Vrije Universiteit Brussel

Temporal safety for stack allocated memory on capability machines

Tsampas, Stelios; Devriese, Dominique; Piessens, Frank

Published in:
32nd IEEE Computer Security Foundations Symposium

DOI:
10.1109/CSF.2019.00024

Publication date:
2019

License:
Unspecified

Document Version:
Accepted author manuscript

Link to publication

Citation for published version (APA):
Tsampas, S., Devriese, D., & Piessens, F. (2019). Temporal safety for stack allocated memory on capability machines. In 32nd IEEE Computer Security Foundations Symposium (pp. 243-255). (IEEE Computer Security Foundations Symposium (CSF)). IEEE. https://doi.org/10.1109/CSF.2019.00024

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 21. Oct. 2019

Temporal safety for stack allocated memory on capability machines

Stelios Tsampas[1], Dominique Devriese[2], Frank Piessens[1]
[1] imec-Distrinet, KU Leuven, Belgium; email `name.surname@cs.kuleuven.be`
[2] Vrije Universiteit Brussel, Belgium; email `dominique.devriese@vub.be`

***Abstract*—Memory capabilities as supported in capability machines are very similar to fat pointers, and hence are very useful for the efficient enforcement of spatial memory safety. Enforcing temporal memory safety however, is more challenging. This paper investigates an approach to enforce temporal memory safety for stack-allocated memory in C-like languages by extending capabilities with a simple dynamic mechanism. This mechanism ensures that capabilities with a certain lifetime can only be stored in memory that has a longer lifetime. Our mechanism prevents temporal memory safety violations, yet is sufficiently permissive to allow typical C coding idioms where addresses of local variables are passed up the call stack. We formalize the desired behavior of a simple C-like language as a dependently typed operational semantics, and we show that existing compilers to capability machines do not simulate this desired behavior: they either have to break temporal safety, or they have to defensively rule out allowed behaviors. Finally, we show that with our proposed dynamic mechanism, our compiler is fully abstract.**

***Index Terms*—capabilities, temporal memory safety, machine-checked proof**

```c
void aFun(void) {
  int ar[100];

  getData(ar, 100);
  sortInt(ar, 100);
}
```

Fig. 1: A C function calling external functions `getData` and `sortInt` to mutate local array `ar`.

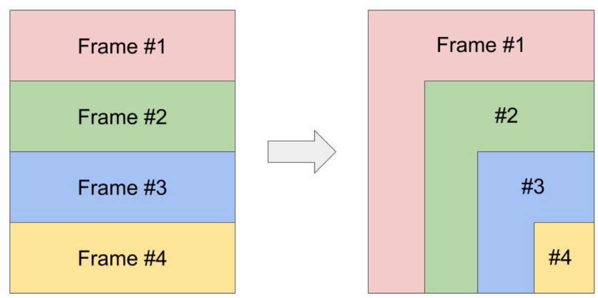

Fig. 2: A call stack and its hierarchy of lifetimes. On the left side there are the stack frames as found in memory, while on the right we have the lifetimes of their objects.

I. Introduction

Capability machines [1], [2] have recently enjoyed a resurgence in the field of secure compilation, not least because of the advent of CHERI [3], a capability processor based on MIPS that supports real-world operating systems and applications. Unlike ordinary processors, capability machines offer a number of low-level features that propels them as compelling candidates for applications where hardware-level security is in high demand. One such feature is *memory capabilities*, a form of fat pointers that grant access to a contiguous region of memory. Another important feature is sandboxed execution via *object capabilities*, a special kind of capabilities that represent bundled executable code. These features are specifically useful for securely compiling C since pointers can be seamlessly translated to capabilities [4], [5].

Yet despite the emergence of capability machines, neither CHERI nor the theoretical models found in published work [6] adequately consider a feature found in most of the eminent imperative programming languages, *passing references to local variables*. For instance, compiling and executing the C program found in Figure 1 to CHERI will cause a violation if `getData` or `sortInt` are exported by a different sandbox than `aFun`. Examples such as this one are found often enough in real-world applications to make this shortcoming especially problematic.

