \mathscr{C} \Rightarrow \mathscr{C}[comp_1]\Downarrow_{\_}^{T_{A,1},\text{stk\_base}_1} \Leftrightarrow \mathscr{C}[comp_2]\Downarrow_{\_}^{T_{A,2},\text{stk\_base}_2}$$

with $T_{A,i} = \text{dom}(comp_i.ms_{\text{code}})$.

On LCM , we define that $comp_1 \approx_{\text{ctx}} comp_2$ iff

$$\forall \mathscr{C}. \emptyset \vdash \mathscr{C} \Rightarrow \mathscr{C}[comp_1]\Downarrow_{\_} \Leftrightarrow \mathscr{C}[comp_2]\Downarrow_{\_}$$

where $\Phi\Downarrow_i^{T_A,\text{stk\_base}}$ iff $\Phi \rightarrow_i^{T_A,\text{stk\_base}}$ halted and $\Phi\Downarrow_{\_}^{T_A,\text{stk\_base}} \stackrel{def}{=} \exists i. \Downarrow_i^{T_A,\text{stk\_base}}$ ♦

With the above defined, we are almost ready to state our full-abstraction, and all that remains is the compiler we claim to be fully-abstract. We only care about the well-formed components, and they sport none of the new syntactic constructs oLCM adds to LCM. This means that the compilation from oLCM components to LCM components is simply the identity function.

Theorem 3.4.8. *For reasonable, well-formed components $comp_1$ and $comp_2$, we have*

$$comp_1 \approx_{\text{ctx}} comp_2 \quad \Leftrightarrow \quad comp_1 \approx_{\text{ctx}} comp_2 \qquad \Diamond$$

Readers unfamiliar with fully-abstract compilation may wonder why Theorem 3.4.8 proves that StkTokens guarantees LSE and WBCF. Generally speaking, behavioral equivalences are preserved and reflected by fully-abstract compilers. This means that any property the source language has must somehow be there after compilation whether or not it is a property of the target language. If the source language has a property that the target language doesn't have, then a compiled source program must use the available target language features to emulate the source language property in a way that it behaviorally matches exactly. In our case, LSE and WBCF was built into the semantics of oLCM, but they are not properties of LCM. In order to enforce these properties, components on LCM use StkTokens. Theorem 3.4.8 proves that StkTokens enforces these properties in a way that behaviorally matches oLCM which means that it enforces LSE and WBCF.

3.5 Proving full abstraction

To prove Theorem 3.4.8, we will essentially show that trusted components in oLCM are related in a certain way to their embeddings in LCM, and that untrusted LCM components are similarly related to their embeddings in oLCM. We will then prove that these relations imply that the combined programs have the same observable behavior, i.e. one terminates if and only if the other does. The difficult part is to define when components are related. In the next section, we give an overview of the relation we define, and then we sketch the full-abstraction proof in Section 3.5.6.

In the previous version, we were forced to omit many aspects of the logical relation, and especially the way it deals with the linking model. This was unfortunate as substantial effort goes into this work, and there are insights that may be useful for similar work. Therefore the current version takes a bit longer to explain those aspects. Readers who are looking for a shorter summary with less detail may refer to the conference version of this paper for a presentation [86] with emphasis on the intuition and fewer details.

3.5.1 Kripke worlds

The relation between oLCM and LCM components is non-trivial: essentially, we will say that components are related if invoking them with related values produces related observable behavior. However, values are often only related under certain assumptions about the rest of the system. For example, the linear data part of a return capability should only be related to the corresponding oLCM capability if no other value in the system references the same inactive stack frame and it is sealed with a seal only used for return pointers to the same code location. To accommodate such conditional relatedness, we construct the relation as a step-indexed Kripke logical relation with recursive worlds.

Assumptions about the system that relatedness is predicated on are gathered in (Kripke) worlds. To a first approximation, a world is a semantic model of the memory. In its simplest form, it is a collection of invariants that the memory must satisfy. The invariants of a world can vary in complexity and expressiveness depending on the application. The possible contents of the memory also influences what the world looks like. For instance, the uniqueness of the linear capabilities on LCM and oLCM is modelled by the worlds. In order to relate LCM and oLCM, we model all the features of oLCM in the world which means we have to model:

- Three kinds of memory: heap, stack of local frames, and free stack
- Linearity
- Call stack
- Seals

The three kinds of memory are modelled by having three sub-worlds where each sub-world is its own little world in a traditional sense. Linearity is modelled by adding ownership to certain parts of the world that capabilities can take ownership of ensuring that they are the sole reference for that part of memory. Memory satisfaction (the relation that decides whether two memories are related in the world) models the call stack by ensuring that the memory is actually shaped like a stack. Finally, the seals are modelled by seal invariants that make sure that seals are only used on permitted sealables. In the following, we present the world and go into details about how each of the four features are modelled.

Triple world and regions

oLCMs memory is split in three: heap, free stack and encapsulated local stack frames. In order to model the three kinds of memory, we simply have

three sub-worlds. That is, our world is defined as a product:

$$\text{World} = \text{World}_{\text{heap}} \times \text{World}_{\text{call\_stack}} \times \text{World}_{\text{free\_stack}}$$

The sub-worlds are partial maps from names RegName (not to be confused with register names), modelled as natural numbers, to invariants. In order to define what this actually means, we need to be more precise about what we mean by invariants.

We call the invariants regions because, as we will see in Section 3.5.1, they turn out to not be invariant. A region describes a collection of related memory segments, so it is simply represented as a relation over memory segments (i.e. a relation in $\text{Rel}(\text{MemSeg} \times \text{MemSeg})$). Intuitively, we want to be able to say that two memory segments are related when their content is related. That is, for every address in the two memory segments, the words that reside there must be related. In Section 3.5.2, we define precisely what it means for words to be related, but for now we provide some intuition. If we have two integers, then they are related when they are equal. If we instead have two capabilities, then they should also be related if they, in some sense, are equal. But what does it mean for capabilities to be *equal*? Intuitively, it should mean that the capabilities give you the same authority, for instance, if you have two executable capabilities, they should observably perform the same computation. As a capability points to a piece of memory, the authority of the capability depends on the contents of that memory. In other words to say whether two capabilities are related, we must know the possible contents of the memory. The world expresses the possible memory contents, so our regions are world indexed, i.e.

$$\text{World}_{\text{heap}} = \text{RegName} \rightharpoonup (\text{World} \to \text{Rel}(\text{MemSeg} \times \text{MemSeg}))$$

At this point, we can see that we have constructed a recursive domain equation. If we inline $\text{World}_{\text{heap}}$ in World, then we have a circular equation with no solution because the self-reference happens in a negative position. Luckily, we can solve circular equations if we move to a different domain. For now, we will ignore the problem and return to the issue in Section 3.5.1.

For the sake of readability, we introduce the following notation

$$\begin{aligned} W.\text{heap} &= \pi_1(W) \\ W.\text{call\_stk} &= \pi_2(W) \\ W.\text{free\_stk} &= \pi_3(W) \end{aligned}$$

Linearity

The linear capabilities of oLCM and LCM guarantee that they have the sole authority over the memory they reference[6]. In order to model this unique-

[6]Under the assumption that the system was initialised with unique linear capabilties.

ness, we need to keep track of which parts of memory are uniquely referenced and make sure that only one linear capability references the unique parts. We use the world to keep track of what parts of memory must be uniquely referenced by having two kinds of regions: shared and spatial. If a memory segment is governed by a shared region, then normal capabilities may reference it. On the other hand, if a memory segment is governed by a spatial region, then only a linear capability may reference it.

We cannot let multiple linear capabilities reference the same memory. The spatial region represent ownership of a part of memory which ensures that a linear capability is not aliased. Even though spatial regions gives the right to reference part of memory, we still need the world to specify the remainder of the memory that may be referenced by linear capabilities. To this end, we have shadow regions. A shadow region is a shadow copy of a spatial region in the sense that it specifies part of memory, but it does not give the right to reference that part of memory. This does not mean that the memory is necessarily not referenced. A compatible world may claim a shadow region as spatial which allows the other world to reference the memory (we return to the notion of compatible worlds w.r.t. ownership in Section 3.5.1). Specifically, we add tags spatial and shadow to the spatial regions:

$$\mathrm{Region}_{\mathrm{spatial}} = \begin{cases} \{\mathrm{spatial}\} \times (\mathrm{World} \to \mathrm{Rel}(\mathrm{MemSeg} \times \mathrm{MemSeg})) \cup \\ \{\mathrm{shadow}\} \times (\mathrm{World} \to \mathrm{Rel}(\mathrm{MemSeg} \times \mathrm{MemSeg})) \end{cases}$$

For readability, we also add a tag shared to the shared regions:

$$\mathrm{Region}_{\mathrm{shared}} = \{\mathrm{shared}\} \times (\mathrm{World} \to \mathrm{Rel}(\mathrm{MemSeg} \times \mathrm{MemSeg}))$$

We will extend the regions further in Sections 3.5.1 and 3.5.1, but for now we continue the definitions of the three sub-worlds. The sub-world $\mathrm{World}_{\mathrm{heap}}$ specifies the heap memory which can be referenced by both linear and normal capabilities, so it should contain both shared and spatial regions. For this reason, it is defined as

$$\mathrm{World}_{\mathrm{heap}} = \mathrm{RegName} \rightharpoonup (\mathrm{Region}_{\mathrm{shared}} \cup \mathrm{Region}_{\mathrm{spatial}})$$

oLCM internalizes the StkTokens stack, so it can only be referenced by linear capabilities which means that the two stack regions should only have spatial regions. For instance the $\mathrm{World}_{\mathrm{free\_stack}}$ is defined as

$$\mathrm{World}_{\mathrm{free\_stack}} = \mathrm{RegName} \rightharpoonup \mathrm{Region}_{\mathrm{spatial}}$$

$\mathrm{World}_{\mathrm{call\_stack}}$ can also only be referenced by linear capabilities, so it should also only have spatial regions. $\mathrm{World}_{\mathrm{call\_stack}}$ not only models the memory contents of the local stack frames, it also models the call stack that consists of the stack frames. Conceptually, this means that each of the stack frames is connected with a return point in some code. In a traditional C calling

convention, the code return point would even be stored in the stack frame. However, with StkTokens a stack frame does not contain any information about the corresponding code return point. Instead, the stack frame and code return point is connected by the capabilities that reference them as they are sealed with the same seal and together then constitute the sealed return pair. In order to ensure that each call actually returns to the correct point of the code, we must still include the address of the return point in our model. To this end, each shared region in $\text{World}_{\text{call\_stack}}$ must be paired with a return address:

$$\text{World}_{\text{call\_stack}} = \text{RegName} \rightharpoonup (\text{Region}_{\text{spatial}} \times \text{Addr})$$

The spatial regions add the necessary bookkeeping to the worlds to model linear capabilities. The logical relation presented in Section 3.5.2 uses this bookkeeping to ensure that linear capabilities *uniquely* references part of memory.

Given a world, we want to be able to express that a capability, that is otherwise valid with respect to the world, is not linear or indirectly depends on a linear capability. This is expressed by stripping the world of all its ownership which corresponds to replacing all spatial regions with shadow regions and then requiring that the capability is valid w.r.t. this new world. We refer to this as the shared part of the world and define the function *sharedPart* which turns all spatial regions into shadow copies.

Definition 3.5.1 (The shared part of a world)**.** For any world W, we define

$$\mathit{sharedPart}(W) \stackrel{\mathit{def}}{=} (\mathit{sharedPart}(W.\text{heap}), \mathit{sharedPart}(W.\text{call\_stk}), \mathit{sharedPart}(W.\text{free\_stk}))$$

$$\mathit{sharedPart}(W_{\mathit{heap}}) \stackrel{\mathit{def}}{=} \lambda r. \begin{cases} (\text{shadow}, H) & \text{if } W_{\mathit{heap}}(r) = (\text{spatial}, H) \\ W_{\mathit{heap}}(r) & \text{otherwise} \end{cases}$$

$$\mathit{sharedPart}(W_{\mathit{call\_stk}}) \stackrel{\mathit{def}}{=} \lambda r. \begin{cases} ((\text{shadow}, H), \mathit{opc}) & \text{if } W_{\mathit{call\_stk}}(r) = ((\text{spatial}, H), \mathit{opc}) \\ W_{\mathit{call\_stk}}(r) & \text{otherwise} \end{cases}$$

$$\mathit{sharedPart}(W_{\mathit{free}}) \stackrel{\mathit{def}}{=} \lambda r. \begin{cases} (\text{shadow}, \mathit{Hs}) & \text{if } W_{\mathit{free}}(r) = (\text{spatial}, \mathit{Hs}) \\ W_{\mathit{free}}(r) & \text{otherwise} \end{cases}$$

♦

Seals

So far, the world represents a collection of assumptions on the memory contents that value correctness may depend on. However, value correctness may depend on other assumptions. Specifically, StkTokens has certain assumption on the seals used for return capabilities and closures. For instance, a return seal must only be used to seal the return pointer of one specific return point. Therefore, in addition to a relation on memory segments, some

regions also carry a *seal interpretation function* that relates the sealables that may be sealed with a given seal.

$$\text{Seal} \rightharpoonup \text{World} \to \text{Rel}(\text{Sealables} \times \text{Sealables})$$

In StkTokens, once a seal has been used for a specific purpose (e.g. for sealing return capability pairs for a specific call site), it can never be reused for a different purpose. This is because there may still be copies of return capabilities out there, signed with the seal. This situation is similar to the situation for non-linear memory capabilities, so we only allow shared regions to carry seal interpretation functions, as we will see that those regions can never be revoked in future worlds.

$$\begin{aligned}\text{Region}_{\text{shared}} = \{\text{shared}\} \times (\text{World} \to \text{Rel}(\text{MemSeg} \times \text{MemSeg})) \times \\ (\text{Seal} \rightharpoonup \text{World} \to \text{Rel}(\text{Sealables} \times \text{Sealables}))\end{aligned}$$

We also refer to the seal interpretation function as the seal invariant, and we will refer to the memory relation as the memory invariant or just invariant when it is unambiguous.

Future worlds and revocation

Very often, relatedness of two capabilities does not change if extra assumptions in the system are added. For example, two related capabilities remain related when an extra invariant is added on unrelated memory, or when a stack frame that it does not reference is dropped. In Kripke logical relations, this kind of assumption changes, that do not invalidate the relatedness of values, are modelled by the future world relation $\sqsupseteq$. The future world relation can also be thought of as the model of changes in memory over time.

The memory invariants in the world should be defined, so they are monotone with respect to the future world relation. This means that the system assumptions capabilities rely on for safety are defined such that capabilities remain safe during execution. Specifically, we require the world-indexed memory invariants to be monotone in the world. This means that a pair of memory segments that are related now will stay related in future worlds. This changes the spatial regions as follows:

$$\text{Region}_{\text{spatial}} = \begin{cases} \{\text{shadow}\} \times (\text{World} \xrightarrow{mon} \text{Rel}(\text{MemSeg} \times \text{MemSeg})) \cup \\ \{\text{spatial}\} \times (\text{World} \xrightarrow{mon} \text{Rel}(\text{MemSeg} \times \text{MemSeg})) \cup \\ \{\text{revoked}\} \end{cases}$$

We make a similar change to $\text{Region}_{\text{shared}}$. The seal invariants are monotone as well.

Kripke future world relations must be defined such that future worlds are extensions of the past world. In other words, future world relations usually

only allow for adding fresh assumptions on fresh memory or protocol steps that the system was designed to support from the start. However in our setting, we sometimes allow dropping assumptions, namely when linearity tells us that no value in the system depends on assumption anymore. Specifically, if we have the only linear capability for a piece of memory, then we can be sure that there are no other capabilities for the same memory which makes it safe to repurpose the memory. We mark dropped regions as revoked in the world following Ahmed [14] and Thamsborg and Birkedal [91] which for all intends and purposes corresponds to actually dropping the region.

We add a revoked tag to the spatial regions $\text{Region}_{\text{spatial}}$ which results in

$$\text{Region}_{\text{spatial}} = \begin{cases} \{\text{shadow}\} \times (\text{World} \to \text{Rel}(\text{MemSeg} \times \text{MemSeg})) \cup \\ \{\text{spatial}\} \times (\text{World} \to \text{Rel}(\text{MemSeg} \times \text{MemSeg})) \cup \\ \{\text{revoked}\} \end{cases}$$

The spatial-region signifies that a linear capability *may* depend on it which means that it cannot be revoked. On the other hand, no capability can depend on a shadow-region, so it can be safely revoked. This may be confusing as a shadow-region does not mean that you "have" the capability, but we just explained that having a linear capability means that you can repurpose the memory it points to. As we will see later, the logical relation splits worlds with respect to their ownership, so each capability can get its own world with unique ownership to depend on. A repurposed linear capability gets an entirely new world to depend on. The new world constructed for the repurposed linear capability would have a revoked region in place of the spatial-region, but the new world would not be a future world of the world the linear capability depended on before. After all, a future world has to respect old invariants, but a repurposed capability needs to depend on new invariants. While the repurposed capability gets a new world to depend on, all other capabilities in the system depend on the same invariants, so they must be valid with respect to worlds that respect the old invariants, i.e. future worlds. Further, the future worlds must be compatible (agree on everything but ownership) with the new world. This means that all the shadow-regions must be replaced with revoked regions which is why shadow regions can be turned into revoked regions.

We define the future world relation in terms of a future region relation which is displayed in Figure 3.16. Apart from being revoked, a region can stay the same, or a shadow region can become spatial which models the affinity of the linear capabilities. Specifically, the linear capabilities in LCM and oLCM can be dropped by overwriting them in registers or memory which makes the linear capabilities affine. A linear capability is safe with respect to a world with some spatial regions. However, this world, and thus the spatial regions, is no longer needed if the linear capability is dropped, so other worlds are free to claim the spatial region in place of their shadow region. The above repurposing discussion is exemplified in Figure 3.17. The

$$\frac{r \in \text{Region}_{\text{spatial}} \cup \text{Region}_{\text{shared}}}{r \sqsupseteq r} \qquad \frac{}{\text{revoked} \sqsupseteq (\text{shadow}, \_)}$$

$$\frac{}{(\text{spatial}, H) \sqsupseteq (\text{shadow}, H)}$$

Figure 3.16: Future region relation.

Figure displays two capabilities c_{normal} and c_{linear} that are normal and linear, respectively. The linear capability is valid with respect to W_1 which has the necessary spatial region, and the normal capability is valid with respect to W_2 which has a shared region. The two worlds are compatible as W_2 has a shadow region that matches the spatial region of W_1. When the linear capability is repurposed, it must be reflected by the worlds W_1' and W_2'. In both new worlds, the r_2 region is replaced with a revoked region. W_1' has a new spatial region at r_3, and W_2' gets a matching shadow region. The new world for the linear capability, W_1', is not a future world of W_1 as it replaces a spatial region with a revoked region which is not allowed by the future region relation. The new world for the normal capability, W_2', is a future world of W_2 as the future region relation allows shadow regions to be revoked. The normal capability c_{normal} remains valid in W_2' as the shared region it depends on is monotone with respect to the future world relation.

With the future region relation in place, we can define the future world relation as follows: For worlds W and W',

$$W' \sqsupseteq W \text{ iff } \begin{cases} \text{for } i \in \{\text{heap}, \text{free\_stk}, \text{call\_stk}\} \\ \quad \exists m_i : \text{RegionName} \to \text{RegionName}, \text{ injective.} \\ \quad\quad \text{dom}(W'.i) \supseteq \text{dom}(m_i(W.i)) \wedge \forall r \in \text{dom}(W.i).\, W'.i(m_i(r)) \sqsupseteq W.i(r) \end{cases}$$

The relation says that each of the three worlds must be an extension of the past world and each of the existing regions must have a future region. Note that the future world relation has a mapping function m_i which allows us to change the naming of regions in future worlds. The definition is a generalization of the standard definition where m_i would be the identity[7].

Solving the recursive domain equation

In the previous sections, we sketched what our worlds should be. However, the worlds we want constitute a self-referential domain equation for which no solution exists in set and domain theory. Therefore, we need to move to

[7] In Skorstengaard et al. [83] the future region relation and the reasoning about the awkward example could have been simplified with this future world relation.

