

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.

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 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

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

Authors' addresses: Aina 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.

2018. 2475-1421/2018/1-ART1 \$15.00

<https://doi.org/>

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 can be found online at <https://github.com/logsem/cerise-stack-monotone>.

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 conventions 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

```

1 void adv(void);
2 void f(void) {
3   int *y; // allocated on
4   *y = 2; // the stack
5   adv();
6   assert (*y == 2); }

```

Listing 1. Integrity: frame

```

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

```

Listing 2. Integrity: environment

```

1 void adv(void);
2 void f(void) {
3   int *y; *y = 2;
4   adv();
5   return; }
6 void g(void) {
7   int *y; *y = 3;
8   adv();
9   return; }

```

Listing 3. Confidentiality: frame private state x.

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

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.

```

1 void adv(void);
2 void f(void) {
3   static int x = 2;
4   adv(); }
5 void g(void) {
6   static int x = 3;
7   adv(); }

```

Listing 4. Confidentiality: environment

```

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

Listing 5. Awkward Example. 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];
5   h(x); }
6 void f(int** x) {
7   char* t[N];
8   int z;
9   *x = &z; }
10 int main(void) {
11   int* x;
12   f(&x);
13   g(x);
14   return 0; }

```

Listing 6. Example violating temporal stack safety

Listing 6. Example violating temporal stack safety. 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.

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

2.2 Two Subtleties of Stack Safety

```

1 void f(void) {
2   static int x = 2;
3   int *y;
4   *y = &x;
5   assert (x == 2); }

```

Listing (7) Dangling stack

```

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

```

Listing (8) Temporal confidentiality

```

1 int g(char* z, char* in)
2 int f(char* in) {
3   int *y = 2;
4   char* z = ...;
5   g(z, in);
6   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 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)

	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	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓

Table 1. Guarantees granted by the calling convention.

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 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

²While 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.

295 $a \in \text{Addr} \triangleq [0, \text{AddrMax}]$ $r \in \text{RegName} ::= \text{pc} \mid r_0 \mid r_1 \mid \dots$
 296 $p \in \text{Perm} ::= \text{o} \mid \text{E} \mid \text{RO} \mid \text{RX} \mid \text{RW} \mid \text{RWX}$ $\text{reg} \in \text{Reg} \triangleq \text{RegName} \rightarrow \text{Word}$
 297 $\mid \text{RWL} \mid \text{RWLX} \mid \text{URW} \mid \text{URWL} \mid \text{URWX} \mid \text{URWLX}$ $m \in \text{Mem} \triangleq \text{Addr} \rightarrow \text{Word}$
 298 $g \in \text{Locality} ::= \text{GLOBAL} \mid \text{LOCAL} \mid \text{DIRECTED}$ $\varphi \in \text{ExecConf} \triangleq \text{Reg} \times \text{Mem}$
 299 $c \in \text{Cap} \triangleq \{(p, g, b, e, a) \mid b, e, a \in \text{Addr}\}$ $\delta \in \text{ExecMode} ::=$
 300 $w \in \text{Word} \triangleq \mathbb{Z} + \text{Cap}$ $\text{Executable} \mid \text{Halted} \mid \text{Failed}$
 301 $\rho \in \mathbb{Z} + \text{RegName}$
 302 $i ::= \text{jmp } r \mid \text{jnz } r \mid \text{move } r \rho \mid \text{load } r r \mid \text{store } r \rho \mid \text{add } r \rho \rho \mid \text{sub } r \rho \rho \mid$
 303 $\text{lt } r \rho \rho \mid \text{lea } r \rho \mid \text{restrict } r \rho \mid \text{subseg } r \rho \rho \mid \text{isptr } r r \mid \text{getp } r r \mid \text{getl } r r \mid$
 304 $\text{getb } r r \mid \text{gete } r r \mid \text{geta } r r \mid \text{fail} \mid \text{halt} \mid \text{loadU } r r \rho \mid \text{storeU } r \rho \rho \mid \text{promoteU } r$
 305

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

EXECSTEP

$$(\text{Executable}, \varphi) \rightarrow \begin{cases} \llbracket \text{decode}(z) \rrbracket(\varphi) & \text{if } \varphi.\text{reg}(\text{pc}) = (p, g, b, e, a) \wedge b \leq a < e \wedge \\ & p \in \{\text{RX}, \text{RWX}, \text{RWLX}\} \wedge \varphi.\text{mem}(a) = z \\ (\text{Failed}, \varphi) & \text{otherwise} \end{cases}$$

Fig. 3. Operational semantics: reduction steps.

312 convention here. There are several reasons: first, linear capabilities can be cumbersome to use,
 313 as only the top part of a stack frame can be passed as parameters. Second, exceptions cannot be
 314 implemented efficiently. Third, and most importantly, *LINEAR* capabilities are expensive to realize
 315 in practice. Moving a *LINEAR* capability requires an atomic move which is believed by hardware
 316 developers to lead to an undesirable overhead in runtime [Skorstengaard 2019, §3.6.2]. Indeed, this
 317 was also the reason why Georges et al. [2021] considered local and uninitialized capabilities instead.

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

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

331 We define a unary model to reason about integrity properties, and a binary model to reason about
 332 confidentiality properties (including temporal confidentiality). We use these models to reason about
 333 small but challenging examples; we focus on examples that depend on properties not previously
 334 considered on a low level capability machine (integrity in the presence of stack objects, and temporal
 335 confidentiality). Furthermore, we follow the methodology presented by Skorstengaard et al. [2019b]
 336 and define an overlay semantics that clearly enforces each of the properties in Table 1. In § 6.2, we
 337 show how our new calling convention is fully abstract with respect to this overlay semantics.³
 338

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

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 $[0, \text{AddrMax}]$, 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 $U\pi$ 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 .

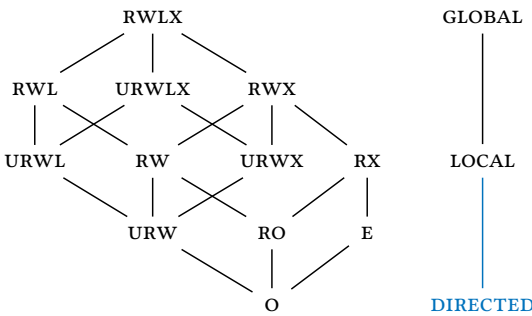


Fig. 4. Permission and locality hierarchy.

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, \dots . 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

393 one rule: if the state is Executable and the program counter contains an in-bounds executable
394 capability pointing to some machine word, then an instruction is decoded and executed; otherwise
395 the machine fails. Recall that, in general, failing is considered safe since it crashes the machine
396 before anything unsafe occurs.

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

420 3.2 A Secure Calling Convention using Local and Uninitialized Capabilities

421 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.

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

427 Bob expects to receive a stack capability from Alice to build his own stack frame. Similarly,
428 when calling Claire, Bob needs to provide the stack capability to her. However, in order to enforce
429 local state encapsulation, it is necessary that Bob does not provide access to his own stackframe to
430 Claire. Thus, when calling Claire, Bob restricts the stack capability to the unused part, using the
431 `subseg` instruction, and then passes it to her. However, Bob needs to be able to restore access to
432 his own stack frame upon return. He can do that using an enter capability: Bob constructs a return
433 capability as an enter capability that restores the local environment when jumped to. This return
434 capability can be safely passed to the callee Claire as its contents cannot be read, but can only be
435 jumped to. This suffices to protect Bob's private state from Claire, but it is not enough to enforce
436 WBCF. Indeed, on its own, this does not prevent the attack explained in § 2.1.2, in which a callee
437 keeps a copy of a previous return capability beyond its "lifetime". To prevent this kind of attack,
438 [Skorstengaard et al.](#) use local capabilities: The stack capability is made write-local, executable
439
440
441

442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490

$$\text{updPC}(\varphi) = \begin{cases} (\text{Executable}, \varphi[\text{reg.pc} \mapsto (p, g, b, e, a + 1)]) & \text{if } \varphi.\text{reg}(\text{pc}) = (p, g, b, e, a) \text{ and } \text{RX} \not\leq p \\ (\text{Failed}, \varphi) & \text{otherwise} \end{cases}$$

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

i	$\llbracket i \rrbracket(\varphi)$	Conditions
fail	(Failed, φ)	
halt	(Halted, φ)	
move $r \rho$	$\text{updPC}(\varphi[\text{reg}.r \mapsto w])$	$w = \text{getWord}(\varphi, \rho)$
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 $p \in \{\text{RO}, \text{RX}, \text{RW}, \text{RWX}, \text{RWL}, \text{RWLX}\}$
store $r \rho$	$\text{updPC}(\varphi[\text{mem}.a \mapsto w])$	$\varphi.\text{reg}(r) = (p, g, b, e, a)$ and $b \leq a < e$ and $p \in \{\text{RW}, \text{RWX}, \text{RWL}, \text{RWLX}\}$ and $w = \text{getWord}(\varphi, \rho)$ and if $w = (_, \text{LOCAL}, _, _, _)$, then $p \in \{\text{RWLX}, \text{RWL}\}$ and if $w = (_, \text{DIRECTED}, _, _, _)$, then $p \in \{\text{RWLX}, \text{RWL}\}$ and $\text{canReadUpTo}(w) \leq a$
jmp r	(Executable, $\varphi[\text{reg.pc} \mapsto \text{newPc}]$)	if $\varphi.\text{reg}(r) = (E, g, b, e, a)$, then $\text{newPc} = (\text{RX}, g, b, e, a)$ otherwise $\text{newPc} = \varphi.\text{reg}(r)$
restrict $r \rho$	$\text{updPC}(\varphi[\text{reg}.r \mapsto w])$	$\varphi.\text{reg}(r) = (p, g, b, e, a)$ and $(p', g') = \text{decodePermPair}(\text{getWord}(\varphi, \rho))$ and $(p', g') \leq (p, g)$ and $w = (p', g', b, e, a)$
subseg $r \rho_1 \rho_2$	$\text{updPC}(\varphi[\text{reg}.r \mapsto w])$	$\varphi.\text{reg}(r) = (p, g, b, e, a)$ 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 $p \neq E$ and $w = (p, g, z_1, z_2, a)$
lea $r \rho$	$\text{updPC}(\varphi[\text{reg}.r \mapsto w])$	$\varphi.\text{reg}(r) = (p, g, b, e, a)$ and $z = \text{getWord}(\varphi, \rho)$ and $p \neq E$ and $w = (p, g, b, e, a + z)$ $p = \cup-$, 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 $r_1 r_2 \rho$	$\text{updPC}(\varphi[\text{reg}.r_1 \mapsto w])$	$\varphi.\text{reg}(r_2) = (p, g, b, e, a)$ and $p = \cup-$ and $\text{off} = \text{getWord}(\varphi, \rho)$ and $b \leq a + \text{off} < a \leq e$ and $w = \varphi.\text{mem}(a + \text{off})$
storeU $r \rho_1 \rho_2$	$\text{updPC}(\varphi'$ $[\text{mem}.(a + \text{off}) \mapsto w])$	$\varphi.\text{reg}(r) = (p, g, b, e, a)$ and $p = \cup-$ and $\text{off} = \text{getWord}(\varphi, \rho_1)$ and $w = \text{getWord}(\varphi, \rho_2)$ and if $w = (_, \ell, _, _, _)$ and $\ell \neq \text{GLOBAL}$ then $p \in \{\text{URWLX}, \text{URWL}\}$ and $b \leq a + \text{off} \leq a < e$ and if $\text{off} \neq 0$ then $\varphi' = \varphi$ else $\varphi' = \varphi[\text{reg}.r \mapsto (p, g, b, e, a + 1)]$ and if $\ell = \text{DIRECTED}$, then $\text{canReadUpTo}(w) \leq a + \text{off}$
promoteU r	$\text{updPC}(\varphi[\text{reg}.r \mapsto w])$	$\varphi.\text{reg}(r) = (p, g, b, e, a)$ and $p = \cup\pi$ and $w = (\pi, g, b, \min(a, e), a)$
...		
_	(Failed, φ)	otherwise

Fig. 5. Operational semantics: instruction semantics.

(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

491 stack frame, using the return capability will only lead to cleared data. As briefly mentioned in § 2,
 492 the original calling convention by Skorstengaard et al. [2018] enforces temporal confidentiality
 493 with an excessive amount of clearing. With the optimization by Skorstengaard et al. [2019a], one
 494 only clears one’s own stack frame on return. Although this improves the efficiency of the calling
 495 convention, every secure closure must clear its own stack frame upon return.

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

507 3.3 Directed Capabilities

508 We now introduce a novel kind of directed capabilities and then explain, in § 3.4, our new improved
 509 calling convention, which relies on directed capabilities to efficiently guarantee LSE, WBCF and
 510 temporal stack safety (as we prove in later sections).
 511

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

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

536 Second, directed capabilities only change the semantics of `load(U)` and `store(U)` by adding an
 537 extra bounds check. The added bounds check is no different from current checks, and we expect
 538 existing optimisation patterns, such as parallelisation, to apply.
 539

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.

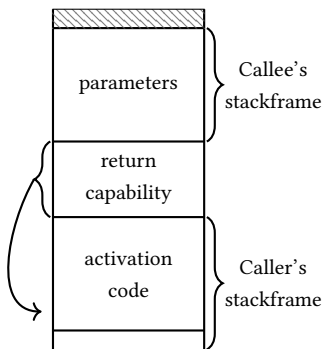
To sum up, our new calling convention is as follows; see also the figure below, which shows what the stack is expected to look like just after a call.

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.

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

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

599 Furthermore, a compiler must also be careful with the order of stack allocations. Consider for
 600 instance the following code snippet:

