Le Temps des Cerises: Efficient Temporal Stack Safety on Capability Machines using Directed Capabilities

AÏNA LINN GEORGES, Aarhus University, Denmark
ALIX TRIEU*, ANSSI, France
LARS BIRKEDAL, Aarhus University, Denmark

Capability machines are a type of CPUs that support fine-grained privilege separation using *capabilities*, machine words that include forms of authority. Formal models of capability machines and associated calling conventions have so far focused on establishing two forms of stack safety properties, namely local state encapsulation and well-bracketed control flow. We introduce a novel kind of *directed* capabilities and show how to use them to make an earlier suggested calling convention more efficient. In contrast to earlier work on capability machine models we do not only consider integrity properties but also confidentiality properties; we provide a unary logical relation to reason about the former and a binary logical relation to reason about the latter, each expressive enough to reason about temporal stack safety. While the logical relations are useful for reasoning about concrete examples, they do not on their own demonstrate that stack safety holds for a large class of programs. Therefore, we also show full abstraction of a compiler from an overlay semantics that internalizes the calling convention as a single call step and explicitly keeps track of the call stack and frame lifetimes to a base capability machine. All results have been mechanized in Coq.

CCS Concepts: • **Security and privacy** → **Logic and verification**; **Formal security models**; *Security in hardware*; **Logic and verification**; • **Theory of computation** → *Logic and verification*; *Logic and verification*.

Additional Key Words and Phrases: capability machines, logical relation, Iris, separation logic, full absraction, security, stack safety

ACM Reference Format:
Aïna Linn Georges, Alix Trieu, and Lars Birkedal. 2022. Le Temps des Cerises: Efficient Temporal Stack Safety on Capability Machines using Directed Capabilities. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 74 (April 2022), 30 pages. https://doi.org/10.1145/3527318

1 INTRODUCTION

Lack of memory safety is an important source of security bugs, for instance, 70% of all issues in Microsoft products [Thomas 2019] and in the Google Chrome browser [Chromium 2020] are memory safety related. It is thus not surprising that a large number of software or hardware protection mechanisms such as shadow stacks, stack canaries, address space layout randomization, etc (see [Szekeres et al. 2013] for a survey) have been proposed. Capability machines have recently risen as a promising solution to memory safety vulnerabilities; quoting a Microsoft study, "[capability machines] would have deterministically mitigated at least two thirds of all those issues" [Joly et al. 2020].

Capability machines are a kind of architecture that enable fine-grained memory protection using tagged memory [Carter et al. 1994; Dennis and Van Horn 1966; Levy 1984] and *capabilities*, a form

*This work was carried out while the author was affiliated with Aarhus University.

Authors' addresses: Aïna Linn Georges, Aarhus University, Denmark, ageorges@cs.au.dk; Alix Trieu, ANSSI, France, alix.trieu@ssi.gouv.fr; Lars Birkedal, Aarhus University, Denmark, birkedal@cs.au.dk.

2475-1421/2022/4-ART74
https://doi.org/10.1145/3527318

of unforgeable memory pointers with a certain amount of authority, in the form of a permission, range, etc. Over the last decade, CHERI [Watson et al. 2020], a family of capability machines, has matured into an extensive design featuring, among other, a full UNIX-style operating system, CheriBSD [Watson et al. 2015]. Ideas from CHERI are currently being adopted by ARM in their Morello project [Arm 2021], which is aimed at developing concrete CPU designs and prototypes that could be implemented in future hardware.

One of the promises of capability machines is that they can enforce security properties that we expect from high-level languages, in particular stack safety, *even when machine code is linked with other untrusted and possibly adversarial machine code*. This potential is not yet realized in practice. In particular, while CheriBSD does make use of so-called local capabilities to limit the impact of potential bugs, it does not rely on them for enforcing security properties. This is likely because a secure calling convention based on local capabilities could be too inefficient as it would require a lot of stack clearing. Indeed, this is the case for the first known provably secure calling convention based on local capabilities [Skorstengaard et al. 2019a] — this calling convention requires to clear the full stack before and after every call. This has led to research on other calling conventions based on novel forms of capabilities. In particular, Skorstengaard et al. [2019b] proposed a calling convention based on so-called linear capabilities, which, however, are believed not to be efficiently implementable in hardware. This motivated another proposal by Georges et al. [2021] who suggested a secure calling convention based on a combination of so-called uninitialized capabilities and local capabilities, and which only involves a modicum of stack clearing per call, on the order of a single stack frame.

The works cited above on provably secure capability-machine-based calling conventions have all focused on spatial memory safety, in particular local state encapsulation and well-bracketed control flow. In another direction, Tsampas et al. [2019] recently proposed a kind of capabilities including "lifetime" information to enforce *temporal* memory safety, e.g., that the content of popped stack frames cannot be accessed. However, one problem with implementing this proposal is that in order to allow for a call depth of size 2^n, n bits would be required in the encoding of the lifetime information for a capability, which renders it impractical.

In this paper, we propose a novel kind of so-called *directed* capabilities and show how they can be used in combination with uninitialized capabilities to realize a new calling convention, which is efficient (it does not involve any stack clearing at all) and which provably enforces both spatial and temporal stack safety properties.

More precisely, we present CeriseM, an extension of the low-level capability machine model of Georges et al. [2021], with a novel form of directed capabilities, for which we present a novel stack-based calling convention. We show that it provably guarantees spatial and temporal stack safety. In light of the fact that it is actually quite subtle to capture stack safety properties formally, as also emphasized in a recent paper by Anderson et al. [2021], we include a detailed discussion of the stack safety properties we consider and how our novel approach improves over earlier proposals, see § 2. We include a discussion of the impact of stack objects on stack safety properties; prior work on local capability machines have largely ignored stack objects, but they have a significant impact on the guarantees provided by the capability machine. In contrast to the earlier formal models for capability machines mentioned above, we do not only consider integrity properties but also (stack) confidentiality properties.

To formally establish *integrity*, we follow the approach of Skorstengaard et al. [2019a] and develop a unary Kripke logical relations model, which captures capability machine safety. Our model is an extension of the one by Georges et al. [2021]; the novelty consists of an extension to account for temporal safety. There are two facets to this: a simple one, which is to extend the definition to also treat directed capabilities, and a challenging one, which is to extend the model, in particular, the

Kripke worlds, to capture the enforcement of temporal properties. The latter means that our model makes use of a novel kind of state transition system for the Kripke worlds.

To formally establish *confidentiality*, we further develop a *binary* logical relations model. We show that the binary logical relation implies contextual refinement so that it can serve as a sound proof method for establishing contextual equivalence and hence confidentiality. To the best of our knowledge, this is the first binary logical relations model for a low-level capability machine model.

We demonstrate that the unary and binary logical relations models can be used to prove stack safety properties for challenging example programs; we focus on examples that have not been considered in the literature before.

To give further evidence for the claim that our novel directed-capability-based calling convention actually does capture stack safety, we follow the approach of Skorstengaard et al. [2019b] and show full abstraction of a compiler from an overlay semantics that internalizes the calling convention as a single call step and explicitly keeps track of the call stack and frame lifetimes to the base capability machine. The idea is that the overlay semantics clearly enforces stack safety; our overlay semantics is related to the one used in [Skorstengaard et al. 2019b] but now accounts for temporal properties by completely removing popped stack frames from the stack once their lifetime is over (technical differences are detailed in § 6).

We have mechanized all of the models and results presented in the paper on top of the mechanization of the Iris program logic [Jung et al. 2016, 2018, 2015; Krebbers et al. 2017a] in Coq [Krebbers et al. 2018, 2017b]. The Iris-Coq mechanization [Georges et al. 2022a] can be found online at https://github.com/logsem/cerise-stack-monotone/releases/tag/OOPSLA2022.

2 ON THE STACK SAFETY OF CAPABILITY MACHINES

In this section, we explore the properties that make up stack safety in the context of capability machines. We follow Anderson et al. [2021], who define multiple degrees of stack safety, as various conjunctions of local state encapsulation (LSE) and well-bracketed control flow (WBCF). In particular, our goal is to reach a notion of stack safety that falls within their definition of observational stack safety, which also covers the temporal aspect of LSE. A key takeaway of this section is to highlight how existing calling convetions incur undesired overhead in order to enforce stack safety. Unlike Anderson et al. [2021], we consider LSE, WBCF and temporal stack safety for machines with both a stack and a heap. For clarity, we illustrate each property with an example written in a C-like language, though we actually consider the underlying assembly code. Next, we explore two interesting aspects of these properties that are particularly tricky. Finally, we survey previously proposed capability machine calling conventions and show where they fall on the spectrum of stack safety, including the novel calling convention and efficient enforcement mechanism we present in this paper.

2.1 A Family of Stack Safety Properties

```
void adv(void);
void f(void) {
  int *y; // allocated on
  *y = 2; // the stack
  adv(); assert (*y == 2)}
```

Listing 1. Integrity: frame

```
void adv(void);
void f(void) {
  static int x = 2;
  adv(); assert (x == 2)}
```

Listing 2. Integrity: environment

2.1.1 Local State Encapsulation. Local state is a concept that exists in both low and high level languages. In a low-level language with a stack, local state often refers to the encapsulation of local variables in a stack frame. For instance, in Lis. 1, the local variable y is a part of f's local stack frame, and is not shared with the arbitrary adversarial code adv; and hence the assert, stating the integrity of y, should succeed.

In high-level languages, local state may also refer to the state encapsulated within the scope of a closure. Consider Lis. 2, where function f possesses some private state x (a static variable persists across calls, similarly to local variables in closures). Upon return, the integrity of x is tested with an assert statement. If x is not properly encapsulated, adv may modify x, and the assertion fails.

The stack is used to store local variables as well as the local environment to be reclaimed upon return of a call. When discussing LSE, we will refer to the local state being the local stack frame not shared with a callee, as well as the state encapsulated within a closure, which ought to stay encapsulated not just from the callee, but from the caller as well. For instance, an adversarial context may call f in Lis. 2 multiple times, but it should never get access to the private state x.

```
void adv(void);
void f(void) {
 int *y; *y = 2; adv()}
void g(void) {
 int *y; *y = 3; adv()}
```

Listing 3. Confidentiality: frame

```
void adv(void);
void f(void) {
 static int x = 2; adv()}
void g(void) {
 static int x = 3; adv()}
```

Listing 4. Confidentiality: environment

Following Anderson et al. [2021], we must distinguish between *local state integrity* and *local state confidentiality*. Local state integrity states that the local stack frame is protected from changes by the callee, local state confidentiality states that the local stack frame cannot be read by the callee and thus influence their behaviour. Hence local state confidentiality is a binary property. For example, Lis. 3 contains two programs f and g with *different* local states that should stay hidden from the arbitrary function adv. Local state confidentiality guarantees the contextual equivalence of the functions f and g. Similarly, Lis. 4 depends on the same local state confidentiality, but for the environment of closures.

Since we also use the stack for the encapsulated environment of closures, our notion of LSE includes the integrity and confidentiality of the environment of a closure against the full context. Indeed, a closure needs to protect its private state against *both* callees and callers.

```
1  void adv(void);
2  void f(void) {
3    static int x = 0;
4    x = 0; adv();
5    x = 1; adv();
6    assert (x == 1); }
```

Listing 5. Awkward Example

2.1.2 Well-Bracketed Control Flow. Another common property in high-level languages is well-bracketed control flow. For example, consider Lis. 5, which is a variant of the classical "awkward example" [Dreyer et al. 2010a]. Here f possesses some local state x (line 3), which is set to 0 before calling some arbitrary adversarial code adv(). After the call returns, x is set to 1 before calling the adversary again. Finally, x is checked to be still equal to 1 at the end. If WBCF is not enforced, then during the second call to adv on line 5, adv could store the return pointer to line 6 in its own private state, and call f, which would then set x to 0 before calling adv again who can finally use the return pointer to line 6 it kept and fail the assertion.

It has been shown that both some form of LSE and WBCF can be enforced on capability machines, even in the presence of arbitrary code [Georges et al. 2021; Skorstengaard et al. 2018, 2019b]. We will give more details on how this is enforced in § 3.2.

```
1   int N, K;
2   void h(int* x) { *x = 0 }
3   void g(int* x) {
4     char* t[K]; h(x) }
5   void f(int** x) {
6     char* t[N];
7     int z; *x = &z }
8   int main(void) {
9     int* x; f(&x);
10    g(x); return 0 }
```

Listing 6. Example violating temporal stack safety

2.1.3 Temporal Stack Safety. In another direction, Tsampas et al. [2019] study the issue of temporal safety. Consider the code in Lis. 6[1], where &x on line 12 is a pointer to a location containing another pointer. After the call to f, there is now a pointer at &x to the location l previously occupied by z on line 8. The value of l depends on a global variable N. It should be noted that l is *stale* after the return and should not be allowed to be passed down. Nevertheless, l is passed to h through g. For well chosen values of K and N, it is possible that l coincides with where the return pointer of h is stored and thus the store at line 2 can lead to the control

[1] An example adapted from [Tsampas et al. 2019].

flow being hijacked. This example shows a temporal stack safety violation that exploits a dangling stack pointer. To address this issue, Tsampas et al. propose that capabilities are extended with "lifetime" information, basically the call depth of the function's stack frame, and that capabilities with longer lifetime may not be used to store a capability with shorter lifetime. This would disallow the store on line 9 in the example. In essence, it disallows dangling stack pointers to be stored on the stack, and thus to be passed down the call stack beyond their lifetime.

Also related to temporal stack safety, Anderson et al. [2021] find that the lazy tagging and clearing micro-policy of Roessler and DeHon [2018] violates the temporal aspect of observable stack safety, and repairs it by generating fresh identifiers for *each* call, requiring an unbounded number of tags.

2.2 Two Subtleties of Stack Safety

```
void f(void) {
  static int x = 2;
  int *y;
  *y = &x;
  assert (x == 2); }
```

Listing 7. Dangling stack

```
void f(void) {
  int *x; *x = 2; }
void g(void) {
  int *x; *x = 3; }
```

Listing 8. Temporal confidentiality

```
int g(char* z, char* in)
int f(char* in) {
  int *y = 2;
  char* z = ...;
  g(z, in);
  assert (y == 2); }
```

Listing 9. Integrity with stack objects

We now highlight two subtleties of stack safety which previous works have mostly glossed over in the context of capability machines.