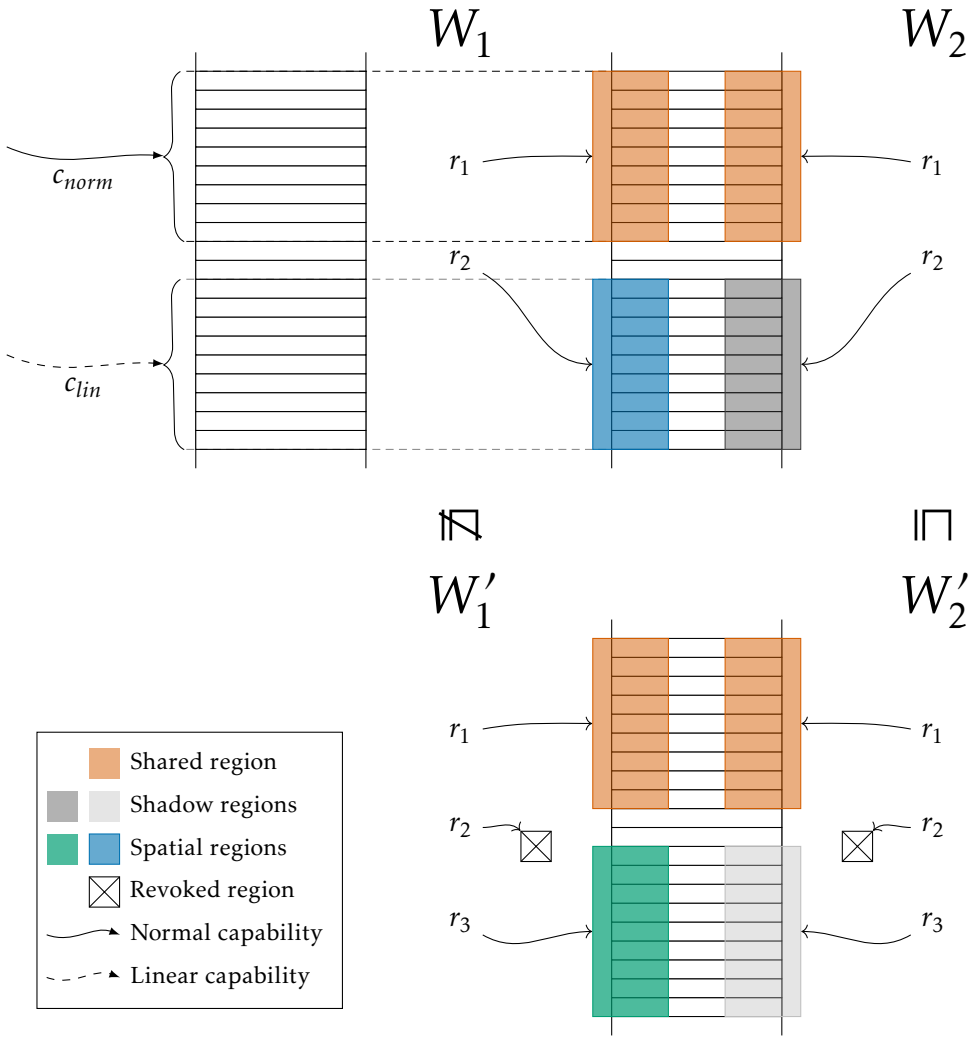


Figure 3.17: Example what happens to worlds when a linear capability c_{lin} is repurposed. The linear capability is originally valid with respect to W_1. In W_1' there is a new spatial region that the linear capability is valid with respect to (the different coloured regions signify different invariants). W_1' still has r_2, but now it points to a revoked region. This means that W_1' is not a future world of W_1. The normal capability c_{norm} is originally valid with respect to W_2 and should stay valid with respect to the new world W_2'. The W_2' is a future world of W_2 as the shadow region in W_2 is replaced with a revoked region which is permitted by the future world relation.

a different domain with enough structure for a solution to exist for recursive equations. nSolutions to recursive domain equations can be found using standard techniques [15, 20, 78]. Essentially, we move to a setting where instead of sets we have c.o.f.e.'s (complete ordered families of equivalences), instead of functions we have non-expansive functions, and instead of relations we have downwards-closed relations. A c.o.f.e. can be thought of as a set with added structure. Specifically, c.o.f.e.'s have a step-index that must satisfy a set of conditions which provide sufficient structure for a solution to exist for the recursive domain equation. In this section, we present some of the technical details on solving our world equation as well as the final world equation (Theorem3.5.14). This section does not provide further intuition about the worlds and can be skipped.

Recursive domain equations are so common and the solution so standard that frameworks implement them lifting the technical burden of constructing the world by hand. The technique we use to solve our recursive domain equation is essentially the same used internally in the program logic framework Iris [51, 52, 56] to construct worlds. We believe the worlds presented here could be developed in Iris. Had we done the development in Iris Coq, then we would have had the world tool support of Iris and the rigorousness of Coq which, in hindsight, would have been very beneficial.

In the following, we present complete ordered family of equivalences [35]. We loosely follow the presentation in Birkedal and Bizjak [18] and introduce the necessary definitions needed for our particular world.

Definition 3.5.2 (Ordered family of equivalences (o.f.e.))**.** An *ordered family of equivalences* is a pair $\left(X, \left(\overset{n}{=}\right)_{n=0}^{\infty}\right)$ where X is a set and $\overset{n}{=}$ is an equivalence relation for each n. The pair must satisfy the following properties:

1. $\overset{0}{=}$ is the total relation.
2. For all $n \in \mathbb{N}$, $\overset{n}{=} \supseteq \overset{n+1}{=}$.
3. For all $x, y \in X$, if for all $n \in \mathbb{N}$ $x \overset{n}{=} y$, then $x = y$.

♦

An o.f.e. can be thought of as a set with extra structure. Intuitively, the index on the family of equivalences is a measure of precision. The larger the index is, the more refined the equivalences are which means that they can distinguish more elements from one another. On the other hand as the index decreases, the equivalences become increasingly imprecise and unable to distinguish certain elements. At index 0, all precision is lost and the equivalence cannot distinguish anything.

Definition 3.5.3 (Complete ordered family of equivalences (c.o.f.e.))**.** A *complete ordered family of equivalences* is an ordered family of equivalences $\left(X, \left(\overset{n}{=}\right)_{n=0}^{\infty}\right)$ such that every Cauchy sequence in X has a limit in X.

Let $\left(X, \left(\overset{n}{=}\right)_{n=0}^{\infty}\right)$ be an o.f.e. and $(x_n)_{n=0}^{\infty}$ be a sequence of elements in X. Then $(x_n)_{n=0}^{\infty}$ is a *Cauchy sequence* if

$$\forall k \in \mathbb{N}, \exists j \in \mathbb{N}, \forall n \geq j, x_j \overset{k}{=} x_n$$

or in words, the elements of the chain get arbitrarily close.

An element $x \in X$ is the *limit* of the sequence $(x_n)_{n=0}^{\infty}$ if

$$\forall k \in \mathbb{N}, \exists j \in \mathbb{N}, \forall n \geq j, x \overset{k}{=} x_n.$$

♦

In order to move from one c.o.f.e. to another, we define functions, just like we would when working with sets. Unlike functions between sets, functions between c.o.f.e.'s must preserve the c.o.f.e. structure which informally means that the measure of precision must be retained. This specifically means that functions between c.o.f.e.'s must be *non-expansive* which basically means that they preserve n-equalities. Functions may not only retain equivalences but make it more precise. Such a function is said to be *contractive*.

Definition 3.5.4. Let $\left(X, \left(\overset{n}{=}_X\right)_{n=0}^{\infty}\right)$ and $\left(Y, \left(\overset{n}{=}_Y\right)_{n=0}^{\infty}\right)$ be two ordered families of equivalences and f a function from the set X to the set Y. The function f is

non-expansive when for all $x, x' \in X$, and all $n \in \mathbb{N}$,

$$x \overset{n}{=}_X x' \implies f(x) \overset{n}{=}_Y f(x')$$

contractive when for any $x, x' \in X$, and any $n \in \mathbb{N}$,

$$x \overset{n}{=}_X x' \implies f(x) \overset{n+1}{=}_Y f(x')$$ ♦

The step-indexing provided by c.o.f.e.'s give enough structure to solve the recursive domain equation. There are a number of standard c.o.f.e. constructions for countably sets, products, sums, and functions. A set can be lifted to c.o.f.e. by having the normal equality for all positive indices and the total relation for $\overset{0}{=}$. The other standard constructions are defined naturally and compose new c.o.f.e.'s from existing c.o.f.e.'s.

In some sense, the standard c.o.f.e. constructions maintain the precision of the underlying c.o.f.e.'s. This, however, does not utilise the c.o.f.e. structure in a way that allows us to construct a solution for a recursive domain equation. Intuitively, the step of the c.o.f.e. needs to go down before the recursive occurrence. To this end, we have the *later* ($\blacktriangleright$) c.o.f.e.:

Lemma 3.5.5 (▶ c.o.f.e.)**.** *Given a c.o.f.e* $\left(X, \left(\overset{n}{=}_X\right)_{n=0}^{\infty}\right)$ *define the later c.o.f.e. as*

$$\blacktriangleright\left(X, \left(\overset{n}{=}_X\right)_{n=0}^{\infty}\right) = \left(X, \left(\overset{n}{=}_{\blacktriangleright}\right)_{n=0}^{\infty}\right)$$

where

$$x \overset{n}{=}_{\blacktriangleright} x' \text{ iff } \begin{cases} n = 0 \text{ or} \\ x \overset{n-1}{=}_X x' \end{cases} \qquad \lozenge$$

The later c.o.f.e. should be placed before the self-reference to make the index decrease. Intuitively, this gives something to recurse on in order to find a solution for the domain equation. If we limit ourselves to a handful of standard constructions (product, sum, arrow, later, and partial functions from the natural numbers) to construct self-referential equations and make sure to guard every occurrence of the self reference with a later, then the equation has a solution.

Unfortunately, the standard c.o.f.e. constructions do not suffice for the worlds we want to define. The worlds we sketched have additional structure in terms of a future world relation. The future world relation is captured by adding a preorder to c.o.f.e.'s:

Definition 3.5.6 (Preordered c.o.f.e.)**.** A preordered c.o.f.e. is a c.o.f.e. equipped with a preorder on X, $\left(X, \left(\overset{n}{=}\right)_{n=0}^{\infty}, \sqsupseteq\right)$.

- The ordering preserves limits. That is, for Cauchy sequences $\{a_n\}_n$ and $\{b_n\}_n$ in X if $\{a_n\}_n \sqsupseteq \{b_n\}_n$, then $\lim\{a_n\}_n \sqsupseteq \lim\{b_n\}_n$.

⧫

The preorder must respect the existing structure of the c.o.f.e.'s which means that it must preserve limits. There are also natural constructions for preordered c.o.f.e.'s that rely on the underlying preorder. We will need the standard construction for later, product, and sum (defined in Appendix 3.A.4).

Sometimes we will refer to c.o.f.e.'s and preordered c.o.f.e.'s simply by their underlying set.

In the following, we will define a number of preordered c.o.f.e.'s that can be combined to get our world equation. First, we define the c.o.f.e. for the memory invariants (we do not need a preorder here). As a reminder, we sketched the memory invariants to as follows

$$\text{World} \xrightarrow{mon} \text{Rel}(\text{MemSeg} \times \text{MemSeg})$$

To construct this, we need to define a c.o.f.e. for the relation and a c.o.f.e. for the monotone and non-expansive functions. We do not need a construction for the world as it will be given as the solution to the recursive equation. Generally, we can lift a set of relations to a c.o.f.e. by adding a downwards closed index to it. This means that if a pair is in the relation for a specific index, then it is also in the relation for all smaller indices.

Definition 3.5.7 (Uniform relation)**.** Given a set $R = X \times Y$, we define $\mathrm{URel}(R) \subseteq \mathcal{P}(\mathbb{N} \times R)$ as

$$\mathrm{URel}(R) = \{A \in \mathcal{P}(\mathbb{N} \times R) \mid \forall n \in \mathbb{N}.\, \forall r \in R.\, (n, r) \in A \implies \forall m \leq n.\, (m, r) \in A\}$$

The uniform relation c.o.f.e. $\left(\mathrm{URel}(R), \left(\overset{n}{=}\right)_{n=0}^{\infty}\right)$ has $\mathrm{URel}(R)$ as the underlying set and the following family of equivalences:

$$A \overset{n}{=} B \text{ iff } \lfloor A \rfloor_n = \lfloor B \rfloor_n$$

where erasure is defined as

$$\lfloor A \rfloor_n \overset{def}{=} \{(k, a) \in A \mid k < n\}$$

to get a preordered c.o.f.e. we use subset as the preorder. ♦

The uniform relation construction uses the canonical family of equivalences which basically ignores everything from the step up and requires the rest to be equal. This means that everything is equal at step 0 as everything is left out.

Next, we define a c.o.f.e. of monotone, non-expansive functions

Definition 3.5.8 (The c.o.f.e. of monotone and non-expansive functions)**.** Given a preordered c.o.f.e. $\left(W, \left(\overset{n}{=}_W\right)_{n=0}^{\infty}, \sqsupseteq_W\right)$ and a preordered c.o.f.e. $\left(U, \left(\overset{n}{=}_U\right)_{n=0}^{\infty}, \sqsupseteq_U\right)$, define the c.o.f.e. $\left(W \xrightarrow{mon,\, ne} U, \left(\overset{n}{=}\right)_{n=0}^{\infty}\right)$ where

$$h \overset{n}{=} h' \text{ iff } \forall w \in \mathrm{dom}(h).\, h(w) \overset{n}{=}_U h'(w)$$

♦

In order to construct the memory invariant, we use Definition 3.5.8 with the world solution as the W c.o.f.e. and a uniform relation over pairs of memory segments (Definition 3.5.7) as the U c.o.f.e.

Next, we construct the seal invariant. As a reminder, we sketched the seal invariant as

$$\mathrm{Seal} \rightharpoonup \mathrm{World} \xrightarrow{mon,\, ne} \mathrm{URel}(\mathrm{Sealables} \times \mathrm{Sealables})$$

In order to define the c.o.f.e. for this, we can use Definitions 3.5.7 and 3.5.8. This leaves us with the task of lifting the set of seals to a c.o.f.e. and defining the c.o.f.e. partial functions. Generally, all countable sets can be lifted to a c.o.f.e. with a family of equivalences that has full precision for all indices except index 0.

Definition 3.5.9 (Sets to c.o.f.e.)**.** A countable set S can be lifted to a c.o.f.e. $\left(S, \left(\overset{n}{=}\right)_{n=0}^{\infty}\right)$ where

- $\overset{0}{=}$ is the total relation
- for $n > 0$ the relation $\overset{n}{=}$ is the normal equality on the set

♦

This definition can be used to lift the set of all seals Seal to a c.o.f.e.

The c.o.f.e. of partial, non-expansive functions is defined similarly to the c.o.f.e. of monotone non-expansive functions.

Definition 3.5.10 (The c.o.f.e. of partial, non-expansive functions)**.** Given a c.o.f.e. $\left(S, \left(\overset{n}{=}_S\right)_{n=0}^{\infty}\right)$ and a c.o.f.e. $\left(H, \left(\overset{n}{=}_H\right)_{n=0}^{\infty}\right)$, define the c.o.f.e. $\left(S \rightharpoonup H, \left(\overset{n}{=}\right)_{n=0}^{\infty}\right)$ where

$$h \overset{n}{=} h' \text{ iff } \mathrm{dom}(h) = \mathrm{dom}(h') \wedge \forall w \in \mathrm{dom}(h).\, h(w) \overset{n}{=}_H h'(w)$$

♦

The seal invariant is defined using Definition 3.5.9 to lift Sealables to a c.of.e. defining a c.o.f.e. similar to the one for memory invariants, and finally combining the two with Definition 3.5.10.

Next, we want to define the regions of the world. As mentioned, our regions have the added structure of the future region relation, so we need to define them as a preordered c.o.f.e. The future region was presented in Section 3.5.1. On shared regions, the relation is equality which makes it straight forward to define:

Definition 3.5.11 (Shared region preordered c.o.f.e.)**.** Given c.o.f.e.'s $\left(MemInv, \left(\overset{n}{=}_{MI}\right)_{n=0}^{\infty}\right)$ and $\left(SealInv, \left(\overset{n}{=}_{SI}\right)_{n=0}^{\infty}\right)$, define the preordered c.o.f.e. of shared regions as

$$\left(\{\text{shared}\} \times MemInv \times SealInv, \left(\overset{n}{=}\right)_{n=0}^{\infty}, =\right)$$

where the family of equivalences distributes to the underlying c.o.f.e.'s, i.e. given shared regions $(\text{shared}, H, H_\sigma), (\text{shared}, H', H'_\sigma) \in \{\text{shared}\} \times MemInv \times SealInv$,

$$(\text{shared}, H, H_\sigma) \overset{n}{=} (\text{shared}, H', H'_\sigma) \text{ iff } H \overset{n}{=}_{MI} H' \wedge H_\sigma \overset{n}{=}_{SI} H'_\sigma$$

♦

Definition 3.5.11 gives us the shared region for preorder c.o.f.e. when we use it with the memory invariant c.o.f.e. as *MemInv* and the seal invariant c.o.f.e. as *SealInv*.

Next, we define the preordered c.o.f.e. for spatial regions. The future region relation on spatial regions is non-trivial, so the definition of the preordered c.o.f.e. is slightly more involved.

Definition 3.5.12 (Spatial region preordered c.o.f.e.)**.** Given c.o.f.e.'s $\left(MemInv, \left(\overset{n}{=}_{MI}\right)_{n=0}^{\infty}\right)$ and $\left(MemInv', \left(\overset{n}{=}_{MI'}\right)_{n=0}^{\infty}\right)$, let

$$\text{Region}_{\text{spatial}} = \{\text{shadow}\} \times MemInv \cup \{\text{spatial}\} \times MemInv' \cup \{\text{revoked}\}$$

define the preordered c.o.f.e. of shared regions as

$$\left(\text{Region}_{\text{spatial}}, \left(\overset{n}{=}\right)_{n=0}^{\infty}, \sqsupseteq\right)$$

where the preorder $\sqsupseteq$ is defined as

$$\frac{r \in \text{Region}_{\text{spatial}}}{r \sqsupseteq r} \qquad \frac{}{\text{revoked} \sqsupseteq (\text{shadow}, \_)} \qquad \frac{}{(\text{spatial}, H) \sqsupseteq (\text{shadow}, H)}$$

and $\left(\overset{n}{=}\right)_{n=0}^{\infty}$ is the total relation for $n = 0$ and otherwise the region tags must be the same and the memory invariants n-equal, i.e. for $r, r' \in \text{Region}_{\text{spatial}}$ we have $r \overset{n}{=} r'$ when one of the following holds

- $n = 0$
- $n > 0$ and one of the following is true
  - $r = r' = \text{revoked}$
  - $r = (\text{shadow}, H)$, $r' = (\text{shadow}, H')$, and $H \overset{n}{=}_{MI} H'$
  - $r = (\text{spatial}, H)$, $r' = (\text{spatial}, H')$, and $H \overset{n}{=}_{MI'} H'$

♦

Definitions 3.5.11 and 3.5.12 define the preordered c.o.f.e.'s necessary for the regions in our worlds.

Next, we need to define the three sub-worlds for heap, encapsulated stack frames, and free stack. The definitions of the sub-worlds are very similar, so we can capture the essence of all three sub worlds in one definition.

Definition 3.5.13 (Sub-world preordered c.o.f.e.)**.** Given preordered c.o.f.e.

$$\left(Reg, \left(\overset{n}{=}_{Reg}\right)_{n=0}^{\infty}, \sqsupseteq_{Reg}\right)$$

define the sub-world preordered c.o.f.e. as

$$\left(\mathbb{N} \rightharpoonup Reg, \left(\overset{n}{=}\right)_{n=0}^{\infty}, \sqsupseteq\right)$$

where the family of equivalences is defined by given $W, W' \in \mathbb{N} \rightharpoonup Reg$,

$$W \overset{n}{=} W' \text{ iff } \text{dom}(W) = \text{dom}(W') \wedge \forall r \in \text{dom}(W).\, W(r) \overset{n}{=} W'(r)$$

and the future world relation is defined by given $W, W' \in \mathbb{N} \rightharpoonup Reg$,

$$W' \sqsupseteq W \text{ iff } \begin{cases} \exists m : \text{RegionName} \to \text{RegionName}, \text{ injective.} \\ \quad \text{dom}(W') \supseteq \text{dom}(m(W)) \wedge \forall r \in \text{dom}(W).\, W'.i(m(r)) \sqsupseteq W(r) \end{cases}$$

♦

The free stack sub-world preordered c.o.f.e. is constructed from the spatial region preordered c.o.f.e. (Definition 3.5.12) and the general sub-world construction in Definition 3.5.13. The result is the sub-world for the free stack

$$\text{World}_{\text{free\_stack}} = \mathbb{N} \rightharpoonup \text{Region}_{\text{spatial}}$$

Next, we define the call stack sub-world. To this end, we use Definition 3.5.12 to get the spatial region which we need to combine this with an address. We lift the set of addresses to a preordered c.o.f.e. A (Definition 3.5.9 with equality as the preorder) and combine the preordered c.o.f.e.'s Addr and $\text{Region}_{\text{spatial}}$ with the standard product construction. All in all, we get the a preordered c.o.f.e.

$$\text{Region}_{\text{spatial}} \times \text{Addr}$$

which we use with Definition 3.5.13 to get the preordered c.o.f.e. for the stack sub-world:

$$\text{World}_{\text{call\_stack}} = \mathbb{N} \rightharpoonup \text{Region}_{\text{spatial}} \times \text{Addr}$$

Finally, we define the sub-world for the heap. To this end, we combine the shared region preordered c.o.f.e. (Definition 3.5.11) with the spatial region preordered c.o.f.e. (Definition 3.5.12) using the standard union preordered c.o.f.e. construction. The result is a preordered c.o.f.e.

$$\text{Region}_{\text{spatial}} \cup \text{Region}_{\text{shared}}$$

which we use in the sub-world construction (Definition 3.5.13) to obtain the preordered c.o.f.e.

$$\text{World}_{\text{heap}} = \mathbb{N} \rightharpoonup (\text{Region}_{\text{spatial}} \cup \text{Region}_{\text{shared}})$$

With the three sub-worlds defined, we combine them with the standard product construction for preordered c.o.f.e.'s to get our worlds.

$$\text{World}_{\text{heap}} \times \text{World}_{\text{call\_stack}} \times \text{World}_{\text{free\_stack}}$$

This is the world, we would like to work with, but we still don't have a solution for the recursive domain equation. In fact, when we defined the regions, we just left world-index in the memory and seal invariants as an unspecified variable, which means that the above world has them as a variable. In order to find a solution, we still need to guard the recursive occurrence. To this

end, we simply use the construction for later preordered c.o.f.e.'s in Definition 3.A.8 with the above to get.

$$\blacktriangleright(\text{World}_{\text{heap}} \times \text{World}_{\text{call\_stack}} \times \text{World}_{\text{free\_stack}})$$

Now the recursive occurrence is guarded which means that there is a solution to the recursive domain equation.