```
601 int* x; int y; x = &y;
```

602
 603 In the above code, the compiler must reorder the allocations for x and y to guarantee the directed
 604 property. Such considerations must also be taken into account when implementing compiler
 605 optimizations.

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

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

612
 613 are not implementable in an linearity-friendly way, since $p1$ and $p2$ are allowed to be aliases.

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

615 4 A UNARY MODEL FOR INTEGRITY

616
 617 In this section, we develop a novel model that captures all the guarantees provided by DIRECTED
 618 capabilities and our associated calling convention. The core novelty lies in its ability to express
 619 temporal stack safety. The model is made up of two components; a program logic to reason about
 620 known and trusted code, and a Kripke logical relation to reason about arbitrary untrusted code.
 621 We use the unary model to reason about the *integrity* of example programs.

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

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

631 Thanks to the dynamic checks of the capability machine, the behaviour of a program is limited by
 632 the capabilities it has access to. Thus, even completely arbitrary code must adhere to rules imposed
 633 by the capability machine, and will satisfy some notion of *capability safety*. The logical relation
 634 captures this notion of capability safety, and serves as a contract between trusted and untrusted
 635 code. The fundamental theorem of logical relations (see below) means that any word that is safe to
 636 read satisfies that contract. As long as arbitrary code is just a list of instructions (and thus does not,
 637

638 e.g., include an embedded capability), a corollary then states that even completely arbitrary code
639 satisfies that contract.

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

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

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

667 In the remainder of this section we first describe the standard states we use to capture stack and
668 heap states, and how these states may evolve, such that they can be used to capture the desired
669 stack properties (§ 4.1). In § 4.2, we will then present the logical relation itself. Finally, we end this
670 section with two examples highlighting what kind of programs we can now verify using our model
671 (§ 4.3).

672 4.1 A New Kripke World

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

682 We now explain each of the standard states. The Permanent state represents an allocated heap
683 region. As soon as a heap region is allocated and shared, it becomes permanent. There is no
684 mechanism to free the heap region from its state. The Temporary state represents a live stack
685

686

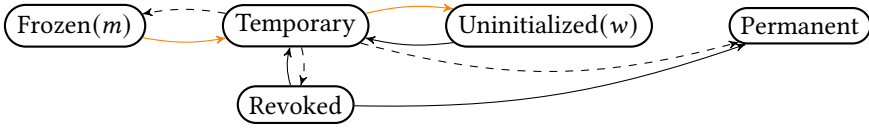


Fig. 6. Standard State Transition System. Full lines indicate public transitions, dashed lines indicate private transitions, orange lines indicate temporal transitions.

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 addresses⁴. If a shared address a is part of the heap, then the associated invariant must be monotone with regards to \sqsupset^{priv} . On the other hand, if a is part of the (live) stack, then the associated invariant must be monotone with regards to \sqsupset^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].

The value relation $\mathcal{V} : \text{WORLD} \rightarrow \text{Word} \rightarrow iProp$ 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 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 0 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 $\phi : \text{WORLD} \rightarrow \text{Word} \rightarrow iProp$ to the address a . It suffices to think of $rel(a, \phi)$ as an invariant, that

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

$$\begin{array}{l}
785 \quad \boxed{\mathcal{E}(W)(v)} \quad \triangleq \forall \text{reg}, \mathcal{R}(W)(\text{reg}) * \text{sharedResources}(W) * \text{stsCollection}(W) * \text{pc} \mapsto v \\
786 \quad \quad \quad * *_{(r,w) \in \text{reg}/\text{pc}} r \mapsto w \text{ —} * \\
787 \quad \quad \quad \text{wp Executable} \left\{ \begin{array}{l} v, v = \text{Halted} \rightarrow \exists W' \text{ reg}', W' \sqsupseteq^{\text{priv}} W \\ * \text{sharedResources}(W') * \text{stsCollection}(W') \\ * *_{(r,w) \in \text{reg}'} r \mapsto w \end{array} \right\} \\
788 \\
789 \\
790 \quad \boxed{\mathcal{R}(W)(\text{reg})} \quad \triangleq *_{(r,w) \in \text{reg}/\text{pc}} \mathcal{V}(W)(w) \\
791 \\
792 \quad \left\{ \begin{array}{l} \mathcal{V}(W)(z) \quad \triangleq \top \\ \mathcal{V}(W)(o, -) \quad \triangleq \top \\ \mathcal{V}(W)(p, g, b, e, a) \quad \triangleq *_{a' \in [b, e]} \left\{ \begin{array}{l} \mathcal{S}^u(W)(a', g, p, a) \quad \text{if } p = \text{U-} \\ \mathcal{S}(W)(a', g, p) \quad \text{otherwise} \end{array} \right. \\ \quad \quad \quad \wedge \left\{ \begin{array}{l} \exists P, \text{rel}(a', P) * \text{rcond}(P) \quad \text{if } p \in \{\text{RO}, \text{RX}\} \\ \text{rel}(a', \mathcal{V}) \quad \text{otherwise} \end{array} \right. \\ \mathcal{V}(W)(\text{E}, g, b, e, a) \quad \triangleq \square \forall W' \sqsupseteq^g W, \triangleright \mathcal{E}(W')(\text{RX}, g, b, e, a) \end{array} \right. \\
793 \\
794 \\
795 \quad \boxed{\mathcal{V}(W)(w)} \\
796 \\
797 \\
798 \\
799 \\
800 \quad \boxed{\text{rcond}(P)} \quad \triangleq \triangleright \square \forall W, w, P(W)(w) \text{ —} * \mathcal{V}(W)(w) \\
801 \quad \quad \quad * \triangleright \square \forall W_1, W_2, z, P(W_1)(z) \text{ —} * P(W_2)(z) \\
802 \\
803 \quad \boxed{\text{State relation}} \\
804 \quad \mathcal{S}(W)(a, g, p) \quad \triangleq \left\{ \begin{array}{l} W^{\text{std}}(a) \in \{\text{Temporary}, \text{Permanent}\} \quad \text{if } \neg \text{write-local}(p) \wedge g = \text{DIRECTED} \\ W^{\text{std}}(a) = \text{Temporary} \quad \text{if } \text{write-local}(p) \wedge g = \text{DIRECTED} \\ W^{\text{std}}(a) = \text{Permanent} \quad \text{if } g \neq \text{DIRECTED} \end{array} \right. \\
805 \\
806 \\
807 \quad \mathcal{S}^u(W)(a, g, p, \text{mid}) \quad \triangleq \left\{ \begin{array}{l} \mathcal{S}(W)(a, g, p) \vee \exists w, W^{\text{std}}(a) = \text{Uninitialized}(w) \quad \text{if } a \geq \text{mid} \\ \quad \quad \quad \wedge g = \text{DIRECTED} \\ \mathcal{S}(W)(a, g, p) \quad \text{if } a < \text{mid} \\ \quad \quad \quad \vee g \neq \text{DIRECTED} \end{array} \right. \\
808 \\
809 \\
810 \\
811
\end{array}$$

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

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 $\text{rcond}(\phi)$, 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 \square . 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 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.

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 [r2] r3` 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.