From a semantics standpoint, many block-structured programming languages like C [7] and C++ [8] operate under a hybrid memory model composed of a global store, more informally knows as *heap*, and a local store, the *stack*. The heap stores dynamically allocated objects and globals, while the stack is used for local variables. During execution the stack grows and shrinks accordingly to accommodate for the various function calls and returns. This forms a hierarchy of scopes and lifetimes: objects in inner function calls have shorter lifetimes compared to objects found in outer function calls as seen in Figure 2. On the contrary, heap objects live indefinitely until freed or garbage collected.

The root of the problem lies in the fact that capability machines typically distinguish lifetimes of capabilities in a coarse manner: a capability may either be global or local [3], [6]. When compiling high level languages to capability machines, global or dynamically allocated variables are mapped to global capabilities while all local variables irrespective of their lifetime are mapped to a single kind of local capability. To alleviate the risk of attacks, all interactions that might lead to undefined behavior are either prohibited or require special permissions. For example, a special store-local permission bit is needed to store local capabilities in memory [9]. More importantly, passing local capabilities as arguments when invoking an object capability is prohibited hence the violation in Figure 1.

At the same time, simply lifting this limitation compromises the security properties of the system, as shown in Figure 3. A malicious sandbox can invoke itself (or another "accomplice" sandbox) and obtain a dangling reference to unallocated space. When the victim sandbox is called and creates its own private stack, the dangling reference, in possession of the attacker, may point to resources in the newly created space. This is a dangerous situation which could violate both confidentiality and integrity. For instance, the attacker may now peek in the victim's private stack or, if the dangling reference is passed over to the victim, break invariants due to the unexpected aliasing.

Just as capabilities protect against buffer overflow attacks in single-sandbox situations [10], there is incentive for temporal safety on the stack outside of sandboxed execution. Temporal safety violations in the heap often lead to vulnerabilities, as is the case with *use-after-free* [11] attacks. The stack-based equivalents are known as *stack-based use-after-free* [12], *use-after-return* or *use-after-scope* [13] and are being exploited in the wild [14], [15].

A simplified, representative example is given at Figure 4. Function `h` assigns argument pointer `p` an address that, after `h` returns, points at unallocated memory. When function `victim` is called and depending on the compiler and stack allocator, `q` might be pointing at the return address of `victim`. Thus any assignment compromises the execution flow of `victim`. Interestingly, this example works both as an exploitable bug in single-sandbox situations or as a security violation in a sandboxed environment: `main` and `f` can also be seen as malicious sandboxes that perform an attack on `victim`.

In this paper, we present our solution to stack-based temporal safety for capability machines. Specifically, we introduce extensions to a generic model of a capability machine to support various levels of object lifetimes. We then claim that the extensions are safe and correct and proceed to formally confirm our intuitions. In particular:

– We introduce a simple block structured imperative language with local scopes and references, which captures the relevant part of languages like C/C++ and, to a lesser extent, Java.

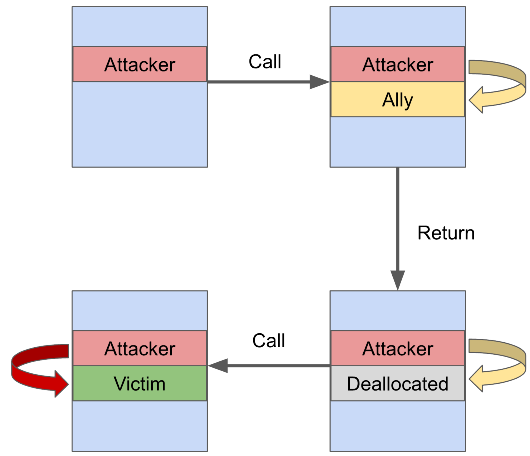

Fig. 3: Malicious aliasing in a capability machine. Straight arrows are sandbox invocations/returns and curved arrows are pointers. An attacking context can create a dangling reference and force resource aliasing later on.

```
void f(int** p) {
  int x;
  *p = &x; // Unsafe assignment
}

void main() {
  int *q;
  f(&q); // q points at unused stack memory
  h(q);
}

void victim(int* q) {
  *q = 0; // May overwrite own return address
}
```

Fig. 4: An example of an exploitable single-sandbox bug or an attack in a sandboxed environment. Function `victim` may end up overwriting its own return address.