Theorem 3.5.14. *There exists a c.o.f.e.* Wor *and preorder* $\sqsupseteq$ *such that* (Wor, $\sqsupseteq$) *is a preordered c.o.f.e., and there exists an isomorphism* ξ *such that*

$$\xi : \text{Wor} \cong \blacktriangleright(\text{World}_{\text{heap}} \times \text{World}_{\text{call\_stack}} \times \text{World}_{\text{free\_stack}})$$

and for $\hat{W}, \hat{W}' \in \text{Wor}$

$$\hat{W}' \sqsupseteq \hat{W} \textit{ iff } \xi(\hat{W}') \sqsupseteq \xi(\hat{W})$$

for $\text{World}_{\text{call\_stack}}$, $\text{World}_{\text{heap}}$, *and* $\text{World}_{\text{free\_stack}}$ *defined as follows*

$$\text{World}_{\text{heap}} = \text{RegionName} \rightharpoonup (\text{Region}_{\text{spatial}} \cup \text{Region}_{\text{shared}})$$

and

$$\text{World}_{\text{call\_stack}} = \text{RegionName} \rightharpoonup (\text{Region}_{\text{spatial}} \times \text{Addr})$$

and

$$\text{World}_{\text{free\_stack}} = \text{RegionName} \rightharpoonup \text{Region}_{\text{spatial}}$$

where $\text{RegionName} = \mathbb{N}$. $\text{Region}_{\text{spatial}}$ *and* $\text{Region}_{\text{shared}}$ *defined as follows*

$$\begin{aligned}\text{Region}_{\text{shared}} = \{\text{shared}\} \times (\text{Wor} \xrightarrow{mon,\, ne} \text{URel}(\text{MemSeg} \times \text{MemSeg})) \times \\ (\text{Seal} \rightharpoonup \text{Wor} \xrightarrow{mon,\, ne} \text{URel}(\text{Sealables} \times \text{Sealables}))\end{aligned}$$

and

$$\text{Region}_{\text{spatial}} = \begin{cases} \{\text{shadow}, \text{spatial}\} \times (\text{Wor} \xrightarrow{mon,\, ne} \text{URel}(\text{MemSeg} \times \text{MemSeg})) \cup \\ \{\text{revoked}\} \end{cases}$$

◊

Theorem 3.5.14 uses the method of Birkedal and Bizjak [18], Birkedal et al. [20] to construct the solution Wor to the recursive equation. Note, that Wor is not equal to $\blacktriangleright(\text{World}_{\text{heap}} \times \text{World}_{\text{call\_stack}} \times \text{World}_{\text{free\_stack}})$; it is isomorphic. This means that whenever we encounter Wor, we have to apply ξ and go under a later before we can actually use the world. This makes it rather inconvenient to have Wor as the world, so instead we define the worlds as

$$\text{World} = \text{World}_{\text{heap}} \times \text{World}_{\text{call\_stack}} \times \text{World}_{\text{free\_stack}}$$

Joining worlds

The world serves multiple purposes as it is both a specification of memory contents as well as a specification of authority. This is best seen in the operators used to join worlds. First when we see the world as a memory specification, we have a pretty standard join $\uplus$ that simply requires the worlds to have different region names.

Definition 3.5.15 (World disjoint union $\uplus$)**.** Given worlds W_1, W_2, W

$$\begin{aligned} W_1 \uplus W_2 = W \text{ iff } & \mathrm{dom}(W.\mathrm{heap}) = \mathrm{dom}(W_1.\mathrm{heap}) \uplus \mathrm{dom}(W_2.\mathrm{heap}) \wedge \\ & \mathrm{dom}(W.\mathrm{free\_stk}) = \mathrm{dom}(W_1.\mathrm{free\_stk}) \uplus \mathrm{dom}(W_2.\mathrm{free\_stk}) \wedge \\ & \mathrm{dom}(W.\mathrm{call\_stk}) = \mathrm{dom}(W_1.\mathrm{call\_stk}) \uplus \mathrm{dom}(W_2.\mathrm{call\_stk}) \end{aligned}$$

♦

The $\uplus$ world join does not guarantee that the result is a sensible world with respect to authority or memory specification. In Section 3.5.1, we define memory satisfaction which also acts as a well-formedness judgement.

When we view the world as a specification of authority, then the world join need to respect the region ownership. That is, when we join the authority of two worlds, then the ownership of the two worlds should not overlap. This is expressed by the $\oplus$ operator.

Definition 3.5.16 ($\oplus$, disjoint union of ownership)**.**

$$\begin{aligned} W_1 \oplus W_2 = W \text{ iff } & \mathrm{dom}(W.\mathrm{heap}) = \mathrm{dom}(W_1.\mathrm{heap}) = \mathrm{dom}(W_2.\mathrm{heap}) \wedge \\ & \mathrm{dom}(W.\mathrm{free\_stk}) = \mathrm{dom}(W_1.\mathrm{free\_stk}) = \mathrm{dom}(W_2.\mathrm{free\_stk}) \wedge \\ & \mathrm{dom}(W.\mathrm{call\_stk}) = \mathrm{dom}(W_1.\mathrm{call\_stk}) = \mathrm{dom}(W_2.\mathrm{call\_stk}) \wedge \\ & \forall r \in \mathrm{dom}(W.\mathrm{heap}).\ W.\mathrm{heap}(r) = W_1.\mathrm{heap}(r) \oplus W_2.\mathrm{heap}(r) \wedge \\ & \forall r \in \mathrm{dom}(W.\mathrm{free\_stk}).\ W.\mathrm{free\_stk}(r) = W_1.\mathrm{free\_stk}(r) \oplus W_2.\mathrm{free\_stk}(r) \wedge \\ & \forall r \in \mathrm{dom}(W.\mathrm{call\_stk}). \\ & \qquad \pi_1(W.\mathrm{call\_stk}(r)) = \pi_1(W_1.\mathrm{call\_stk}(r)) \oplus \pi_1(W_2.\mathrm{call\_stk}(r)) \end{aligned}$$

where $\oplus$ for regions is defined as

$$\begin{aligned} (\mathrm{shared}, H, H_\sigma) \oplus (\mathrm{shared}, H, H_\sigma) &= (\mathrm{shared}, H, H_\sigma) \\ (\mathrm{shadow}, H) \oplus (\mathrm{shadow}, H) &= (\mathrm{shadow}, H) \\ \mathrm{revoked} \oplus \mathrm{revoked} &= \mathrm{revoked} \\ (\mathrm{spatial}, H) \oplus (\mathrm{shadow}, H) &= (\mathrm{shadow}, H) \oplus (\mathrm{spatial}, H) \\ &= (\mathrm{spatial}, H) \end{aligned}$$

♦

The $\oplus$ operator, like the $\uplus$ operator, does not guarantee that the resulting world is sensible (e.g., the regions are not overlapping). Only when we

know that a certain memory satisfies the world (as a memory specification, see Section 3.5.1), will we be sure that the world's specifications are actually non-contradictory.

The $\uplus$ operator is used in the logical relation for components (Section 3.5.4) which specifies (among other things) that the world should specify the presence of the component's data memory. Linking two components then produces a new component with both components' data memory. The linked component is valid in a world that has the combined memory presence specifications, not the combined authority.

Note also that this picture is further complicated by our usage of non-authority-carrying shadow regions. They are really only in a world W as a shadow copy of a spatial region in another world W' that W will be combined with. The shadow copy is used for specifying when a memory satisfies a world: the memory should contain all memory ranges that anyone has authority over, not just the ones whose authority belongs to the memory itself. For example, if a register contains a linear pointer to a range of memory, then the register file will be valid in a world where the corresponding region is spatial, while the memory will be valid in a world with the corresponding region only shadow. However, for the memory to satisfy the world, the block of memory needs to be there, i.e. the memory should contain blocks of memory satisfying every region that is spatial, shared, but also just shadow (because it may be spatial in, for example, the register file's world).

Memory satisfaction

The world can be seen as a specification of the memory contents. This means that we need to define what it means for a pair of LCM and oLCM memories to satisfy the specification. The world also keeps track of the structure of the call stack, the allowed uses of designated seals, and linear capability authority, so these things also influence the definition of memory satisfaction. The world definition on its own allows the invariants imposed by regions to be overlapping. However, to be able to easily determine what invariants a memory segment must respect, we want the invariants of the regions to impose requirements on disjoint parts of the memory. The memory satisfaction relation makes sure that this is the case, so in a sense the memory satisfaction also acts as a well-formedness judgement for worlds.

Memory satisfaction is split into four definitions. At the top-level we have

$$ms_S, ms_{stk}, stk, ms_T :^{gc}_{n} W$$

which relates the source memory triple, ms_S (heap), ms_{stk} (free stack), and stk (stack frames), from oLCM to the target memory ms_T from LCM. The top-level definition of memory satisfaction splits the target memory in three parts, one for each of the three kinds of source memory. It also splits the

world in three to distribute the authority of the world. The three ms_T partitions are related to ms_S, ms_{stk}, and stk by the relations $\mathcal{H}$, $\mathcal{S}$, and $\mathcal{F}$, respectively. The $\mathcal{H}$ relation relates the oLCM heap (non-stack memory) to the corresponding heap memory on LCM. The $\mathcal{H}$ relation also ensures that the seal invariants associated with each region have invariants on disjoint sets of seals. The $\mathcal{S}$ relation relates the stack of encapsulated stack frames on oLCM to the corresponding memory on LCM. The layout of the stack determines the call order, so $\mathcal{S}$ also makes sure that the placement in memory of the stack frames have the correct order relative to each other and the base address of the stack. Finally, $\mathcal{F}$ relates the free part of the stack of oLCM to the corresponding memory segment on LCM. $\mathcal{F}$ also makes sure that the base address of the stack is actually part of the free stack.

Definition 3.5.17 (Heap relation)**.** For a set of seals $\overline{\sigma}$, memory segments ms and ms_T, and worlds W and W', we define the heap relation $\mathcal{H}$ as:

$$
\begin{aligned}
&(n, (\overline{\sigma}, ms, ms_T)) \in \mathcal{H}(W.\text{heap})(W') = \\
&\quad \left\{ \begin{array}{l}
\exists R_{ms} : \text{dom}(\mathit{active}(W.\text{heap})) \to \text{MemSeg} \times \text{MemSeg} \wedge \\
\quad ms_T = \biguplus_{r \in \text{dom}(\mathit{active}(W.\text{heap}))} \pi_2(R_{ms}(r)) \wedge \\
\quad ms = \biguplus_{r \in \text{dom}(\mathit{active}(W.\text{heap}))} \pi_1(R_{ms}(r)) \wedge \\
\quad \exists R_W : \text{dom}(\mathit{active}(W.\text{heap})) \to \text{World}. \\
\qquad W' = \oplus_{r \in \text{dom}(\mathit{active}(W.\text{heap}))} R_W(r) \wedge \\
\qquad \forall r \in \text{dom}(\mathit{active}(W.\text{heap})), n' < n. \\
\qquad\quad (n', R_{ms}(r)) \in W.\text{heap}(r).H\ \xi^{-1}(R_W(r)) \wedge \\
\exists R_{seal} : \text{dom}(\mathit{active}(W.\text{heap})) \to \mathcal{P}(\text{Seal}) \wedge \\
\quad \biguplus_{r \in \text{dom}(\mathit{active}(W.\text{heap}))} R_{seal}(r)) \subseteq \overline{\sigma} \wedge \\
\quad \forall r \in \text{dom}(\lfloor W.\text{heap} \rfloor_{\{\text{shared}\}}). \text{dom}(W.\text{heap}(r).H_\sigma) = R_{seal}(r)
\end{array} \right.
\end{aligned}
$$

♦

Memory satisfaction, and thus also the heap relation, only considers the non-revoked regions. The $\mathcal{H}$-relation uses the function active to erase all the revoked regions from the world.

To a large extend, the definition of $\mathcal{H}$ is pretty standard. $\mathcal{H}$ assumes the existence of a partitioning of the LCM and oLCM heap memories that can be turned into memory segment pairs each satisfying the invariant of a region. The heap satisfaction must also respect the world as an authority specification, so the heap satisfaction partitions the authority of the world using $\oplus$. Each of the memory segment pairs must be in the region invariant with respect to a specific world partition which makes sure that uniqueness of linearity of capabilities is respected.

The heap sub-world contains all seal invariants. Similar to memory segments, only one seal invariant should impose restrictions on a seal, which $\mathcal{H}$ makes sure is the case.

Definition 3.5.18 (Free stack relation)**.**

$$(n,(ms_{stk},ms_T)) \in \mathcal{F}^{gc}(W) \text{ iff}$$

$$\left\{\begin{array}{l} gc = (\_,\text{stk\_base},\_,\_)\wedge \\ W_{stack} = W.\text{free\_stk}\wedge \\ \exists R_{ms} : \text{dom}(active(W_{stack})) \to \text{MemSeg}\times\text{MemSeg}\wedge \\ \quad ms_T = \biguplus_{r\in\text{dom}(active(W_{stack}))} \pi_2(R_{ms}(r))\wedge \\ \quad ms_{stk} = \biguplus_{r\in\text{dom}(active(W_{stack}))} \pi_1(R_{ms}(r))\wedge \\ \quad \text{stk\_base}\in\text{dom}(ms_T)\wedge\text{stk\_base}\in\text{dom}(ms_{stk})\wedge \\ \quad \exists R_W : \text{dom}(active(W_{stack})) \to \text{World}. \\ \quad\quad W = \bigoplus_{r\in\text{dom}(active(W_{stack}))} R_W(r)\wedge \\ \quad\quad \forall r\in\text{dom}(active(W_{stack})), n' < n. \\ \quad\quad\quad (n', R_{ms}(r)) \in W_{stack}(r).H\ \xi^{-1}(R_W(r)) \end{array}\right.$$

♦

The free stack relation $\mathcal{F}$ is in most regards like the heap relation, $\mathcal{H}$. The free stack relation, partitions the oLCM and LCM free stack memory, it partitions the authority of the world, and it requires the memory segment pairs to be related under part of the world. For StkTokens to work, it should always work on the same stack. As discussed in Section 3.3, we make sure that it is always the same stack by requiring the address stk_base to be the "top" address of the free stack address space. As the free stack relation relates the stack of oLCM with the memory that represents the stack on LCM, it makes sure that stk_base is the top address of the free stack address space.

Definition 3.5.19 (Stack relation)**.**

$$(n,(stk,ms_T)) \in \mathcal{S}^{gc}(W) \text{ iff}$$

$$\left\{\begin{array}{l} \exists m, opc_0,\ldots,opc_m, ms_0,\ldots,ms_m, W_{stack}. \\ \quad gc = (\_,\text{stk\_base},\_,\_)\wedge \\ \quad W_{stack} = W.\text{call\_stk}\wedge \\ \quad stk = (opc_0,ms_0),\ldots(opc_m,ms_m)\wedge \\ \quad \forall i\in\{0,\ldots,m\}.(\text{dom}(ms_i)\neq\emptyset\wedge \\ \quad\quad \forall j<i.\forall a\in\text{dom}(ms_i).\forall a'\in\text{dom}(ms_j).\text{stk\_base}<a<a')\wedge \\ \quad \exists R_{ms} : \text{dom}(active(W_{stack})) \to \text{MemSeg}\times\text{Addr}\times\text{MemSeg}. \\ \quad\quad ms_T = \biguplus_{r\in\text{dom}(active(W_{stack}))} \pi_3(R_{ms}(r))\wedge \\ \quad\quad ms_0\uplus\cdots\uplus ms_m = \biguplus_{r\in\text{dom}(active(W_{stack}))} \pi_1(R_{ms}(r))\wedge \\ \quad\quad \exists R_W : \text{dom}(active(W_{stack})) \to \text{World}. \\ \quad\quad\quad W = \bigoplus_{r\in\text{dom}(active(W_{stack}))} R_W(r)\wedge \\ \quad\quad\quad \forall r\in\text{dom}(active(W_{stack})), n' < n. \\ \quad\quad\quad\quad (n',(\pi_1(R_{ms}(r)),\pi_3(R_{ms}(r)))\in W_{stack}(r).H\ \xi^{-1}(R_W(r))\wedge \\ \quad\quad\quad\quad \pi_2(R_{ms}(r)) = W_{stack}(r).opc\wedge \\ \quad\quad\quad\quad \exists i.\ opc_i = W_{stack}(r).opc\wedge ms_i = \pi_1(R_{ms}(r)) \end{array}\right.$$

♦

The stack relation $\mathcal{S}$ is similar to the heap relation in some ways. The $\mathcal{S}$ relation also partitions the LCM memory but only the LCM as the oLCM partitions are given as the stack frames. The stack relation also partitions the authority of the world, so it can relate the stack frames in a way that respect linearity. The stack on oLCM represents the call stack which means that each stack frame corresponds to a call and its local data. The operational semantics of LCM does not have a built-in stack, so we emulate it by requiring that a stack like data structure resides in LCM memory. That is for a memory segment that represents a stack frame, all the addresses of memory frames lower in the stack should have strictly smaller memory addresses. Further, the stack frames should be in the part of the memory we agree to be the stack which means that the addresses should be smaller than stk_base. Informally, this just means that the stack should be laid out in memory as a downwards growing stack with no addresses below stk_base.

According to StkTokens (described in Section 3.3), stack frames must be non-empty, so they are distinguishable from a missing stack frame. For this reason, $\mathcal{S}$ requires every stack frame to be non-empty. Note that each stack frame corresponds to a trusted call. Untrusted calls are not protected which means that untrusted stack frames reside in the free stack memory (this does not mean that the untrusted stack frames are necessarily unprotected; it only means that they are not natively protected by StkTokens). This means that the protected stack frames are not necessarily packed tightly in memory, and the memory in between is part of the free stack. The free stack in between stack frames may be used by untrusted code for their local stack frames (Untrusted stack frames are not protected by oLCM, but this does not prevent untrusted code from securing their own stack frames). Figure 3.18 sketches this.

Each stack frame in the oLCM stack contains an old program pointer which corresponds to the old program counter recorded in the region of the world associated with the stack frame. To achieve this, the partition function R_{ms} also records an *opc* for each region, and this *opc* should establish the link between the region and the stack frame.

In order to tie Definitions 3.5.17, 3.5.18, and 3.5.19 together, we define memory satisfaction. Memory satisfaction defines when a oLCM memory, consisting of a heap, a stack, and a free stack, relates to a LCM memory under a world.

Definition 3.5.20 (Memory satisfaction)**.** For memory segments ms_S, ms_{stk},

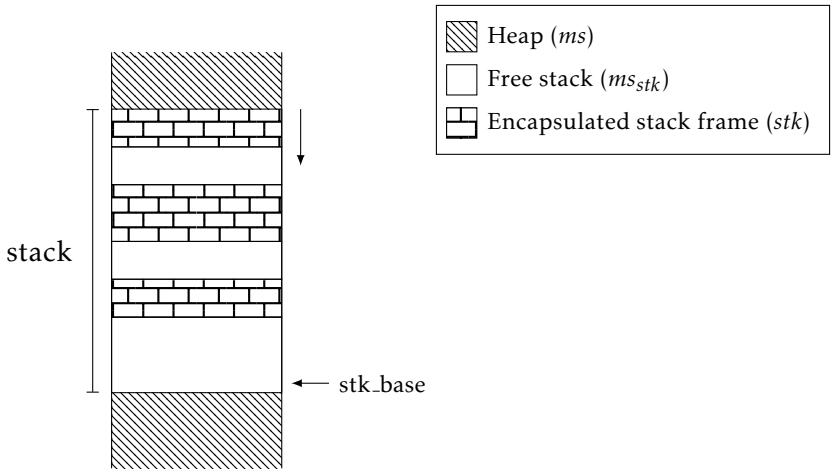


Figure 3.18: A sketch of how heap, encapsulated stack and free stack are laid out in memory. The encapsulated stack frames and the free stack constitutes the stack. The encapsulated stack frames may have free stack in between them where stack frames of non-trusted code may reside. The stack grows downwards in memory with stk_base as the top address (and the base address of the free stack capability).

and ms_T, stack stk, and world W we define memory satisfaction as

$$ms_S, ms_{stk}, stk, ms_T :^{gc}_n W \text{ iff}$$

$$\left\{\begin{array}{l}
\exists m, opc_0, \ldots opc_m, ms_0, \ldots, ms_m, W_{stack}, W_{free_stack}, W_{heap}. \\
\quad stk = (opc_0, ms_0) :: \cdots :: (opc_m, ms_m) \wedge \\
\quad ms_S \# ms_{stk} \# ms_0 \# \ldots \# ms_m \wedge \\
\quad W = W_{stack} \oplus W_{free_stack} \oplus W_{heap} \wedge \\
\quad \exists ms_{T,stack}, ms_{T,free_stack}, ms_{T,heap}, ms_{T,f}, ms_{S,f}, ms'_S, \overline{\sigma}. \\
\qquad ms_S = ms_{f,S} \uplus ms'_S \wedge \\
\qquad ms_T = ms_{T,stack} \uplus ms_{T,free_stack} \uplus ms_{T,heap} \uplus ms_{T,f} \wedge \\
\qquad \mathrm{dom}(ms_{T,stack} \uplus ms_{T,free_stack}) = [b_{stk}, e_{stk}] \wedge \\
\qquad b_{stk} - 1, e_{stk} + 1 \in \mathrm{dom}(ms_{T,f}) \wedge \\
\qquad (n, (stk, ms_{T,stack})) \in \mathcal{S}^{gc}(W_{stack}) \wedge \\
\qquad \left(n, (ms_{stk}, ms_{T,free_stack})\right) \in \mathcal{F}^{gc}(W_{free_stack}) \wedge \\
\qquad \left(n, (\overline{\sigma}, ms'_S, ms_{T,heap})\right) \in \mathcal{H}(W.\mathrm{heap})(W_{heap})
\end{array}\right.$$

♦

Memory satisfaction partitions the LCM memory in a heap, stack frames, a free stack and a frame. The oLCM heap is split in two: the active heap and a frame. Our configurations describe the complete machine state, but we may only be interested in the invariants on part of it. The frame allows us to ignore the part of the memory that won't affect the computation. Just

like the previous memory relations, the world is split in three to make sure that linearity is respected. Each part of the oLCM memory is related to the appropriate part of the memory from LCM by the relevant relation under a partition of the world.

StkTokens requires the stack to not be adjacent to heap or code memory. This is enforced in the memory satisfaction by requiring that the addresses adjacent to the memory are in the frame.