```

859 (closure creation around f1)
860 g1: malloc r2
861     store r2 2
862     move r3 pc
863     lea r3 offset
864     crtcls [r2] r3
865     jmp r0
866 f1: prepstack rstk
867     loadU r0 rstk (-1)
868 (intentional leak)
869     push renv
870     load renv renv
871 (integrity assertion)
872     assert renv 2
873     rclear RegName\{pc, r0}
874     jmp r0

```

`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.

Fig. 8. Assembly of Lis. 7 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*

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

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});
- $\text{reg}(pc) = c_{\text{temp}}$, $\text{reg}(r_{\text{stk}}) = c_{\text{stk}}$, $\text{reg}(r_0) = c_{\text{adv}}$ and $\text{reg}(r) \in \mathbb{Z}$ otherwise;
- flag denotes the assertion flag, initialized to 0;

If $(\text{Executable}, (\text{reg}, m)) \rightarrow^* (\mu, (\text{reg}', m'))$ 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 ; $\mathcal{E}(W)(E, \text{GLOBAL}, b_{\text{temp}}, e_{\text{temp}}, g1)$. Next, we conclude by applying the adequacy of weakest preconditions. \square

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.

```

887 (closure creation around f2)
888 g2: move r1 pc
889     lea r1 offset
890     restrict r1 encodePerm(e)
891     jmp r0
892 f2: (the linked code is a heap closure)
893     reqglob radv
894     (calling convention)
895     prepstack rstk
896     loadU r1 rstk (-1)
897     (integrity protection)
898     reqRA r1
899     checkintregion r1
900     (hidden part of stack)
901     push "secret"
902     (new stack object)
903     createstackobj r2 "obj"
904     (call linked code)
905     scall radv [r0; renv] [r1; r2]
906     lea rstk (-6)
907     (load hidden part of stack)
908     loadU radv rstk (-2)
909     (assert its integrity)
910     assert radv "secret"
911     (return to caller)
912     loadU r1 rstk (-4)
913     rclear RegName\{pc, r1}
914     jmp r1

```

Fig. 9. Assembly of Lis. 9

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.

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.

THEOREM 4.3. (Correctness of the stack object example)

Let $\text{reg} \in \text{Reg}$, $m \in \text{Mem}$ and

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

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});

- $\text{reg}(pc) = c_{\text{stkobj}}$, $\text{reg}(r_{\text{stk}}) = c_{\text{stk}}$, $\text{reg}(r_0) = c_{\text{adv}}$ and $\text{reg}(r) \in \mathbb{Z}$ otherwise;
- flag denotes the assertion flag, initialized to 0;

If $(\text{Executable}, (\text{reg}, m)) \rightarrow^* (\mu, (\text{reg}', m'))$ 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 [Fruhin 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 $r \mapsto w$, the state of specification memory, denoted $a \mapsto w$, and the current execution mode of the specification program, denoted $\Rightarrow \mu$. 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

$$\begin{array}{l}
981 \quad \boxed{\mathcal{E}(W)(v_1, v_2)} \quad \triangleq \forall \text{reg}_1, \text{reg}_2, \\
982 \quad \mathcal{R}(W)(\text{reg}_1, \text{reg}_2) * \text{sharedResources}(W) * \text{stsCollection}(W) \\
983 \quad * \text{pc} \mapsto v_1 * \text{pc} \mapsto v_2 \\
984 \quad * \Rightarrow \text{Executable} \\
985 \quad * *_{(r, w_1) \in \text{reg}_1/\text{pc} \wedge (r, w_2) \in \text{reg}_2/\text{pc}} r \mapsto w_1 * r \mapsto w_2 \text{ ---} \\
986 \\
987 \quad \text{wp Executable} \left\{ \begin{array}{l} v, v = \text{Halted} \rightarrow \\ \exists W' \text{reg}'_1 \text{reg}'_2, W' \sqsupseteq^{\text{priv}} W \\ * \Rightarrow \text{Halted} \\ * \text{sharedResources}(W') * \text{stsCollection}(W') \\ * *_{(r, w_1) \in \text{reg}'_1 \wedge (r, w_2) \in \text{reg}'_2} r \mapsto w_1 * r \mapsto w_2 \end{array} \right\} \\
988 \\
989 \\
990 \\
991 \quad \boxed{\mathcal{R}(W)(\text{reg}_1, \text{reg}_2)} \quad \triangleq *_{(r, w_1) \in \text{reg}_1/\text{pc} \wedge (r, w_2) \in \text{reg}_2/\text{pc}} \mathcal{V}(W)(w_1, w_2) \\
992 \\
993 \quad \left\{ \begin{array}{l} \mathcal{V}(W)(z_1, z_2) \quad \triangleq z_1 = z_2 \\ \mathcal{V}(W)((o, g, b, e, a), w_2) \quad \triangleq (o, g, b, e, a) = w_2 \\ \mathcal{V}(W)((E, g, b, e, a), w_2) \quad \triangleq (E, g, b, e, a) = w_2 \\ \\ * \square \forall W' \sqsupseteq^g W, \triangleright \mathcal{E}(W') \left(\begin{array}{l} (RX, g, b, e, a), \\ (RX, g, b, e, a) \end{array} \right) \\ \\ \mathcal{V}(W)((p, g, b, e, a), w_2) \quad \triangleq (p, g, b, e, a) = w_2 \\ \\ * *_{a' \in [b, e]} \begin{cases} \mathcal{S}^u(W)(a', g, p, a) & \text{if } p = \text{U-} \\ \mathcal{S}(W)(a', g, p) & \text{otherwise} \end{cases} \\ \\ \wedge \begin{cases} \exists P, \text{rel}(a', P) * \text{rcond}(P) & \text{if } p = \text{RO} | \text{RX} \\ \text{rel}(a', \mathcal{V}) & \text{otherwise} \end{cases} \end{array} \right. \\
994 \\
995 \\
996 \\
997 \\
998 \quad \boxed{\mathcal{V}(W)(w_1, w_2)} \\
999 \\
1000 \\
1001 \\
1002 \\
1003 \\
1004 \quad \boxed{\text{rcond}(P)} \quad \triangleq \triangleright \square \forall W, w_1, w_2, P(W)(w_1, w_2) \text{ ---} \mathcal{V}(W)(w_1, w_2) \\
1005 \quad * \triangleright \square \forall W_1, W_2, z_1, z_2, P(W_1)(z_1, z_2) \text{ ---} P(W_2)(z_1, z_2) \\
1006 \\
1007 \quad \boxed{\text{State relation}} \\
1008 \quad \mathcal{S}(W)(a, g, p) \quad \triangleq \begin{cases} W^{\text{std}}(a) \in \{\text{Temporary, Permanent}\} & \text{if } \neg \text{write-local}(p) \wedge g = \text{DIRECTED} \\ W^{\text{std}}(a) = \text{Temporary} & \text{if } \text{write-local}(p) \wedge g = \text{DIRECTED} \\ W^{\text{std}}(a) = \text{Permanent} & \text{if } g \neq \text{DIRECTED} \end{cases} \\
1009 \\
1010 \\
1011 \quad \mathcal{S}^u(W)(a, g, p, \text{mid}) \quad \triangleq \begin{cases} \mathcal{S}(W)(a, g, p) \\ \vee \exists w_1, w_2, W^{\text{std}}(a) = \text{Uninitialized}(w_1, w_2) & \text{if } a \geq \text{mid} \\ & \wedge g = \text{DIRECTED} \\ \mathcal{S}(W)(a, g, p) & \text{otherwise} \end{cases} \\
1012 \\
1013 \\
1014 \\
1015 \\
1016 \\
1017 \\
1018 \\
1019 \\
1020 \\
1021 \\
1022 \\
1023 \\
1024 \\
1025 \\
1026 \\
1027 \\
1028 \\
1029
\end{array}$$

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

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 $\text{Uninitialized}(w_1, w_2)$. 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 $p = \text{RX}$, $p = \text{RWX}$ or $(p = \text{RWLX} \wedge g = \text{DIRECTED})$. 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 Appendix B. We use them to formally define contextual equivalence.

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

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