2.2.1 Elaborating on Temporal Stack Safety. Stack frame lifetime intuitively dictates that the content of a popped frame should not be read once popped. Tsampas et al. define temporal stack safety as the absence of dangling stack pointers passed down the call stack (cf. Lis. 6). Here we wish to emphasize that the absence of dangling stack pointers should also mean that no caller should be able to (re)gain access to a dangling stack pointer when they resume. Concretely, consider Lis. 7, where `f` possesses some local state `x`, initialised to 2, which is copied to the local variable `y`, after which its integrity is tested with an assert statement. This assert statement appears entirely trivial. However, recall that `x` is statically allocated and that `f` may be called multiple times. Each invocation of `f` may therefore leave a copy of `x`'s address on the stack. Subsequently, if a caller can read `f`'s old stack frame then it may break the integrity of `x` in-between calls. This dangling stack attack is an additional threat in low level languages where callers may create activation records containing dangling stack pointers.

We will distinguish the absence of dangling stack pointers property from a slightly different notion of temporal stack safety, which we call *temporal confidentiality*, and which can be thought of as the temporal aspect of local state confidentiality. Consider two programs `f` and `g` whose only difference is to leave different traces on their respective stack frames, e.g., as in Lis. 8. Then, as long as temporal confidentiality is enforced, no caller should be able to distinguish `f` from `g`. We remark that the complete absence of any dangling stack pointer (passed down or otherwise) implies temporal confidentiality, without having to clear any parts of the stack.

2.2.2 Stack Safety in the Presence of Stack Objects. We now explain how *stack objects* may influence stack safety properties. In prior work on local capability machines, stack objects have largely been ignored. However, they have a significant impact on the guarantees provided by the capability machine. Disallowing stack objects altogether is too restrictive as it is a common programming idiom in C-like languages to pass stack references as arguments.

Let us consider what happens to local state integrity in the presence of stack objects. Consider for instance the example in Lis. 9 in which `f` receives an input from a caller and passes it along with

Table 1. Guarantees granted by the calling convention.

	LOCAL + clear		LOCAL + U		LINEAR		TEMPORAL		DIRECTED + U	
	w\o	w	w\o	w	w\o	w	w\o	w	w\o	w
LSE integrity	✓	✗	✓	✗	✓[2]	✓[2]	N\A	N\A	✓	✗
LSE confidentiality	✓	✗	✓	✗	✓[2]	✓[2]	N\A	N\A	✓	✗
WBCF	✓	✗	✓	✗	✓	✓	N\A	N\A	✓	✗
Temp. confidentiality	✓	✓	✓	✓	✗	✗	N\A	N\A	✓	✓
Dangling stack	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓

its own stack object to its callee. In this scenario, neither f's caller, nor g are trusted. In fact, they may collaborate to break y's integrity. Indeed, if no precaution is taken by f, it may be possible that the stack object passed by its caller actually possesses *write* authority on f's stackframe, which could be abused by g.

2.3 Enforcing Stack Safety in Capability Machines

Let us recap the stack safety properties we have isolated thus far. (1) *Local state integrity* (LSE integrity) guarantees that a callee cannot break the integrity of local stack frames (Listings 1 and 9), and that neither the callee nor a caller can break the integrity of the private environment associated with a closure (Lis. 2). (2) *Local state confidentiality* (LSE confidentiality) guarantees that the local stack frame cannot influence the behaviour of a callee (Lis. 3), and that the private environment of a closure cannot influence the behaviour of a callee or a caller (Lis. 4). (3) *Well-bracketed control flow* (WBCF) guarantees that a callee returns to its immediate caller in the call stack (Lis. 5). (4) *Temporal stack safety* according to Tsampas et al. guarantees that a callee cannot return a dangling stack pointer (a property that is violated in Lis. 6). We expand on that notion and additionally guarantee that no caller can restore a dangling stack pointer upon return either (Lis. 7). Finally (5) *temporal confidentiality* guarantees that a caller is unable to read popped stack frames, or, in other words, that popped stack frames cannot influence the behaviour of the caller (Lis. 8). Temporal confidentiality can also be interpreted as the temporal aspect of local state confidentiality.

Each of these properties can be investigated with or without the presence of stack-object parameters. It is in general easier to guarantee these properties by altogether disallowing stack objects. Passing a stack object from a caller to a callee is not safe, unless certain conditions are dynamically checked in between, in case an overlapping stack pointer (potentially breaking integrity) can be reached from that stack object.

Together, these properties make up Anderson et al.'s notion of observable stack safety. In order for the calling convention of a capability machine to be fully stack safe, it must enforce each of these properties. Unfortunately, the current state of the art enforce them at a varying degree of efficiency. In particular, enforcing temporal stack safety appears always to come at a significant cost.

Table 1 relates previous work on capability machine stack safety and also the work presented in this paper to these properties, each in a situation where stack objects are not passed from caller to callee (w\o), and in a situation stack objects are allowed (w). A ✓ means that the property is guaranteed by the associated calling convention. The different calling conventions have more or less overhead, in terms of the amount of stack clearing required by the calling convention. A ✓ depicts a

[2] While we mark the StkTokens calling convention [Skorstengaard et al. 2019b] as enforcing LSE as the authors claim, it is actually unclear whether it does protect more than just the local stackframe as done in the other works [Georges et al. 2021; Skorstengaard et al. 2018]. Indeed, the calling convention does not seem to prevent one from leaving a capability to some private state on the stack and returning without clearing the stackframe.

high overhead, on the order of the full stack size, a ✓ depicts a relatively low overhead, on the order of a single stack frame, and a ✓ depicts a low overhead of constant time. A ✗ means some additional check is needed to guarantee the property (it does not mean it is impossible to guarantee a given property, but rather that it requires some additional mechanism beyond the calling convention). Finally, `N\A` means that the property is assumed to hold given the granularity of the capability machine language. (The marks come from our understanding of the earlier work, supported by the various examples verified in each model.) The first column outlines the calling convention using LOCAL capabilities and full stack clearing [Skorstengaard et al. 2018]. The second column outlines the calling convention using uninitialized capabilities and partial clearing [Georges et al. 2021]. The third column outlines the calling convention using LINEAR capabilities [Skorstengaard et al. 2019b], and the fourth column outlines a more high level language using TEMPORAL capabilities [Tsampas et al. 2019]. The rightmost column gives an overview of the novel calling convention using the DIRECTED capabilities we introduce in this paper.

We remark that the LINEAR column shows a calling convention that checks many of the boxes and, in fact, we conjecture that a (LINEAR + uninitialized)-based calling convention could check all boxes. Thus the reader may wonder why we introduce a new kind of capability and a new calling convention here. There are several reasons: first, linear capabilities can be cumbersome to use, as only the top part of a stack frame can be passed as parameters. Second, exceptions cannot be implemented efficiently. Third, and most importantly, LINEAR capabilities are expensive to realize in practice. Moving a LINEAR capability requires an atomic move which is believed by hardware developers to lead to an undesirable overhead in runtime [Skorstengaard 2019, §3.6.2]. Indeed, this was also the reason why Georges et al. [2021] considered local and uninitialized capabilities instead.

In fact, in order to reach full stack safety à la [Anderson et al. 2021], there is a cost to each existing calling convention. The excessive stack clearing of [Skorstengaard et al. 2018] was improved upon in [Georges et al. 2021], however the latter only achieves temporal stack safety by clearing local stack frames upon return. Tsampas et al. [2019] propose temporal capabilities as an enforcement mechanism to prevent dangling stack pointers. However, they would require an expensive amount of bits to represent.

In this paper, we propose DIRECTED capabilities as an *efficient* enforcement mechanism (both wrt. space and time complexity) of full stack safety. § 3.3 presents the definition of DIRECTED capabilities, and in § 4 to 6 we show how DIRECTED capabilities can be used to enforce stack safety. Our calling convention does not use any stack clearing at all. Furthermore, DIRECTED capabilities can be efficiently realized in practice, requiring only one additional bit in the representation of capabilities, and with only one additional dynamic bounds check which is similar to existing ones (and hence efficient).

We define a unary model to reason about integrity properties, and a binary model to reason about confidentiality properties (including temporal confidentiality). We use these models to reason about small but challenging examples; we focus on examples that depend on properties not previously considered on a low level capability machine (integrity in the presence of stack objects, and temporal confidentiality). Furthermore, we follow the methodology presented by Skorstengaard et al. [2019b] and define an overlay semantics that clearly enforces each of the properties in Table 1. In § 6.2, we show how our new calling convention is fully abstract with respect to this overlay semantics.[3]

[3] In light of this full abstraction result, the reader may wonder why we also develop the logical relations models. The reason is that while the overlay semantics makes some properties obvious (e.g., popping stack frames upon return), it is not easy to use the overlay semantics for reasoning about concrete examples. This is not so surprising: even for high-level languages like ML, scientists have had to invent Kripke logical relations (and other kinds of) models to reason about local state encapsulation, e.g., [Ahmed et al. 2009; Dreyer et al. 2010a; Sumii and Pierce 2007].

$$\begin{array}{lcll}
a & \in & \mathrm{Addr} & \triangleq\ [0, \mathrm{AddrMax}] \\
p & \in & \mathrm{Perm} & ::=\ \textsc{o} \mid \textsc{e} \mid \textsc{ro} \mid \textsc{rx} \mid \textsc{rw} \mid \textsc{rwx} \\
 & & \mid \textsc{rwl} \mid \textsc{rwlx} \mid \textsc{urw} \mid \textsc{urwl} \mid \textsc{urwx} \mid \textsc{urwlx} & \\
g & \in & \mathrm{Locality} & ::=\ \textsc{global} \mid \textsc{local} \mid \textsc{directed} \\
c & \in & \mathrm{Cap} & \triangleq\ \{(p, g, b, e, a) \mid b, e, a \in \mathrm{Addr}\} \\
w & \in & \mathrm{Word} & \triangleq\ \mathbb{Z} + \mathrm{Cap}
\end{array}
\qquad
\begin{array}{lcll}
r & \in & \mathrm{RegName} & ::=\ \mathsf{pc} \mid \mathsf{r}_0 \mid \mathsf{r}_1 \mid \ldots \\
reg & \in & \mathrm{Reg} & \triangleq\ \mathrm{RegName} \rightarrow \mathrm{Word} \\
m & \in & \mathrm{Mem} & \triangleq\ \mathrm{Addr} \rightarrow \mathrm{Word} \\
\varphi & \in & \mathrm{ExecConf} & \triangleq\ \mathrm{Reg} \times \mathrm{Mem} \\
\delta & \in & \mathrm{ExecMode} & ::= \\
 & & \multicolumn{2}{l}{\mathsf{Executable} \mid \mathsf{Halted} \mid \mathsf{Failed}}
\end{array}$$

$$\begin{array}{lcl}
\rho & \in & \mathbb{Z} + \mathrm{RegName} \\
i & ::= & \texttt{jmp}\ r \mid \texttt{jnz}\ r\ r \mid \texttt{move}\ r\ \rho \mid \texttt{load}\ r\ r \mid \texttt{store}\ r\ \rho \mid \texttt{add}\ r\ \rho\ \rho \mid \texttt{sub}\ r\ \rho\ \rho \mid \\
 & & \texttt{lt}\ r\ \rho\ \rho \mid \texttt{lea}\ r\ \rho \mid \texttt{restrict}\ r\ \rho \mid \texttt{subseg}\ r\ \rho\ \rho \mid \texttt{isptr}\ r\ r \mid \texttt{getp}\ r\ r \mid \texttt{getl}\ r\ r \mid \\
 & & \texttt{getb}\ r\ r \mid \texttt{gete}\ r\ r \mid \texttt{geta}\ r\ r \mid \texttt{fail} \mid \texttt{halt} \mid \texttt{loadU}\ r\ r\ \rho \mid \texttt{storeU}\ r\ \rho\ \rho \mid \texttt{promoteU}\ r
\end{array}$$

Fig. 2. Machine words, machine state and instructions.

$$\textsc{ExecStep}$$

$$(\mathsf{Executable},\ \varphi) \rightarrow \begin{cases} [\![decode(z)]\!](\varphi) & \text{if } \ \varphi.\mathrm{reg}(\mathsf{pc}) = (p, g, b, e, a) \wedge b \le a < e \wedge \\ & \qquad p \in \{\textsc{rx}, \textsc{rwx}, \textsc{rwlx}\} \wedge \varphi.\mathrm{mem}(a) = z \\ (\mathsf{Failed},\ \varphi) & \text{otherwise} \end{cases}$$

Fig. 3. Operational semantics: reduction steps.

3 CAPABILITY MACHINE: OPERATIONAL SEMANTICS AND CALLING CONVENTION

In this section we present the operational semantics of the capability machine we consider. Our capability machine is based on the one by Georges et al. [2021] and is, transitively, inspired by CHERI [Watson et al. 2015] and the M-Machine [Carter et al. 1994].

In § 3.1, we first recall from Georges et al. [2021] how the operational semantics for a capability machine with local and uninitialized capabilities is defined. Then, in § 3.2, we further recall how said capabilities can be used to enforce LSE and WBCF via a secure calling convention. We then add support for directed capabilities in § 3.3, and present our new improved calling convention, which can finally *efficiently* guarantee temporal stack safety, in § 3.4. Figures 2 to 5 summarize the operational semantics; components marked in blue are for the novel directed capabilities and will be detailed in § 3.3.

Fig. 2 describes the syntax of our capability machine. We model a capability machine with finite memory. The set of addresses Addr is defined as the interval $[0, \mathrm{AddrMax}]$, where AddrMax is the top address and cannot be dereferenced. Memory contains machine words w that are represented by either an (unbounded) integer or a capability. A capability is a quintuple (p, g, b, e, a) representing the authority to exert the permission p over the memory range $[b, e[$ and currently pointing to a.