3.5.2 The Logical Relation

Using these Kripke worlds as assumptions, we can then define when different oLCM and LCM entities are related: values, jump targets, memories, execution configurations, components etc. The most important relations are summarised in the following table, where we mention the general form of the relations, what type of things they relate and extra conditions that some of them imply:

General form	Relates ...	and ...
$(n, (w_S, w_T)) \in \mathcal{V}^{\square,gc}_{\text{untrusted}}(W)$	values (machine words)	safe to pass to adversarial code
$(n, (w_S, w_T)) \in \mathcal{V}^{\square,gc}_{\text{trusted}}(W)$	values (machine words)	
$(n, (reg_S, reg_T)) \in \mathcal{R}^{\square,gc}(W)$	register files	safe to pass to adversarial code
$(n, \Phi_S, \Phi_T) \in \mathcal{O}^{\square,gc}$	execution configurations	
$(n, (w_S, w_T)) \in \mathcal{E}^{\square,gc}(W)$	`jmp` targets	
$\begin{pmatrix} (w_{S,1}, w_{S,2}), \\ (w_{T,1}, w_{T,2}) \end{pmatrix} \in \mathcal{E}^{\square,gc}_{\text{xjmp}}(W)$	`xjmp` targets	
$ms_S, stk, ms_{stk}, ms_T :^{gc}_n W$	memory	satisfy the assumptions in W

In Section 3.5.1, we already defined memory satisfaction, the relation for memories. In the following, we define each of the remaining relations and give some intuition about the definitions. The logical relation we define ends up as a cyclic definition. The circularity is resolved by another use of step-indexing in the definitions, but the circularity also poses a chicken and egg problem with respect to the order in which the definitions of the relations should be presented. There is no canonical way of presenting the logical relation as we are bound to make forward references. For this reason, we suggest making a cursory first read through to get an overview followed by a more thorough read.

Observation relation

The observation relation defines what machine configurations have related and permissible observable effects. Generally speaking, an observation relation captures the property we want to prove. Ultimately, we want to prove a

full-abstraction theorem which is defined in terms of contextual equivalence for components that in turn is defined as co-termination in any context. This means that the observation relation should capture co-termination.

So far, we have talked about the logical relation as though we define one. However, we actually define two logical approximations that only differ in the observation relation. We define a oLCM configuration to logically approximate a LCM configuration when the halting termination of the oLCM configuration implies the halting termination of the LCM configuration. This also means that oLCM configurations that terminates by failing termination are related to any LCM configuration. Intuitively, this is because the failed configuration signals that there was an attempt to break the guarantees of the capability machine. For instance, a piece of code could have attempted to read from a part of memory it does not have access to, or a callee could have attempted to return out of order. In both cases, we haven't defined a way to recover from such attempts to break the guarantees, so we are content with failure.

$$\mathcal{O}^{\leq,(T_A,\text{stk\_base},\_,\_)} \stackrel{def}{=}$$

$$\left\{ \left(n, \begin{pmatrix} (ms_S, reg_S, stk_S, ms_{stk,S}), \\ (ms_T, reg_T) \end{pmatrix} \right) \middle| \begin{array}{l} \forall i \leq n. \\ \quad (ms_S, reg_S, stk_S, ms_{stk,S}) \Downarrow_i^{T_A,\text{stk\_base}} \\ \qquad \Rightarrow (ms_T, reg_T) \Downarrow_- \end{array} \right\}$$

The step-indexing plays a role here because we are only interested in oLCM configurations that terminate in n or fewer steps. However, if the oLCM configuration terminates successfully in n steps, then the LCM configuration should just terminate in any number of step (possibly more than n steps). For the most part, it would make sense to require the LCM configuration to terminate in the same amount of steps as the oLCM configuration as they run in lockstep for most of the computation. However, when it comes to calls and returns, the two configurations stop running in lockstep. The oLCM configuration handles calls and returns in one step whereas LCM configurations need to execute each instruction of the call preparation as well as the return code.

We define a LCM configuration to approximate a oLCM configuration in a dual way to the above.

$$\mathcal{O}^{\geq,(T_A,\text{stk\_base},\_,\_)} \stackrel{def}{=}$$

$$\left\{ \left(n, \begin{pmatrix} (ms_S, reg_S, stk_S, ms_{stk,S}), \\ (ms_T, reg_T) \end{pmatrix} \right) \middle| \begin{array}{l} \forall i \leq n. \\ \quad (ms_T, reg_T) \Downarrow_i \\ \qquad \Rightarrow (ms_S, reg_S, stk_S, ms_{stk,S}) \Downarrow_-^{T_A,\text{stk\_base}} \end{array} \right\}$$

The remainder of our logical relation will be the same for both $\leq$ and $\geq$, so we will write $\square$ instead of the approximation.

Value Relations

The value relation relates LCM words to oLCM words. The oLCM machine has special tokens that represent the stack capabilities and the return pointer components. These tokens do not exist on LCM, but all of the tokens correspond to capabilities on LCM, and the value relation establishes the link between then. Skorstengaard et al. [83] defines a logical relation that can be seen as a notion of capability safety. When they define their value relation, they define based on the question "What is the most an adversary can be allowed to do with this word without breaking memory invariants?" This allows them to use the logical relation to reason about arbitrary (untrusted) programs. We also want to be able to say something about arbitrary (untrusted) programs, but we also want to be able to say something about somewhat arbitrary trusted programs. In our setting, a trusted program is a well-formed, reasonable program that follows the StkTokens calling convention, and an untrusted program is an arbitrary well-formed program. In order for a trusted program to use StkTokens, it needs access to return seals, but we cannot allow untrusted programs access to the return seals. A value relation based on what it is safe for an adversary to have should prohibit return seals, so such a relation cannot be used to reason about trusted programs. For this reason, we define two value relations a trusted $\mathcal{V}_{\text{trusted}}$ and an untrusted $\mathcal{V}_{\text{untrusted}}$. Anything safe for unstrusted programs is also safe to give to a trusted program, so the trusted value relation is defined as a super set of the untrusted value relation.

From time to time in this section, we will refer to safety of a capability or a word. In some sense, our logical relation actually ends up as the definition of safety, so when we refer to a capability as *safe* it is in an informal sense where it means that the capability cannot be used break memory invariants.

In Figure 3.19, we have sketched the two value relations. This shows that for the most part, words on oLCM are related to words on LCM that are syntactical identical. The only exception is stack pointers on oLCM that are related to linear capabilities on LCM. Note that the return pointers of oLCM are not related to anything as it is never safe for any program, trusted or not, to have them. The oLCM return pointers should only occur under a return seal, and they should only be used in a jump in which case the oLCM semantics transforms them to the capabilities they correspond to.

The value relation is defined in terms of a number of auxiliary definitions. In the following, we introduce a number of *standard regions* that express common requirements on memory. Based on the standard regions, we define what we call *permission based conditions*, conditions that a capability with a specific permission must satisfy to be safe.

Standard regions The notion of regions we defined in Section 3.5.1 is general enough to allow a wide variety of regions. There are, however, some

$$
\begin{aligned}
\mathcal{V}^{\square,gc}_{\text{untrusted}}(W) = {} & \{(n,(i,i)) \mid i \in \mathbb{Z}\} \cup \\
& \{(n,(\text{stack-ptr}(p,b,e,a),((p,\text{linear}),b,e,a))) \mid \ldots\} \cup \\
& \{(n,(\text{seal}(\sigma_b,\sigma_e,\sigma),\text{seal}(\sigma_b,\sigma_e,\sigma))) \mid \ldots\} \cup \\
& \{(n,(\text{sealed}(\sigma,sc_S),\text{sealed}(\sigma,sc_T))) \mid \ldots\} \cup \\
& \{(n,(((p,l),b,e,a),((p,l),b,e,a))) \mid \ldots\} \\
\mathcal{V}^{\square,gc}_{\text{trusted}}(W) = {} & \mathcal{V}^{\square,gc}_{\text{untrusted}}(W) \cup \\
& \{(n,(\text{seal}(\sigma_b,\sigma_e,\sigma),\text{seal}(\sigma_b,\sigma_e,\sigma))) \mid \ldots\} \cup \\
& \{(n,(((p,\text{normal}),b,e,a),((p,\text{normal}),b,e,a))) \mid p \leq \text{RX} \wedge \ldots\}
\end{aligned}
$$

Figure 3.19: Sketches of the trusted and untrusted value relation. The untrusted and trusted value relation both relates oLCM and LCM words. The untrusted value relation $\mathcal{V}_{\text{untrusted}}$ relates words that are safe to give to untrusted programs and $\mathcal{V}_{\text{trusted}}$ relates words that are safe to give to trusted programs.

regions that may seem more natural or standard than others. In particular, when it comes to capability safety, it seems natural to have a region that requires everything in memory to be safe. This is exactly what we refer to as a *standard region* because we usually define a region like that along with a logical relation.

We define a shared, shadow, and spatial standard region. They all have the same invariant which is defined as follows:

$$
H^{\text{std},\square}_A\ gc\ \hat{W} \stackrel{def}{=} \left\{ (n, ms_S, ms_T) \;\middle|\; \begin{array}{l} \text{dom}(ms_S) = \text{dom}(ms_T) = A \wedge \\ \exists S : A \to \text{World}.\, \xi(\hat{W}) = \oplus_{a \in A} S(a) \wedge \\ \quad \forall a \in A.\, (n,(ms_S(a), ms_T(a))) \in \mathcal{V}^{\square,gc}_{\text{untrusted}}(S(a)) \end{array} \right\}
$$

The standard region invariant requires the memory segment pairs to have a specific address space A. Further, the two memory segments must contain words from the untrusted value relation. The memory segments may contain linear capabilities, so we must distribute the ownership of the world between each memory cell which the function S takes care of. Note that the invariant takes a $\hat{W}$ from Wor as argument which means that we must apply the isomorphism ξ before the world can be used. Using this invariant, we define the standard shadow and spatial regions as follows:

$$
\iota^{\text{std},v}_{A,gc} \stackrel{def}{=} (v, H^{\text{std},\square}_A\ gc), v \in \{\text{shadow}, \text{spatial}\}
$$

and the standard shared regions as follows

$$
\iota^{\text{std,shared}}_{A,gc} \stackrel{def}{=} (\text{p}, H^{\text{std},\square}_A\ gc, \lambda\_\ \_.\, \emptyset)
$$

Note that the standard shared region has an empty seal invariant and thus puts no requirements on seals.

Sometimes we need to know that the contents of a memory segment stays the same. For instance, the contents of encapsulated stack frames do not change which we need to be able to rely on. To express this, we define a *static region*. The static region is parameterised with a memory segment pair which is the only memory segment pair the region accepts. The memory invariant is defined as follows

$$H^{\text{sta},\square}_{(ms_S,ms_T)}\ gc\ \hat{W} \stackrel{def}{=}$$
$$\left\{ (n,(ms_S,ms_T)) \;\middle|\; \begin{array}{l} \text{dom}(ms_S) = \text{dom}(ms_T) \wedge \\ \exists S : \text{dom}(ms_S) \to \text{World}.\ \xi(\hat{W}) = \oplus_{a\in\text{dom}(ms)} S(a) \wedge \\ \quad \forall a \in \text{dom}(ms_S).\ (n,(ms_S(a),ms_T(a))) \in \mathcal{V}^{\square,gc}_{\text{untrusted}}(S(a)) \end{array} \right\}$$

The region also requires the static memory to contain words from the untrusted value relation. This means that the stack should not be used to store return seals, closure seals, and code pointers for trusted code. With the memory invariant, we define the static region as follows:

$$\iota^{\text{sta},v,\square}_{(ms_S,ms_T),gc} \stackrel{def}{=} (v, H^{\text{sta},\square}_{(ms_S,ms_T)}\ gc), v \in \{\text{shadow}, \text{spatial}\}$$

A shared static region can be defined in a similar fashion to that of the standard region.

In our result, we assume well-formed components which puts certain syntactic constraints on the components. We also have the semantic assumption that trusted components are reasonable. Both assumptions need to be captured in the logical relation in order for us to rely on them. To this end, we define a *code region* which captures the syntactic and semantic assumptions we make on components. The memory invariant of the code region is defined as

$$H^{\text{code}}\ \overline{\sigma_{\text{ret}}}\ \overline{\sigma_{\text{clos}}}\ code\ (T_A, \_, \overline{\sigma_{\text{glob\_ret}}}, \overline{\sigma_{\text{glob\_clos}}})\ \hat{W} =$$
$$\left\{ \begin{array}{l} \left(n, \left(code \uplus ms_{\text{pad}}, code \uplus ms_{\text{pad}}\right)\right) \Big| \\ \text{dom}(code) = [b,e] \wedge \\ ([b-1,e+1] \subseteq T_A \wedge \overline{\sigma_{\text{ret}}} \subseteq \overline{\sigma_{\text{glob\_ret}}} \wedge \overline{\sigma_{\text{clos}}} \subseteq \overline{\sigma_{\text{glob\_clos}}} \wedge tst = \text{trusted}) \vee \\ \quad ([b-1,e+1] \# T_A \wedge \overline{\sigma_{\text{ret}}} = \emptyset \wedge tst = \text{untrusted}) \wedge \\ ms_{\text{pad}} = [b-1 \mapsto 0] \uplus [e+1 \mapsto 0] \wedge \\ \overline{\sigma_{\text{ret}}}, \overline{\sigma_{\text{clos}}}, T_A \vdash_{\text{comp-code}} code \wedge \\ \forall a \in \text{dom}(code). \\ \quad (n, (code(a), code(a))) \in \mathcal{V}^{\square,gc}_{tst}(sharedPart(\xi(\hat{W}))) \end{array} \right\}$$

The code region is more restrictive than the standard region. It only allows one memory segment, namely *code* padded with zeroes that make sure that

two capabilities cannot be spliced to cause unintended control-flow. We use the relation to reason about trusted components (well-formed and reasonable) as well as untrusted components (well-formed). The assumptions we can make on the code depends on whether it is part of a trusted or untrusted component. This is captured by requiring the contents of the code memory to be in the trusted or untrusted value relation depending on the trustworthiness of the code. That is, if all the code memory addresses are in the trusted address space and the seals are from the global seals, then the component is trusted. On the other hand, if the code memory addresses are disjoint from the trusted addresses and there are no return seals, then the component is untrusted. In either case, the words should be in the value relation with respect to the *sharedPart* of the world which means that the code memory cannot contain linear capabilities.

StkTokens rely on proper seal usage to guarantee well-bracketed control-flow and local state encapsulation. This means that components must use return and closure seals for their intended purpose for StkTokens to work. The code region has a seal invariant $H_\sigma^{\text{code},\square}$ to guarantee that the return and closure seals of the region are used correctly. The seal invariant is displayed in Figure 3.20. The return seals $\overline{\sigma_{\text{ret}}}$ in a code region should only be used to seal return pointers. That is on oLCM, the return seals should only be used to seal ret-ptr-code and ret-ptr-data. If we allowed any ret-ptr-code to be sealed, then we could not be sure that the ret-ptr-code came from a call even though it should only be possible to get a return pointer from a call. For this reason, we require that the oLCM return pointer actually points to the first address after a call. For a LCM capability related to a oLCM code return pointer, we require it to point to the first address of the return code, not the first address after the call, as the return instructions must be executed.

For sealed data return pointers, we need to know that the world contains a region that governs the local stack frame. That is, there should be a static region with the contents of the stack frame. The fact that it is static signifies that the contents will remain the same. The region that governs the stack frame must come from the call-stack sub-world which means that it is paired with a return address. The return address should correspond to an actual return address of a call in *code*.

Unlike return seals, both trusted and untrusted components can have closure seals. For untrusted components (components with their code address space disjoint from the trusted address space), we allow everything in the untrusted value relation to be sealed. Intuitively, untrusted components are assumed to have access to words from the untrusted value relation, and we cannot know how the words are used, so we need to assume that an untrusted component may seal untrusted words. Trusted components only use closure seals for sealed capability pairs that represent actual closures. The code capability for a closure must point to the code memory because it is the only

$$
\begin{aligned}
&H_\sigma^{\text{code},\square}\ \overline{\sigma_{\text{ret}}}\ \overline{\sigma_{\text{clos}}}\ \mathit{code}\ (T_A, \text{stk\_base}, \_, \overline{\sigma_{\text{glob\_ret}}})\ \sigma\ \hat{W} \stackrel{def}{=} \\
&\left\{
\begin{array}{l}
\left(n, \left(\begin{array}{l}\text{ret-ptr-code}(b,e,a'+\text{call\_len}),\\ ((\textsc{rx}, \text{normal}), b, e, a)\end{array}\right)\right)\Bigg| \\
\overline{\sigma_{\text{ret}}} \subseteq \overline{\sigma_{\text{glob\_ret}}} \wedge \\
\text{dom}(\mathit{code}) \subseteq T_A \wedge \\
\mathit{decode}(\mathit{code}([a', a'+\text{call\_len}-1])) = \overline{\texttt{call}^{\mathit{off}_{\text{pc}}, \mathit{off}_\sigma}\ r_1\ r_2} \wedge \\
a = a' + \text{ret\_pt\_offset} \wedge \\
\mathit{code}(a' + \mathit{off}_{\text{pc}}) = \text{seal}(\sigma_b, \sigma_e, \sigma_b) \wedge \sigma = \sigma_b + \mathit{off}_\sigma \in \overline{\sigma_{\text{ret}}} \wedge \\
[a', a'+\text{call\_len}-1] \subseteq [b, e]
\end{array}
\right\} \cup \\
&\left\{
\begin{array}{l}
\left(n, \left(\begin{array}{l}\text{ret-ptr-data}(b,e),\\ ((\textsc{rw}, \text{linear}), b, e, b-1)\end{array}\right)\right)\Bigg| \\
\overline{\sigma_{\text{ret}}} \subseteq \overline{\sigma_{\text{glob\_ret}}} \wedge \\
\text{dom}(\mathit{code}) \subseteq T_A \wedge \\
\exists r \in \mathit{addressable}(\text{linear}, \xi(\hat{W}).\text{call\_stk}). \\
\quad \xi(\hat{W}).\text{call\_stk}(r) \stackrel{n}{=} (\iota^{\text{sta,spatial},\square}_{(ms_S, ms_T)}\ (T_A, \text{stk\_base}), a'+\text{call\_len}) \wedge \\
\quad \text{dom}(ms_S) = \text{dom}(ms_T) = [b, e] \wedge \\
\quad \mathit{decode}(\mathit{code}([a', a'+\text{call\_len}-1])) = \overline{\texttt{call}^{\mathit{off}_{\text{pc}}, \mathit{off}_\sigma}\ r_1\ r_2} \wedge \\
\quad \mathit{code}(a' + \mathit{off}_{\text{pc}}) = \text{seal}(\sigma_b, \sigma_e, \sigma_b) \wedge \sigma = \sigma_b + \mathit{off}_\sigma \in \overline{\sigma_{\text{ret}}}
\end{array}
\right\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{for } \sigma \in \overline{\sigma_{\text{ret}}}
\end{aligned}
$$

$$
\begin{aligned}
&H_\sigma^{\text{code},\square}\ \overline{\sigma_{\text{ret}}}\ \overline{\sigma_{\text{clos}}}\ \mathit{code}\ (T_A, \text{stk\_base}, \overline{\sigma_{\text{glob\_clos}}}, \overline{\sigma_{\text{glob\_ret}}})\ \sigma\ \hat{W} \stackrel{def}{=} \\
&\left\{
\begin{array}{l}
(n, (sc, sc'))| \\
(\text{dom}(\mathit{code}) \# T_A \wedge (n, (sc, sc')) \in \mathcal{V}^{\square,gc}_{\text{untrusted}}\ \xi(\hat{W})) \vee \\
(\text{dom}(\mathit{code}) \subseteq T_A \wedge \overline{\sigma_{\text{clos}}} \subseteq \overline{\sigma_{\text{glob\_clos}}} \wedge \overline{\sigma_{\text{ret}}} \subseteq \overline{\sigma_{\text{glob\_ret}}} \wedge \\
\quad (\mathit{executable}(sc) \wedge (n, (sc, sc')) \in \mathcal{V}^{\square,gc}_{\text{trusted}}\ \xi(\hat{W})) \vee \\
\quad (\mathit{nonExec}(sc) \wedge (n, (sc, sc')) \in \mathcal{V}^{\square,gc}_{\text{untrusted}}\ \xi(\hat{W})))
\end{array}
\right\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{for } \sigma \in \overline{\sigma_{\text{clos}}}
\end{aligned}
$$

Figure 3.20: The seal invariant for code regions.

part of memory that is executable. Untrusted components cannot safely have a capability for a trusted components code (it could be used to read return capabilities or start execution in the middle of a call), so capabilities for the code memory of a trusted component is in the trusted value relation. While it is not safe to give a bare capability for a trusted components code memory, it can be perfectly safe to give a sealed capability for a trusted components code. For this reason, the seal invariant allows executable capabilities from the trusted value relation to be sealed with a closure seal.

When it comes to the data capability of a closure, we just require that it comes from the untrusted value relation because the trusted value relation contains nothing that makes sense to seal as the data capability (we return to the specific contents of the two value relations later in this section).

With the memory invariant and seal invariant in hand, we define the code region as follows:

$$\iota^{\text{code}}_{\overline{\sigma_{\text{ret}}},\overline{\sigma_{\text{clos}}},code,gc} \stackrel{def}{=} (\text{shared}, H^{\text{code},\square}\ \overline{\sigma_{\text{ret}}}\ \overline{\sigma_{\text{clos}}}\ code\ gc, H^{\text{code}}_{\sigma}\ \overline{\sigma_{\text{ret}}}\ \overline{\sigma_{\text{clos}}}\ code\ gc)$$

The code region is shared because it needs to contain a seal invatiant and because we assume that code pointers are normal capabilities.

Permission based conditions The safe capabilities will be defined by the value relation. However, the safety requirements for a capability depends on the authority the capability gives. Therefore, rather than bundling everything into the value relation, we first present a number of *permission based conditions* that each spell out what the requirements are for each permission.