– We formalize *idealized*, dependently-typed semantics for our language that intrinsically provide guarantees of temporal safety and rule out unsafe operations. Our semantics are slightly stricter than the C/C++ standard [7], [8], but are more convenient for formal proofs and sufficiently permissive to allow typical and recommended C programming practices.
– We present a low-level capability machine that is not yet augmented with our extensions and formally confirm that it fails to simulate our language. This machine treats local objects in the same manner found in the current iteration of CHERI.
– We then introduce our extensions and prove that the identity compiler from the ideal semantics to the extended capability machine is fully abstract.

– We analyze and discuss potential implementations in CHERI.

A. Notation and typesetting

The semantics and the proofs are all mechanized using the dependently typed programming language **Agda** [16]. We shall spare the reader the frustration of navigating through cryptic proof terms by introducing a more intuitive presentation that (mostly) follows common practices in the literature.

Datatypes (sets), predicates, relations and function are typeset in blue, singletons and data constructors in green. Variables are typeset in *gray* and we write $a \in \mathsf{Set}$ to denote that variable a belongs in set Set. We shall use the same notation for predicates and relations, unless it makes more sense to use a more intuitive notation, for instance $a \equiv b$.

Simple datatypes and relations are presented in a direct, Haskell-like style with "|" denoting disjoint union. For example, a inductive definition for $\mathbb{N}$ would look like:

$$\mathsf{Nat} := \mathsf{zero} \mid \mathsf{suc}\ \mathsf{Nat}$$

We shall be using "$\times$" for Cartesian products, "," as the product constructor and we typeset projection functions in pink.

Complex datatypes and relations will be presented using inference rules. Variable declarations the datatype of which are obvious will be omitted and the reader may assume that each of the variables is universally quantified.

Lists are used throughout the paper and we shall be using a special notation for them. A list of head a and tail t is written as $a :: t$, while the empty list is $[]$. We write $a \triangleright_i L$ to denote that $L[i] \equiv a$.

Finally, because of the widespread use of colors for typesetting we recommend printing this paper in color for optimal readability.

II. The language

We begin by introducing our block structured imperative language, **ImpR**. ImpR is at its core a safe, simplified version of C focusing on language constructs relevant to our problem: it has blocks, local variables and references. It is also statically typed but unlike C it does not support type casting.

We define the set of types of ImpR, Ty, as

$$\mathsf{Ty} := \mathsf{Unit} \mid \mathsf{Nat} \mid \mathsf{Ref}\ \mathsf{Ty}$$

Apart from the two bases types Unit and Nat there is a constructor Ref to model references in the obvious way: given $t : \mathsf{Ty}$ then $\mathsf{Ref}\ t$ denotes a reference to t. We then define the record Ns (for namespace) as a pair of Ty:

$$\mathsf{Ns} := \mathsf{Ty} \times \mathsf{Ty}$$
$$\mathsf{TypeOf} \in \mathsf{Ns} \times (\mathrm{arg} \uplus \mathrm{local}) \to \mathsf{Ty}$$

Ns represents a restricted, local *typing* environment for the functions of ImpR, much like in C where the compiler keeps track of the types of the local variables during type checking. The difference is that the namespace of ImpR consists of two entries, one argument and a local variable, rather than a user-defined amount. This assumption simplifies the formal development but it is straightforward to generalize our results. For convenience we shall be using $(\mathrm{arg} \uplus \mathrm{local}) \cong$ `Bool` when choosing between local variable and argument.