3.1 A Capability Machine with Local and Uninitialized Capabilities

A permission p can either be opaque (o), enter (e), read-only (ro), read-execute (rx), read-write (rw), read-write-execute (rwx), read-write-local (rwl), read-write-local-execute (rwlx), or uninitialized-rw (urw), uninitialized-rwl (urwl), uninitialized-rwx (urwx), uninitialized-rwlx (urwlx). Permissions form a lattice as illustrated in Fig. 4. The permissions ro, rx, rw, rwx are standard. Permission o provides no authority. Enter (e) capabilities represent opaque closures encapsulating code and data. As such, they cannot be read, written to, executed nor modified. They can only be jumped to, which will load them into the program counter register and unseal them into a rx capability. Their usage will be further illustrated when describing the operational semantics and the calling convention. Locality g is either global or local, and forms a lattice as illustrated in Fig. 4. local capabilities are meant to represent stack derived capabilities, while global ones represent heap derived ones. They will be described further in § 3.2. Write-local permissions (RWL and RWLX) are similar to their regular counterparts, but additionally provide the authority to write LOCAL capabilities to memory. That is, a regular RW capability cannot be used to write a LOCAL capability to memory, only GLOBAL ones. Finally, uninitialized capabilities Uπ represent a form of use-after-write authority: they provide permission π over the range $[b, a[$ and write permission on range $[a, e[$ — the boundary is automatically increased when the capability is used to write at a.

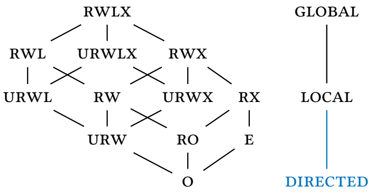


Fig. 4. Permission and locality hierarchy.

Machine instructions i operate over registers or constants and their behaviour will be detailed later. A register is either the program counter pc or a general purpose register $r_0, r_1, \ldots$. A machine state is composed of a mode describing whether the machine is in an Executable state, or in a terminal Failed or Halted state, and an execution configuration. An execution configuration is a pair of a register file, mapping registers to their values, and a memory state, mapping addresses to their values. The operational semantics of the machine shown in Fig. 3 is given in smallstep style and has only one rule: if the state is Executable and the program counter contains an in-bounds executable capability pointing to some machine word, then an instruction is decoded and executed; otherwise the machine fails. Recall that, in general, failing is considered safe since it crashes the machine before anything unsafe occurs.

The semantics of instructions is given in Fig. 5. Most instructions increment the program counter at the end of their execution, except for branching and terminating instructions. This process is defined by updPC, which fails if the program counter does not contain an executable capability. Terminating instructions `fail` and `halt` change the machine state respectively to a Failed and Halted state. The `move` instruction copies a machine word in a register. The `load` instruction reads from the memory into a register, provided that a capability with sufficient authority is given. Similarly, the `store` instruction is used to write to memory. Additionally, if the word that is stored is a local capability, it must be that the capability provided has write-local authority. The `jmp` instruction copies a word to the program counter, additionally, if the word is an enter capability, it is unsealed into a RX one. The instructions `restrict` and `subseg` are used to decrease the authority of a capability. The former decreases the permission and the locality of a capability following the lattice given in Fig. 4. The latter decreases the range of authority of a capability. The `lea` instruction is used to change the current address a capability points to. As explained earlier, an enter capability cannot be modified, and `subseg` and `lea` will thus fail. Furthermore, as the current address of an uninitialized capability indicates the boundary between where its regular authority applies and where it can only write, it is only safe to decrease the current address using `lea`. The instructions `getp`, `getl`, `getb`, `gete` and `geta` can be used to retrieve respectively the permission, locality, base address, end address and current address of a capability. `loadU` and `storeU` are similar to their regular counterparts, but operate only with uninitialized capabilities. An additional offset parameter is provided in order to be able to access the range that has already been written to. Moreover, if the offset provided is 0, then the boundary of the capability is incremented by `storeU`. The `promoteU` instruction can promote an uninitialized capability to its regular counterpart by discarding the memory range that has not been written to yet.

3.2 A Secure Calling Convention using Local and Uninitialized Capabilities

We now give an intuitive account of how the calling conventions of Skorstengaard et al. [2018, 2019a] and Georges et al. [2021] enforce local state encapsulation and well-bracketed control flow.

$$\text{updPC}(\varphi) = \begin{cases} (\text{Executable}, \varphi[\text{reg.pc} \mapsto (p, g, b, e, a+1)]) & \text{if } \varphi.\text{reg}(\text{pc}) = (p, g, b, e, a) \text{ and } \textsc{rx} \npreccurlyeq p \\ (\text{Failed}, \varphi) & \text{otherwise} \end{cases}$$

$$\text{getWord}(\varphi, \rho) = \begin{cases} \rho & \text{if } \rho \in \mathbb{Z} \\ \varphi.\text{reg}(\rho) & \text{if } \rho \in \text{RegName} \end{cases} \quad \text{canReadUpTo}(w) = \begin{cases} \bot_{\textsc{Addr}} & \text{if } w \in \mathbb{Z} \\ \min(a, e) & \text{if } w = (\textsc{u}\pi, \_, \_, e, a) \\ e & \text{if } w = (\pi, \_, \_, e, \_) \end{cases}$$

i	$[\![i]\!](\varphi)$	Conditions
`fail`	(Failed, φ)	
`halt`	(Halted, φ)	
`move` r ρ	$\text{updPC}(\varphi[\text{reg}.r \mapsto w])$	$w = \text{getWord}(\varphi, \rho)$
`load` r_1 r_2	$\text{updPC}(\varphi[\text{reg}.r_1 \mapsto w])$	$\varphi.\text{reg}(r_2) = (p, g, b, e, a)$ and $w = \varphi.\text{mem}(a)$ and $b \leq a < e$ and $p \in \{\textsc{ro}, \textsc{rx}, \textsc{rw}, \textsc{rwx}, \textsc{rwl}, \textsc{rwlx}\}$
`store` r ρ	$\text{updPC}(\varphi[\text{mem}.a \mapsto w])$	$\varphi.\text{reg}(r) = (p, g, b, e, a)$ and $b \leq a < e$ and $p \in \{\textsc{rw}, \textsc{rwx}, \textsc{rwl}, \textsc{rwlx}\}$ and $w = \text{getWord}(\varphi, \rho)$ and if $w = (\_, \textsc{local}, \_, \_, \_)$, then $p \in \{\textsc{rwlx}, \textsc{rwl}\}$ and if $w = (\_, \textsc{directed}, \_, \_, \_)$, then $p \in \{\textsc{rwlx}, \textsc{rwl}\}$ and $\text{canReadUpTo}(w) \leq a$
`jmp` r	$(\text{Executable},$ $\varphi[\text{reg.pc} \mapsto \mathit{newPc}])$	if $\varphi.\text{reg}(r) = (\textsc{e}, g, b, e, a)$, then $\mathit{newPc} = (\textsc{rx}, g, b, e, a)$ otherwise $\mathit{newPc} = \varphi.\text{reg}(r)$
`restrict` r ρ	$\text{updPC}(\varphi[\text{reg}.r \mapsto w])$	$\varphi.\text{reg}(r) = (p, g, b, e, a)$ and $(p', g') = \text{decodePermPair}(\text{getWord}(\varphi, \rho))$ and $(p', g') \preccurlyeq (p, g)$ and $w = (p', g', b, e, a)$
`subseg` r ρ_1 ρ_2	$\text{updPC}(\varphi[\text{reg}.r \mapsto w])$	$\varphi.\text{reg}(r) = (p, g, b, e, a)$ and for $i \in \{1, 2\}$, $z_i = \text{getWord}(\varphi, \rho_i)$ and $z_i \in \mathbb{Z}$ and $b \leq z_1$ and $0 \leq z_2 \leq e$ and $p \neq \textsc{e}$ and $w = (p, g, z_1, z_2, a)$
`lea` r ρ	$\text{updPC}(\varphi[\text{reg}.r \mapsto w])$	$\varphi.\text{reg}(r) = (p, g, b, e, a)$ and $z = \text{getWord}(\varphi, \rho)$ and $p \neq \textsc{e}$ and $w = (p, g, b, e, a+z)$ $p = \textsc{u}$-, then $z \leq 0$
`geta` r_1 r_2	$\text{updPC}(\varphi[\text{reg}.r_1 \mapsto a])$	$\varphi.\text{reg}(r_2) = (\_, \_, \_, \_, a)$
`loadU` r_1 r_2 ρ	$\text{updPC}(\varphi[\text{reg}.r_1 \mapsto w])$	$\varphi.\text{reg}(r_2) = (p, g, b, e, a)$ and $p = \textsc{u}$- and $\mathit{off} = \text{getWord}(\varphi, \rho)$ and $b \leq a + \mathit{off} < a \leq e$ and $w = \varphi.\text{mem}(a + \mathit{off})$
`storeU` r ρ_1 ρ_2	$\text{updPC}(\varphi'$ $[\text{mem}.(a + \mathit{off}) \mapsto w])$	$\varphi.\text{reg}(r) = (p, g, b, e, a)$ and $p = \textsc{u}$- and $\mathit{off} = \text{getWord}(\varphi, \rho_1)$ and $w = \text{getWord}(\varphi, \rho_2)$ and if $w = (\_, \ell, \_, \_, \_)$ and $\ell \neq \textsc{global}$ then $p \in \{\textsc{urwlx}, \textsc{urwl}\}$ and $b \leq a + \mathit{off} \leq a < e$ and if $\mathit{off} \neq 0$ then $\varphi' = \varphi$ else $\varphi' = \varphi[\text{reg}.r \mapsto (p, g, b, e, a+1)]$ and if $\ell = \textsc{directed}$, then $\text{canReadUpTo}(\mathrm{w}) \leq a + \mathit{off}$
`promoteU` r	$\text{updPC}(\varphi[\text{reg}.r \mapsto w])$	$\varphi.\text{reg}(r) = (p, g, b, e, a)$ and $p = \textsc{u}\pi$ and $w = (\pi, g, b, \min(a, e), a)$
…		
_	(Failed, φ)	otherwise

Fig. 5. Operational semantics: instruction semantics.

The calling convention of Skorstengaard et al. [2018] uses local capabilities (not uninitialized capabilities) and requires that a program is initially provided with a stack capability with authority over the whole stack in a register r_{stk} when executed. Assume the following scenario where Alice calls Bob who calls Claire. We explain how Bob can protect himself from both Alice and Claire using the calling convention of Skorstengaard et al. [2018].

Bob expects to receive a stack capability from Alice to build his own stack frame. Similarly, when calling Claire, Bob needs to provide the stack capability to her. However, in order to enforce local state encapsulation, it is necessary that Bob does not provide access to his own stackframe to Claire. Thus, when calling Claire, Bob restricts the stack capability to the unused part, using the `subseg` instruction, and then passes it to her. However, Bob needs to be able to restore access to his own stack frame upon return. He can do that using an enter capability: Bob constructs a return capability as an enter capability that restores the local environment when jumped to. This return capability can be safely passed to the callee Claire as its contents cannot be read, but can only be jumped to. This suffices to protect Bob's private state from Claire, but it is not enough to enforce WBCF. Indeed, on its own, this does not prevent the attack explained in § 2.1.2, in which a callee keeps a copy of a previous return capability beyond its "lifetime". To prevent this kind of attack, Skorstengaard et al. use local capabilities: The stack capability is made write-local, executable (RWLX) and LOCAL, and all other (heap) capabilities are non write-local. This guarantees that if the return capability is built on the stack (and therefore LOCAL), then the only place Claire can keep a copy of a return capability, is on the stack itself. Consequently, *by clearing the stack* before passing it to Claire, Bob can be sure that she will not be able to recover a previously left copy of a return capability. Finally, to protect himself from Alice, Bob also clears the whole stack and the registers before returning, so that Alice cannot access anything. Skorstengaard et al. later point out that it is sufficient for Bob to only clear his own stack frame, as anything that Claire may leave on the stack either originally came from Alice, or is a return capability from Bob, and, as Bob clears his own stack frame, using the return capability will only lead to cleared data. As briefly mentioned in § 2, the original calling convention by Skorstengaard et al. [2018] enforces temporal confidentiality with an excessive amount of clearing. With the optimization by Skorstengaard et al. [2019a], one only clears one's own stack frame on return. Although this improves the efficiency of the calling convention, every secure closure must clear its own stack frame upon return.

Furthermore, even with the optimization mentioned by Skorstengaard et al. [2018], Bob still needs to clear the whole stack before being able to call Claire safely, and hence the calling convention is still very inefficient. To address this issue, Georges et al. [2021] proposed to make the stack capability into an uninitialized capability. By passing an uninitialized URWLX stack capability to Claire, we are guaranteed that Claire cannot read anything left on the stack, without overwriting it beforehand, and thus there is no need to clear the whole stack before calling Claire. However, Bob still needs to clear his own stack frame before returning, as there is no guarantee that Alice did not keep a "fully initialized" stack capability, which would allow her to read leftover data on the stack. For this calling convention, Georges et al. [2021] prove that LSE and WBCF are enforced; however dangling stack pointers are still a possibility, and thus some clearing is still needed in order to guarantee temporal confidentiality.

3.3 Directed Capabilities

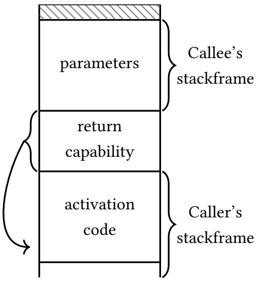
We now introduce a novel kind of directed capabilities and then explain, in § 3.4, our new improved calling convention, which relies on directed capabilities to efficiently guarantee LSE, WBCF and temporal stack safety (as we prove in later sections).

The intention of directed capabilities is to restrict where they can be stored in memory. This is done by adding a new locality DIRECTED, as illustrated in the locality lattice in Fig. 4. To write a DIRECTED capability to memory it is then necessary to have permit-write-local authority (similarly to writing LOCAL capabilities to memory), as shown in the operational semantics of `store` and `storeU` in Fig. 5. The distinguishing feature of a directed capability is that it cannot be stored "below" where it can read memory up to. That is, for a directed capability with a *regular* permission (i.e., not uninitialized) with authority over range $[b, e[$, it can only be stored at an address a such that $e \leq a$. For an uninitialized directed capability $(\mathrm{U}\pi, \ell, b, e, a)$, the part $[a, e[$ can only be written to, therefore it can only be stored at an address a' such that $a \leq a'$. The intuition is that, for a stack that grows upwards, the address a stack capability can read up to implicitly approximates the lifetime of the capability. Given two directed capabilities, if the first can read at a lower address than the second, then the first is owned by an "older" stack frame than the second and has thus a longer lifetime.