```

f3: prepstack r_stk
    loadU r0 r_stk (-1)
    push 2
    rclear RegName\{pc, r0}
    jmp r0

h3: prepstack r_stk
    loadU r0 r_stk (-1)
    push 3
    rclear RegName\{pc, r0}
    jmp r0

```

Fig. 11. Assembly of Lis. 8

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:

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*

$$comp_{f3}.exports \triangleq \{\emptyset : (E, GLOBAL, \dots, f3)\} \quad comp_{h3}.exports \triangleq \{\emptyset : (E, GLOBAL, \dots, h3)\}$$

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 \emptyset , and a memory segment with instructions (integers) only. Then

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

PROOF. The heart of the proof lies in showing $\mathcal{V}(W)((E, GLOBAL, \dots, f3), (E, GLOBAL, \dots, h3))$ (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 uninitialized 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' . \square

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](#).

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

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 $(\text{reg}, h, \text{stk}, \text{cs})$ 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 $\text{Stk}(d, p, b, e, a)$, and return capabilities $\text{Ret}(b, e, a)$. A capability $\text{Stk}(d, p, b, e, a)$ 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 $(\text{reg}, h, \text{stk}, \text{cs})$, then $d = 0$ provides access to the *oldest* stack frame (i.e., at the tail of cs), while $d = |\text{cs}|$ provides access to the current stack frame stk . For instance, if $|\text{cs}| = 1$, then it means that the current executing function is at depth 1, its caller is necessarily the main entrypoint function. Return capabilities $\text{Ret}(b, e, a)$ 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 $\text{call } r \vec{r}_{\text{args}}$ 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.

$$\begin{array}{l}
 \text{EXEC CALL} \\
 \dagger \varphi.\text{reg}(\text{pc}) = (p, \ell, b, e, a) \quad \dagger p \in \{\text{RX}, \text{RWX}, \text{RWLX}\} \quad \dagger [a, a + |\text{call } r \vec{r}_{\text{args}}| \subseteq [b, e[\\
 \dagger \varphi.h([a, a + |\text{call } r \vec{r}_{\text{args}}|]) = [\text{call}_0 r \vec{r}_{\text{args}}; \text{call}_1 r \vec{r}_{\text{args}}; \dots] \quad \dagger \forall r, r \in \vec{r}_{\text{args}}, \text{safe}(\varphi.\text{reg}(r)) \\
 \S \varphi.\text{reg}(r) = (\mathbb{E}, \ell', b', e', a') \quad \star \varphi.\text{reg}(r_{\text{stk}}) = \text{Stk}(d, \text{URWLX}, b_{\text{stk}}, e_{\text{stk}}, a_{\text{stk}}) \\
 \star d = |\varphi.\text{cs}| \quad \star [a, a_{\text{stk}} + |\text{act}_{\text{code}}| + 1 + |\vec{r}_{\text{args}}| \subseteq [b_{\text{stk}}, e_{\text{stk}}[\\
 \star \forall i, \text{canBeStored}(\varphi.\text{reg}(r_i), a_{\text{stk}} + i) \quad \varphi' = (\text{reg}', \varphi.h, \text{stk}', (\text{reg}^*, \text{stk}^*) :: \varphi.\text{cs}) \\
 \hline
 (\text{Executable}, \varphi) \rightarrow (\text{Executable}, \varphi')
 \end{array}$$

where

- reg' is defined such that $\text{reg}'(\text{pc}) = (\text{RX}, \ell', b', e', a')$, $\text{reg}'(r_{\text{stk}}) = \text{Stk}(d + 1, \text{URWLX}, a_{\text{stk}} + |\text{act}_{\text{code}}|, e_{\text{stk}}, a_{\text{stk}} + |\text{act}_{\text{code}}| + 1 + |\vec{r}_{\text{args}}|)$ and $\text{reg}'(r') = \text{reg}(r)$ if $r' = r$ then $\text{reg}(r)$ else 0;
- $\text{stk}' = \emptyset[a_{\text{stk}} + |\text{act}_{\text{code}}| \mapsto \text{Ret}(\dots), \dots];$

- $reg^* = reg[pc \mapsto (p, \ell, b, e, a + |\text{call } r \vec{r}_{args}|)];$
- $stk^* = stk[a_{stk} \mapsto \dots, \dots, a_{stk} + |\text{act}_{code}| - 1 \mapsto \dots].$

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 \dagger in EXEC CALL). 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 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 EXEC STEP, and simply executes one instruction at a time. The formal rule for call, denoted EXEC CALL has been formalized in Coq.

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

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

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. EXEC CALL 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. EXEC CALL 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 than 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.

- 1177 (1) Let ϕ and ψ be the initial states of p_1 and p_2 , then $\phi \sim \psi$.
 1178 (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
 1179 $\phi' \sim \psi'$.
 1180 (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 .

1181 A forward simulation implies preservation of termination [Leroy 2009]:
 1182

1183 **Lemma 6.3.** *If there exists a forward simulation between programs p_1 and p_2 , then $p_1 \Downarrow \Rightarrow p_2 \Downarrow$.*

1184 Before proving Theorem 6.1, we first show the following lemma.
 1185

1186 **Lemma 6.4.** *For all well-formed closed programs (see Appendix B for a precise definition) \mathcal{P} in the
 1187 overlay semantics, there exists a forward simulation between \mathcal{P} and its counterpart \mathbf{P} in the base
 1188 semantics.*

1189 **PROOF.** The detailed proof can be found in the accompanying Coq development, we only provide
 1190 a proof sketch here. The crux of the proof is to build a relation \sim and show it is a forward simulation.

1191 We say that $(\mathit{reg}, h, \mathit{stk}, \mathit{cs}) \sim (\mathit{reg}, \mathit{mem})$ when reg and reg contain related words in each
 1192 register. An integer is only related to itself, while a stack capability $\mathit{Stk}(d, p, b, e, a)$ is related to
 1193 a directed capability $(p, \text{DIRECTED}, b, e, a)$. Similarly, return capabilities $\mathit{Ret}(b, e, a)$ are related to
 1194 enter directed capabilities $(E, \text{DIRECTED}, b, e, a)$ and regular capabilities (p, ℓ, b, e, a) are related to
 1195 themselves. Furthermore, the heap h , the current stack stk and saved stack frames in the call stack
 1196 cs must have disjoint domains, as well as have related words at corresponding addresses with mem .
 1197

1198 When this relation is defined, it is relatively straightforward, though tedious, to show that it
 1199 is indeed a forward simulation. The simulation operates in a “lockstep” fashion, except for the
 1200 EXEC CALL and EXEC RETURN steps where one step in the overlay semantics is simulated by multiple
 1201 ones in the base semantics. \square

1202 We can now prove our full-abstraction theorem (Theorem 6.1).
 1203

1204 **PROOF.** By unfolding the definitions, we need to prove the following:
 1205

$$1206 (\forall C, C[\mathit{comp}_1] \Downarrow \Leftrightarrow C[\mathit{comp}_2] \Downarrow) \Leftrightarrow (\forall C, C[\mathbf{comp}_1] \Downarrow \Leftrightarrow C[\mathbf{comp}_2] \Downarrow)$$

1207 By combining Lemma 6.3 and Lemma 6.4, we know that for all closed programs p , $p \Downarrow \Rightarrow \mathbf{p} \Downarrow$.
 1208 Furthermore, as the base capability machine semantics is deterministic, we can actually build a
 1209 backward simulation from the forward simulation [Leroy 2009]. We thus actually have that for all
 1210 closed programs p , $p \Downarrow \Leftrightarrow \mathbf{p} \Downarrow$.

1211 Without loss of generality, we can thus consider the \Rightarrow implication (the other direction is
 1212 similar). By symmetry it suffices once again to consider only the \Rightarrow direction of $(\forall C, C[\mathbf{comp}_1] \Downarrow \Leftrightarrow$
 1213 $C[\mathbf{comp}_2] \Downarrow)$.

1214 We prove this by following the proof structure shown on the left. Let C be a context such that
 1215 $C[\mathbf{comp}_1] \Downarrow$, we need to prove that $C[\mathbf{comp}_2] \Downarrow$, knowing that $C[\mathit{comp}_1] \Downarrow \Leftrightarrow C[\mathit{comp}_2] \Downarrow$.

1216 $C[\mathit{comp}_1] \Downarrow \xRightarrow{(1)\uparrow} C[\mathit{comp}_2] \Downarrow$ The steps described in the figure can then be proved as follows.
 1217 $(1)\uparrow$ (2) $\Downarrow(3)$ (1) We use the backward simulation to prove that $C[\mathit{comp}_1] \Downarrow$.
 1218 $C[\mathbf{comp}_1] \Downarrow \xRightarrow{?} C[\mathbf{comp}_2] \Downarrow$ (2) By assumption, we have that $C[\mathit{comp}_1] \Downarrow \Leftrightarrow C[\mathit{comp}_2] \Downarrow$, and
 1219 therefore $C[\mathit{comp}_2] \Downarrow$.
 1220 (3) We use the forward simulation to conclude that $C[\mathbf{comp}_2] \Downarrow$.
 1221 \square

1222 7 RELATED WORK

1223 We now discuss some related work that has not already been discussed in the paper.
 1224
 1225

1226 In this paper we have emphasized temporal stack safety, but capabilities have also recently