$$\frac{}{\mathsf{unit} \in \mathsf{Expr}\ \Gamma\ \mathsf{Unit}} \qquad \frac{n \in \mathbb{N}}{\mathsf{nat}\ n \in \mathsf{Expr}\ \Gamma\ \mathsf{Nat}}$$

$$\frac{f \in \mathbb{N} \to \mathbb{N} \to \mathbb{N} \qquad a, b \in \mathsf{Expr}\ \Gamma\ \mathsf{Nat}}{\mathsf{bOp}\ f\ a\ b \in \mathsf{Expr}\ \Gamma\ \mathsf{Nat}}$$

$$\frac{vr \in \mathrm{arg} \uplus \mathrm{local} \qquad t = \mathsf{TypeOf}\ \Gamma\ vr}{\mathsf{var}\ vr \in \mathsf{Expr}\ \Gamma\ t}$$

$$\frac{vr \in \mathrm{arg} \uplus \mathrm{local} \qquad t = \mathsf{TypeOf}\ \Gamma\ vr}{\&vr \in \mathsf{Expr}\ \Gamma\ (\mathsf{Ref}\ t)}$$

$$\frac{expr \in \mathsf{Expr}\ \Gamma\ (\mathsf{Ref}\ t)}{*expr \in \mathsf{Expr}\ \Gamma\ t}$$

Fig. 5: Expressions in ImpR. All rules apply to any $\Gamma \in \mathsf{Ns}$.

The set of expressions Expr in ImpR is listed in Figure 5. It is parameterized over a local typing environment $\Gamma \in \mathsf{Ns}$ and indexed by a type Ty. Expressions in ImpR are intrinsically typed [17] so they are described via inference rules. The conclusion of the rules can be interpreted as term constructors, for instance the rule for $\&$ means that if vr is either arg or local then $\&vr$ is a pointer expression of the appropriate type. Other terms are unit expressions, natural numbers, arithmetic operations, variables (var) and dereferencing ($*$). It is worth noting that the underlying type system is exactly analogous to the scoping/typing rules of C[1]: expressions can only refer to in-scope variables and dereferencing or arithmetic operations require arguments that make sense.

Commands in ImpR, shown in Figure 6, denote actions with side effects. Like Expr they are parameterized over a local typing environment $\Gamma \in \mathsf{Ns}$ and indexed by Ty. There is however an extra parameter in $X \in \mathsf{Exports}$, an alias for $\mathsf{List}\ \mathsf{Decl}$, which is defined as follows:

$$\mathsf{Decl} := \mathsf{Ty} \times \mathsf{Ty}$$
$$\mathsf{a}, \mathsf{r} \in \mathsf{Ns} \to \mathsf{Ty}$$

[1]Not considering type-casting.

$$\frac{rv \in \mathsf{Expr}\ \Gamma\ t}{\mathsf{return}\ rv \in \mathsf{Cmd}\ X\ \Gamma\ t}$$

$$\frac{loc \in \mathsf{Expr}\ \Gamma\ (\mathsf{Ref}\ t) \qquad v \in \mathsf{Expr}\ \Gamma\ t}{loc := v \in \mathsf{Cmd}\ X\ \Gamma\ \mathsf{Unit}}$$

$$\frac{c \in \mathsf{Cmd}\ X\ \Gamma\ t_1 \qquad d \in \mathsf{Cmd}\ X\ \Gamma\ t_2}{c\ \mathbf{;}\ d \in \mathsf{Cmd}\ X\ \Gamma\ t_2}$$

$$\frac{\begin{array}{c} cond \in \mathsf{Expr}\ \Gamma\ \mathsf{Nat} \\ c \in \mathsf{Cmd}\ X\ \Gamma\ t \qquad d \in \mathsf{Cmd}\ X\ \Gamma\ t \end{array}}{\mathsf{if}\ cond\ \mathsf{then}\ c\ \mathsf{else}\ d \in \mathsf{Cmd}\ X\ \Gamma\ t}$$

$$\frac{decl \in \mathsf{Decl} \qquad decl \rhd_i X \qquad a \in \mathsf{Expr}\ \Gamma\ (\mathsf{a}\ decl)}{\mathsf{call}\ i\ a \in \mathsf{Cmd}\ X\ \Gamma\ \mathsf{Unit}}$$

$$\frac{\begin{array}{cc} decl \in \mathsf{Decl} & decl \rhd_i X \\ a \in \mathsf{Expr}\ \Gamma\ (\mathsf{a}\ decl) & r \in \mathsf{Expr}\ \Gamma\ (Ref(\mathsf{r}\ decl)) \end{array}}{\mathsf{c}\!\leftarrow\ i\ a\ r \in \mathsf{Cmd}\ X\ \Gamma\ \mathsf{Unit}}$$