We remark that, from an hardware implementation point of view, directed capabilities should be quite easy to implement. First, uninitialized directed capabilities require only two additional bits, one for the directed locality, and one indicating whether it is uninitialized. Since increased pointer size can severely affect performance, CHERI Concentrate [Woodruff et al. 2019] employs a rigorous compression scheme to achieve realistic performance. Within this scheme, 2 and 7 bits are reserved for future use in the CHERI-64 and CHERI-128 respective compression formats. The two necessary additional bits are thus already available in either format. We can contrast that to temporal capabilities [Tsampas et al. 2019], which require n bits to encode the lifetime information for a call depth of size 2^n. The required number of bits is thus unbounded, and it is unclear how to determine the ideal least number of bits. Tsampas et al. discuss this exact point, and propose various workarounds. Directed capabilities, on the other hand, already fit within existing formats.

Second, directed capabilities only change the semantics of `load(U)` and `store(U)` by adding an extra bounds check. The added bounds check is no different from current checks, and we expect existing optimisation patterns, such as parallelisation, to apply.

3.4 A New Calling Convention using Directed Capabilities



Let us revisit the example in § 3.2, now assuming that the stack capability is directed instead of local. When Bob is called by Alice, Bob can ensure that Alice did not keep a capability with *read* authority on the unused part of the stack by checking that the return capability that he received is not "above" the stack capability he received. This check can be avoided if the return capability is passed on the stack as part of the calling convention. Indeed, if the return capability is stored at some address a', Bob knows by property of directed capabilities that Alice can only have kept a copy of the stack capability with read authority at most up to a'. Thus, Bob will not have to clear his stackframe on return. On the other hand, if Bob passes some stack references to Claire as parameters, Claire will not be able to store anything from her own stackframe in them, thus avoiding the issues described in § 2.2. In fact, Bob can take advantage of this property to ensure that Claire only returns safe values. By passing a stack capability rather than a dedicated register as the return value destination, Bob knows that any return value cannot grant read authority over popped stack frames. The calling convention assumes this strategy, and clears all general purpose registers upon return.

To sum up, our new calling convention is as follows; see also the figure above, which shows what the stack is expected to look like just after a call.

At program start-up. A directed URWLX stack capability is in register r_{stk}.

When called by an adversary. Check that the received stack capability is a capability of the form $(\text{URWLX}, \text{DIRECTED}, b, e, a)$, the return capability is expected to be stored at address b.

Before calling an adversary. Push activation record to the stack and create a directed E-capability to use as return capability. `Subseg` the stack capability to the unused part. Push the return pointer on the stack, as well as all parameters. Clear all registers except r_{stk} and the program counter before calling.

Before returning to an adversary. Clear all general purpose registers.

Let's consider the cost of one secure call. Our new calling convention does not require any memory clearing, and thus incurs a constant overhead, mainly of clearing registers. Register clearing can be done efficiently using the `CClearRegs` instruction [Watson et al. 2020]. On the other hand, memory clearing is a costly operation. Previous calling conventions based on LOCAL capabilities all require some amount of stack clearing. Although Georges et al. [2021] improves the situation by only clearing the local stack frame, the calling convention still has an undesired linear cost in the amount of stack memory used.

In summary, we present a faster calling convention, which can realistically be implemented in hardware. Moreover, the calling convention enforces all desired stack safety properties as is proved in the next sections.

3.5 Discussion

While directed capabilities lead to an efficient calling convention, the questions remains whether their new restrictions render them impractical otherwise. Specifically, do directed capabilities hinder critical C idioms, or compiler optimizations to any significant extent. A full investigation of this question is beyond the scope of this paper. However, we conjecture that directed capabilities are sufficiently permissive in practice, and in some cases more practical than, e.g., linear capabilities.

Some objects cannot be allocated on the stack. For instance, a locally allocated circular linked list breaks the directed property. Similarly, any locally allocated node cannot be added to an ambient heap allocated linked list. Indeed, nor should they: stack objects and heap objects often have different lifetimes and are thus generally incompatible.

We do not expect uninitialized capabilities to have major impact on code generation, though one must be careful to initialize an uninitialized object in "increasing" order. Similarly, it takes linear time to allocate and pass stack objects, while C assumes constant time allocation. However, we hasten to point out that it is in general only safe to pass stack objects for which the previous contents cannot be read. [Roessler and DeHon 2018] enforce this efficiently using a lazy tagging and clearing scheme; it would interesting to investigate a similar scheme for uninitialized capabilties.

Furthermore, a compiler must also be careful with the order of stack allocations. Consider for instance the following code snippet: **int*** x; **int** y; x = &y. In this code, the compiler must reorder the allocations for x and y to guarantee the directed property. Such considerations must also be taken into account when implementing compiler optimizations.

All in all, directed capabilities are more restrictive (by design) than local capabilities, but we argue they remain more practical than linear capabilities. For instance, it is a common idiom in C to pass pointers as an argument but not return them to the caller. With linear capabilities, a compiler would need to ensure that all linear capabilities be returned when used in this fashion. Furthermore, some library functions such as

```
int memcmp(const void* p1, const void* p2, size_t size)
```

are not implementable in an linearity-friendly way, since p1 and p2 are allowed to be aliases.

We leave a full practical evaluation of directed capabilities to future work.

4 A UNARY MODEL FOR INTEGRITY

In this section, we develop a novel model that captures all the guarantees provided by DIRECTED capabilities and our associated calling convention. The core novelty lies in its ability to express temporal stack safety. The model is made up of two components; a program logic to reason about known and trusted code, and a Kripke logical relation to reason about arbitrary untrusted code. We use the unary model to reason about the *integrity* of example programs.

The program logic is a variant of the one by Georges et al. [2021]; the main change is that some proof rules have been updated to account for directed capabilities, following the operational semantics defined in § 3.1. Thus we refrain from describing the program logic in detail.

Here it suffices to know that the program logic is defined using Iris' weakest preconditions [Jung et al. 2018] (which means that we can re-use the Iris program logic infrastructure to reason formally in Coq) and that the weakest precondition predicate wp Executable $\{v, Q(v)\}$ intuitively means the program pointed to by the program counter can execute without getting stuck, and that if it terminates, then $Q(v)$ is guaranteed to hold for some final mode v, which can be either Halted or Failed.

Thanks to the dynamic checks of the capability machine, the behaviour of a program is limited by the capabilities it has access to. Thus, even completely arbitrary code must adhere to rules imposed by the capability machine, and will satisfy some notion of *capability safety*. The logical relation captures this notion of capability safety, and serves as a contract between trusted and untrusted code. The fundamental theorem of logical relations (see below) means that any word that is safe to read satisfies that contract. As long as arbitrary code is just a list of instructions (and thus does not, e.g., include an embedded capability), a corollary then states that even completely arbitrary code satisfies that contract.

Our logical relation is an extension of the one by Georges et al. [2021] and our presentation focuses on the key challenge, which is to extend the model, in particular, the Kripke worlds, to capture the enforcement of temporal properties qua directed capabilities. The execution of a program depends on the physical state of memory. Since we want to reason about stack safety, we are particularly interested in the state of the stack. During execution, different parts of the stack are in different states (e.g. used, unused, etc.). Following Georges et al., we use a Kripke world to model the abstract state of the stack. In essence, the Kripke world is an abstraction of physical memory. Concretely, it tracks which parts are in heap space, which parts are in stack space, and at what particular state a location is. The stack may change in ways that accord with the specific properties we attribute to stack safety. We capture stack-based properties by carefully describing the possible changes to the Kripke world so that they reflect the expected behaviour of the physical stack.

From a technical point of view, a WORLD has two parts: W^{std} maps addresses to so-called *standard states*, governing *shared* regions, such as the stack; and W^{cus} maps a countably infinite set of region names to custom state transition systems, governing *owned* regions, such as the private environment of closures.

The logical relation imposes certain invariants on regions of memory that is shared between trusted and untrusted code. Those invariants depend on the state of the stack, and thus on the Kripke world that represents it. We use Iris ghost state to track not only the physical machine state, but also what we call the *instrumented machine state*, which captures the connection between the physical memory, and the abstract state of memory. It uses an Iris predicate called the *standard resource*. A standard resource has two functions: (1) it associates an address of a shared region of memory, in particular each stack address, to its physical state, and (2) it associates that physical state to an invariant. The invariant may depend on the current state of the Kripke world. Our model extends Georges et al. [2021] insofar as it uses the same structure for the instrumented machine state. However, in order to capture temporal properties, we define a *novel* Kripke world, upon which we build new definitions for the standard resources. In this paper, we focus a large part of our attention on the new Kripke world.

In the remainder of this section we first describe the standard states we use to capture stack and heap states, and how these states may evolve, such that they can be used to capture the desired stack properties (§ 4.1). In § 4.2, we will then present the logical relation itself. Finally, we end this section with two examples highlighting what kind of programs we can now verify using our model (§ 4.3).

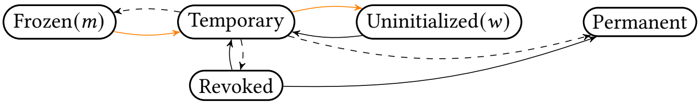


Fig. 6. Standard State Transition System. Full lines indicate public transitions, dashed lines indicate private transitions, orange lines indicate temporal transitions.

4.1 A New Kripke World

The standard states represent the various states a shared memory address can be in. An address is shared if it lies within the region of authority of a capability that crosses the boundary between known and arbitrary code. The physical state of these addresses will be imposed by invariants, which may in turn depend on the current state of the rest of the machine. For instance, the invariant of a heap region should not be able to depend on the state of the stack and its changes, since locality dictates that the heap is unable to contain stack pointers. Likewise, the invariant of a stack address connected to a lower stack frame will be different from the invariant of the currently live, or popped stack frame. The standard states must therefore also reflect the very specific lifetime properties of a stack frame.

We now explain each of the standard states. The Permanent state represents an allocated heap region. As soon as a heap region is allocated and shared, it becomes permanent. There is no mechanism to free the heap region from its state. The Temporary state represents a live stack frame, i.e., the stack frame owned by the currently executing function. Specifically, it will refer to the *readable* parts of the stack shared between calls. A live stack frame does not need to be Temporary at every step of execution. Rather, the Temporary state is meant to represent the live parts of the stack at the point of change of control. The Uninitialized state represents the unused part of the stack, i.e., a Freed frame or some part of the stack that has never been used. In general, the Uninitialized state will simply refer to the parts of the stack that cannot be read from, only written to. The Frozen(m) state represents a frozen stack frame, i.e., the parts of a frame not shared with a new callee. Here m maps the addresses of that particular frame to their frozen values. The Revoked state represents a part of the stack which is currently owned by an executing function. A region is not part of the shared stack while it is Revoked.

Next we define how a standard state may evolve in order to reflect physical stack changes that accord with the key properties of stack safety (LSE, WBCF, and temporal stack safety). Some stack changes are not observable by any caller or callee, whereas other changes are public and observable by both. As long as temporal stack safety is enforced, some stack changes (such as popping a stack frame) are safe to observe by the caller, but not by higher stack frames. After all, it is not safe to pop a frame if there are still live frames on top of it.

We use three kinds of transitions to reflect these distinctions. A transition is *observable* whenever some entity is oblivious to that transformation. Public transitions (depicted in Fig. 6 as straight black lines) are those that are observable by all functions. Private transitions (dashed lines) can only be observed by the currently executing function. Finally, temporal transitions (orange lines) are only observable by functions that are still present on the call stack.

Using these transitions, we isolate three future world relations. A world W' is a public future world of W, $W' \sqsupseteq^{pub} W$, if all the states in W' are either connected to a fresh address or region name, or were updated from W through public transitions only. A private future world $W' \sqsupseteq^{priv} W$ allows for public, temporal or private transitions. Finally, the third relation we consider is in fact a family of future world relations, where each relation is indexed by an address. We say that W' is a future world of W relative to address a, written as $W' \sqsupseteq^{a} W$, when the state of all addresses *below*

a were updated via public transitions only, while addresses *at or above* a were updated via public or temporal transitions.

For instance, consider an address a' that is Temporary in W. If that address lies below a, it must still be Temporary in W' if $W' \sqsupseteq^a W$. However, if that address lies at or above a, it may change to an uninitialized state. Likewise, any address at or above a with an Uninitialized(w) state in W may change to a new Uninitialized(w') state.

A relative future world relation captures the changes to a stack *relative* to a specific stack frame (delimited by its upper bound address). From the point of view of a particular stack frame, a new call will push then pop new stack frames, whereas that stack frame *remains* frozen or initialized. Upon return of a well-bracketed call, a stack frame is safe to pop. In other words, invoking a return capability should be safe to do in a world where the current as well as all higher stack frames are uninitialized. In $W' \sqsupseteq^a W$, world W' represents such a world, from the perspective of a stack frame with upper bound a.

The instrumented machine state imposes monotonicity requirements on the invariants associated with shared addresses[4]. If a shared address a is part of the heap, then the associated invariant must be monotone with regards to $\sqsupseteq^{priv}$. On the other hand, if a is part of the (live) stack, then the associated invariant must be monotone with regards to $\sqsupseteq^a$.

We finish this section by highlighting some interesting transitions, relating them to the corresponding physical state changes of the stack:

- Temporary $\dashrightarrow$ Frozen(m): local variables freeze when their associated function makes a new call. These local variables are stored in a stack frame. The frozen part of a stack frame cannot be written to while it is frozen. A stack frame cannot stay frozen forever. In particular, a frozen stack frame must thaw when control is returned to its caller. A caller should therefore not be able to observe that the stack frame was at any point frozen during execution, hence the private transition.
- Frozen(m) $\longrightarrow$ Temporary: as indicated in the previous point, a frozen stack frame must thaw before it can be written to again. A frozen stack frame is thawed only after a callee invokes the callback. Temporal stack safety dictates that only higher stack frames can invoke the callback. Invoking the callback effectively pops the callee's stack frame. Thus a local stack frame is thawed once all higher stack frames have been popped, but has no effect on lower stack frames.
- Temporary $\longrightarrow$ Uninitialized(w): Finally, local stack frames are popped upon return. Thanks to temporal stack safety, popping a stack frame has no effect on lower stack frames, since they cannot read its content.