1227 been proposed in the CHERI project as a mechanism for ensuring temporal memory safety for the
1228 heap. In particular, in CHERIVoke [Xia et al. 2019] and Cornucopia [Filardo et al. 2020] the authors
1229 suggest to use capabilities to efficiently and securely reclaim memory managed by a dedicated
1230 memory allocator using a garbage-collector-like approach. In contrast to our work, they do not
1231 formally state nor prove the guarantees provided by their mechanism, and it would be interesting
1232 to do that in future work.

1233 We have already discussed the most closely related work on formalising capability safety. Other
1234 related approaches include the work of Nienhuis et al. [2020] who define a syntactic notion of
1235 capability safety as a monotonicity guarantee of reachable objects (the machine does not create
1236 new capabilities out of thin air); in contrast to our approach Nienhuis et al. [2020] do not consider
1237 safety across calls to possibly adversarial code, so they only show that security properties hold
1238 within a single component. On the other hand, they consider a capability machine model with all
1239 of the instructions found on a real machine whereas we consider a core capability machine model.
1240 Devriese et al. [2016] propose a semantic approach to capturing capability safety for a high-level
1241 language with object capabilities using logical relations; this approach was further expanded upon
1242 by Swasey et al. [2017], who showed the robustness of several object capability patterns. Recently,
1243 El-Korashy et al. [2021] have studied the formal security guarantees of PAC (pointers-as-capabilities)
1244 compilers for partial programs and characterized them via a full abstraction result.

1245 Full abstraction [Abadi 1999] is a well-known property in the field of secure compilation [Pa-
1246 trignani et al. 2019] and has been used in recent works to characterize the security properties
1247 provided by different capability machines. Our approach follows the one proposed by Skorstengaard
1248 et al. [2019b] who use a fully abstract overlay semantics to define and show that their protection
1249 mechanisms enforce WBCF and LSE. Their proof uses a complex cross-language logical relation and
1250 is not mechanized, whereas we use a simpler simulation argument. Van Strydonck et al. [2021] also
1251 use a simulation-based argument for a fully abstract compiler from a statically verified language to
1252 an unverified language with support for linear capabilities. Likewise, Tsampas et al. [2019] also use
1253 a simulation proof for proving full abstraction between an “ideal” semantics with native temporal
1254 safety and an imperative language equipped with capabilities. It is interesting to note that their
1255 higher-level semantics already assumes well-bracketed control-flow with automatic deallocation of
1256 stackframes on return; we provably enforce that using directed capabilities.

1257 While we use an overlay semantics to characterize the notion of stack safety our calling conven-
1258 tion guarantees, Anderson et al. [2021] define stack safety as a trace property, expressed as the
1259 conjunction of LSE and WBCF. Anderson et al. distinguish between the integrity of local state, and
1260 the confidentiality of local state. Likewise, we develop a unary model to reason about the integrity
1261 of specific examples, and a binary model to reason about confidentiality. Unlike Anderson et al., we
1262 consider a machine with both a stack and a heap. Anderson et al. use their definition to validate the
1263 stack safety micro-policies proposed by Roessler and DeHon [2018], who use a general-purpose
1264 tagged architecture to design stack protection security policies. Unlike our capability-based calling
1265 convention, their policies do not incur an overhead when passing stack objects, but require more
1266 sophisticated tags, and a mechanism of lazy tagging to achieve the low overhead.

1267 Tagged architectures have also been used to enforce more general properties such as information-
1268 flow control [Azevedo de Amorim et al. 2014], secure compartmentalization [Abate et al. 2018],
1269 among other micro-policies [Azevedo de Amorim et al. 2015]. However, micro-policies must be
1270 expertly designed in order to leverage cache and be efficient.

1271 In a different line of work, software-based fault isolation (SFI) aims to provide process-based
1272 isolation by compartmentalizing (sandboxing) processes in different regions of the memory [Wahbe
1273 et al. 1993]. Our calling convention provides similar guarantees *despite* a shared stack. Recently,
1274

1275 Kolosick et al. [2022] have used an overlay semantics to characterize sufficient conditions for which
1276 so-called “heavyweight transitions” (context switching) can be safely replaced by “near zero cost
1277 transitions” akin to regular function calls. They further show that WebAssembly code compiled by
1278 a *correct* compiler would satisfy these conditions.

1279 Finally, we remark that our unary and binary models are built on a large body of work on
1280 characterizing security through logical relations. We use a logical approach [Dreyer et al. 2011,
1281 2010b; Turon et al. 2013] to step-indexed Kripke logical relations [Ahmed 2004; Birkedal et al. 2011],
1282 mechanized in the Coq implementation of Iris [Krebbbers et al. 2017b]. We use private and public
1283 transitions to characterise well-bracketed control flow [Dreyer et al. 2010a], and a new kind of
1284 transition that we call temporal transitions to characterise the temporal aspect of stack safety.
1285

1286 8 CONCLUSION AND FUTURE WORK

1287 We have demonstrated how directed capabilities can be used to enforce a strong degree of stack
1288 safety, including local state encapsulation, well-bracketed control flow, and temporal stack safety,
1289 with no stack clearing, and with only one additional bit. We have presented two new logical
1290 relations to reason about the integrity and confidentiality of specific examples, and proved a full
1291 abstraction result for an overlay semantics that defines our notion of stack safety. Finally, we
1292 discussed interesting subtleties of stack safety properties in a capability machine with a stack and a
1293 heap, and in the presence of stack objects crossing the boundary from caller to callee.
1294

1295 We have used contextual equivalence to formalize confidentiality, whereas Anderson et al. [2021]
1296 and Azevedo de Amorim et al. [2018] intuitively link confidentiality to a kind of non-interference
1297 property. We believe our calling convention based on directed capabilities also guarantees non-
1298 interference and in future work, it would be interesting to show this formally. To that end, one
1299 would probably have to extend the capability machine language with security labels.

1300 We have shown how our capability machine can implement function calls, as found in higher-
1301 level languages, in a secure manner. We believe it is easy to show that it can also implement
1302 tail-calls securely and conjecture that it is also possible to implement non-standard control flow
1303 such as C-style `setjmp/longjmp` efficiently and securely. Indeed, we may implement `setjmp` by
1304 creating a `Jmp` capability pointing to some activation code, similarly to how `call` creates a return
1305 capability. These capabilities can then be safely passed up the stack to callees. `longjmp` would then
1306 be implemented by jumping to such a capability. Such an implementation is efficient as `longjmp`
1307 is just a jump, and it is not necessary to unwind the stack. It is also safe in the sense that we
1308 can guarantee temporal safety of a `setjmp` environment: a caller will not be able to `longjmp` to
1309 a `setjmp` environment set up by one of its descendants (this is similar to how we ensure WBCF
1310 and guarantee that a return capability cannot be smuggled away). Temporal confidentiality of
1311 stack frames can still be enforced, and stack frames do *not* need to be scrubbed because directed
1312 capabilities guarantee that a caller cannot read them. Previously proposed calling conventions for
1313 capability machines either cannot provide such guarantees or would require careful unwinding or
1314 extensive memory clearing.
1315

1316 ACKNOWLEDGMENTS

1317 We thank the anonymous reviewers for excellent comments and suggestions. This work was sup-
1318 ported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program
1319 Verification (CPV), from the VILLUM Foundation. We would also like to thank Dominique De-