The world can be seen as an authority specification which means that it dictates what kind of capabilities can address a certain part of memory. Specifically, linear capabilities can only address memory governed by a spatial region, and normal capabilities can only address memory governed by a shared region. All the permission based condition we define project the regions that the capability may address from the world. The addressable *addressable* function takes care of the projection:

$$addressable(l, W) \stackrel{def}{=} \begin{cases} \{r \mid W(r) = (\text{shared}, \_)\} & \text{if } l = \text{normal} \\ \{r \mid W(r) = (\text{spatial}, \_)\} & \text{otherwise (i.e. } l = \text{linear)} \end{cases}$$

We capture the essence of what it means for a capability with read permission to be safe in the condition *readCondition*. The main purpose of *readCondition* is to make sure that only safe words can be read from the memory governed by a read capability. This is done by putting an upper bound on what requirements an invariant can impose on the memory segments governed by the capability. In particular a region that governs the memory a read capability has access to can at most allow safe values to be read. Without this requirement, a read capability could potentially be used to break memory

invariants if it were used to read capabilities that has the authority to break memory invariants. The read condition is defined as follows

$$readCondition^{\square,gc}(l,W) = \left\{ (n,A) \;\middle|\; \begin{array}{l} \exists S \subseteq addressable(l, W.\text{heap}). \\ \quad \exists R : S \to \mathcal{P}(\mathbb{N}). \\ \qquad \biguplus_{r \in S} R(r) \supseteq A \wedge \\ \qquad (l = \text{linear} \Rightarrow \forall r \in S. |R(r)| = 1) \wedge \\ \qquad \forall r \in S. W.\text{heap}(r).H \overset{n}{\subseteq} \iota^{\text{std,p}}_{R(r),gc}.H \end{array} \right\}$$

The *readCondition* is compatible with all the operations that can be performed on capabilities. This means that if two capabilities, for which *readCondition* holds, are spliced together, we can establish that *readCondition* holds for the resulting capability. To support this, we require the presence of a set of regions S that governs the addresses the capability has authority over rather than just a single region. If we need to establish the *readCondition* after a splice, we can simply use the union of the regions that witnessed the *readCondition* of the two individual capabilities. We also need to support splitting which is no problem for normal capabilities as the same shared region can be used to establish the *readCondition* for multiple normal capabilities. On the other hand, a spatial region can only be used to establish the *readCondition* for one linear capability because the ownership of a spatial region can only go to one world when splitting the ownership. None the less, we need to support arbitrary splitting of linear capabilities, which means that *readCondition* must make sure that the necessary regions are in the world to argue that the result of a split preserves *readCondition*. This is why, *readCondition* requires all regions to only govern one address when the capability is linear. This means that after a split, the authority of the regions for the bottom half of the split can go to one capability and the remaining regions can go to the top half.

A safe read capability only gives authority to read safe words. The invariant on the memory a read capability gives access to may be even more restrictive than just requiring safe words. For instance, the invariant may require a flag to stay unchanged. We express the fact that a region may be more restrictive by making the standard region $\iota^{\text{std,p}}_{R(r),gc}$, which permits all memory segments with safe words, the upper bound of what a region may require when it governs a memory segment that can be accessed through a safe read capability.

Similarly to *readCondition*, we define a condition that captures the essence of what it means for a capability with writer permission to be safe. We call this condition *writeCondition*. A capability with write permission can be used to write to memory. The question is, what can we safely allow to be written to memory without any memory invariants being broken. The answer to this is anything - even words that are unsafe. Say, you manage to write something that can break memory invariants, then it would not be possible to read it

back again as write permission, generally speaking, does not entail read permission. If the capability had read permission, then *readCondition* would make sure that the word would have to be safe[8]. It should always be possible to write safe values, so we impose this as a lower bound.

A safe write capability must respect the memory invariant of the region that governs the memory the capability gives access to. Now consider the case, where the invariant permits two memory segments that differs in two or more addresses. In this case, the write capability cannot be used to transform the memory from one memory segment to the other because only one memory address can be updated at a time. If an adversary had such a capability, then it should be possible for them to transform the memory in a way that is consistent with the region. In other words, the adversarial code should be able to transform the memory segment to any memory segment permitted by the region. This is captured by address stratification (Definition 3.5.21) which basically says that if a region permits two memory segments, then all the intermediate memory segments you may end up with when transforming one memory segment to the other must be permitted as will.

Definition 3.5.21. We say that a region $\iota = (\_, H, \_)$ is address stratified iff

$$\begin{array}{l}\forall n, ms_S, ms_T, ms'_S, ms'_T, s, \hat{W}.\\ \quad (n,(ms_S, ms_T)), \left(n, \left(ms'_S, ms'_T\right)\right) \in H\ \hat{W} \wedge\\ \quad \mathrm{dom}(ms_S) = \mathrm{dom}(ms_T) = \mathrm{dom}(ms'_S) = \mathrm{dom}(ms'_T)\\ \quad \Rightarrow\\ \qquad \forall a \in \mathrm{dom}(ms_S). \left(n, (ms_S[a \mapsto ms'_S(a)], ms_T[a \mapsto ms'_T(a)])\right) \in H\ \hat{W}\end{array}$$

♦

With address stratification defined, we define the write condition.

Definition 3.5.22.

$$writeCondition^{\square,gc}(l, W) \stackrel{def}{=} \left\{ (n, A) \;\middle|\; \begin{array}{l}\exists S \subseteq addressable(l, W.\mathrm{heap}).\\ \quad \exists R : S \to \mathcal{P}(\mathbb{N})\\ \qquad \biguplus_{r \in S} R(r) \supseteq A \wedge\\ \qquad (l = \mathrm{linear} \Rightarrow \forall r \in S. |R(r)| = 1) \wedge\\ \qquad \forall r \in S. W.\mathrm{heap}(r). H \overset{n}{\supseteq} \iota^{\mathrm{std,p}}_{R(r),gc}. H \wedge\\ \qquad\quad W.\mathrm{heap}(r) \text{ is address-stratified}\end{array} \right\}$$

♦

[8] It should not be possible to obtain a capability that can be used to break invariants. After all, if such a capability was obtained, memory invariants could be broken. However, the *writeCondition* tries to capture the essence of safety and in principle it is safe to write an unsafe capability that cannot be read back.

The definition of *writeCondition* is very similar to *readCondition*. Support for split and splice is done in the same way, and the bound is defined in terms of the standard region.

The *readCondition* and *writeCondition* specifically uses the heap sub-world which means that it can only be used for heap capabilities. This means that we cannot use it for stack capabilities. To take care of stack capabilities, we define two more conditions a *stackReadCondition* and *stackWriteCondition*. The two new condition are essentially the same as the *readCondition* and *writeCondition* except that they use the free stack sub-world and assume that the capability is linear as all stack capabilities are linear. Note that we do not have any condition that talks about the stack-frames sub world because we should never have a capability that allows us to directly read from or write to that part of memory.

Definition 3.5.23.

$$stackReadCondition^{\square,gc}(W) = \left\{ (n,A) \;\middle|\; \begin{array}{l} \exists S \subseteq addressable(\text{linear}, W.\text{free\_stk}). \\ \quad \exists R : S \to \mathcal{P}(\mathbb{N}). \\ \qquad \biguplus_{r \in S} R(r) \supseteq A \wedge \\ \qquad \forall r \in S. |R(r)| = 1 \\ \qquad \forall r \in S. W.\text{free\_stk}(r).H \overset{n}{\subseteq} \iota^{\text{std,p}}_{R(r),gc}.H \end{array} \right\}$$

♦

Definition 3.5.24.

$$stackWriteCondition^{\square,gc}(W) = \left\{ (n,A)) \;\middle|\; \begin{array}{l} \exists S \subseteq addressable(\text{linear}, W.\text{free\_stk}). \\ \quad \exists R : S \to \mathcal{P}(\mathbb{N}) \\ \qquad \biguplus_{r \in S} R(r) \supseteq A \wedge \\ \qquad \forall r \in S. |R(r)| = 1 \wedge \\ \qquad \forall r \in S. W.\text{free\_stk}(r).H \overset{n}{\supseteq} \iota^{\text{std,p}}_{R(r),gc}.H \wedge \\ \qquad\quad W.\text{free\_stk}(r) \text{ is address-stratified} \end{array} \right\}$$

♦

The final permission, we define conditions for is the execute permission. We define two conditions *executeCondition* and *readXCondition*. The *executeCondition* captures what operations an execute-capability can be used for, i.e. execution. The *readXCondition* captures some additional read assumptions we can make on a capability when we know the capability is executable.

The *executeCondition* intuitively says that an execute capability is safe when any capability that can be derived from it is safe as a program counter now and in the future. We later define the $\mathcal{E}$-relation which captures what

it means for a word to be safe as a program counter, but for now it suffices to think of it as a program counter that causes an execution that does not break memory invariants. An executable capability can have its range of authority shrunk or its current address changed which changes what instructions are executed and thus potentially whether the code respects memory invariants. For this reason, the condition requires that any executable capability with a derived range of authority and a current address in that range is safe to use for execution. The *executeCondition* is quantified over all future worlds of the *sharedPart* of W. We do not know when the executable capability will be used, so it should be safe even in the future when the memory has changed. The *sharedPart* function turns the spatial regions of a world into shadow copies. This means that the capability cannot depend on linear capabilities and thus the contents of the stack. When we define the logical relation, we even require the executable capability to not be linear. Linear executable capabilities would likely not be useful because they cannot be moved from the pc-register without crashing the execution. This may sound like an ideal primitive for constructing something that can be executed once, however, most programs rely on loading other capabilities or seal sets using the program counter capability which is not possible when the program counter is linear.

$$executeCondition^{\square,gc}(W) = \left\{ (n,A) \;\middle|\; \begin{array}{l} \forall n' < n, W' \sqsupseteq sharedPart(W).\, \forall b', e'.\, \forall a \in [b', e'] \subseteq A. \\ \quad \left(n', \begin{pmatrix} ((\text{RX}, \text{normal}), b', e', a), \\ ((\text{RX}, \text{normal}), b', e', a) \end{pmatrix}\right) \in \mathcal{E}^{\square,gc}(W') \end{array} \right\}$$

The *readCondition* condition by itself allows many different regions and thus potentially many different memory segments. However, when we have a read capability with execute permission, we know that the capability must point to a piece of code memory. For this reason, we define the *readXCondition* to capture the additional assumptions that we can make when a capability is executable.

$$readXCondition^{\square,gc}(W) = \left\{ (n,A) \;\middle|\; \begin{array}{l} \exists r \in addressable(\text{normal}, W.\text{heap}), code. \\ \quad W.\text{heap}(r) \overset{n}{=} \iota^{\text{code}}_{\_,\_,code,gc} \wedge \\ \quad \text{dom}(code) \supseteq A \end{array} \right\}$$

The *readXCondition* requires that the memory segment an executable capability has authority over is governed by a code region. Note that we do not define *executeCondition* and *readXCondition* for the stack because the stack is not executable.

The *executeCondition* handles normal jumps, but it does not cover the case of `xjmp`. Executable capabilities can be used on their own whereas sealed capabilities must be jumped to in pairs. However, we do not need to consider

arbitrary pairs: given a sealed capability we only have to consider the capabilities permitted by the relevant seal invariant. Just like the $\mathcal{E}$ relation captures what it means for a word to be safe as a program counter, we later define $\mathcal{E}_{\text{xjmp}}$ that define what it means for a code and data capability pair to be safe together as program counter and data capability, respectively. A sealed capability is safe when it can be paired with any sealed capability from the seal invariant such that the pair is in the $\mathcal{E}_{\text{xjmp}}$ relation. Just like safe executable capabilities, a sealed capability may be stored, so it should also be safe to use in future worlds. The condition for sealed capabilities is defined by *sealedCondition*.

$$
\begin{aligned}
&\mathit{sealedCondition}^{\square,gc}(W, H_\sigma) = \\
&\left\{ (n, (\sigma, sc_S, sc_T)) \;\middle|\; \begin{array}{l} \forall W' \sqsupseteq W, W_o, n' < n, \left(n', \left(sc'_S, sc'_T\right)\right) \in H_\sigma\ \sigma\ \xi^{-1}(W_o). \\ \quad \left(n', sc_S, sc'_S, sc_T, sc'_T\right) \in \mathcal{E}^{\square,gc}_{\text{xjmp}}(W' \oplus W_o) \end{array} \right\}
\end{aligned}
$$

The untrusted value relation The untrusted value relation $\mathcal{V}_{\text{untrusted}}$ relates all the words that untrusted components can safely posses. That is words that cannot be used to break any memory invariants. The relation is displayed in Figure 3.21.

The untrusted value relation has five cases: data, capabilities, stack pointers, sealed capabilities, seal sets, and stack pointers. In the following, we will give some intuition about why it is safe to give these words to untrusted code as well motivate the conditions they are safe under.

The first case is data. Data grant no authority, so data is always safe. Further unlike capabilities, it is always possible to construct a new integer with the move instruction.

Next we have capabilities that do not have a special representation on oLCM, i.e. all capabilities but stack pointers and return pointers. For two capabilities to be related, they should be syntactically equal. That is, they should have the same range of authority, linearity and so one. Generally speaking, untrusted components should not have direct access to a trusted components code, so we require that capabilities must have a range of authority outside the trusted address space if they are to be related. The safety of a capability also depends on the world and whether the capability can be used to break the memory invariants of the world. For instance, if a capability has read-permission, then it should not be possible to read something unsafe, i.e. something that can break memory invariants. This condition and conditions for the other permissions are captured by the permission based conditions, so we use them to express the necessary conditions. That is, if a capability has read permission, then *readCondition* must be satisfied, if it has write permission, then *writeCondition* must be satisfied, and if it has execute permission, then *executeCondition* and *readXCondition* must be satisfied. If the capability has execute permission, then it must also be a normal ca-

pability. Finally, the capability cannot have read/write/execute permission because that would break the write-XOR-execute assumption, i.e. the code memory in non-writable and data memory is non-executable.

Stack pointers on oLCM are represented with the special token stack-ptr(p,b,e,a). The corresponding capability on LCM is a linear capability with the same permission and addresses. We assume that the stack is non-executable, so the permission for a stack pointer cannot have execute permission. Similarly to the normal capabilities, we use the permission based conditions for the stack to ensure that the stack capability is safe to use.

A sealed capability encapsulates the authority of the underlying capability, and the authority is only released when the sealed capability is used in an `xjmp`. The `xjmp` takes a pair of sealed capabilities, so the authority of a sealed capability depends on what other sealed capabilities it might be used with. The seal invariant specifies the capabilities that may be sealed with a given seal and thus the capabilities that may be used together as a sealed pair. As discussed, closure and return seals must be used in specific ways which is captured in the code region seal invariant. In order for a pair of oLCM and LCM sealed capabilities to be in the untrusted value relation, they must be sealed with the same seal σ and related in the appropriate seal invariant. Further, they should satisfy the *sealedCondition* which means that they can safely be paired up with any other pair of capabilities from the seal invariant and used safely for execution.

For sets of seals to be related related in the untrusted relation they must be syntactically equal. Further, the seals in the set should be disjoint from the return seals and trusted closure seals ($\overline{\sigma_{\text{glob\_ret}}}$ and $\overline{\sigma_{\text{glob\_clos}}}$) because the trusted code relies on having the sole access to them. We do not know what an adversary may seal or what seal they may use, so, for every seal in the seal set, we require the seal invariant to be essentially equal to the untrusted value relation.

When we give a word to untrusted code, we can make no assumptions on when they will use it. For instance, they may store it in memory and use it in a later call. This means that a safe word must not only be safe now but also at any point in the future. The untrusted value relation ensures this as it is monotone with respect to future worlds.

Lemma 3.5.25 (Untrusted value relation monotonicity)**.** *For all integers n, words w_1 and w_2, and worlds $W' \sqsupseteq W$, if $(n,(w_1,w_2)) \in \mathcal{V}^{\square,gc}_{\text{untrusted}}(W)$, then $(n,(w_1,w_2)) \in \mathcal{V}^{\square,gc}_{\text{untrusted}}(W')$.* ◊

The trusted value relation The trusted value relation $\mathcal{V}_{\text{trusted}}$ relates everything safe for a trusted component to have without breaking memory invariants. For the most part, we allow them to contain the same words as the untrusted components, but we also need to allow them to have seal sets with trusted closure seals and return seals which we cannot allow untrusted

$$
\begin{array}{l}
\mathcal{V}^{\square,gc}_{\text{untrusted}}(W) = \{(n,(i,i)) \mid i \in \mathbb{Z}\} \cup \\
\left\{ \left(n, \begin{pmatrix} ((p,l),b,e,a), \\ ((p,l),b,e,a) \end{pmatrix}\right) \;\middle|\; \begin{array}{l} [b,e] \# T_A \wedge \\ p \in \mathit{readAllowed} \\ \quad \Rightarrow (n,[b,e]) \in \mathit{readCondition}^{\square,gc}(l,W) \wedge \\ p \in \mathit{writeAllowed} \\ \quad \Rightarrow (n,[b,e]) \in \mathit{writeCondition}^{\square,gc}(l,W) \wedge \\ p \neq \text{RWX} \wedge \\ p = \text{RX} \Rightarrow \left\{ \begin{array}{l} (n,[b,e]) \in \mathit{executeCondition}^{\square,gc}(W) \wedge \\ (n,[b,e]) \in \mathit{readXCondition}^{\square,gc}(W) \wedge \\ l = \text{normal} \end{array} \right. \end{array} \right\} \cup \\
\left\{ \left(n, \begin{pmatrix} \text{stack-ptr}(p,b,e,a), \\ ((p,\text{linear}),b,e,a) \end{pmatrix}\right) \;\middle|\; \begin{array}{l} p \notin \{\text{RX},\text{RWX}\} \wedge \\ p \in \mathit{readAllowed} \\ \quad \Rightarrow (n,[b,e]) \in \mathit{stackReadCondition}^{\square,gc}(W) \wedge \\ p \in \mathit{writeAllowed} \\ \quad \Rightarrow (n,[b,e]) \in \mathit{stackWriteCondition}^{\square,gc}(W) \end{array} \right\} \cup \\
\left\{ \left(n, \begin{pmatrix} \text{sealed}(\sigma, sc_S), \\ \text{sealed}(\sigma, sc_T) \end{pmatrix}\right) \;\middle|\; \begin{array}{l} (\mathit{isLinear}(sc_S) \text{ iff } \mathit{isLinear}(sc_T)) \wedge \\ \exists r \in \text{dom}(W.\text{heap}), \overline{\sigma_{\text{ret}}}, \overline{\sigma_{\text{clos}}}, ms_{\text{code}}. \\ \quad W.\text{heap}(r) = (\text{shared}, \_, H_\sigma) \wedge \\ \quad H_\sigma\ \sigma \overset{n}{=} H^{\text{code},\square}_\sigma\ \overline{\sigma_{\text{ret}}}\ \overline{\sigma_{\text{clos}}}\ ms_{\text{code}}\ gc\ \sigma \wedge \\ \quad (n',(sc_S,sc_T)) \in H_\sigma\ \sigma\ \xi^{-1}(W) \text{ for all } n' < n \wedge \\ \quad \mathit{isLinear}(sc_S) \\ \qquad \Rightarrow (n,(\sigma,sc_S,sc_T)) \in \mathit{sealedCondition}^{\square,gc}(W,H_\sigma) \wedge \\ \quad \mathit{nonLinear}(sc_S) \\ \qquad \Rightarrow (n,(\sigma,sc_S,sc_T)) \in \mathit{sealedCondition}^{\square,gc}(\mathit{purePart}(W),H_\sigma) \end{array} \right\} \cup \\
\left\{ \left(n, \begin{pmatrix} \text{seal}(\sigma_b,\sigma_e,\sigma), \\ \text{seal}(\sigma_b,\sigma_e,\sigma) \end{pmatrix}\right) \;\middle|\; \begin{array}{l} [\sigma_b,\sigma_e] \# (\overline{\sigma_{\text{glob\_ret}}} \cup \overline{\sigma_{\text{glob\_clos}}}) \wedge \\ \forall \sigma' \in [\sigma_b,\sigma_e]. \exists r \in \text{dom}(W.\text{heap}). \\ \quad W.\text{heap}(r) = (\text{shared}, \_, H_\sigma) \wedge H_\sigma\ \sigma' \overset{n}{=} (\mathcal{V}^{\square,gc}_{\text{untrusted}} \circ \xi) \end{array} \right\}
\end{array}
$$

Figure 3.21: The untrusted value relation relates all the words on oLCM to all the words on LCM that are safe for non-trusted components to posses.

components to have. Further, we need to allow trusted components to have capabilities for the trusted code which, again, is something that we cannot allow untrusted components to have. The untrusted value relation is defined in Figure 3.22.

The words in $\mathcal{V}_{\text{trusted}}$ but not in the $\mathcal{V}_{\text{untrusted}}$ have the potential to break the system invariants StkTokens rely on. We can only let trusted components have words from $\mathcal{V}_{\text{trusted}}$ because the trusted component promises to not break the invariant by behaving reasonably. This promise is expressed formally in $\mathcal{V}_{\text{trusted}}$ by requiring the presence of a code region in both cases specific to $\mathcal{V}_{\text{trusted}}$. As explained previously, the code region essentially captures the requirements put on components by well-formedness and the reasonability condition which constitutes the promise to use seals and trusted code pointers in a way that does not break invariants.

The trusted closure seals and return seals serve a specific purpose, namely they must be used for return pointers and closures. To make sure this is the case, there must be a code region in the world that governs the code. The code region contains a seal invariant that makes sure that the seals are only used for their intended purpose. This is why the trusted value relation only relates seal sets of trusted closure seals and return seals when the world contains an appropriate code region.

Two capabilities are related as trusted code pointers if they are normal, has a permission derivable from read-execute, and has a range of authority within the trusted address space, T_A. Further, we need to know that the capabilities actually point to a piece of code which is why we require the *readXCondition* to be satisfied. This makes sure that the region that governs the memory the capability points to is a code region. Note that even though the capability has read permission, we do not require the read condition to hold. The code memory contains trusted closure seals and return seals that we cannot let untrusted code have and the read condition requires everything to be in $\mathcal{V}_{\text{untrusted}}$, so the read condition would not hold. However, trusted code can have access to such seals because we expect the trusted code to treat the seals reasonably. Like the untrusted value relation, the trusted value relation is monotone. Intuitively, the two relations are monotone for the same reason; words are potentially used at any point in time. If words are safe now (in the current world), then they should also be safe to use later (in any possible future world).