In summary, we have presented the standard states and transitions that make up the new Kripke world used to model LSE, WBCF and now also temporal stack safety. The world differs from [Georges et al. 2021] in the following way: we distinguish between the Uninitialized state (shared write access but no shared read access) and the Frozen state (no shared write or read access), and we introduce a new kind of transition for defining a relative future world relation, capturing the temporal properties of the stack.

4.2 A Unary Logical Relation

Fig. 7 defines the unary Kripke logical relation with support for temporal stack safety. We depict in blue the parts of the definition that are different from the unary logical relation used by Georges et al. [2021].

[4] The formal definitions of the instrumented machine state and standard resources are here omitted for brevity, full definitions can be found in the Coq formalisation.

$$\boxed{\mathcal{E}(W)(v)} \triangleq \forall reg,\ \mathcal{R}(W)(reg) * sharedResources(W) * stsCollection(W) * \mathrm{pc} \mapsto v * \mathop{\Large\ast}_{(r,w)\in reg/\mathrm{pc}} r \mapsto w \mathrel{-\!\!*}$$
$$\text{wp Executable} \left\{ \begin{array}{l} v, v = \text{Halted} \rightarrow \exists W'\ reg',\ W' \sqsupseteq^{priv} W \\ \quad * \ sharedResources(W') * stsCollection(W') * \mathop{\Large\ast}_{(r,w)\in reg'} r \mapsto w \end{array} \right\}$$

$$\boxed{\mathcal{R}(W)(reg)} \triangleq \mathop{\Large\ast}_{(r,w)\in reg/\mathrm{pc}} \mathcal{V}(W)(w)$$

$$\boxed{\mathcal{V}(W)(w)} \begin{cases} \mathcal{V}(W)(z), \mathcal{V}(W)(\textsc{o}, -) & \triangleq \top \\ \mathcal{V}(W)(p, g, b, e, a) & \triangleq \mathop{\Large\ast}_{a'\in[b,e[} \begin{cases} \mathcal{S}^u(W)(a', g, p, a) & \text{if } p = \textsc{u-} \\ \mathcal{S}(W)(a', g, p) & \text{otherwise} \end{cases} \\ & \quad \wedge \begin{cases} \exists P, rel(a', P) * \mathrm{rcond}(P) & \text{if } p \in \{\textsc{ro}, \textsc{rx}\} \\ rel(a', \mathcal{V}) & \text{otherwise} \end{cases} \\ \mathcal{V}(W)(\textsc{e}, g, b, e, a) & \triangleq \square\ \forall W' \sqsupseteq^g W, \triangleright\ \mathcal{E}(W')(\textsc{rx}, g, b, e, a) \end{cases}$$

$$\boxed{\mathrm{rcond}(P)} \triangleq \triangleright \square \forall W, w, P(W)(w) \mathrel{-\!\!*} \mathcal{V}(W)(w) * \triangleright \square \forall W_1, W_2, z, P(W_1)(z) \mathrel{-\!\!*} P(W_2)(z)$$

$$\boxed{\text{State relation}} \quad \begin{array}{l} \mathcal{S}(W)(a, g, p) \triangleq \begin{cases} W^{std}(a) \in \{\text{Temporary}, \text{Permanent}\} & \text{if } \neg\text{write-local}(p) \wedge g = \textsc{dir.} \\ W^{std}(a) = \text{Temporary} & \text{if write-local}(p) \wedge g = \textsc{dir.} \\ W^{std}(a) = \text{Permanent} & \text{if } g \neq \textsc{dir.} \end{cases} \\ \mathcal{S}^u(W)(a, g, p, mid) \triangleq \begin{cases} \mathcal{S}(W)(a, g, p) \vee \exists w, W^{std}(a) = \text{Uninitialized}(w) & \text{if } a \geq mid \wedge\ g = \textsc{dir.} \\ \mathcal{S}(W)(a, g, p) & \text{if } a < mid \vee\ g \neq \textsc{dir.} \end{cases} \end{array}$$

Fig. 7. A Logical Relation with Support for Temporal Stack Safety. $\sqsupseteq^g$ equals $\sqsupseteq^{priv}$ whenever g is GLOBAL or LOCAL, and $\sqsupseteq^e$ whenever g is DIRECTED. DIR. stands for DIRECTED

The value relation $\mathcal{V}$: WORLD → Word → *iProp* defines the validity of a word relative to a world; the register relation $\mathcal{R}$ defines the validity of a register state; and the expression relation $\mathcal{E}$ defines when it is is safe to use a word as the program counter. Note that all relations are defined in the Iris program logic (cf. the type *iProp* for Iris propositions). We now explain the definition, and first consider the value relation. Integers and words with O capability are always valid. A word with a read and/or write permission is valid in a world W only if certain conditions on W are met. A GLOBAL capability with a read and/or write permission imposes a Permanent state on its range of authority. An uninitialized capability with a write-local (i.e., it can be used to store LOCAL and DIRECTED words) permission imposes a Temporary state on its initialized readable range of authority, whereas its uninitialized part may be either Temporary or Uninitialized. The state relations $\mathcal{S}$ and $\mathcal{S}^u$ define the exact conditions on W for regular and uninitialized capabilities respectively.

A valid capability with read and/or write permission grants access to the so-called standard resources alluded to in the beginning of this section: $rel(a, \phi)$ associates the memory predicate ϕ : WORLD → Word → *iProp* to the address a. It suffices to think of $rel(a, \phi)$ as an invariant, that can be used to access the ghost state of a, while guaranteeing that ϕ holds at the current physical state of a in the current world W. Normally, the predicate we associate with such an address a would be $\mathcal{V}$ itself. However, we distinguish between a read-only and a read-write permission by allowing the associated predicate of an address within a read-only region to be stronger than $\mathcal{V}$. The predicate in question then needs to satisfy the read condition $\mathrm{rcond}(\phi)$, which imposes two restrictions on ϕ. First, it enforces that $\phi(W)(w)$ always implies validity, regardless of W and w. Second, it enforces that $\phi(W)(w)$ never depends on W when w is an integer. In other words, only capabilities can depend on the instrumented machine state. Notice how each condition is guarded by a later ($\triangleright$) modality; this is to guarantee that the definition of $\mathcal{V}$ is well defined (here we use that Iris supports the definition of guarded recursive predicates).

An E capability can only be jumped to, hence its validity is defined in terms of its safe execution. Such a capability can be jumped to at any moment and hence the property should be persistent (i.e., not rely on any ephemeral resources); this is expressed by Iris' persistence modality □. The execution of a capability may depend on the current state of the stack. For instance, a GLOBAL E capability represents a global function closure, and is safe to jump to regardless of the state of the stack. On the other hand, a DIRECTED closure (used for return pointers) is only safe to jump to at the end of a function's execution. This distinction is made by quantifying over the possible future worlds an E capability may be invoked from, see $W' \sqsupseteq^g W$. A GLOBAL or LOCAL closure can be invoked in any private future world of W, whereas a DIRECTED closure can only be invoked in a relative future world $W' \sqsupseteq^e W$, where e will represent the upper bound of the stack frame being returned to.

Finally, the safe execution of a word is defined using the expression relation $\mathcal{E}$. The expression relation is defined in terms of the program logic that the logical relation is built upon. Specifically, $\mathcal{E}(W)(w)$ expresses that given an instrumented machine state beginning at world W, and a safe register state $\mathcal{R}(r)$, the word w is safe to execute. That is, the weakest precondition holds for a configuration in which the pc contains w, with a post condition that enforces the instrumented machine state, i.e., all the established invariants hold at some private future world.

Now that we have our definition of the logical relation in place, we can state the fundamental theorem of logical relations. We refer to the Coq mechanisation for its full proof, and the technical report [Georges et al. 2022b, Appendix A] for a proof sketch.

THEOREM 4.1 (FTLR). *Assume that p = RX, p = RWX or (p = RWLX $\wedge$ g = DIRECTED). Assume also that $\mathcal{V}(W)(p, g, b, e, a)$. Then we have that $\mathcal{E}(W)(p, g, b, e, a)$.*

4.3 Examples

We show how to use the unary model to prove safety of two example programs: Listings 7 and 9 in capability machine code. These two examples illustrate two properties that have not been explored in previous formalizations. Each program uses an assert subroutine that tests the integrity of encapsulated state. Although we will not present it in this paper, we have also proved the safety of the awkward example (Lis. 5), which can be found in the Coq mechanization.

```
(closure creation around f1)
g1: malloc r2
    store r2 2
    move r3 pc
    lea r3 offset
    crtcls [r2] r3
    jmp r0
f1: prepstack rstk
    loadU r0 rstk (-1)
(intentional leak)
    push renv
    load renv renv
(integrity assertion)
    assert renv 2
    rclear RegName\{pc, r0}
    jmp r0
```

Fig. 8. Assembly of Lis. 7

4.3.1 Protection against Dangling Stack Pointers. Fig. 8 depicts a program with an assertion whose success depends on the absence of dangling stack pointers. g1 creates a closure around some code f1 and a dynamically allocated location containing the integer 2. The macro `crtcls [`r_2`]` r_3 allocates a closure where r_3 points to the closure's code (created using the *offset* from g1 to f1), and r_2 points to the newly allocated environment; the resulting closure is an enter capability. f1 applies the calling convention from § 3.4: (1) it prepares the stack by checking that the stack has permission URWLX and lowers its address to point to the bottom of its bound, and (2) it loads the return capability parameter which has been passed on the stack itself. Now the idea is that since the parameter was stored on the stack, it must either be a heap closure, or a stack allocated activation record of an *older* stack frame qua temporal stack safety. Thus, when f1 attempts to leak the private capability of the closure by pushing a copy of the private capability onto the stack, temporal stack safety ought to ensure that the content of the stack frame cannot be read once popped and thus that the leaked capability remains inaccessible from the environment after we return from f1. Finally, f1 clears the registers and returns to the caller by invoking the return capability that was passed on the stack. Note that f1's stack frame is *not* cleared.

Recall that f1's assertion hinges on the fact that any caller to the closure created by g1 is not able to read the popped stack frame (§ 2.2.1). We use the unary model to prove that within any arbitrary context of a certain layout, the assertion flag associated with the assert subroutine stays at 0 at every step of the execution, meaning that the assertion never fails.

Theorem 4.2. *(Correctness of the temporal stack safety example)* *Let* $reg \in \text{Reg}$, $m \in \text{Mem}$ *and*

$$c_{\text{temp}} \triangleq (\mathit{RX}, \mathit{GLOBAL}, \ldots) \quad c_{\text{stk}} \triangleq (\mathit{URWLX}, \mathit{DIRECTED}, \ldots) \quad c_{\text{adv}} \triangleq (\mathit{RWX}, \mathit{GLOBAL}, \ldots)$$

where the capabilities have an appropriate range of authority and pointer. Furthermore, assume that:

- *m has been initialized with the code of the program and subroutines (pointed to by c_{temp}), an uninitialized stack (pointed to by c_{stk}), and unknown adversarial code (pointed to by c_{adv});*
- *$reg(pc) = c_{\text{temp}}$, $reg(r_{\text{stk}}) = c_{\text{stk}}$, $reg(r_0) = c_{\text{adv}}$ and $reg(r) \in \mathbb{Z}$ otherwise;*
- flag *denotes the assertion flag, initialized to* 0*;*

If $(\text{Executable}, (reg, m)) \longrightarrow^* (\mu, (reg', m'))$ *then* $m'(\text{flag}) = 0$.

Proof. We prove this statement in two stages. First, we show that g1 is safe according to the expression relation in any world W; $\mathcal{E}(W)(\text{E}, \text{GLOBAL}, b_{\text{temp}}, e_{\text{temp}}, \texttt{g1})$. Next, we conclude by applying the adequacy of weakest preconditions. □

4.3.2 Local State Integrity and Stack Objects. We now consider what happens with local state encapsulation in the presence of stack objects. One might expect that this property is rather straightforward: a shared stack object is not encapsulated, but capability bounds ensure that the other parts of the stack are hidden from the context, and that their integrity is guaranteed. However, subtle issues creep up if one is not careful about the parameters exposed to the context, in particular in the cases where a stack object is passed from the caller to a callee. Consider Fig. 9, where g2 creates a closure around some code f2, which in turn calls some unknown linked code, to which f2 passes two parameters: a stack object that was passed to f2 from its caller, and a new stack object created by f2. In the callback, the integrity of the unshared parts of f2's stack frame is tested with an assertion. f2 begins by checking that the linked unknown code is indeed a heap closure (using a macro reqglob). f2 then applies the calling convention from § 3.4 by checking the permission of the stack capability, and loading a parameter from the stack; here the parameter of interest is the older stack object passed to f2 by the caller.

```
(closure creation around f2)
g2: move r1 pc
    lea r1 offset
    restrict r1 encodePerm(e)
    jmp r0
f2: (the linked code is a
heap closure) reqglob r_adv
    (calling convention)
    prepstack r_stk
    loadU r1 r_stk (-1)
    (integrity protection)
    reqRA r1
    checkintregion r1
    (hidden part of stack)
    push "secret"

(new stack object)
createstackobj r2 "obj"
(call linked code)
scall r_adv [r0;r_env] [r1;r2]
lea r_stk (-6)
(load hidden part of stack)
loadU r_adv r_stk (-2)
(assert its integrity)
assert r_adv "secret"
(return to caller)
loadU r1 r_stk (-4)
rclear RegName\{pc,r1}
jmp r1
```

Fig. 9. Assembly of Lis. 9

Since this stack object was passed through the stack, its *read* authority must be smaller (i.e. lower) than f2's stack frame. However, directed capabilities do not enforce any restrictions on the *write* authority of that stack object. In fact, this passed stack object could in principle be an uninitialized capability with a write authority that overlaps with f2's stack frame, thus presenting a threat to the integrity of the stack frame if passed to some unknown code.