1320 vriese, Thomas Van Strydonck, Amin Timany, Armaël Guéneau and Frank Piessens for invaluable
1321 discussions and feedback.
1322
1323

REFERENCES

- 1324
1325 Martin Abadi. 1999. Protection in Programming-Language Translations. In *Secure Internet Programming, Security Issues for*
1326 *Mobile and Distributed Objects*. 19–34. https://doi.org/10.1007/3-540-48749-2_2
- 1327 Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Catalin Hritcu, Théo
1328 Laurent, Benjamin C. Pierce, Marco Stronati, and Andrew Tolmach. 2018. When Good Components Go Bad: Formally
1329 Secure Compilation Despite Dynamic Compromise. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer*
1330 *and Communications Security, CCS 2018, Toronto, ON, Canada, October 15–19, 2018*. 1351–1368. <https://doi.org/10.1145/3243734.3243745>
- 1331 Amal Ahmed. 2004. *Semantics of Types for Mutable State*. Ph.D. Dissertation. Princeton University.
- 1332 Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-dependent representation independence. In *Proceedings of*
1333 *the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA,*
1334 *January 21–23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 340–353. <https://doi.org/10.1145/1480881.1480925>
- 1335 Sean Noble Anderson, Leonidas Lampropoulos, Roberto Blanco, Benjamin C. Pierce, and Andrew Tolmach. 2021. Security
1336 Properties for Stack Safety. *CoRR abs/2105.00417* (2021). arXiv:2105.00417 <https://arxiv.org/abs/2105.00417>
- 1337 Arm. 2021. Morello project. Retrieved July 6, 2021 from <https://www.morello-project.org/>
- 1338 Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Catalin Hritcu, David Pichardie, Benjamin C.
1339 Pierce, Randy Pollack, and Andrew Tolmach. 2014. A verified information-flow architecture. In *The 41st Annual ACM*
1340 *SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20–21,*
1341 *2014*. 165–178. <https://doi.org/10.1145/2535838.2535839>
- 1342 Arthur Azevedo de Amorim, Maxime Dénès, Nick Giannarakis, Catalin Hritcu, Benjamin C. Pierce, Antal Spector-Zabusky,
1343 and Andrew Tolmach. 2015. Micro-Policies: Formally Verified, Tag-Based Security Monitors. In *2015 IEEE Symposium on*
1344 *Security and Privacy, SP 2015, San Jose, CA, USA, May 17–21, 2015*. 813–830. <https://doi.org/10.1109/SP.2015.55>
- 1345 Arthur Azevedo de Amorim, Catalin Hritcu, and Benjamin C. Pierce. 2018. The Meaning of Memory Safety. In *Principles of*
1346 *Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and*
1347 *Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14–20, 2018, Proceedings (Lecture Notes in Computer Science),*
1348 Lujio Bauer and Ralf Küsters (Eds.), Vol. 10804. Springer, 79–105. https://doi.org/10.1007/978-3-319-89722-6_4
- 1349 Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. 2011. Step-
1350 indexed Kripke models over recursive worlds. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles*
1351 *of Programming Languages, POPL 2011, Austin, TX, USA, January 26–28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM,
1352 119–132. <https://doi.org/10.1145/1926385.1926401>
- 1353 Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. 1994. Hardware Support for Fast Capability-Based Addressing.
1354 In *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 319–327.
1355 <https://doi.org/10.1145/195473.195579>
- 1356 Chromium. 2020. Memory safety. Retrieved July 6, 2021 from [https://www.chromium.org/Home/chromium-security/](https://www.chromium.org/Home/chromium-security/memory-safety)
1357 [memory-safety](https://www.chromium.org/Home/chromium-security/memory-safety)
- 1358 Jack B. Dennis and Earl C. Van Horn. 1966. Programming Semantics for Multiprogrammed Computations. *Commun. ACM*
1359 9, 3 (March 1966), 143–155. <https://doi.org/10.1145/365230.365252>
- 1360 Dominique Devriese, Lars Birkedal, and Frank Piessens. 2016. Reasoning about Object Capabilities with Logical Relations
1361 and Effect Parametricity. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany,*
1362 *March 21–24, 2016*. IEEE, 147–162. <https://doi.org/10.1109/EuroSP.2016.22>
- 1363 Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2011. Logical Step-Indexed Logical Relations. *LMCS* 7, 2:16 (June 2011),
1364 1–37.
- 1365 Derek Dreyer, Georg Neis, and Lars Birkedal. 2010a. The impact of higher-order state and control effects on local relational
1366 reasoning. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010,*
1367 *Baltimore, Maryland, USA, September 27–29, 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM, 143–156. <https://doi.org/10.1145/1863543.1863566>
- 1368 Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. 2010b. A relational modal logic for higher-order stateful
1369 ADTs. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010,*
1370 *Madrid, Spain, January 17–23, 2010*. 185–198. <https://doi.org/10.1145/1706299.1706323>
- 1371 Akram El-Korashy, Stelios Tsampas, Marco Patrignani, Dominique Devriese, Deepak Garg, and Frank Piessens. 2021.
1372 CapablePtrs: Securely Compiling Partial Programs Using the Pointers-as-Capabilities Principle. In *2021 IEEE 34th*
1373 *Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, Los Alamitos, CA, USA, 421–436. <https://doi.org/10.1109/CSF51468.2021.00036>
- 1374 Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan
1375 Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka,
1376 Alexandre Joannou, A. Theodore Markettos, Alfredo Mazinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey
1377 Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. 2020. Cornucopia: Temporal

- 1373 Safety for CHERI Heaps. In *IEEE Symposium on Security and Privacy*. IEEE.
- 1374 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>
- 1375
- 1376 Aina Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique
- 1377 Devriese, and Lars Birkedal. 2021. Efficient and provable local capability revocation using uninitialized capabilities. *Proc.*
- 1378 *ACM Program. Lang.* 5, POPL (2021), 1–30. <https://doi.org/10.1145/3434287>
- 1379 Nicolas Joly, Saif ElSherei, and Saar Amar. 2020. Security Analysis of CHERI ISA. Retrieved July 6, 2021 from [https://msrc-](https://msrc-blog.microsoft.com/2020/10/14/security-analysis-of-cheri-isa/)
- 1380 [blog.microsoft.com/2020/10/14/security-analysis-of-cheri-isa/](https://msrc-blog.microsoft.com/2020/10/14/security-analysis-of-cheri-isa/)
- 1381 Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *Proceedings of the 21st ACM*
- 1382 *SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18–22, 2016*. 256–269. <https://doi.org/10.1145/2951913.2951943>
- 1383 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the
- 1384 ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. <https://doi.org/10.1017/S0956796818000151>
- 1385 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris:
- 1386 Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM*
- 1387 *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>
- 1388 Matthew Kolosick, Shraavan Narayan, Conrad Watt, Michael LeMay, Deepak Garg, Ranjit Jhala, and Deian Stefan. 2022.
- 1389 Isolation Without Taxation: Near Zero Cost Transitions for SFI. In *ACM SIGPLAN Symposium on Principles of Programming*
- 1390 *Languages (POPL)*. ACM.
- 1391 Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud,
- 1392 and Derek Dreyer. 2018. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *PACMPL*
- 1393 2, ICFP (2018), 77:1–77:30. <https://doi.org/10.1145/3236772>
- 1394 Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017a. The Essence
- 1395 of Higher-Order Concurrent Separation Logic. In *Programming Languages and Systems - 26th European Symposium on*
- 1396 *Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017,*
- 1397 *Uppsala, Sweden, April 22–29, 2017, Proceedings*. 696–723. https://doi.org/10.1007/978-3-662-54434-1_26
- 1398 Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017b. Interactive proofs in higher-order concurrent separation logic.
- 1399 In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France,*
- 1400 *January 18–20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 205–217. [https://doi.org/10.1145/3009837.](https://doi.org/10.1145/3009837.3009855)
- 1401 [3009855](https://doi.org/10.1145/3009837.3009855)
- 1402 Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. 2017. A relational model of types-and-effects in higher-
- 1403 order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming*
- 1404 *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>
- 1405 Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *J. Autom. Reason.* 43, 4 (2009), 363–446. [https://doi.org/10.](https://doi.org/10.1007/s10817-009-9155-4)