Lemma 3.5.26 (Trusted value relation monotonicity)**.** *For all integers n, words w_1 and w_2, and worlds $W' \sqsupseteq W$, if $(n,(w_1,w_2)) \in \mathcal{V}^{\square,gc}_{\text{trusted}}(W)$, then $(n,(w_1,w_2)) \in \mathcal{V}^{\square,gc}_{\text{trusted}}(W')$.* ◊

Another, perhaps unsurprising property, of the trusted value relation it that non-linear words do not depend on the spatial regions that may be in

$$
\begin{aligned}
&\mathcal{V}^{\square,gc}_{\text{trusted}}(W) = \\
&\quad \mathcal{V}^{\square,gc}_{\text{untrusted}}(W) \cup \\
&\quad \left\{ \left(n, \begin{pmatrix} \text{seal}(\sigma_b, \sigma_e, \sigma), \\ \text{seal}(\sigma_b, \sigma_e, \sigma) \end{pmatrix} \right) \;\middle|\; \begin{array}{l} gc = (T_A, \text{stk\_base}, \overline{\sigma_{\text{glob\_ret}}}, \overline{\sigma_{\text{glob\_clos}}}) \wedge \\ \exists r \in \text{dom}(W.\text{heap}). \\ \quad W.\text{heap}(r) \overset{n}{=} \iota^{\text{code}}_{\overline{\sigma_{\text{ret}}}, \overline{\sigma_{\text{clos}}}, code, gc} \wedge \\ \quad \text{dom}(code) \subseteq T_A \wedge [\sigma_b, \sigma_e] \subseteq (\overline{\sigma_{\text{ret}}} \cup \overline{\sigma_{\text{clos}}}) \wedge \\ \quad \overline{\sigma_{\text{ret}}} \subseteq \overline{\sigma_{\text{glob\_ret}}} \wedge \overline{\sigma_{\text{clos}}} \subseteq \overline{\sigma_{\text{glob\_clos}}} \end{array} \right\} \cup \\
&\quad \left\{ \left(n, \begin{pmatrix} ((p, \text{normal}), b, e, a), \\ ((p, \text{normal}), b, e, a) \end{pmatrix} \right) \;\middle|\; \begin{array}{l} p \sqsubseteq \text{RX} \wedge \\ gc = (T_A, \text{stk\_base}, \overline{\sigma_{\text{glob\_ret}}}, \overline{\sigma_{\text{glob\_clos}}}) \wedge \\ [b, e] \subseteq T_A \wedge \\ (n, [b, e]) \in \mathit{readXCondition}^{\square,gc}(W) \end{array} \right\}
\end{aligned}
$$

Figure 3.22: The trusted value relation $\mathcal{V}_{\text{trusted}}$ relates all the words that are safe for trusted components to contain. A trusted component may contain untrusted words (Figure 3.21), return seals and trusted closure seals, and code pointers for trusted code.

the world. This is unsurprising as normal capabilities do not necessarily reference memory uniquely.

Lemma 3.5.27 (Non-linear words are independent of spatial regions). *If* $(n, (w_1, w_2)) \in \mathcal{V}^{\square,gc}_{\text{trusted}}(W)$ *and either* $\mathit{nonLinear}(w_1)$ *or* $\mathit{nonLinear}(w_2)$*, then*

$$(n, (w_1, w_2)) \in \mathcal{V}^{\square,gc}_{\text{trusted}}(\mathit{sharedPart}(W)). \quad \Diamond$$

This is similar to Lemma 3.5.27 for the untrusted value relation.

Register file relation

The register file relation relates oLCM register files to LCM register files. Intuitively, two register files are related when they only contain safe words, i.e. words from the value relation. This raises the question "which value relation?" We only use the register file relation to relate register files for components we do not trust, so the answer is the untrusted value relation. The definition of the register-file relation is straightforward. It distributes the authority of the world among the registers and requires each of the registers to contain a safe word with respect to the authority it is given. The register file never takes into account the pc and it can leave out further registers. We use this to not relate register content that will be overwritten anyway. We write $\mathcal{R}(W)$ to mean $\mathcal{R}(\emptyset)(W)$. That is, if we do not need to exclude additional registers, then we simply omit that argument. The register file relation is defined in Figure 3.23.

$$\mathcal{R}^{\square,gc}(R)(W) = \left\{ (n,(reg_S, reg_T)) \;\middle|\; \begin{array}{l} \exists S : (\text{RegName} \setminus (\{\text{pc}\} \cup R)) \to \text{World}. \\ \quad W = \bigoplus_{r \in (\text{RegName} \setminus (\{\text{pc}, \text{r}_{\text{data}}\} \cup R))} S(r) \wedge \\ \quad \forall r \in \text{RegName} \setminus (\{\text{pc}\} \cup R). \\ \quad\quad (n,(reg_S(r), reg_T(r))) \in \mathcal{V}^{\square,gc}_{\text{untrusted}}(S(r)) \end{array} \right\}$$

Figure 3.23: The register file relation relates register files. Two register files are related when their content is related.

Expression relations

The expression relation $\mathcal{E}$ defines when two capabilities can be used in the pc-register to produce related executions, i.e. the capabilities can be used to construct configurations in the observation relation. The $\mathcal{E}$ relation can be used to reason about the safety of an executable capability, i.e. capabilities that can change the control flow during execution when a `jmp` instruction is executed. In the setting of oLCM and LCM, we can also use sealed capabilities to change the control flow by using the `xjmp` instruction. The `xjmp` instruction updates the pc register and the r_{data} register, however, the $\mathcal{E}$ relation only updates the *pc*-register, so we cannot use $\mathcal{E}$ to reason about sealed capabilities. Instead, we define the relation $\mathcal{E}_{\text{xjmp}}$ which relates two pairs of capabilities when they are safe to use with the `xjmp` instruction.

Executions are related when the observable effect of the executions are permissible. The permissible observations are defined by the observation relation, so we define the expression relation in terms of the observation relation. However, the observation relation relates configurations, not capabilities. We lift the capabilities to configurations simply by plugging the two capabilities into the pc-register of two configurations. We cannot pick arbitrary configurations because an arbitrary configuration may contain words that can be used to break memory invariants and thus create unacceptable observable effects. Instead, we need to pick configurations made out of related components, i.e. related register files and related memories that respect linearity. The type of execution captured by $\mathcal{E}$ corresponds to a normal jump. When a `jmp` r instruction is interpreted, the pc-register is replaced with the contents of register r, i.e. the current configuration is plugged with a new pc. The $\mathcal{E}$ relation is defined in Figure 3.24

The $\mathcal{E}_{\text{xjmp}}$ relation looks very much like the $\mathcal{E}$ relation. It takes related memories and register files (ignoring the r_{data} register) and combines them into two configurations. Each of the configurations are plugged with a code capability and a data capability just like the `xjmp` instruction would do it and requires the resulting configurations to be in the $\mathcal{O}$ relation. The $\mathcal{E}_{\text{xjmp}}$ relation is defined in Figure 3.24.

$$\mathcal{E}^{\Box,gc}(W) =$$
$$\left\{ (n,(v_{c,S},v_{c,T})) \;\middle|\; \begin{array}{l} \forall n' \leq n, reg_S, reg_T, ms_S, ms_T, ms_{stk}, stk. \\ \quad \forall W_{\mathcal{R}}, W_{\mathcal{M}}. \\ \quad\quad (n',(reg_S,reg_T)) \in \mathcal{R}^{\Box,gc}(W_{\mathcal{R}}) \wedge \\ \quad\quad ms_S, stk, ms_{stk}, ms_T \overset{gc}{:}_{n'} W_{\mathcal{M}} \\ \quad\quad \Phi_S = (ms_S, reg_S, stk, ms_{stk}) \\ \quad\quad \Phi'_S = \Phi_S[reg.\mathrm{pc} \mapsto v_{c,S}] \\ \quad\quad \Phi_T = (ms_T, reg_T) \\ \quad\quad \Phi'_T = \Phi_T[reg.\mathrm{pc} \mapsto v_{c,T}] \\ \quad\quad W \oplus W_{\mathcal{R}} \oplus W_{\mathcal{M}} \\ \quad\quad \Rightarrow \left(n', \left(\Phi'_S, \Phi'_T\right)\right) \in \mathcal{O}^{\Box,gc} \end{array} \right\}$$

$$\mathcal{E}^{\Box,gc}_{\mathrm{xjmp}}(W) =$$
$$\left\{ (n,(v_{c,S},v_{d,S},v_{c,T},v_{d,T})) \;\middle|\; \begin{array}{l} \forall n' \leq n, reg_S, reg_T, ms_S, ms_T, ms_{stk}, stk. \\ \quad \forall W_{\mathcal{R}}, W_{\mathcal{M}}. \\ \quad\quad (n',(reg_S,reg_T)) \in \mathcal{R}^{\Box,gc}(\{\mathrm{r}_{\mathrm{data}}\})(W_{\mathcal{R}}) \wedge \\ \quad\quad ms_S, stk, ms_{stk}, ms_T \overset{gc}{:}_{n'} W_{\mathcal{M}} \wedge \\ \quad\quad \Phi_S = (ms_S, reg_S, stk, ms_{stk}) \wedge \\ \quad\quad \Phi_T = (ms_T, reg_T) \wedge \\ \quad\quad W \oplus W_{\mathcal{R}} \oplus W_{\mathcal{M}} \text{ is defined} \\ \quad\quad \Rightarrow \exists \Phi'_S, \Phi'_T. \\ \quad\quad\quad \Phi'_S = xjumpResult(v_{c,S}, v_{d,S}, \Phi_S) \wedge \\ \quad\quad\quad \Phi'_T = xjumpResult(v_{c,T}, v_{d,T}, \Phi_T) \wedge \\ \quad\quad\quad \left(n', \left(\Phi'_S, \Phi'_T\right)\right) \in \mathcal{O}^{\Box,gc} \end{array} \right\}$$

Figure 3.24: The expression relation relates capabilities capabilities that can safely be used for execution. The xjmp expression relation can be used to relate capabilities that are safe as sealed capabilties.

3.5.3 Fundamental Theorem

An important lemma in our proof of full abstraction of the embedding of oLCM into LCM, is the fundamental theorem of logical relations (FTLR). The name indicates that it is an instance of a general pattern in logical relations proofs, but is otherwise unimportant.

Theorem 3.5.28 (FTLR)**.** *For all n, W, l, b, e, a, If*

- $(n, [b, e]) \in readXCondition^{\square, gc}(W)$

and one of the following sets of requirements holds:

- $[b, e] \subseteq T_A$ *and* $(((\text{RX}, \text{normal}), b, e, a)$ *behaves reasonably up to n steps (see Section 3.4.2).*
- $[b, e] \# T_A$

then

$$(n, (((\text{RX}, \text{normal}), b, e, a), ((\text{RX}, \text{normal}), b, e, a))) \in \mathcal{E}^{\square, gc}(W) \qquad \Diamond$$

Roughly speaking, this lemma says that under certain conditions, executing any executable capability under oLCM and LCM semantics will produce the same observable behavior. The conditions require that the capability points to a memory region where code is loaded and that code must be either trusted and behave reasonably (i.e. respect the restrictions that StkTokens relies on, see Section 3.4.2) or untrusted (in which case, it cannot have WBCF or LSE expectations, see Section 3.4.2).

The proof of the lemma consists of a big induction where each possible instruction is proven to behave the same in source and target in related memories and register files. After that first step, the induction hypothesis is used for the rest of the execution.

3.5.4 Related Components

In order to show full abstraction (Theorem 3.4.8), we need not only to relate the words on oLCM with words on LCM we also need to relate oLCM components with LCM components. Specifically, we say that two components are related when they are syntactically equal, after all, a oLCM component is in some sense the same as a LCM as we only see the difference during execution when a call happens and when we lift a component to a configuration where we need to introduce a stack pointer. However, we cannot take arbitrary components as they could potentially break memory invariants. For related base components, we require that if the imports are satisfied with related words, then the resulting memory should be safe. Further, related components must have safe exports. Components with a main are related when the base components are related and the main capabilities are in the public interface, that is they must be in the exports.

Definition 3.5.29 (Component relation). $\mathrm{C}^{\square,gc}(W) =$

$$\left\{ \begin{array}{l} (n, comp, comp) \mid \\ \quad comp = (ms_{\text{code}}, ms_{\text{data}}, \overline{a_{\text{import}} \hookleftarrow s_{\text{import}}}, \overline{s_{\text{export}} \mapsto w_{\text{export}}}, \overline{\sigma_{\text{ret}}}, \overline{\sigma_{\text{clos}}}) \text{ and} \\ \quad \text{For all } W' \sqsupseteq W. \\ \quad\quad \text{If } \overline{\left(n', (w_{\text{import}}, w_{\text{import}})\right)} \in \mathcal{V}^{\square,gc}_{\text{untrusted}}(sharedPart(W')) \text{ for all } n' < n \\ \quad\quad \text{and } ms'_{\text{data}} = ms_{\text{data}}[\overline{a_{\text{import}} \mapsto w_{\text{import}}}] \\ \quad\quad \text{then } \left(n, (\overline{\sigma_{\text{ret}}} \uplus \overline{\sigma_{\text{clos}}}, ms_{\text{code}} \uplus ms'_{\text{data}}, ms_{\text{code}} \uplus ms'_{\text{data}})\right) \\ \quad\quad\quad \in \mathcal{H}(W.\text{heap})(W') \text{ and} \\ \quad\quad \overline{\left(n, (w_{\text{export}}, w_{\text{export}})\right)} \in \mathcal{V}^{\square,gc}_{\text{untrusted}}(sharedPart(W')) \end{array} \right\}$$

$$\cup \left\{ \begin{array}{l} (n, (comp_0, c_{\text{main},c}, c_{\text{main},d}), (comp_0, c_{\text{main},c}, c_{\text{main},d})) \mid \\ \quad (n, (comp_0, comp_0)) \in \mathcal{C}^{\square,gc}(W) \text{ and} \\ \quad \{(\_ \mapsto c_{\text{main},c}), (\_ \mapsto c_{\text{main},d})\} \subseteq \overline{w_{\text{export}}} \end{array} \right\}$$

♦

3.5.5 Related Execution Configuration

The full abstraction theorem (Theorem 3.4.8) is stated in terms of contextual equivalence. Contextual equivalence (Definition 3.4.7) plugs two components into a context and requires equitermination of the resulting executable configurations. This means that we need to lift relatedness one step further than the components, namely to the level of execution configurations. To this end, we define $\mathcal{EC}$.

Definition 3.5.30 (Related execution configuration).

$$\mathcal{EC}^{\square,gc}(W) = \left\{ \begin{array}{l} (n, ((ms_S, reg_S, stk, ms_{stk}), (ms_T, reg_T)) \mid \\ \quad gc = (T_A, \text{stk\_base}) \wedge \\ \quad \exists W_M, W_R, W_{\text{pc}}.\ W = W_M \oplus W_R \oplus W_{\text{pc}} \wedge \\ \quad (n, ((reg_S(\text{pc}), reg_S(\text{r}_{\text{data}})), (reg_T(\text{pc}), reg_T(\text{r}_{\text{data}})))) \in \mathcal{E}^{\square,gc}_{\text{xjmp}}(W_{\text{pc}}) \wedge \\ \quad reg_S(\text{pc}) \neq \text{ret-ptr-code}(\_) \wedge reg_S(\text{r}_{\text{data}}) \neq \text{ret-ptr-data}(\_) \wedge \\ \quad nonExec(reg_S(\text{r}_{\text{data}})) \wedge nonExec(reg_T(\text{r}_{\text{data}})) \\ \quad ms_S, ms_{stk}, stk, ms_T :^{gc}_{n} W_M \wedge \\ \quad (n, (reg_S, reg_T)) \in \mathcal{R}^{\square,gc}(\{\text{r}_{\text{data}}\})(W_R) \end{array} \right\}$$

♦

Definition 3.5.30 essentially says that two executable configurations are related when they are made out of related components. That is, the authority of the world must be distributed such that the code and data pointer pairs are safe for execution, i.e. the contents of the pc and r_{data} registers are related by the $\mathcal{E}_{xjmp}$ relation. Further, the two memories and the two register files should be related. This means that the executable configuration only contains words that respect memory invariants.

3.5.6 Full Abstraction Proof Sketch

Using Lemma 3.5.28, we can now proceed to proving Theorem 3.4.8 (full abstraction).

Using Lemma 3.5.28 and the definitions of the logical relations, we can then prove the following two lemmas. The first is a version of the FTLR for components, stating that all components are related to themselves if they are either (1) well-formed and untrusted or (2) well-formed, reasonable and trusted.

Lemma 3.5.31 (FTLR for components)**.** *If comp is a well-formed component, i.e.* $\vdash$ *comp and either* $\mathrm{dom}(comp.ms_{\mathrm{code}}) \subseteq T_A$ *and comp is a reasonable component; or* $\mathrm{dom}(comp.ms_{\mathrm{code}}) \# T_A$*, then there exists a W such that* $(n,(comp,comp)) \in \mathcal{C}^{\square,gc}(W)$. ◊

Another lemma then relates the component relation and context plugging: plugging related components into related contexts produces related execution configurations.

Lemma 3.5.32. *If* $(n,(\mathscr{C}_S,\mathscr{C}_T)) \in \mathcal{C}^{\square,gc}(W_1)$ *and* $(n,(comp_S,comp_T)) \in \mathcal{C}^{\square,gc}(W_2)$ *and* $W_1 \oplus W_2$ *is defined, then* $\mathscr{C}_S[comp_S]$ *terminates iff* $\mathscr{C}_T[comp_T]$ *terminates.* ◊

Finally, we use these two lemmas to prove Theorem 3.4.8.

Proof of Theorem 3.4.8. Both directions of the proof are similar, so we only show the right direction. To show the LCM contextual equivalence, assume w.l.o.g a well-formed context $\mathscr{C}$ such that $\mathscr{C}[comp_1]\Downarrow$. The proof is sketched in Figure 3.25. By the statement of Theorem 3.4.8, we may assume that the trusted components $comp_1$ and $comp_2$ are well-formed and reasonable. We prove arrow (1) in the figure by using the mentioned assumptions about $comp_1$ and $\mathscr{C}$ along with Lemma 3.5.31 and 3.5.32. Now we know that $\mathscr{C}[comp_1]\Downarrow$, so by the assumption that $comp_1$ and $comp_2$ are contextually equivalent on oLCM we get $\mathscr{C}[comp_2]\Downarrow$, i.e. arrow (2) in the figure. To prove arrow (3), we again apply Lemma 3.5.31, 3.5.32; but this time, we use the assumption that $comp_2$ is well-formed and reasonable and that $\mathscr{C}$ is well-formed. □

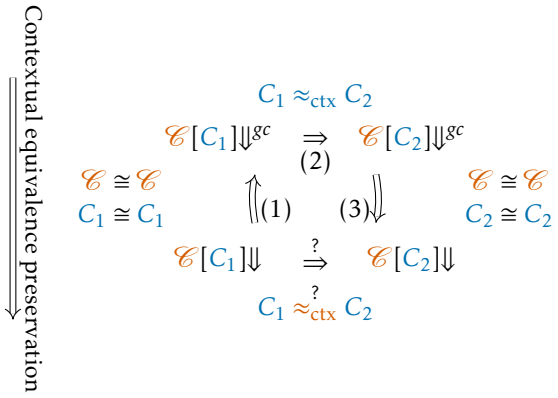


Figure 3.25: Proving one direction of fully abstract compilation (contextual equivalence preservation).

3.6 Discussion

3.6.1 Full Abstraction

Our formulation of WBCF and LSE using a fully abstract overlay semantics has an important advantage with respect to others. Imagine that you are implementing a fully abstract compiler for a high-level language, i.e. a secure compiler that enforces high-level abstractions when interacting with untrusted target-language components. Such a compiler would need to perform many things and enforce other high-level properties than just WBCF and LSE.

If such a compiler uses the StkTokens calling convention, then the security proof should not have to reprove security of StkTokens. Ideally, it should just combine security proofs for the compiler's other functionality with our results about StkTokens. We point out that our formulation enables such reuse. Specifically, the compiler could be factored into a part that targets oLCM, followed by our embedding into LCM. If the authors of the secure compiler can prove full abstraction of the first part (relying on WBCF and LSE in oLCM) and they can also prove that this first part generates well-formed and reasonable components, then full abstraction of the whole compiler follows by our result and transitivity of fully abstract compilation. Perhaps other reusable components of secure compilers could be formulated similarly using some form of fully abstract overlay semantics, to obtain similar reusability of their security proofs.

A compiler is secure when it enforces the properties of high-level languages which begs the question what properties should we enforce. When it comes to fully-abstract compilation, then the answer is that all the properties of the high-level language should be enforced, so the real question is what high-level language we would like. StkTokens ensures a standard call-return

control-flow, but if we want a different kind of control-flow, for instance call/CC, then we need to come up with a different enforcement scheme. Further, many high-level languages have exceptions which is yet another form of control-flow which is also not supported by StkTokens. This goes to show how we must consider what high-level language we want in order to answer the question of what properties we must enforce to get a fully-abstract compiler.

We have not investigated support for continuations and exceptions in StkTokens thoroughly but we expect such support could be added. For exceptions, one approach would let callers provide callees with an additional capability for exceptional returns. This second capability would be similar to the code part of the return capability pair and signed with the same seal. The callee would be able to invoke it to signal that an exception has been thrown after which the caller's code would handle the exception, either by executing an exception handler or by unwinding its stack frame and passing the exception on to its own caller. Essentially, this would mean every function would be made responsible for unwinding its own stack frame. Continuations are more complicated but could perhaps be treated using similar ideas. Alternatively, it might also be possible to have a piece of central trusted code that does the stack unravelling for all stack frames. To do this, this the trusted code would need to receive a copy of all return seals from the linker.