To mitigate that threat, f2 must dynamically check not only that the stack object itself is fully initialized, but also that it transitively does not provide any write access to f2's stack frame. In this particular example, f2 expects a stack object containing simply integers; checkintregion is a macro for checking this. For other examples, other mitigations could be done to inspect the permission of all reachable capabilities within the stack object.

$$\boxed{\mathcal{E}(W)(v_1, v_2)} \triangleq \forall\ \mathit{reg}_1, \mathit{reg}_2, \mathcal{R}(W)(\mathit{reg}_1, \mathit{reg}_2) * \mathit{sharedResources}(W) * \mathit{stsCollection}(W) * \Rrightarrow \text{Executable}$$
$$* \text{ pc} \mapsto v_1 * \text{pc} \rightarrowtail v_2 * \mathop{\scalebox{1.5}{$*$}}_{(r,w_1)\in \mathit{reg}_1/\text{pc} \wedge (r,w_2)\in \mathit{reg}_2/\text{pc}} r \mapsto w_1 * r \rightarrowtail w_2 \mathrel{-\!\!*}$$
$$\text{wp Executable} \left\{ \begin{array}{l} v, v = \text{Halted} \rightarrow \exists W'\ \mathit{reg}'_1\ \mathit{reg}'_2,\ W' \sqsupseteq^{\mathit{priv}} W * \Rrightarrow \text{Halted} \\ * \ \mathit{sharedResources}(W') * \mathit{stsCollection}(W') \\ * \mathop{\scalebox{1.5}{$*$}}_{(r,w_1)\in \mathit{reg}'_1 \wedge (r,w_2)\in \mathit{reg}'_2} r \mapsto w_1 * r \rightarrowtail w_2 \end{array} \right\}$$

$$\boxed{\mathcal{R}(W)(\mathit{reg}_1, \mathit{reg}_2)} \triangleq \mathop{\scalebox{1.5}{$*$}}_{(r,w_1)\in \mathit{reg}_1/\text{pc} \wedge (r,w_2)\in \mathit{reg}_2/\text{pc}} \mathcal{V}(W)(w_1, w_2)$$

$$\boxed{\mathcal{V}(W)(w_1, w_2)} \quad \left\{ \begin{array}{ll} \mathcal{V}(W)(z_1, z_2) & \triangleq z_1 = z_2 \\ \mathcal{V}(W)((\textsc{o}, g, b, e, a), w_2) & \triangleq (\textsc{o}, g, b, e, a) = w_2 \\ \mathcal{V}(W)((\textsc{e}, g, b, e, a), w_2) & \triangleq (\textsc{e}, g, b, e, a) = w_2 \ * \Box\ \forall W' \sqsupseteq^{g} W, \triangleright\ \mathcal{E}(W')\left(\begin{array}{l} (\textsc{rx}, g, b, e, a), \\ (\textsc{rx}, g, b, e, a) \end{array} \right) \\ \mathcal{V}(W)((p, g, b, e, a), w_2) & \triangleq (p, g, b, e, a) = w_2 \\ & \quad * \mathop{\scalebox{1.5}{$*$}}_{a' \in [b,e[} \begin{cases} \mathcal{S}^u(W)(a', g, p, a) & \text{if } p = \textsc{u-} \\ \mathcal{S}(W)(a', g, p) & \text{otherwise} \end{cases} \\ & \qquad \wedge \begin{cases} \exists P, \mathit{rel}(a', P) * \text{rcond}(P) & \textit{if } p = \textsc{ro}|\textsc{rx} \\ \mathit{rel}(a', \mathcal{V}) & \text{otherwise} \end{cases} \end{array} \right.$$

$$\boxed{\text{rcond}(P)} \triangleq \triangleright \Box\, \forall W, w_1, w_2, P(W)(w_1, w_2) \mathrel{-\!\!*} \mathcal{V}(W)(w_1, w_2)$$
$$* \triangleright \Box\, \forall W_1, W_2, z_1, z_2, P(W_1)(z_1, z_2) \mathrel{-\!\!*} P(W_2)(z_1, z_2)$$

$$\boxed{\text{State relation}} \quad \mathcal{S}(W)(a, g, p) \triangleq \begin{cases} W^{\mathit{std}}(a) \in \{\text{Temporary}, \text{Permanent}\} & \text{if } \neg\text{write-local}(p) \wedge g = \textsc{dir}. \\ W^{\mathit{std}}(a) = \text{Temporary} & \text{if write-local}(p) \wedge g = \textsc{dir}. \\ W^{\mathit{std}}(a) = \text{Permanent} & \text{if } g \neq \textsc{dir}. \end{cases}$$

$$\mathcal{S}^u(W)(a, g, p, \mathit{mid}) \triangleq \begin{cases} \mathcal{S}(W)(a, g, p) & \\ \quad \vee\ \exists w_1, w_2, W^{\mathit{std}}(a) = \text{Uninitialized}(w_1, w_2) & \text{if } a \geq \mathit{mid} \wedge\ g = \textsc{dir}. \\ \mathcal{S}(W)(a, g, p) & \text{otherwise} \end{cases}$$

Fig. 10. A Binary Logical Relation with Support for Temporal Stack Safety

Theorem 4.3. *(Correctness of the stack object example)*
Let reg ∈ Reg, *m* ∈ Mem *and*

$$c_{\text{stkobj}} \triangleq (\mathit{RX}, \mathit{GLOBAL}, \ldots) \quad c_{\text{stk}} \triangleq (\mathit{URWLX}, \mathit{DIRECTED}, \ldots) \quad c_{\text{adv}} \triangleq (\mathit{RWX}, \mathit{GLOBAL}, \ldots)$$

where the capabilities have an appropriate range of authority and pointer. Furthermore, assume that:

- *m has been initialized with the code of the program and subroutines (pointed to by c_{stkobj}), an uninitialized stack (pointed to by c_{stk}), and unknown adversarial code (pointed to by c_{adv});*
- *$reg(pc) = c_{\text{stkobj}}$, $reg(r_{\text{stk}}) = c_{\text{stk}}$, $reg(r_0) = c_{\text{adv}}$ and $reg(r) \in \mathbb{Z}$ otherwise;*
- flag *denotes the assertion flag, initialized to* 0*;*

If $(\text{Executable}, (reg, m)) \longrightarrow^* (\mu, (reg', m'))$ *then* $m'(\text{flag}) = 0$.

Discussion. We emphasize that the dynamic checking of the content of a stack object would always have been necessary, including in calling conventions based on local capabilities [Skorstengaard et al. 2018]. However, in all prior examples, including the awkward example considered by Georges et al. [2021], this subtlety never arose, since they did not consider stack objects at all. For instance, the awkward example only allows *global heap closures* as input. The issue is worse for local or directed stack closures, for which no dynamic check can be done. In these cases, the only safe option is to very carefully control what other parameters are passed alongside the closure.

5 A BINARY MODEL FOR CONFIDENTIALITY

So far, we have shown how to use the unary model to reason about examples that depend on integrity properties. To reason about confidentiality properties we must use a binary model. For the temporal aspect of local state confidentiality, we expect that a popped frame should not be able to influence the caller, and thus that two programs whose only difference is to leave different traces on their stack frames ought to be contextually equivalent. We define a binary logical relation and use it to show the contextual equivalence of assembly versions of `f` and `g` from Lis. 8. We follow well-known techniques for defining binary logical relations in Iris [Frumin et al. 2018; Krebbers et al. 2017b; Krogh-Jespersen et al. 2017], but apply them here for the first time to a low-level capability machine language. The logical relation is parameterised by a binary version of the world. The key technical aspect of the definition is to allow for the uninitialized part of the stack to be uninitialized at different words. We will return to this key aspect later. First, let's examine some high-level aspects of the definition of the binary logical relation.

The logical relation, presented in fig. 10, captures program refinement. We depict in blue the parts of the definition that are different from the unary logical relation. The expression relation $\mathcal{E}(v_1, v_2)$ describes that the program pointed to by capability v_1, thought of as the *implementation*, refines the program pointed to by capability v_2, thought of as the *specification*. The trick in defining logical refinements in Iris is to use ghost state (separate from the state interpretation) to track the current state and expression of the *specification*.

In our low level capability machine, this means we use ghost state to track the state of specification registers, denoted $r \rightarrowtail w$, the state of specification memory, denoted $a \rightarrowtail w$, and the current execution mode of the specification program, denoted $\Mapsto \mu$. As usual, these ghost state assertions depict fragmental views of the ghost state. We store the full authoritative view in an Iris invariant, henceforth denoted *specCtx*.

The expression relation can roughly be interpreted as follows: given an implementation register state that refines a specification register state, where both the implementation and specification are in Executable mode, if the implementation halts, then the specification must also halt, and all established invariants of the (binary) instrumented machine state hold at some private future world.

Capability machine programs are able to observe and compare words. As a first step towards a value relation, we thus observe that a word w_1 can only refine w_2 if they are equal. However, syntactic equivalence is not enough. Capabilities that grant read authority must themselves point to refined memory fragments. Just as in the unary case, we use the (binary) instrumented machine state to relate the memory fragments within the authority of a valid capability. The binary state relation enforces appropriate standard states on the world. These standard states are as in fig. 6, except the Uninitialized state now records two words, one for the specification and one for the implementation. Crucially, since a directed capability only grants write authority to its uninitialized part, the content of the latter cannot affect program execution. The state interpretation thus allows the content of implementation side uninitialized memory to differ from its specification counterpart, as reflected by the standard state Uninitialized(w_1, w_2). This is the key point that allows us to verify the contextual equivalence of programs that depend on the temporal aspect of confidentiality.

Theorem 5.1 (Binary FTLR). *Assume that* $p = \textsc{rx}$, $p = \textsc{rwx}$ *or* $(p = \textsc{rwlx} \wedge g = \textsc{directed})$. *Assume also that* $\mathcal{V}(W)((p, g, b, e, a), (p, g, b, e, a))$ *and the invariant specCtx. Then we have that* $\mathcal{E}(W)((p, g, b, e, a), (p, g, b, e, a))$.

We use the logical relation to show contextual equivalence of components. Informally, two components are contextually equivalent if no context can distinguish them through termination. A component is either a library or a main component. A component C can be considered a context for component *comp* when their linking $C[comp]$ generates a closed program, that is, there are no imports left to satisfy. We only consider components that are well-formed. A component is well-formed if its memory segment does not overlap with the stack memory, and all capabilities that it contains only address its own memory segment, and are not permit-write-local. The initial state of a closed program is a state where the register file contains only 0, except for the program counter, which should be initialized to the entry point of the program, and r_{stk}, which should be initialized with the stack capability. The memory is empty except for the part specified by the memory segment of the program, and the stack, which is initialized with arbitrary words. These definitions are formally stated in [Georges et al. 2022b, Appendix B]. We use them to formally define contextual equivalence.

$$comp_1 \approx_{ctx} comp_2 \quad \triangleq \quad \forall C, C[comp_1] \Downarrow \Longleftrightarrow C[comp_2] \Downarrow$$

We use $c \Downarrow$ to denote that the configuration c terminates in a Halted state.

Let's use the binary model to prove the equivalence of the two programs from lis. 8. Fig. 11 depicts their low level implementation. f3 pushes 2 onto the stack, clears its registers and jumps to a return capability loaded from the stack. h3 behaves similarly, except it pushes 3 onto the stack. The two programs leave different traces on their respective stack frames, but if temporal confidentiality is enforced, no caller can distinguish them. We use the binary model to show that the two programs refine each other according to our logical refinement definition.

We then apply the adequacy of weakest preconditions to prove the following contextual equivalence:

```
f3: prepstack r_stk
    loadU r_0 r_stk (-1)
    push 2
    rclear RegName\{pc, r_0}
    jmp r_0

h3: prepstack r_stk
    loadU r_0 r_stk (-1)
    push 3
    rclear RegName\{pc, r_0}
    jmp r_0
```

Fig. 11. Assembly of Lis. 8

Theorem 5.2. *(Correctness of the temporal confidentiality example) Let $comp_{f3}$ and $comp_{h3}$ be two components containing the programs* f3 *and* h3 *respectively, where*

$$comp_{f3}.exports \triangleq \{\texttt{0} : (\textsc{e}, \textsc{global}, ..., \texttt{f3})\} \quad comp_{h3}.exports \triangleq \{\texttt{0} : (\textsc{e}, \textsc{global}, ..., \texttt{h3})\}$$

in which the respective exported entry points have an appropriate range of authority and pointer.

Furthermore, assume that contexts are defined as well-formed components with no exports, a single import 0*, and a memory segment with instructions (integers) only. Then*

$$comp_{f3} \approx_{ctx} comp_{h3}$$

Proof. The heart of the proof lies in showing $\mathcal{V}(W)((\textsc{e}, \textsc{global}, \cdots, \texttt{f3}), (\textsc{e}, \textsc{global}, \cdots, \texttt{h3}))$ (and in the other direction) for the appropriate bounds and world W. The difficulty lies in handling the stack address with potentially distinguishing content. Let a be the stack address that contains different integers in the two runs. Upon invoking the return capability, the instrumented machine state must reflect that distinction. However, since we are jumping to an older frame, we can uninitialize the popped part of the stack, including address a. We thus end up with a world W' in which $W'^{std}(a) = \text{Uninitialized}(2, 3)$. We finish the proof by applying the fundamental theorem on the return capability, which by monotonicity is in the binary logical relation at world W'. □

6 CHARACTERIZING SECURITY USING A FULLY ABSTRACT OVERLAY SEMANTICS

The unary and binary model can be used to prove the integrity and confidentiality of example programs that may depend on any of the five properties presented in § 2.1. While the unary and binary model capture integrity and confidentiality properties, and proving examples increase our confidence in the calling convention, they do not *define* any notion of stack safety. Rather than detailing examples that vaguely cover all properties, we wish to truly capture this notion of stack safety, and prove that our new calling convention enforces it.

To that end, we follow the same approach as Skorstengaard et al. [2019b] for proving that our calling convention does enforce the security properties we claim. We first start by describing an overlay semantics (§ 6.1) whose aim is to clearly capture the properties included in our notion of stack safety. Then, we prove a full abstraction theorem between the overlay semantics and the original base capability machine semantics (§ 6.2).