- 1406 [1007/s10817-009-9155-4](https://doi.org/10.1007/s10817-009-9155-4)
- 1407 Henry M. Levy. 1984. *Capability-Based Computer Systems*. Digital Press. <https://homes.cs.washington.edu/~levy/capabook/>
- 1408 Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony Fox, Michael Roe, Brian Campbell, Matthew Naylor,
- 1409 Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson, and Peter Sewell. 2020. Rigorous
- 1410 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)*.
- 1411 Marco Patrignani, Amal Ahmed, and Dave Clarke. 2019. Formal Approaches to Secure Compilation: A Survey of Fully Abstract
- 1412 Compilation and Related Work. *ACM Comput. Surv.* 51, 6, Article 125 (Feb. 2019), 36 pages. <https://doi.org/10.1145/3280984>
- 1413 Nick Roessler and André DeHon. 2018. Protecting the Stack with Metadata Policies and Tagged Hardware. In *2018 IEEE*
- 1414 *Symposium on Security and Privacy, SP 2018, Proceedings, 21–23 May 2018, San Francisco, California, USA*. IEEE Computer
- 1415 Society, 478–495. <https://doi.org/10.1109/SP.2018.00066>
- 1416 Lau Skorstengaard. 2019. *Formal Reasoning about Capability Machines*. Ph.D. Dissertation. Aarhus University.
- 1417 Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2018. Reasoning About a Machine with Local Capabilities -
- 1418 Provably Safe Stack and Return Pointer Management. In *Programming Languages and Systems - 27th European Symposium*
- 1419 *on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS*
- 1420 *2018, Thessaloniki, Greece, April 14–20, 2018, Proceedings*. 475–501. https://doi.org/10.1007/978-3-319-89884-1_17
- 1421 Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019a. Reasoning about a Machine with Local Capabilities:
- 1422 Provably Safe Stack and Return Pointer Management. *ACM Transactions on Programming Languages and Systems* 42, 1
- 1423 (Dec. 2019), 5:1–5:53. <https://doi.org/10.1145/3363519>

- 1422 Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019b. StkTokens: Enforcing Well-Bracketed Control Flow
1423 and Stack Encapsulation Using Linear Capabilities. *Proc. ACM Program. Lang.* 3, POPL, Article 19 (Jan. 2019), 28 pages.
1424 <https://doi.org/10.1145/3290332>
- 1425 Eijiro Sumii and Benjamin C. Pierce. 2007. A bisimulation for type abstraction and recursion. *J. ACM* 54, 5 (2007), 26.
1426 <https://doi.org/10.1145/1284320.1284325>
- 1427 David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and Compositional Verification of Object Capability Patterns.
1428 In *OOPSLA*. ACM. <https://people.mpi-sws.org/~swasey/papers/ocpl/ocpl-20170418.pdf>
- 1429 Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *2013 IEEE Symposium on*
1430 *Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. 48–62. <https://doi.org/10.1109/SP.2013.13>
- 1431 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/>
- 1432 Stelios Tsampas, Dominique Devriese, and Frank Piessens. 2019. Temporal Safety for Stack Allocated Memory on Capability
1433 Machines. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. 243–255.
1434 <https://doi.org/10.1109/CSF.2019.00024>
- 1435 Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and Hoare-style reasoning in a logic for higher-
1436 order concurrency. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA -*
1437 *September 25 - 27, 2013*. 377–390. <https://doi.org/10.1145/2500365.2500600>
- 1438 Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. 2021. Linear capabilities for fully abstract compilation of
1439 separation-logic-verified code. *J. Funct. Program.* 31 (2021), e6. <https://doi.org/10.1017/S0956796821000022>
- 1440 Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-Based Fault Isolation. In
1441 *Proceedings of the Fourteenth ACM Symposium on Operating System Principles, SOSP 1993, The Grove Park Inn and Country*
1442 *Club, Asheville, North Carolina, USA, December 5-8, 1993*. 203–216. <https://doi.org/10.1145/168619.168635>
- 1443 Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John
1444 Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard
1445 Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan
1446 Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. 2020. *Capability*
1447 *Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8)*. Technical Report UCAM-CL-TR-951.
1448 University of Cambridge, Computer Laboratory. <https://doi.org/10.48456/tr-951>
- 1449 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.
1450 Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable
1451 Software Compartmentalization. In *IEEE Symposium on Security and Privacy*. 20–37. <https://doi.org/10.1109/SP.2015.9>
- 1452 Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony C. J. Fox, Robert M. Norton, David Chisnall, Brooks Davis,
1453 Khilan Gudka, Nathaniel Wesley Filardo, A. Theodore Markettos, Michael Roe, Peter G. Neumann, Robert N. M. Watson,
1454 and Simon W. Moore. 2019. CHERI Concentrate: Practical Compressed Capabilities. *IEEE Trans. Computers* 68, 10 (2019),
1455 1455–1469. <https://doi.org/10.1109/TC.2019.2914037>
- 1456 Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg,
1457 Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. 2019. CHERIvoke: Characterising
1458 Pointer Revocation Using CHERI Capabilities for Temporal Memory Safety. In *IEEE/ACM International Symposium on*
1459 *Microarchitecture*. ACM. <https://doi.org/10.1145/3352460.3358288>
- 1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470

1471 A PROOF SKETCH OF THE FUNDAMENTAL THEOREM

1472 One of the key facts supporting that our logical relation models temporal stack safety is the
 1473 following monotonicity lemma. It describes the monotonicity of \mathcal{V} relative to a word's current
 1474 read authority. It entails in particular that a stack capability that is valid in some world W is also
 1475 valid in a relative-to-its-current-address-future world; and thus it does not depend on the contents
 1476 of higher stack frames.

1477 **Lemma A.1** (Address Relative Monotonicity).

1478 (1) $p \in \{URWLX, URWL, URWX, URW\} \wedge W' \sqsupseteq^a W \rightarrow \mathcal{V}(W)(p, g, b, e, a) \multimap \mathcal{V}(W')(p, g, b, e, a)$

1479 (2) $p \notin \{URWLX, URWL, URWX, URW\} \wedge W' \sqsupseteq^e W \rightarrow \mathcal{V}(W)(p, g, b, e, a) \multimap \mathcal{V}(W')(p, g, b, e, a)$

1482 Furthermore, the address relative temporal future world relation can be weakened by lowering
 1483 the associated address, and the value relation is monotone wrt. private future worlds (for all
 1484 non-directed capabilities).

1485 **Lemma A.2** (Address Relative Weakening). $a' \leq a \rightarrow W' \sqsupseteq^a W \rightarrow W' \sqsupseteq^{a'} W$

1487 **Lemma A.3** (Private Monotonicity). *Let w be a word that is not DIRECTED. Then $W' \sqsupseteq^{priv} W \rightarrow$
 1488 $\mathcal{V}(W)(w) \multimap \mathcal{V}(W')(w)$.*

1489 **THEOREM 4.1** (FTLR). *Assume that $p = RX$, $p = RWX$ or $(p = RWLX \wedge g = DIRECTED)$. Assume also
 1490 that $\mathcal{V}(W)(p, g, b, e, a)$. Then we have that $\mathcal{E}(W)(p, g, b, e, a)$.*

1492 **PROOF.** Upon unfolding the definition of \mathcal{E} , our goal is to prove that the weakest precondition
 1493 holds for a program pointed to by the program counter (p, g, b, e, a) , given the predicates of a safe
 1494 register state. According to the underlying definition of Iris weakest preconditions, this amounts to
 1495 showing that the program does not get stuck; either it executes the next instruction and continues,
 1496 or the program attempts to reach outside its bounds of authority and subsequently crashes into a
 1497 failed configuration.

1498 The fundamental theorem is proved by Löb induction (the proof principle for guarded recursion),
 1499 generalized for all W, p, g, b, e and a . The induction hypothesis states that the program will safely
 1500 execute *later*, in other words after at least one step of execution. The fundamental theorem is thus
 1501 proved by stepping through the execution of the next instruction, currently pointed to by a , (which
 1502 may either fail or succeed), and then applying the induction hypothesis *one step later*.

1503 First, we extract the resources needed for executing the next instruction. Since p has at least
 1504 read authority over its range of authority, the assumption $\mathcal{V}(W)(p, g, b, e, a)$ grants access to the
 1505 standard resources within the range $[b, e[$. In particular, if the program counter is valid, we can
 1506 extract the standard resources for a , including the points-to predicate $a \mapsto w$ for some Word w .
 1507 The proof then proceeds by case analysis on $decode(w)$.