Full-abstraction is a property of the whole language. In other words, a full-abstraction proof must consider all features of a language to make sure that the features don't interact in a way that breaks language abstractions. If we have a fully abstract compiler and add a feature to the source and target language, then the new compiler is no necessarily fully-abstract. Full abstraction would have to be proven for the new compiler to make sure that the new feature does not break existing abstractions in the language. Our proof of full-abstraction for StkTokens targets a simple capability machine that may not be able to enforce the high-level language abstractions we want, e.g. address hiding. In other words, the full-abstraction proof cannot be reused immediately. However, StkTokens is still a good candidate for the enforcement mechanism for well-bracketed control-flow and local-state encapsulation in a real fully abstract compiler. Generally speaking, it is worth investigating enforcement mechanisms for full abstraction in a simple setting that allows to quickly try out ideas and verify that the enforcement works.

Our full-abstraction theorem, Theorem 3.4.8, is not pure full-abstraction as it requires the components to be reasonable and well-formed. In other words, if we were to define a compiler phase that targets oLCM, then we would also have to show that every program it generates is well-formed and reasonable in order to use the full-abstraction result. Without the reasonability constraint, StkTokens would have to enforce reasonability instead. That is, StkTokens would have to dynamically ensure that no return seals or means to get return seals are passed in calls. Essentially, such checks

would protect the trusted code against itself which shouldn't be necessary in the first place. Instead, the compiler should generate reasonable code that never exhibits the unreasonable behaviour. This should be done in a compiler phase where more information about the original program is available, e.g. the compilation phase that commits to the low-level translation. Similarly for the syntactic constraints given by well-formedness. The compiler should make sure to generate code that satisfies the well-formedness judgement, so it can be executed by the machine.

One challenge in full-abstraction proofs is to relate the translation of program to the program it was translated from. Such a relation is often expressed as a back-translation [34], i.e. a translation from the target language to the source language. When we use an overlay semantics, the back-translation becomes trivial because the source and target language are syntactically the same, so the identity can be used as the back-translation. If we have native call and return instructions in the source language, then the source language would be different from the target language, and we would have to use a non-trivial back-translation. Specifically, the back-translation would need to distinguish sequences of instructions that is the translation of a call from sequences of instructions that just look like a call. With overlay semantics, this is not a concern because everything that looks like a call is interpreted as a call.

3.6.2 Practical Applicability

We believe there are good arguments for practical applicability of StkTokens. The strong security guarantees are proven in a way that is reusable as part of a bigger proof of compiler security. Its costs are

- a constant and limited amount of checks on every boundary crossing.
- possibly a small memory overhead because stack frames must be of non-zero length

The main caveat is that we rely on the assumption that capability machines like CHERI can be extended with linear capabilities in an efficient way.

Although this assumption can only be discharged by demonstrating an actual implementation with efficiency measurements, the following notes are based on private discussions with people from the CHERI team as well as our own thoughts on the matter. As we understand it, the main problems to solve for adding linear capabilities to a capability machine like CHERI are related to the move semantics for instructions like `move`, `store` and `load`. Processor optimizations like pipelining and out-of-order execution rely on being able to accurately predict the registers and memory that an instruction will write to and read from. Our instructions are a bit clumsy from this point-of-view because, for example, `move` or `store` will zero the source register

resp. memory location if the value being written is linear. A solution for this problem could be to add separate instructions for moving, storing and loading linear registers at the cost of additional opcode space. Adding splice and split will also consume some opcode space.

Another problem is caused by the move semantics for `load` in the presence of multiple hardware threads. In this setting, zeroing out the source memory location must happen atomically to avoid race conditions where two hardware threads end up reading the same linear capability to their registers. This means that a `load` of a linear capability should behave atomically, similar to a primitive compare-and-swap instruction. This is in principle not a problem except that atomic instructions are significantly slower than a regular `load` (on the order of 10x slower or more). When using StkTokens, loads of linear capabilities happen only when a thread has stored its return data capability on the stack and loads it back from there after a return. Because the stack is a region of memory with very high thread affinity (no other hardware thread should access it, in principle), and which is accessed quite often, well-engineered caching could perhaps reduce the high overhead of atomic loads of linear capabilities. The processor could perhaps also (be told to) rely on the fact that race conditions should be impossible for loads from linear capabilities (which should in principle be non-aliased) and just use a non-atomic load in that case.

Programming languages with a C-like calling convention often allow programs to pass stack references in calls. StkTokens supports stack references but with a couple of caveats. First of all, the stack capability is linear, so all references to the stack have to be linear. This means that the callee has to treat references linearly. Next, like the stack capability, the stack references must be given back to the caller on return, so they can reconstruct their original stack capability (which allows them to return). Finally, the encapsulated local stack frame should be a continuous piece of memory (because it has to be addressable by a single capability: the data part of the return capability pair). Because of this, stack-allocated objects for which references are passed to callees must be allocated at the top or bottom of the caller's stack frame. An escape analysis could be used to statically determine where to put allocations and, in principle, the allocations could be reordered dynamically before a call. In summary, support for passing stack references as arguments to callees could be added to StkTokens, but this would probably require some changes in the compiler and, more importantly, would require the callee to take special care when manipulating such references. We are unsure whether it is realistic to apply this approach for existing C code.

3.7 Related Work

In this section, we discuss related work on securely enforcing control flow correctness and/or local state encapsulation or the linear capabilities we use to do it. We do not repeat the work we discussed in Section 3.1.

Capability machines originate with Dennis and Van Horn [32] and we refer to Levy [60] and N. M. Watson et al. [69] for an overview of previous work. The capability machine formalized in Section 3.2 is modelled after CHERI [69, 100]. This is a recent, relatively mature capability machine which combines capabilities with a virtual memory approach in the interest of backwards compatibility and gradual adoption. For simplicity, we have omitted features of CHERI that were not needed for StkTokens (e.g. local capabilities, virtual memory).

Plenty of other papers enforce well-bracketed control flow at a low level but most are restricted to preventing particular types of attacks and enforce only partial correctness of control flow. This includes particularly the line of work on control-flow integrity [10]. This technique prevents certain classes of attacks by sanitizing addresses before direct and indirect jumps based on static control graph information and a protected shadow stack. Contrary to StkTokens, CFI can be implemented on commodity hardware rather than capability machines. However, its attacker model is different, and its security goals are weaker. They assume an attacker that is unable to execute code but can overwrite arbitrary data at any time during execution (to model buffer overflows). In terms of security goals, the technique does not enforce local stack encapsulation. Also, it only enforces a weak form of control flow correctness saying that jumps stay within the program's static control flow graph [10]. Such a property ignores temporal properties and seems hard to use for reasoning. There is also more and more evidence that these partial security properties are not enough to prevent realistic attacks in practice [23, 40].

More closely related to our work are papers that use separate per-component stacks, a trusted stack manager and some form of memory isolation to enforce control-flow correctness as part of a secure compilation result [50, 74]. Our work differs from theirs in that we use a different low-level security primitive (a capability machine with local capabilities rather than a machine with a primitive notion of compartments), and we do not use per-component stacks or a trusted stack manager but a single shared stack and a decentralized calling convention based on linear capabilities. Both prove a secure compilation result from a high-level language which clearly implies a general form of control-flow correctness, but that result is not separated from the results about other aspects of their compiler.

CheriBSD applies a similar approach with separate per-component stacks and a trusted stack manager on a capability machine [69]. The authors use local capabilities to prevent components from accidentally leaking their stack

pointer to other components, but there is no actual capability revocation in play. They do not provide many details on this mechanism and it is, for example, not clear if and how they intend to deal with higher-order interfaces (C function pointers) or stack references shared across component boundaries.

The fact that our full abstraction result only applies to reasonable components (see Section 3.4) makes it related to full abstraction results for unsafe languages. In their study of compartmentalization primitives, Juglaret et al. [50] discuss the property of Secure Compartmentalizing Compilation (SCC): a variant of full abstraction that applies to unsafe source languages. Essentially, they modify standard full abstraction so that preservation and reflection of contextual equivalence are only guaranteed for components that are *fully defined*, which means essentially that they do not exhibit undefined behavior in any fully defined context. In follow-up work, Abate et al. [11] extend this approach to scenarios where components only start to exhibit undefined behavior after a number of well-defined steps. If we see reasonable behavior as defined behavior, then our full abstraction result can be seen as an application of this same idea. Our results do not apply to dynamic compromise scenarios because they are intended to be used in the verification of a secure compiler where these scenarios are not relevant.

Local capabilities can be used to ensure well-bracketed control-flow and local-state encapsulation as demonstrated by Skorstengaard et al. [83]. Recently, Tsampas et al. [92] demonstrated that an extension of local capabilities with multiple linearly ordered colours can be used to enforce the life time of stack references. Specifically, a stack reference should not be able to outlive the stack frame it points to. If StkTokens was extended with stack references, then it would also enforce reference life times. Specifically in order to return from a call, we must use the return token, i.e. the stack. The stack is linear, so if there are references to it, aside from the stack capability itself, then we cannot have a complete return token. This means that we have to splice all the stack references together with the stack capability to complete the return token in order to return. Tsampas et al. [92] allow (almost) normal references that can be stored in multiple places at the same time. This means that their approach is more like C than what StkTokens has to offer. As explained in Section 3.1, these approaches have the downside that they require stack clearing (including unused parts) on boundary crossings.

In addition to the already-mentioned work on linear capabilities, Van Strydonck et al. [93] have recently used them in a secure (fully abstract) compiler for a C-like language with separation logic contracts. A form of linear capabilities was also used in the SAFE machine developed within the CRASH/SAFE project [30, 31]. Abate et al. [11] used micro-policy enforced linear return capabilities to ensure a cross-component stack discipline. Their linear capabilities were designed specifically to enforce the stack discipline but behave similarly to ours with the notable difference that their linear re-

turn pointers are destroyed in a jump.

There are other notions of secure compilation than full-abstraction [8]. Abate et al. [12] present an overview of trace-based secure compilation properties. Full abstraction is only one, relatively weak, property in their hierarchy. It would be interesting to investigate if our compiler, i.e. the embedding function from oLCM into LCM, also satisfies some of their other properties. While our current result implies that we can prove contextual equivalences in LCM components using StkTokens, by proving them instead in the more well-behaved oLCM semantics, such stronger properties would imply that we can also prove robust preservation of other (hyper-)properties in a similar manner. As our back-translation works for arbitrary programs, we expect that, in addition to full abstraction, our embedding also satisfies Robust Relational Hyperproperty Preservation (RrHP, the strongest property in the hierarchy of Abate et al.) and that a large part of our current proof (the back-translation and the logical relation) could be reused to establish this. However, to do this, we would first need to extend our semantics with some form of traces and we have not investigated how best to do this.

Acknowledgements

This research was supported in part by the ModuRes Sapere Aude Advanced Grant from The Danish Council for Independent Research for the Natural Sciences (FNU) and by Cost Action CA15123 EUTypes. Dominique Devriese held a Postdoctoral Fellowship from the Research Foundation - Flanders (FWO) during most of this research.

3.A Appendix

3.A.1 LCM instruction interpretation

In Figure 3.26, we present the interpretation of the LCM instructions left out of Figure 3.4. The predicate *nonZero* is defined as

$$nonZero(w) \stackrel{def}{=} \begin{cases} \bot & w \in \mathbb{Z} \wedge w = 0 \\ \top & \text{otherwise} \end{cases}$$

3.A.2 oLCM instruction interpretation

In Figure 3.27, we present the interpretation of the oLCM instructions left out of Figure 3.4. For the instructions that only change slightly on oLCM compared to LCM, we include the oLCM specific things in blue and the things both have in common in black.

3.A.3 World definitions

Definition 3.A.1. Given $W_{\text{free}} \in \text{World}_{\text{free\_stack}}$

$$\lfloor W_{\text{free}} \rfloor_{\{S\}} = \lambda r. \begin{cases} W_{\text{free}}(r).\text{v} \in S \\ \bot \end{cases}$$

♦

Definition 3.A.2 (Private stack sub-world erasure)**.** Given $W_{\text{priv}} \in \text{World}_{\text{call\_stack}}$

$$\lfloor W_{\text{priv}} \rfloor_{\{S\}} = \lambda r. \begin{cases} W_{\text{priv}}(r).\text{region.v} \in S \\ \bot \end{cases}$$

♦

Definition 3.A.3 (Heap sub-world erasure)**.** Given $W_{\text{heap}} \in \text{World}_{\text{heap}}$

$$\lfloor W_{\text{heap}} \rfloor_{\{S\}} = \lambda r. \begin{cases} W_{\text{heap}}(r).\text{v} \in S \\ \bot \end{cases}$$

♦

Definition 3.A.4 (World erasure)**.** Given world $(W_{\text{heap}}, W_{\text{priv}}, W_{\text{free}})$ define world erasure as

$$\lfloor (W_{\text{heap}}, W_{\text{priv}}, W_{\text{free}}) \rfloor_{\{S\}} = \left(\lfloor W_{\text{heap}} \rfloor_{\{S\}}, \lfloor W_{\text{priv}} \rfloor_{\{S\}}, \lfloor W_{\text{free}} \rfloor_{\{S\}} \right)$$

$$\lfloor W_{\text{heap}} \rfloor_{\{S\}} = \lambda r. \begin{cases} W_{\text{heap}}(r).\text{v} \in S \\ \bot \end{cases}$$

$i \in \text{Instr}$	$[\![i]\!](\Phi)$	Conditions
`getb` r_1 r_2	$updPc(\Phi[reg.r_1 \mapsto w])$	If $\Phi(r_2) = ((\_,\_),b,\_,\_)$ or$\Phi(r_2) = \text{seal}(b,_,_)$, then $w = b$ and otherwise $w = -1$
`gete` r_1 r_2	$updPc(\Phi[reg.r_1 \mapsto w])$	If $\Phi(r_2) = ((\_,\_),\_,e,\_)$ or $\Phi(r_2) = \text{seal}(\_,e,\_)$, then $w = e$ and otherwise $w = -1$
`gettype` r_1 r_2	$updPc(\Phi[reg.r_1 \mapsto w])$	$w = encodeType(\Phi(r_2))$[9]
`getl` r_1 r_2	$updPc(\Phi[reg.r_1 \mapsto w])$	If $isLinear(\Phi(r_2))$, then $w = encodeLin(\text{linear})$, otherwise $w = encodeLin(\text{normal})$
`getp` r_1 r_2	$updPc(\Phi[reg.r_1 \mapsto w])$	If $\Phi(r_2) = ((p,\_),\_,\_,\_)$, then $w = encodePerm(p)$ and otherwise $w = -1$
`jnz` r rn	$\Phi[reg.r \mapsto w][\text{pc} \mapsto \Phi(r)]$	$nonZero(\Phi(rn))$ and $w = linClear(\Phi(r))$
`jnz` r rn	$updPc(\Phi)$	If not $nonZero(\Phi(rn))$
`plus` r rn_1 rn_2	$updPc(\Phi[reg.r \mapsto n_1 + n_2])$	If for $i \in \{1,2\}$ $\Phi(rn_i) = n_i \in \mathbb{Z}$
`minus` r rn_1 rn_2	$updPc(\Phi[reg.r \mapsto n_1 - n_2])$	If for $i \in \{1,2\}$ $\Phi(rn_i) = n_i \in \mathbb{Z}$
`lt` r rn_1 rn_2	$updPc(\Phi[reg.r \mapsto 1])$	If for $i \in \{1,2\}$ $\Phi(rn_i) = n_i \in \mathbb{Z}$ and $n_1 < n_2$
`lt` r rn_1 rn_2	$updPc(\Phi[reg.r \mapsto 0])$	If for $i \in \{1,2\}$ $\Phi(rn_i) = n_i \in \mathbb{Z}$ and $n_1 \not< n_2$
`seta2b` r	$updPc(\Phi[reg.r \mapsto c])$	$r \neq \text{pc}$, $\Phi(r) = ((p,l),b,e,\_)$, and $c = ((p,l),b,e,b)$
`seta2b` r	$updPc(\Phi[reg.r \mapsto c])$	$r \neq \text{pc}$, $\Phi(r) = \text{seal}(\sigma_b,\sigma_e,\_)$, and $c = \text{seal}(\sigma_b,\sigma_e,\sigma_b)$
`restrict` r rn	$updPc(\Phi[reg.r \mapsto c])$	If $\Phi(r) = ((p,l),b,e,a)$ and $\Phi(rn) = n$ and $decodePerm(n) \leq p$ and $c = ((decodePerm(n),l),b,e,a)$
`split` r_1 r_2 r_3 rn	$updPc(\Phi[reg.r_1 \mapsto c_1]$ $[reg.r_2 \mapsto c_2])$	$\Phi(r_3) = \text{seal}(\sigma_b,\sigma_e,\sigma)$ and $\Phi(rn) = \sigma_n \in \mathbb{N}$ and $\sigma_b \leq \sigma_n < \sigma_e$ and $c_1 = \text{seal}(\sigma_b,\sigma_n,\sigma)$ and $c_2 = \text{seal}(\sigma_n + 1,\sigma_e,\sigma)$
`splice` r_1 r_2 r_3	$updPc(\Phi[reg.r_1 \mapsto c])$	$\Phi(r_2) = \text{seal}(\sigma_b,\sigma_n,\_)$ and $\Phi(r_3) = \text{seal}(\sigma_n + 1,\sigma_e,\sigma)$ and $\sigma_b \leq \sigma_n < \sigma_e$ and $c = \text{seal}(\sigma_b,\sigma_e,\sigma)$

Figure 3.26: Interpretation of LCM instructions.

$i \in$ Instr	$[\![i]\!](\Phi)$	Conditions
`store` r_1 r_2	$updPc(\Phi[reg.r_2 \mapsto w_2]$ $[mem.a \mapsto \Phi(r_2)])$	$\Phi(r_1) = ((p, \_), b, e, a)$ and $p \in \{\text{RWX}, \text{RW}\}$ and $b \leq a \leq e$ and $w_2 = linClear(\Phi(r_2))$ and $a \in \text{dom}(\Phi.mem)$
`load` r_1 r_2	$updPc(\Phi[reg.r_1 \mapsto w_1]$ $[mem.a \mapsto w_a])$	$\Phi(r_2) = \text{stack-ptr}(p, b, e, a)$ and $b \leq a \leq e$ and $p \in \{\text{RWX}, \text{RW}, \text{RX}, \text{R}\}$ and $a \in \text{dom}(\Phi.ms_{stk})$ and $w_1 = \Phi.ms_{stk}(a)$ and $isLinear(w_1) \Rightarrow p \in \{\text{RWX}, \text{RW}\}$ and $w_a = linClear(w_1)$
`geta` r_1 r_2	$updPc(\Phi[reg.r_1 \mapsto w])$	If $\Phi(r_2) = ((\_, \_), \_, \_, a)$, then $w = a$ and otherwise $w = -1$
`getb` r_1 r_2	$updPc(\Phi[reg.r_1 \mapsto w])$	If $\Phi(r_2) = ((\_, \_), b, \_, \_)$, then $w = b$ and otherwise $w = -1$
`gete` r_1 r_2	$updPc(\Phi[reg.r_1 \mapsto w])$	If $\Phi(r_2) = ((\_, \_), \_, e, \_)$, then $w = e$ and otherwise $w = -1$
`getp` r_1 r_2	$updPc(\Phi[reg.r_1 \mapsto w])$	If $\Phi(r_2) = ((p, \_), \_, \_, \_)$, then $w = encodePerm(p)$ and otherwise $w = -1$
`seta2b` r	$updPc(\Phi[reg.r \mapsto c])$	$r \neq \text{pc}$, $\Phi(r) = \text{stack-ptr}(p, b, e, \_)$, and $c = \text{stack-ptr}(p, b, e, b)$
`restrict` r rn	$updPc(\Phi[reg.r \mapsto c])$	If $\Phi(r) = \text{stack-ptr}(p, b, e, a)$ and $\Phi(rn) = n$ and $decodePerm(n) \leq p$ and $c =$ $\text{stack-ptr}(decodePerm(n), lin), b, e, a)$
`splice` r_1 r_2 r_3	$updPc(\Phi[reg.r_2 \mapsto 0]$ $[reg.r_3 \mapsto 0]$ $[reg.r_1 \mapsto c])$	$\Phi(r_2) = \text{stack-ptr}(p, b, n, \_)$ and $\Phi(r_3) = \text{stack-ptr}(p, n+1, e, a)$ and $b \leq n < e$ and $c = \text{stack-ptr}(p, b, e, a)$
`split` r_1 r_2 r_3 rn	$updPc(\Phi[reg.r_3 \mapsto 0]$ $[reg.r_1 \mapsto c_1]$ $[reg.r_2 \mapsto c_2])$	$\Phi(r_3) = \text{stack-ptr}(p, b, e, a)$ and $\Phi(rn) = n \in \mathbb{N}$ and $b \leq n < e$ and $c_1 = \text{stack-ptr}(p, b, n, a)$ and $c_2 = \text{stack-ptr}(p, n+1, e, a)$

Figure 3.27: Interpretation of oLCM instructions.

$$\lfloor W_{\text{priv}} \rfloor_{\{S\}} = \lambda r. \begin{cases} W_{\text{priv}}(r).\text{region.v} \in S \\ \bot \end{cases}$$

$$\lfloor W_{\text{free}} \rfloor_{\{S\}} = \lambda r. \begin{cases} W_{\text{free}}(r).\text{v} \in S \\ \bot \end{cases}$$

The *active* function takes a world and filters away all the revoked regions, so

$$active(W) = \lfloor W \rfloor_{\{\text{shadow,spatial,shared}\}}$$

♦

3.A.4 Standard c.o.f.e. definitions

Definition 3.A.5 (Product c.o.f.e.)**.** Given two c.o.f.e.'s $\left(X, \left(\overset{n}{=}_X\right)_{n=0}^{\infty}\right)$ and $\left(Y, \left(\overset{n}{=}_Y\right)_{n=0}^{\infty}\right)$ define the product c.o.f.e. as $\left(X \times Y, \left(\overset{n}{=}\right)_{n=0}^{\infty}\right)$ where the equivalence family is defined as for $(x,y),(x',y') \in X \times Y$

$$(x,y) \overset{n}{=} (x',y') \text{ iff } x \overset{n}{=} x' \wedge y \overset{n}{=} y'$$