6.1 Overlay Semantics

The overlay semantics augments the base semantics of the capability machine with additional structure to model the properties we wish to enforce, these components are indicated in blue.

$$\begin{aligned}
c \in \mathrm{Cap} &\triangleq \{(p, \ell, b, e, a) \mid b, e, a \in \mathrm{Addr}\} \\
&\quad \cup \{\mathrm{Stk}(d, p, b, e, a) \mid d \in \mathbb{N}, p \in \mathrm{Perm}, b, e, a \in \mathrm{Addr}\} \\
&\quad \cup \{\mathrm{Ret}(b, e, a) \mid b, e, a \in \mathrm{Addr}\} \\
instr &::= \ \ldots \mid \mathrm{call}\ r\ \vec{r} \\
sf \in \mathrm{Stackframe} &\triangleq (\mathrm{Reg} \times \mathrm{Mem}) \\
\varphi \in \mathrm{ExecConf} &\triangleq \mathrm{Reg} \times \mathrm{Mem} \times \mathrm{Mem} \times \text{list Stackframe}
\end{aligned}$$

Syntax. Configurations in the overlay semantics now track a list of overlay stack frames, and natively separate the heap and stack memory. Configurations are now quadruples (reg, h, stk, cs) where reg and stk are the current register state and current stack frame. h is the state of the heap, while cs corresponds to the call stack, a list of saved register states and stack frames.

The overlay semantics has two additional kinds of capabilities; stack derived capabilities $\mathrm{Stk}(d, p, b, e, a)$, and return capabilities $\mathrm{Ret}(b, e, a)$. A capability $\mathrm{Stk}(d, p, b, e, a)$ provides access to the d^{th} stack frame with permission p over range $[b, e[$, and currently points to address a. That is, if the current state is (reg, h, stk, cs), then $d = 0$ provides access to the *oldest* stack frame (i.e., at the tail of cs), while $d = |cs|$ provides access to the current stack frame stk. For instance, if $|cs| = 1$, then it means that the current executing function is at depth 1, its caller is necessarily the main entrypoint function. Return capabilities $\mathrm{Ret}(b, e, a)$ make the overlay semantics return from a call by deallocating the topmost frame, the addresses b, e, a do not matter except for the full abstraction proof which will be explained § 6.2. The regular capabilities (p, ℓ, b, e, a) are now specifically for accessing only the heap in the overlay semantics.

Call. The overlay semantics provide a new instruction `call` $r\ \vec{r}_{args}$ which calls the function given in the register r, and passing the arguments in $\vec{r}_{args}$. The operational semantics is given an additional rule for executing calls as follows.

$$\textsc{ExecCall}\quad \dfrac{\begin{array}{c}
{}^{\dagger}\varphi.reg(\mathrm{pc}) = (p, \ell, b, e, a) \qquad {}^{\dagger}p \in \{\textsc{rx}, \textsc{rwx}, \textsc{rwlx}\} \qquad {}^{\dagger}[a, a + |\mathrm{call}\ r\ \vec{r}_{args}|[\ \subseteq [b, e[\\
{}^{\dagger}\varphi.h([a, a + |\mathrm{call}\ r\ \vec{r}_{args}|[) = [\mathrm{call}_0\ r\ \vec{r}_{args}; \mathrm{call}_1\ r\ \vec{r}_{args}; \ldots] \qquad {}^{\P}\forall r, r \in \vec{r}_{args}, \mathrm{safe}(\varphi.reg(r)) \\
{}^{\S}\varphi.reg(r) = (\textsc{e}, \ell', b', e', a') \qquad {}^{\bigstar}\varphi.reg(r_{\mathrm{stk}}) = \mathrm{Stk}(d, \textsc{urwlx}, b_{stk}, e_{stk}, a_{stk}) \\
{}^{\bigstar}d = |\varphi.cs| \qquad {}^{\bigstar}[a, a_{stk} + |act_{code}| + 1 + |\vec{r}_{args}|[\ \subseteq [b_{stk}, e_{stk}[\\
{}^{\bullet}\forall i, \mathrm{canBeStored}(\varphi.reg(r_i), a_{stk} + i) \qquad \varphi' = (reg', \varphi.h, stk', (reg^*, stk^*) :: \varphi.cs)
\end{array}}{(\mathrm{Executable}, \varphi) \longrightarrow (\mathrm{Executable}, \varphi')}$$

where

- reg' is defined such that $reg'(\mathrm{pc}) = (\textsc{rx}, \ell', b', e', a')$, $reg'(r_{\mathrm{stk}}) = \mathrm{Stk}(d + 1, \textsc{urwlx}, a_{stk} + |act_{code}|, e_{stk}, a_{stk} + |act_{code}| + 1 + |\vec{r}_{args}|)$ and $reg'(r') = \text{if } r' = r \text{ then } reg(r) \text{ else } 0$;
- $stk' = \emptyset[a_{stk} + |act_{code}| \mapsto \mathrm{Ret}(\ldots), \ldots]$;
- $reg^* = reg[\mathrm{pc} \mapsto (p, \ell, b, e, a + |\mathrm{call}\ r\ \vec{r}_{args}|)]$;
- $stk^* = stk[a_{stk} \mapsto \ldots, \ldots, a_{stk} + |act_{code}| - 1 \mapsto \ldots]$.

First, the overlay machine dynamically checks that the current pc contains a valid program counter, pointing to the code pattern implementing call $r\ \vec{r}_{args}$ (indicated with a † in ExecCall). It then checks (¶) the parameters of the call are *safe* (i.e. no return capabilities, nor a capability that can be used to overwrite the activation code of an active return capability, see [Georges et al.

2022b, Appendix C] for a formal definition) and (§) the call target is an enter capability. (★) It then checks that the provided capability in r_{stk} is actually a capability for the current stack, with enough range to store the activation code and parameters. The local state is stored on the stack, followed by the activation record. (●) The overlay semantics requires that all of these can be legally stored on the stack. The callee is given a fresh register state (all cleared except pc, r_{stk} and r), and receives a fresh stackframe with the return capability at the bottom, and all parameters above it. Finally, the local state (reg^*, stk^*) of the caller is pushed on the call stack. If a condition is not satisfied, the overlay semantics falls back on the rule ExecStep, and simply executes one instruction at a time. The formal rule for call, denoted ExecCall has been formalized in Coq.

Return. A jump using a return capability is interpreted as a return.

$$[\![\mathrm{jmp}\ r]\!](reg, h, stk, cs) = (reg', h, stk', cs') \text{ when } \begin{cases} reg(r) = \mathrm{Ret}(b, e, a) \\ cs = (reg', stk') :: cs' \end{cases}$$

The semantics of return capture temporal stack safety. The topmost stack frame *stk* is deallocated and the local environment *reg′* and *stk′* is restored. By deallocating the topmost stack frame, we capture that the caller loses any read access to its old content. This is in contrast with [Skorstengaard et al. 2019b], where the stack frames *stk* and *stk′* are instead merged together, giving the caller access to whatever was left on the stack.

Properties of the overlay semantics. ExecCall and jumping using a return capability are the only way in the semantics to push or pop the call stack, it is thus obvious that WBCF is enforced by the overlay semantics. As the topmost stack frame is entirely removed when returning, temporal stack safety is also natively enforced by the semantics. Finally, a stack frame can only be accessed using a corresponding Stk capability with the right depth. ExecCall is the only rule that creates a Stk capability with an increasing depth and only provides it to the callee, a caller cannot thus access to a callee's stack frame. Conversely, a callee is only given access to its Stk capability and parameters, it thus cannot access its caller's local state, local state is also natively enforced by the overlay semantics.

6.2 A Full Abstraction Theorem

In order to show that our new calling convention does enforce stack safety, we prove a full abstraction theorem between the overlay semantics and the base capability machine. Full abstraction states that two components are indistinguishable by other components in the overlay semantics if and only if they are indistinguishable by other components in the base semantics. Informally, this shows that adversarial contexts in the base capability machine are not stronger that those in the overlay semantics. The theorem is stated as follows. We use *blue* for the overlay machine, and **red** for the regular capability machine.

Theorem 6.1. *For well-formed components $comp_1$ and $comp_2$, we have*

$$comp_1 \approx_{ctx} comp_2 \Leftrightarrow \mathbf{comp_1} \approx_{ctx} \mathbf{comp_2}$$

The theorem states that contextual equivalences are preserved and reflected, it is proved using a simple simulation argument. A forward simulation is defined as follows.

Definition 6.2 (Forward simulation). We say that $\sim$ is a forward simulation between programs p_1 and p_2 when the following holds.

(1) Let ϕ and ψ be the initial states of p_1 and p_2, then $\phi \sim \psi$.
(2) Let ϕ and ψ such that $\phi \sim \psi$, then if $\phi \rightarrow \phi'$ then there exists ψ' such that $\psi \rightarrow^* \psi'$ and $\phi' \sim \psi'$.

(3) Let ϕ and ψ such that $\phi \sim \psi$, then if ϕ is a final state of p_1, then ψ is a final state of p_2.

A forward simulation implies preservation of termination [Leroy 2009]:

Lemma 6.3. *If there exists a forward simulation between programs p_1 and p_2, then $p_1\Downarrow \Rightarrow p_2\Downarrow$.*

Before proving Theorem 6.1, we first show the following lemma.

Lemma 6.4. *For all well-formed closed programs (see [Georges et al. 2022b] for a precise definition) $\mathcal{P}$ in the overlay semantics, there exists a forward simulation between $\mathcal{P}$ and its counterpart $\mathbf{P}$ in the base semantics.*

Proof. The detailed proof can be found in the accompanying Coq development, we only provide a proof sketch here. The crux of the proof is to build a relation $\sim$ and show it is a forward simulation.

We say that $(\mathit{reg}, h, \mathit{stk}, \mathit{cs}) \sim (\mathbf{reg}, \mathbf{mem})$ when reg and $\mathbf{reg}$ contain related words in each register. An integer is only related to itself, while a stack capability $\mathsf{Stk}(d, p, b, e, a)$ is related to a directed capability $(p, \text{DIRECTED}, b, e, a)$. Similarly, return capabilities $\mathsf{Ret}(b, e, a)$ are related to enter directed capabilities $(\text{E}, \text{DIRECTED}, b, e, a)$ and regular capabilities (p, ℓ, b, e, a) are related to themselves. Furthermore, the heap h, the current stack stk and saved stack frames in the call stack cs must have disjoint domains, as well as have related words at corresponding addresses with $\mathbf{mem}$.

When this relation is defined, it is relatively straightforward, though tedious, to show that it is indeed a forward simulation. The simulation operates in a "lockstep" fashion, except for the ExecCall and ExecReturn steps where one step in the overlay semantics is simulated by multiple ones in the base semantics. □

We can now prove our full-abstraction theorem (Theorem 6.1).

Proof. By unfolding the definitions, we need to prove the following:

$$(\forall C, C[\mathit{comp}_1]\Downarrow \Leftrightarrow C[\mathit{comp}_2]\Downarrow) \Leftrightarrow (\forall \mathbf{C}, \mathbf{C}[\mathbf{comp}_1]\Downarrow \Leftrightarrow \mathbf{C}[\mathbf{comp}_2]\Downarrow)$$

By combining Lemma 6.3 and Lemma 6.4, we know that for all closed programs p, $p\Downarrow \implies \mathbf{p}\Downarrow$. Furthermore, as the base capability machine semantics is deterministic, we can actually build a backward simulation from the forward simulation [Leroy 2009]. We thus actually have that for all closed programs p, $p\Downarrow \Leftrightarrow \mathbf{p}\Downarrow$.

Without loss of generality, we can thus consider the $\Rightarrow$ implication (the other direction is similar). By symmetry it suffices once again to consider only the $\Rightarrow$ direction of $(\forall \mathbf{C}, \mathbf{C}[\mathbf{comp}_1]\Downarrow \Leftrightarrow \mathbf{C}[\mathbf{comp}_2]\Downarrow)$.

We prove this by following the proof structure shown on the left. Let $\mathbf{C}$ be a context such that $\mathbf{C}[\mathbf{comp}_1]\Downarrow$, we need to prove that $\mathbf{C}[\mathbf{comp}_2]\Downarrow$, knowing that $C[\mathit{comp}_1]\Downarrow \Leftrightarrow C[\mathit{comp}_2]\Downarrow$.

$$\begin{tikzcd}
C[\mathit{comp}_1]\Downarrow \arrow[r, Rightarrow, "(2)"'] & C[\mathit{comp}_2]\Downarrow \arrow[d, Rightarrow, "(3)"] \\
\mathbf{C}[\mathbf{comp}_1]\Downarrow \arrow[u, Rightarrow, "(1)"] \arrow[r, Rightarrow, "?"'] & \mathbf{C}[\mathbf{comp}_2]\Downarrow
\end{tikzcd}$$

The steps described in the figure can then be proved as follows.

(1) We use the backward simulation to prove that $C[\mathit{comp}_1]\Downarrow$.
(2) By assumption, we have that $C[\mathit{comp}_1]\Downarrow \Leftrightarrow C[\mathit{comp}_2]\Downarrow$, and therefore $C[\mathit{comp}_2]\Downarrow$.
(3) We use the forward simulation to conclude that $\mathbf{C}[\mathbf{comp}_2]\Downarrow$.

□

7 RELATED WORK

We now discuss some related work that has not already been discussed in the paper.

In this paper we have emphasized temporal stack safety, but capabilities have also recently been proposed in the CHERI project as a mechanism for ensuring temporal memory safety for the heap. In particular, in CHERIvoke [Xia et al. 2019] and Cornucopia [Filardo et al. 2020] the authors suggest to use capabilities to efficiently and securely reclaim memory managed by a dedicated memory allocator using a garbage-collector-like approach. In contrast to our work, they do not formally state nor prove the guarantees provided by their mechanism, and it would be interesting to do that in future work.

We have already discussed the most closely related work on formalising capability safety. Other related approaches include the work of Nienhuis et al. [2020] and Bauereiss et al. [2022], who define a syntactic notion of capability safety as a monotonicity guarantee of reachable objects (the machine does not create new capabilities out of thin air); in contrast to our approach, they do not consider safety across calls to possibly adversarial code, so they only show that security properties hold within a single component. On the other hand, they consider a capability machine model with all of the instructions found on a real machine (Morello in the case of [Bauereiss et al. 2022]) whereas we consider a core capability machine model. Devriese et al. [2016] propose a semantic approach to capturing capability safety for a high-level language with object capabilities using logical relations; this approach was further expanded upon by Swasey et al. [2017], who showed the robustness of several object capability patterns. Recently, El-Korashy et al. [2021] have studied the formal security guarantees of PAC (pointers-as-capabilities) compilers for partial programs and characterized them via a full abstraction result.