1508 Let us focus on the particularly interesting case $storeU\ r_{dst}\ r_{src}\ 0$ in which the register r_{dst}
 1509 contains a stack capability, currently pointing to some address a_{stk} . In this case, we will be overriding
 1510 some (uninitialized) word with a new possibly DIRECTED and valid word w_{src} from r_{src} . Since the
 1511 offset is 0, the capability in r_{dst} will increment, thus initializing address a_{stk} . We will therefore have
 1512 to update the standard resource for a_{stk} to an initialized standard resource, now pointing to the
 1513 word w_{src} . In particular, since this is a stack address, we will have to prove that w_{src} satisfies the
 1514 conditions of a stack standard resource, namely that w_{src} is monotone with regards to $\sqsupseteq^{a_{stk}}$. We
 1515 first use Lemma A.1, which asserts that w_{src} is monotone relative to the upper bound of its own
 1516 read authority. Storing a DIRECTED capability on the stack will dynamically check that this upper
 1517 bound is *smaller* than the current stack address, in other words below a_{stk} . We can therefore apply
 1518 Lemma A.2 to assert that w_{src} is indeed monotone with regards to $\sqsupseteq^{a_{stk}}$.

1520 The remainder of the cases are similar. Instructions that interact with memory require invariants
1521 to be opened, whereas instructions that only change the register state will only have to establish
1522 the validity of updated registers. □

1523

1524

1525

1526

1527

1528

1529

1530

1531

1532

1533

1534

1535

1536

1537

1538

1539

1540

1541

1542

1543

1544

1545

1546

1547

1548

1549

1550

1551

1552

1553

1554

1555

1556

1557

1558

1559

1560

1561

1562

1563

1564

1565

1566

1567

1568

B COMPONENTS, LINKING AND CONTEXTS

$ms \in \text{MemFrag} ::= \text{Addr} \rightarrow \text{Word}$

$\overline{a \leftarrow s} \in \text{Imports} ::= \overline{\text{Symbols} \times \text{Addr}}$

$\overline{s \mapsto w} \in \text{Exports} ::= \text{Symbols} \rightarrow \text{Word}$

$\text{basecomp} \in \text{BaseComp} ::= \text{Memfrag} \times \text{Imports} \times \text{Exports}$

$\text{Comp} ::= \text{basecomp} \mid (\text{basecomp}, c_{\text{main}})$

(a) Components.

$$\begin{array}{l} \text{comp}_1 = (ms_1, \overline{a \leftarrow s^1}, \overline{s \mapsto w^1}) \\ \text{comp}_2 = (ms_2, \overline{a \leftarrow s^2}, \overline{s \mapsto w^2}) \quad \text{comp} = (ms, \overline{a \leftarrow s}, \overline{s \mapsto w}) \quad \overline{s \mapsto w} = \overline{s \mapsto w^1} \cup \overline{s \mapsto w^2} \\ ms = (ms_1 \uplus ms_2)[a \mapsto w \mid a \leftarrow s \in (\overline{a \leftarrow s^1} \cup \overline{a \leftarrow s^2}), s \mapsto w \in \overline{s \mapsto w}] \\ \overline{a \leftarrow s} = (\overline{a \leftarrow s^1} \cup \overline{a \leftarrow s^2}) \setminus \{ _ \leftarrow s \mid s \mapsto w \in \overline{s \mapsto w} \} \end{array}$$

$$\text{comp} = \text{comp}_1 \bowtie \text{comp}_2$$

(b) Linking.

Fig. 12. Components and linking.

A closed program is created by linking multiple components. A component (defined in Fig. 12a) is either a base component representing a library waiting to be linked with other components. A main component is a base component with a main entrypoint. A base component is defined by a memory fragment representing the part of the memory owned by the component and its content, a list of imports indicates where a word associated with a symbol will be stored by the linker, and a list of exports associating words with symbols.

Linking components comp_1 and comp_2 , noted $\text{comp}_1 \bowtie \text{comp}_2$, is defined in Fig. 12b. Two components can only be linked if they have disjoint memories, and at most one of them is a main component. The new component $\text{comp}_1 \bowtie \text{comp}_2$ is obtained by taking the union of both components' memory fragments where imports satisfied by an export have the memory updated with the exported word on the imported address. The satisfied imports are removed from the resulting linking, and the exports are obtained by combining the components' exports.

We can now define a notion of context and what is a closed program.

Definition B.1 (Closed programs and contexts). A main component is a closed program when its imports list is empty.

A component C is a context for component comp , written $C[\text{comp}]$ when $\text{comp} \bowtie C$ is a closed program.

We now need to define how to run a closed program, this is done by defining its initial state. However, we are only interested in well-formed programs.

Definition B.2 (Well-formed components). A base component $(ms, \overline{a \leftarrow s}, \overline{s \mapsto w})$ is well-formed when:

- import and export symbols are disjoint;
- all exported words can only address memory owned by the component, i.e., ms ;
- all import addresses are part of the memory owned by the component;
- ms only contains integers or capabilities that can address memory within ms , are not permit-write-local (i.e., have permission URWL, URWLX, RWL or RWLX) and are global;

- ms is disjoint from the memory reserved for the call stack.

A main component is well-formed when its base component is well-formed and its main entrypoint can only address memory it owns.

These properties are required so that initial assumptions of the capability machine are not violated. For instance, stack safety cannot be enforced if any of the components already have direct access to the call stack or can keep a copy of stack derived capabilities in their own memory by having permit-write-local capabilities.

Finally, assuming that $[b_{stk}, e_{stk}[$ is the range reserved for the stack, the initial configuration of a closed program $((ms, [], \overline{s} \mapsto \overline{w}), c_{main})$ is a configuration (reg, m) in the base semantics such that:

- $reg(pc) = c_{main}$;
- $reg(r_{stk}) = (URWLX, Directed, b_{stk}, e_{stk}, b_{stk})$;
- $reg(r) = 0$ if $r \neq pc$ or $r \neq r_{stk}$;
- $mem(a) = ms(a)$ if $a \in \text{dom}(ms)$, and $mem(a) = 0$ otherwise.

In the overlay semantics, the initial configuration of a closed program $((ms, [], \overline{s} \mapsto \overline{w}), c_{main})$ is the configuration $(reg, ms, \emptyset, [])$ such that:

- $reg(pc) = c_{main}$;
- $reg(r_{stk}) = \text{Stk}(0, URWLX, b_{stk}, e_{stk}, b_{stk})$;
- $reg(r) = 0$ if $r \neq pc$ or $r \neq r_{stk}$.

C DEFINITION OF SAFE WORD

In the definition of EXEC CALL, arguments must be words that are *safe* to pass. This notion is defined as follows.

Definition C.1 (Safe word). Given an overlay configuration $\varphi = (reg, h, stk, cs)$, a word w is *safe* to pass as argument in that configuration, written $safe(\varphi, w)$ if one of the following holds.

- w is an integer ($w \in \mathbb{Z}$);
- w is a heap capability (p, ℓ, b, e, a) ;
- w is a stack derived capability $Stk(d, p, b, e, a)$ such that
 - $d < |cs|$ and cs is of the form $(reg_n, stk_n) :: \dots :: (reg_0, stk_0)$ and $reg_d(r_{stk}) = Stk(d, p', b', e', a')$ and $e < a'$ and for all $a'' \in [b, canReadUpTo(w)[, safe(\varphi, stk_d(a''))]$;
 - or $d = |cs|$ and the above conditions hold using $reg_d = reg$ and $stk_d = stk$ instead.

Recall that, intuitively, a word is safe to pass if it cannot tamper the activation code of active return capabilities. As such, integers and heap capabilities are obviously safe. Return capabilities are unsafe to pass as this would break WBCF, though allowing to pass them would actually implement a form of long jmp. The activation code of return capabilities is stored on the stack, just above the stackframe used by a function, hence the $e < a'$ condition above. We must also ensure that the capability can only give access to safe words itself.

We point out that while the definition may seem not well-founded as a stack capability can only be safe if its contents are also safe, which may itself contain the original capability at first glance. This is actually not the case as stack capabilities are *directed capabilities*, and this definition provides a terminating algorithm. Indeed, in the recursive check $safe(\varphi, stk_d(a''))$, it is either a word that is immediately safe (i.e., an integer or a heap capability), or immediately unsafe (a return capability), or a stack capability w' such that $canReadUpTo(w') \leq a''$ (by property of directed capabilities) and $a'' < canReadUpTo(w)$ (by definition of a''). There is therefore a decreasing measure ensuring termination of the check.

In the Coq formalization, we use a more *restrictive* notion of safe words that instead only allow passing integers and heap capabilities.