♦

Definition 3.A.6 (Product preordered c.o.f.e.)**.** Given two c.o.f.e.'s $\left(X, \left(\overset{n}{=}_X\right)_{n=0}^{\infty}, \sqsupseteq_X\right)$ and $\left(Y, \left(\overset{n}{=}_Y\right)_{n=0}^{\infty}, \sqsupseteq_y\right)$, define the product preordered c.o.f.e. as

$$\left(X \times Y, \left(\overset{n}{=}\right)_{n=0}^{\infty}, \sqsupseteq\right)$$

where the preorder $\sqsupseteq$ distributes to the underlying preorder, i.e.

$$\text{for } (x,y),(x',y') \in X \times Y, \quad (x',y') \sqsupseteq (x,y) \text{ iff } x' \sqsupseteq_X x \wedge y' \sqsupseteq_Y y$$

and the family of equivalences distributes to the underlying families of equivalences, i.e.

$$\text{for } (x,y),(x',y') \in X \times Y, \quad (x,y) \overset{n}{=} (x',y') \text{ iff } x \overset{n}{=}_X x' \wedge y \overset{n}{=}_Y y'$$

♦

Definition 3.A.7 (Union preordered c.o.f.e.)**.** Given two c.o.f.e.'s $\left(X, \left(\overset{n}{=}_X\right)_{n=0}^{\infty}, \sqsupseteq_X\right)$ and $\left(Y, \left(\overset{n}{=}_Y\right)_{n=0}^{\infty}, \sqsupseteq_y\right)$, define the product preordered c.o.f.e. as

$$\left(X \cup Y, \left(\overset{n}{=}\right)_{n=0}^{\infty}, \sqsupseteq\right)$$

where the preorder $\sqsupseteq$ distributes to the underlying preorder, i.e.

$$\text{for } z,z' \in X \cup Y, \quad z' \sqsupseteq z \text{ iff } \begin{cases} z,z' \in X \wedge z' \sqsupseteq_x z \vee \\ z,z' \in Y \wedge z' \sqsupseteq_Y z \end{cases}$$

and the family of equivalences distributes to the underlying families of equivalences, i.e.

$$\text{for } z,z' \in X \cup Y, \quad z \overset{n}{=} z' \text{ iff } \begin{cases} z,z' \in X \wedge z \overset{n}{=}_x z' \vee \\ z,z' \in Y \wedge z \overset{n}{=}_Y z' \end{cases}$$

♦

Definition 3.A.8 (▶ preordered c.o.f.e.)**.** Given a preordered c.o.f.e $\left(X, \left(\overset{n}{=}_X\right)_{n=0}^{\infty}, \sqsupseteq_X\right)$ define

$$\blacktriangleright\left(X, \left(\overset{n}{=}_X\right)_{n=0}^{\infty}, \sqsupseteq_X\right) = \left(X, \left(\overset{n}{=}_{\blacktriangleright}\right)_{n=0}^{\infty}, \sqsupseteq_X\right)$$

where

$$x \overset{n}{=}_{\blacktriangleright} x' \text{ iff } \begin{cases} n = 0 \vee \\ x \overset{n-1}{=}_X x' \end{cases}$$

♦

Bibliography

[1] Caja.
https://developers.google.com/caja/.
Accessed: 25-07-2019.
18, 19

[2] Capability hardware enhanced risc instructions (cheri).
https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/, .
Accessed: 08-07-2019.
10, 16

[3] Fork of llvm adding cheri support.
https://github.com/CTSRD-CHERI/llvm-project, .
Accessed: 08-07-2019.
10

[4] Newspeak.
https://newspeaklanguage.org/.
Accessed: 25-07-2019.
18

[5] ocaps.
https://wsargent.github.io/ocaps/index.html.
Accessed: 25-07-2019.
18

[6] Pony.
https://www.ponylang.io/.
Accessed: 25-07-2019.
18

[7] *iAPX 432 General Data Processor Architecture Reference Manual*, 1981.
6

[8] Martín Abadi.
Protection in Programming-Language Translations: Mobile Object Systems.

In *European Conference on Object-Oriented Programming*. Springer Berlin Heidelberg, 1998.
21, 78, 189

[9] Martín Abadi.
Protection in programming-language translations.
In *Secure Internet programming*. Springer-Verlag, 1999.
104, 122

[10] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti.
Control-flow Integrity.
In *Conference on Computer and Communications Security*. ACM, 2005.
32, 78, 85, 103, 120, 187

[11] Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Catalin Hritcu, Théo Laurent, Benjamin C. Pierce, Marco Stronati, and Andrew Tolmach.
When good components go bad: Formally secure compilation despite dynamic compromise.
In *Computer and Communications Security*. ACM, 2018.
101, 188

[12] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault.
Journey beyond full abstraction: Exploring robust property preservation for secure compilation.
In *Computer Security Foundations Symposium*. IEEE Computer Society Press, 2019.
21, 26, 189

[13] Amal Ahmed, Derek Dreyer, and Andreas Rossberg.
State-dependent representation independence.
In *Symposium on Principles of Programming Languages*. ACM, 2009.
86

[14] Amal Jamil Ahmed.
Semantics of types for mutable state.
PhD thesis, Princeton University, 2004.
50, 86, 144

[15] Pierre America and Jan Rutten.
Solving reflexive domain equations in a category of complete metric spaces.
Journal of Computer and System Sciences, 39, 1989.
91, 147

[16] Andrew W. Appel and David McAllester.
An indexed model of recursive types for foundational proof-carrying code.
ACM Transactions on Programming Languages and Systems, 23, 2001.
47, 86

[17] Nick Benton and Chung-Kil Hur.
Biorthogonality, Step-indexing and Compiler Correctness.
In *International Conference on Functional Programming*. ACM, 2009.
86

[18] Lars Birkedal and Aleš Bizjak.
A Taste of Categorical Logic - tutorial notes.
http://cs.au.dk/~birke/modures/tutorial/categorical-logic-tutorial-notes.pdf, 2014.
50, 89, 147, 154

[19] Lars Birkedal, Kristian Støvring, and Jacob Thamsborg.
The category-theoretic solution of recursive metric-space equations.
Theoretical Computer Science, 411, 2010.
91

[20] Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang.
Step-indexed Kripke models over recursive worlds.
In *Symposium on Principles of Programming Languages*. ACM, 2011.
50, 86, 147, 154

[21] Aleš Bizjak.
Some theorems about mutually recursive domain equations in the category of preordered COFEs.
Manuscript. Available at http://alesb.com/documents/notes/mutually-recursive-domain-eq.pdf, 2017.
91

[22] Gilad Bracha.
On the interaction of method lookup and scope with inheritance and nesting.
In *3rd ECOOP Workshop on Dynamic Languages and Applications*, 2007.
18

[23] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross.
Control-flow bending: On the effectiveness of control-flow integrity.
In *USENIX Security*. USENIX Association, 2015.
187

[24] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally.
Hardware support for fast capability-based addressing.
In *Architectural Support for Programming Languages and Operating Systems*. ACM, 1994.
9, 10, 32, 35, 37, 85

[25] S. Chiricescu, A. DeHon, D. Demange, S. Iyer, A. Kliger, G. Morrisett, B. C. Pierce, H. Reubenstein, J. M. Smith, G. T. Sullivan, A. Thomas, J. Tov, C. M. White, and D. Wittenberg.
Safe: A clean-slate architecture for secure systems.
In *2013 IEEE International Conference on Technologies for Homeland Security (HST)*, 2013.
15, 24

[26] David Chisnall, Brooks Davis, Khilan Gudka, David Brazdil, Alexandre Joannou, Jonathan Woodruff, A. Theodore Markettos, J. Edward Maste, Robert Norton, Stacey Son, Michael Roe, Simon W. Moore, Peter G. Neumann, Ben Laurie, and Robert N.M. Watson.
CHERI JNI: Sinking the Java Security Model into the C.
In *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017.
40

[27] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil.
Deny capabilities for safe, fast actors.
In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. ACM, 2015.
18

[28] William Dally, Stephen W. Keckler, Nick Carter, Andrew Chang, Marco Fillo, and Whay Lee.
M-machine microarchitecture v1.1.
Technical report, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1994.
9

[29] William J. Dally, Stephen W. Keckler, Nick Carter, Andrew Chang, Marco Fillo, and Whay S. Lee.
The M-Machine Instruction Set Reference Manual v1.55, 1997.
6, 9, 10

[30] Arthur Azevedo de Amorim, Maxime Dénès, Nick Giannarakis, Catalin Hritcu, Benjamin C. Pierce, Antal Spector-Zabusky, and Andrew Tolmach.
Micro-policies: Formally verified, tag-based security monitors.

In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, 2015.
188

[31] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Catalin Hritcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach.
A verified information-flow architecture.
Journal of Computer Security, 24, 2016.
188

[32] Jack B. Dennis and Earl C. Van Horn.
Programming semantics for multiprogrammed computations.
Communications of the ACM, 9, 1966.
4, 6, 32, 85, 187

[33] Dominique Devriese, Lars Birkedal, and Frank Piessens.
Reasoning about object capabilities using logical relations and effect parametricity.
In *European Symposium on Security and Privacy*. IEEE Computer Society, 2016.
20, 23, 33, 53, 59, 83, 86

[34] Dominique Devriese, Marco Patrignani, Frank Piessens, and Steven Keuchel.
Modular, fully-abstract compilation by approximate back-translation.
Logical Methods in Computer Science, 13, 2017.
101, 185

[35] Pietro Di Gianantonio and Marino Miculan.
A unifying approach to recursive and co-recursive definitions.
In *TYPES*, volume 2646, 2002.
147

[36] Derek Dreyer, Georg Neis, and Lars Birkedal.
The impact of higher-order state and control effects on local relational reasoning.
Journal of Functional Programming, 22, 2012.
23, 33, 34, 73, 74, 76, 84, 86

[37] Akram El-Korashy.
A formal model for capability machines: An illustrative case study towards secure compilation to cheri.
Master's thesis, Saarland University, 2016.
87

[38] Dhammika Elkaduwe, Gerwin Klein, and Kevin Elphinstone.
Verified protection model of the sel4 microkernel.
In *Verified Software: Theories, Tools, Experiments*. Springer Berlin Heidelberg, 2008.
15

[39] D. M. England.
Capability concept mechanisms and structure in system 250.
In *Proceedings of the International Workshop on Protection in Operating Systems, IRIA, Paris*, 1974.
6

[40] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos.
Control jujutsu: On the weaknesses of fine-grained control flow integrity.
In *Computer and Communications Security*. ACM, 2015.
187

[41] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee.
The m-machine multicomputer.
International Journal of Parallel Programming, 25, 1997.
9

[42] Stephanie Forrest, Anil Somayaji, and David H. Ackley.
Building diverse computer systems.
In *Hot Topics in Operating Systems*. IEEE Computer Society, 1997.
32

[43] Bill Frantz, Norm Hardy, Jay Jonekait, and Charlie Landau.
Gnosis: A secure operating system for the '90s. share proceedings, 1979.
http://cap-lore.com/CapTheory/upenn/Gnosis/Gnosis.html.
15

[44] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi.
The essence of javascript.
In *ECOOP 2010 – Object-Oriented Programming*. Springer Berlin Heidelberg, 2010.
20

[45] Norm Hardy.
The confused deputy:(or why capabilities might have been invented).
ACM SIGOPS Operating Systems Review, 22, 1988.
5, 13

[46] Merle E. Houdek, Frank G. Soltis, and Roy L. Hoffman.
Ibm system/38 support for capability-based addressing.
In *Proceedings of the 8th Annual Symposium on Computer Architecture*. IEEE Computer Society Press, 1981.
6

[47] Chung-Kil Hur and Derek Dreyer.
A kripke logical relation between ml and assembly.
In *Symposium on Principles of Programming Languages*. ACM, 2011.
83, 86

[48] Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W. Moore, Alex Bradbury, Hongyan Xia, Robert N. M. Watson, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alfredo Mazzinghi, Alex Richardson, Stacey D. Son, and A. Theodore Markettos.
Efficient tagged memory.
In *IEEE International Conference on Computer Design*. IEEE Computer Society, 2017.
80

[49] Anita K. Jones, Robert J. Chansler, Jr., Ivor Durham, Karsten Schwans, and Steven R. Vegdahl.
Staros, a multiprocessor operating system for the support of task forces.
In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*. ACM, 1979.
15

[50] Yannis Juglaret, Catalin Hritcu, Arthur Azevedo de Amorim, and Benjamin C. Pierce.
Beyond Good and Evil: Formalizing the Security Guarantees of Compartmentalizing Compilation.
In *Computer Security Foundations Symposium*. IEEE Computer Society Press, 2016.
86, 101, 187, 188

[51] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer.
Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning.
In *Symposium on Principles of Programming Languages*. ACM, 2015.
86, 147

[52] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer.
Higher-order ghost state.

In *International Conference on Functional Programming*. ACM, 2016.
86, 147

[53] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood.
sel4: Formal verification of an os kernel.
In *Symposium on Operating Systems Principles*. ACM, 2009.
15

[54] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser.
Comprehensive formal verification of an os microkernel.
ACM Transactions on Computer Systems, 32, 2014.
15

[55] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal.
The essence of higher-order concurrent separation logic.
In *European Symposium on Programming Languages and Systems*. Springer, 2017.
86

[56] Robbert Krebbers, Amin Timany, and Lars Birkedal.
Interactive proofs in higher-order concurrent separation logic.
In *Symposium on Principles of Programming Languages*. ACM, 2017.
26, 86, 147

[57] Jean-Louis Krivine.
Classical logic, storage operators and second-order lambda-calculus.
Annals of Pure and Applied Logic, 68, 1994.
83

[58] Butler W. Lampson and Howard E. Sturgis.
Reflections on an operating system design.
Communications of the ACM, 19, 1976.
15

[59] Xavier Leroy.
Formal verification of a realistic compiler.
Commun. ACM, 52, 2009.
27

[60] Henry M. Levy.
Capability-Based Computer Systems.
Digital Press, 1984.

6, 15, 85, 102, 187

[61] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley.
The Java Virtual Machine Specification.
Pearson Education, 2014.
32

[62] R. J. Lipton and L. Snyder.
A linear time algorithm for deciding subject security.
Journal of the ACM, 24, 1977.
14

[63] Roger M. Needham and R D. H. Walker.
The cambridge cap computer and its protection system.
Operating Systems Review - SIGOPS, 11, 1977.
6

[64] Sergio Maffeis, John C. Mitchell, and Ankur Taly.
Object capabilities and isolation of untrusted web applications.
In *Symposium on Security and Privacy*. IEEE Computer Society, 2010.
33, 59

[65] Adrian Mettler, David Wagner, and Tyler Close.
Joe-E: A security-oriented subset of Java.
In *Network and Distributed System Security*. The Internet Society, 2010.
18

[66] Mark Miller, Ka-ping Yee, and Jonathan Shapiro.
Capability myths demolished.
2003.
13

[67] Mark Samuel Miller.
Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.
PhD thesis, Johns Hopkins University, 2006.
17, 18

[68] Greg Morrisett, David Walker, Karl Crary, and Neal Glew.
From System F to Typed Assembly Language.
ACM Transactions on Programming Languages and Systems, 21, 1999.
32

[69] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav H. Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey D. Son, and Munraj Vadera.

Cheri: A hybrid capability-system architecture for scalable software compartmentalization.
In *Symposium on Security and Privacy*. IEEE Computer Society, 2015.
6, 10, 12, 23, 25, 32, 35, 37, 80, 85, 102, 105, 106, 187

[70] Max S. New, William J. Bowman, and Amal Ahmed.
Fully abstract compilation via universal embedding.
In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ACM, 2016.
101

[71] Zhaozhong Ni and Zhong Shao.
Certified assembly programming with embedded code pointers.
In *Symposium on Principles of Programming Languages*. ACM, 2006.
86

[72] Leo Osvald, Grégory Essertel, Xilun Wu, Lilliam I. González Alayón, and Tiark Rompf.
Gentrification Gone Too Far? Affordable 2Nd-class Values for Fun and (Co-)Effect.
In *Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2016.
84

[73] Marco Patrignani and Deepak Garg.
Secure compilation and hyperproperty preservation.
In *Computer Security Foundations*. IEEE, 2017.
101

[74] Marco Patrignani, Dominique Devriese, and Frank Piessens.
On Modular and Fully-Abstract Compilation.
In *Computer Security Foundations Symposium*. IEEE Computer Society, 2016.
86, 187

[75] Marco Patrignani, Amal Ahmed, and Dave Clarke.
Formal approaches to secure compilation: A survey of fully abstract compilation and related work.
ACM Computing Surveys, 51, 2019.
21, 87, 101

[76] Andrew M. Pitts and Ian D. B. Stark.
Operational reasoning for functions with local state.
In *Higher Order Operational Techniques in Semantics*. Cambridge University Press, 1998.
23, 48, 73, 74, 83, 86

[77] Jerome H. Saltzer and Michael D. Schroeder.
The protection of information in computer systems.
In *Proceedings of the IEEE*. IEEE, 1975.
5

[78] Dana S. Scott.
Data types as lattices.
SIAM J. Comput., 5, 1976.
147

[79] Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein.
sel4 enforces integrity.
In *Interactive Theorem Proving*. Springer Berlin Heidelberg, 2011.
84

[80] Hovav Shacham.
The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86).
In *Proceedings of the 14th ACM Conference on Computer and Communications Security*. ACM, 2007.
22

[81] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber.
EROS: A Fast Capability System.
In *Symposium on Operating Systems Principles*. ACM, 1999.
15, 32

[82] Lau Skorstengaard.
An introduction to logical relations.
https://arxiv.org/abs/1907.11133.
23

[83] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal.
Reasoning about a machine with local capabilities.
In *Programming Languages and Systems*. Springer International Publishing, 2018.
23, 34, 103, 104, 120, 145, 163, 188

[84] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal.
Reasoning about a machine with local capabilities: Provably safe stack and return pointer management - technical appendix including proofs and details.
Technical report, Dept. of Computer Science, Aarhus University, 2018.
URL https://arxiv.org/abs/1902.05283.
26, 34, 39, 59, 64, 70, 78, 88, 92

[85] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal.
StkTokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities - technical report with proofs and details.
Technical report, Dept. of Computer Science, Aarhus University, 2018.
URL https://arxiv.org/abs/1811.02787.
26, 105, 113

[86] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal.
StkTokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities.
Proceedings of the ACM on Programming Languages, 3, 2019.
24, 78, 105, 138

[87] David Swasey, Deepak Garg, and Derek Dreyer.
Robust and compositional verification of object capability patterns.
Proceedings of the ACM on Programming Languages, 1, 2017.
86

[88] Nick Szabo.
Formalizing and Securing Relationships on Public Networks.
First Monday, 2, 1997.
103

[89] Nick Szabo.
Scarce objects, 2004.
URL https://nakamotoinstitute.org/scarce-objects/.
103

[90] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song.
Sok: Eternal war in memory.
In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2013.
32

[91] Jacob Thamsborg and Lars Birkedal.
A Kripke logical relation for effect-based program transformations.
In *International Conference on Functional Programming*. ACM, 2011.
50, 144

[92] Stelios Tsampas, Dominique Devriese, and Frank Piessens.
Temporal safety for stack allocated memory on capability machines.
In *2019 IEEE 32nd Computer Security Foundations Symposium*. IEEE Computer Society Press, 2019.
188

[93] Thomas Van Strydonck, Frank Piessens, and Dominique Devriese.
Linear capabilities for fully abstract compilation of separation-logic-verified code.
In *ICFP*, 2019.
188

[94] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham.
Efficient Software-based Fault Isolation.
In *Symposium on Operating Systems Principles*. ACM, 1993.
32

[95] R. N. M. Watson, R. M. Norton, J. Woodruff, S. W. Moore, P. G. Neumann, J. Anderson, D. Chisnall, B. Davis, B. Laurie, M. Roe, N. H. Dave, K. Gudka, A. Joannou, A. T. Markettos, E. Maste, S. J. Murdoch, C. Rothwell, S. D. Son, and M. Vadera.
Fast Protection-Domain Crossing in the CHERI Capability-System Architecture.
IEEE Micro, 36, September 2016.
102, 106

[96] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway.
Capsicum: Practical capabilities for UNIX.
In *USENIX Security Symposium*. USENIX, 2010.
15, 16

[97] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Jonathan Anderson, David Chisnall, Brooks Davis, Alexandre Joannou, Ben Laurie, Simon W. Moore, Steven J. Murdoch, Robert Norton, and Stacey Son.
Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture.
Technical report, University of Cambridge, Computer Laboratory, 2015.
URL http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-876.html.
102, 103

[98] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, David Chisnall, Brooks Davis, Nathaniel Wesley Filardo, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alex Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia.

Capability hardware enhanced risc instructions: Cheri instruction-set architecture (version 7).
Technical Report UCAM-CL-TR-927, University of Cambridge, Computer Laboratory, 2019.
10, 12, 24, 103

[99] Robert NM Watson, Peter G Neumann, Jonathan Woodruff, Jonathan Anderson, Ross Anderson, Nirav Dave, Ben Laurie, Simon W Moore, Steven J Murdoch, Philip Paeps, and others.
CHERI: A research platform deconflating hardware virtualization and protection.
In *Workshop on Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE)*, 2012.
103

[100] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe.
The CHERI Capability Model: Revisiting RISC in an Age of Risk.
In *International Symposium on Computer Architecture*, pages 457–468. IEEE Press, 2014.
10, 15, 16, 32, 44, 85, 187

[101] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Brooks Davis, Peter G. Neumann, Robert Nicholas Maxwell Watson, Simon Moore, Anthony Fox, Robert Norton, and David Chisnall.
Cheri concentrate: Practical compressed capabilities.
IEEE Transactions on Computers, PP, 2019.
11

[102] William A. Wulf, Ellis S. Cohen, William M. Corwin, Anita K. Jones, Roy Levin, C. Pierson, and Fred J. Pollack.
Hydra: The kernel of a multiprocessor operating system.
Communications of the ACM, 17, 1974.
15

[103] Victor H. Yngve.
The chicago magic number computer.
In *ICR Quarterly Report*. Institute for Computer Research, University of Chicago, 1968.
6