Full abstraction [Abadi 1999] is a well-known property in the field of secure compilation [Patrignani et al. 2019] and has been used in recent works to characterize the security properties provided by different capability machines. Our approach follows the one proposed by Skorstengaard et al. [2019b] who use a fully abstract overlay semantics to define and show that their protection mechanisms enforce WBCF and LSE. Their proof uses a complex cross-language logical relation and is not mechanized, whereas we use a simpler simulation argument. Van Strydonck et al. [2021] also use a simulation-based argument for a fully abstract compiler from a statically verified language to an unverified language with support for linear capabilities. Likewise, Tsampas et al. [2019] also use a simulation proof for proving full abstraction between an “ideal” semantics with native temporal safety and an imperative language equipped with capabilities. It is interesting to note that their higher-level semantics already assumes well-bracketed control-flow with automatic deallocation of stackframes on return; we provably enforce that using directed capabilities.

While we use an overlay semantics to characterize the notion of stack safety our calling convention guarantees, Anderson et al. [2021] define stack safety as a trace property, expressed as the conjunction of LSE and WBCF. Anderson et al. distinguish between the integrity of local state, and the confidentiality of local state. Likewise, we develop a unary model to reason about the integrity of specific examples, and a binary model to reason about confidentiality. Unlike Anderson et al., we consider a machine with both a stack and a heap. Anderson et al. use their definition to validate the stack safety micro-policies proposed by Roessler and DeHon [2018], who use a general-purpose tagged architecture to design stack protection security policies. Unlike our capability-based calling convention, their policies do not incur an overhead when passing stack objects, but require more sophisticated tags, and a mechanism of lazy tagging to achieve the low overhead.

Tagged architectures have also been used to enforce more general properties such as information-flow control [Azevedo de Amorim et al. 2014], secure compartmentalization [Abate et al. 2018], among other micro-policies [Azevedo de Amorim et al. 2015]. However, micro-policies must be expertly designed in order to leverage cache and be efficient.

In a different line of work, software-based fault isolation (SFI) aims to provide process-based isolation by compartmentalizing (sandboxing) processes in different regions of the memory [Wahbe et al. 1993]. Our calling convention provides similar guarantees *despite* a shared stack. Recently, Kolosick et al. [2022] have used an overlay semantics to characterize sufficient conditions for which so-called “heavyweight transitions” (context switching) can be safely replaced by “near zero cost transitions" akin to regular function calls. They further show that WebAssembly code compiled by a *correct* compiler would satisfy these conditions.

Finally, we remark that our unary and binary models are built on a large body of work on characterizing security through logical relations. We use a logical approach [Dreyer et al. 2011, 2010b; Turon et al. 2013] to step-indexed Kripke logical relations [Ahmed 2004; Birkedal et al. 2011], mechanized in the Coq implementation of Iris [Krebbers et al. 2017b]. We use private and public transitions to characterise well-bracketed control flow [Dreyer et al. 2010a], and a new kind of transition that we call temporal transitions to characterise the temporal aspect of stack safety.

8 CONCLUSION AND FUTURE WORK

We have demonstrated how directed capabilities can be used to enforce a strong degree of stack safety, including local state encapsulation, well-bracketed control flow, and temporal stack safety, with no stack clearing, and with only one additional bit. We have presented two new logical relations to reason about the integrity and confidentiality of specific examples, and proved a full abstraction result for an overlay semantics that defines our notion of stack safety. Finally, we discussed interesting subtleties of stack safety properties in a capability machine with a stack and a heap, and in the presence of stack objects crossing the boundary from caller to callee.

We have used contextual equivalence to formalize confidentiality, whereas Anderson et al. [2021] and Azevedo de Amorim et al. [2018] intuitively link confidentiality to a kind of non-interference property. We believe our calling convention based on directed capabilities also guarantees non-interference and in future work, it would be interesting to show this formally. To that end, one would probably have to extend the capability machine language with security labels.

We have shown how our capability machine can implement function calls, as found in higher-level languages, in a secure manner. We believe it is easy to show that it can also implement tail-calls securely and conjecture that it is also possible to implement non-standard control flow such as C-style `setjmp/longjmp` efficiently and securely. Indeed, we may implement `setjmp` by creating a Jmp capability pointing to some activation code, similarly to how call creates a return capability. These capabilities can then be safely passed up the stack to callees. `longjmp` would then be implemented by jumping to such a capability. Such an implemention is efficient as `longjmp` is just a jump, and it is not necessary to unwind the stack. It is also safe in the sense that we can guarantee temporal safety of a `setjmp` environment: a caller will not be able to `longjmp` to a `setjmp` environment set up by one of its descendants (this is similar to how we ensure WBCF and guarantee that a return capability cannot be smuggled away). Temporal confidentiality of stack frames can still be enforced, and stack frames do *not* need to be scrubbed because directed capabilities guarantee that a caller cannot read them. Previously proposed calling conventions for capability machines either cannot provide such guarantees or would require careful unwinding or extensive memory clearing.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for excellent comments and suggestions. This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation. We would also like to thank Dominique Devriese, Thomas Van Strydonck, Amin Timany, Armaël Guéneau and Frank Piessens for invaluable discussions and feedback.

REFERENCES

Martín Abadi. 1999. Protection in Programming-Language Translations. In *Secure Internet Programming, Security Issues for Mobile and Distributed Objects*. 19–34. https://doi.org/10.1007/3-540-48749-2_2

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. 2018. When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018.* 1351–1368. https://doi.org/10.1145/3243734.3243745

Amal Ahmed. 2004. *Semantics of Types for Mutable State.* Ph.D. Dissertation. Princeton University.

Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-dependent representation independence. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 340–353. https://doi.org/10.1145/1480881.1480925

Sean Noble Anderson, Leonidas Lampropoulos, Roberto Blanco, Benjamin C. Pierce, and Andrew Tolmach. 2021. Security Properties for Stack Safety. *CoRR* abs/2105.00417 (2021). arXiv:2105.00417 https://arxiv.org/abs/2105.00417

Arm. 2021. Morello project. Retrieved July 6, 2021 from https://www.morello-project.org/

Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Catalin Hritcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach. 2014. A verified information-flow architecture. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014.* 165–178. https://doi.org/10.1145/2535838.2535839

Arthur Azevedo de Amorim, Maxime Dénès, Nick Giannarakis, Catalin Hritcu, Benjamin C. Pierce, Antal Spector-Zabusky, and Andrew Tolmach. 2015. 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.* 813–830. https://doi.org/10.1109/SP.2015.55

Arthur Azevedo de Amorim, Catalin Hritcu, and Benjamin C. Pierce. 2018. The Meaning of Memory Safety. In *Principles of Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science)*, Lujo Bauer and Ralf Küsters (Eds.), Vol. 10804. Springer, 79–105. https://doi.org/10.1007/978-3-319-89722-6_4

Thomas Bauereiss, Brian Campbell, Thomas Sewell, Alasdair Armstrong, Lawrence Esswood, Ian Stark, Graeme Barnes, Robert N. M. Watson, and Peter Sewell. 2022. Verified Security for the Morello Capability-enhanced Prototype Arm Architecture. In *Proceedings of the 31st European Symposium on Programming.*

Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. 2011. Step-indexed Kripke models over recursive worlds. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 119–132. https://doi.org/10.1145/1926385.1926401

Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. 1994. Hardware Support for Fast Capability-Based Addressing. In *International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM, 319–327. https://doi.org/10.1145/195473.195579

Chromium. 2020. Memory safety. Retrieved July 6, 2021 from https://www.chromium.org/Home/chromium-security/memory-safety

Jack B. Dennis and Earl C. Van Horn. 1966. Programming Semantics for Multiprogrammed Computations. *Commun. ACM* 9, 3 (March 1966), 143–155. https://doi.org/10.1145/365230.365252

Dominique Devriese, Lars Birkedal, and Frank Piessens. 2016. Reasoning about Object Capabilities with Logical Relations and Effect Parametricity. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016.* IEEE, 147–162. https://doi.org/10.1109/EuroSP.2016.22

Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2011. Logical Step-Indexed Logical Relations. *LMCS* 7, 2:16 (June 2011), 1–37.

Derek Dreyer, Georg Neis, and Lars Birkedal. 2010a. The impact of higher-order state and control effects on local relational reasoning. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM, 143–156. https://doi.org/10.1145/1863543.1863566

Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. 2010b. A relational modal logic for higher-order stateful ADTs. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010.* 185–198. https://doi.org/10.1145/1706299.1706323

Akram El-Korashy, Stelios Tsampas, Marco Patrignani, Dominique Devriese, Deepak Garg, and Frank Piessens. 2021. CapablePtrs: Securely Compiling Partial Programs Using the Pointers-as-Capabilities Principle. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF).* IEEE Computer Society, Los Alamitos, CA, USA, 421–436. https://doi.org/10.1109/CSF51468.2021.00036

Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. 2020. Cornucopia: Temporal

Safety for CHERI Heaps. In *IEEE Symposium on Security and Privacy*. IEEE.

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 442–451. https://doi.org/10.1145/3209108.3209174

Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. 2021. Efficient and provable local capability revocation using uninitialized capabilities. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30. https://doi.org/10.1145/3434287

Aïna Linn Georges, Alix Trieu, and Lars Birkedal. 2022a. *Artifact for Le Temps des Cerises: Efficient Temporal Stack Safety on Capability Machines using Directed Capabilities.* https://doi.org/10.5281/zenodo.5821862

Aïna Linn Georges, Alix Trieu, and Lars Birkedal. 2022b. *Le Temps des Cerises: Efficient Temporal Stack Safety on Capability Machines using Directed Capabilities.* Technical Report. https://cs.au.dk/~ageorges/publications_pdfs/monotone-technical.pdf

Nicolas Joly, Saif ElSherei, and Saar Amar. 2020. Security Analysis of CHERI ISA. Retrieved July 6, 2021 from https://msrc-blog.microsoft.com/2020/10/14/security-analysis-of-cheri-isa/

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. 256–269. https://doi.org/10.1145/2951913.2951943

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 637–650. https://doi.org/10.1145/2676726.2676980

Matthew Kolosick, Shravan Narayan, Conrad Watt, Michael LeMay, Deepak Garg, Ranjit Jhala, and Deian Stefan. 2022. Isolation Without Taxation: Near Zero Cost Transitions for SFI. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM.

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *PACMPL* 2, ICFP (2018), 77:1–77:30. https://doi.org/10.1145/3236772

Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017a. The Essence of Higher-Order Concurrent Separation Logic. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. 696–723. https://doi.org/10.1007/978-3-662-54434-1_26

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017b. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 205–217. https://doi.org/10.1145/3009837.3009855

Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. 2017. A relational model of types-and-effects in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 218–231. https://doi.org/10.1145/3009837.3009877

Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *J. Autom. Reason.* 43, 4 (2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

Henry M. Levy. 1984. *Capability-Based Computer Systems.* Digital Press. https://homes.cs.washington.edu/~levy/capabook/

Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson, and Peter Sewell. 2020. Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (SP).*

Marco Patrignani, Amal Ahmed, and Dave Clarke. 2019. Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work. *ACM Comput. Surv.* 51, 6, Article 125 (Feb. 2019), 36 pages. https://doi.org/10.1145/3280984

Nick Roessler and André DeHon. 2018. Protecting the Stack with Metadata Policies and Tagged Hardware. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 478–495. https://doi.org/10.1109/SP.2018.00066

Lau Skorstengaard. 2019. *Formal Reasoning about Capability Machines.* Ph.D. Dissertation. Aarhus University.

Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2018. Reasoning About a Machine with Local Capabilities - Provably Safe Stack and Return Pointer Management. In *Programming Languages and Systems - 27th European Symposium* *on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings.* 475–501. https://doi.org/10.1007/978-3-319-89884-1_17

Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019a. Reasoning about a Machine with Local Capabilities: Provably Safe Stack and Return Pointer Management. *ACM Transactions on Programming Languages and Systems* 42, 1 (Dec. 2019), 5:1–5:53. https://doi.org/10.1145/3363519

Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019b. StkTokens: Enforcing Well-Bracketed Control Flow and Stack Encapsulation Using Linear Capabilities. *Proc. ACM Program. Lang.* 3, POPL, Article 19 (Jan. 2019), 28 pages. https://doi.org/10.1145/3290332

Eijiro Sumii and Benjamin C. Pierce. 2007. A bisimulation for type abstraction and recursion. *J. ACM* 54, 5 (2007), 26. https://doi.org/10.1145/1284320.1284325

David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and Compositional Verification of Object Capability Patterns. In *OOPSLA*. ACM. https://people.mpi-sws.org/~swasey/papers/ocpl/ocpl-20170418.pdf

Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013.* 48–62. https://doi.org/10.1109/SP.2013.13

Gavin Thomas. 2019. A proactive approach to more secure code. Retrieved July 6, 2021 from https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/

Stelios Tsampas, Dominique Devriese, and Frank Piessens. 2019. Temporal Safety for Stack Allocated Memory on Capability Machines. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019.* 243–255. https://doi.org/10.1109/CSF.2019.00024

Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013.* 377–390. https://doi.org/10.1145/2500365.2500600

Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. 2021. Linear capabilities for fully abstract compilation of separation-logic-verified code. *J. Funct. Program.* 31 (2021), e6. https://doi.org/10.1017/S0956796821000022

Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles, SOSP 1993, The Grove Park Inn and Country Club, Asheville, North Carolina, USA, December 5-8, 1993.* 203–216. https://doi.org/10.1145/168619.168635

Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. 2020. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8).* Technical Report UCAM-CL-TR-951. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-951

R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *IEEE Symposium on Security and Privacy.* 20–37. https://doi.org/10.1109/SP.2015.9

Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony C. J. Fox, Robert M. Norton, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel Wesley Filardo, A. Theodore Markettos, Michael Roe, Peter G. Neumann, Robert N. M. Watson, and Simon W. Moore. 2019. CHERI Concentrate: Practical Compressed Capabilities. *IEEE Trans. Computers* 68, 10 (2019), 1455–1469. https://doi.org/10.1109/TC.2019.2914037

Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. 2019. CHERIvoke: Characterising Pointer Revocation Using CHERI Capabilities for Temporal Memory Safety. In *IEEE/ACM International Symposium on Microarchitecture.* ACM. https://doi.org/10.1145/3352460.3358288