Fig. 6: Commands in ImpR. The first four rules apply for all $X \in$ Exports and $\Gamma \in$ Ns.

The intuition here is that Decl is a function declaration in the sense that it specifies an interface; a function exposes the type of its argument a and its return value r. Consequently, Exports is a list of callable functions. Going back to the Cmd datatype, the parameter $X \in$ Exports is only relevant to the constructors c← and call, which denote function invocations (with or without return values respectively). More precisely, calling a function requires picking a declaration from X and respecting its types. We then have the assignment command :=, the sequencing operator **;**, the conditional if_then_else_ and the return command.

Moving on we have definitions relevant to the notions of programs and contexts, starting with the function definition FunDef, which is essentially a declaration along with a body.

$$\begin{aligned}&\mathsf{FunDef}\ (X \in \mathsf{Exports}) := \\ &(t \in \mathsf{Ty}) \times (n \in \mathsf{Ns}) \times (\mathsf{Cmd}\ X\ n\ t)\end{aligned}$$

A program Prog X is an alias of DefList X X with the latter defined as follows:

$$\begin{aligned}&\mathsf{DefList}\ (X \in \mathsf{Exports})\ (Y \in \mathsf{Exports}) := \\ &[\forall\ d\ \rhd_i\ Y.\ \exists!\ fd \in \mathsf{FunDef}\ X.\ fd\ |\mathsf{i}|\ d]\end{aligned}$$

The judgment $fd\ |\mathsf{i}|\ d$ where $fd \in \mathsf{FunDef} X$ and $d \in$ Decl means that a function definition fd satisfies interface d by matching the type of the argument and the return value. A DefList X Y is a list of functions definitions interfacing

```c
void aFun(int **r) {
  int k = 5;
  *r = &k;
}
```

Fig. 7: A C function with an unsafe assignment. After the function returns, any dereference of the argument pointer results to undefined behavior.

with $X \in$ Exports implementing $Y \in$ Exports. A program Prog is a DefList that interfaces with and implements the same $X \in$ Exports so that every function call corresponds to an implemented function.

This is the last case of an intrinsic property that greatly simplifies reasoning. Crucially, most such properties are no more than what a real-world block-structured programming language with local references like C statically enforces. Recall for example that the syntax of C allows unsafe assignments like the one found in Figure 7. It is the run-time semantics that flag this as a *potential* [2] source of undefined behavior [7]. Should ImpR intrinsically disallowed such terms then there would be no issue and consequently no argument.

Finally we define program contexts, Ctx d X, as incomplete programs that require an additional FunDef X satisfying interface d in order to be complete (definition omitted). We write ctx | def] $impl$ to denote that $def \in$ FunDef completes $ctx \in$ Ctx by satisfying specification $impl$.

III. Ideal semantics

In this section we formalize the *ideal* semantics for our language in the form of dependently typed operational semantics. Their function is not to describe a potential implementation but to act as a reference point for the capability semantics described in Section IV and Section V. They are ideal in the sense that they describe how a hypothetical, idealized machine would evaluate ImpR according to its specification and, as they are devoid of the problems that mar the capability semantics from Section IV, work well as a benchmark. In other words, these semantics are to ImpR what a machine that respects the C standard [7] would be to C.

The first step towards specifying the ideal semantics is to define the set of values.

$$\mathsf{Val}\ (L \in \mathsf{StoreTy}) := \mathsf{unit} \mid \mathsf{nat}\ \mathbb{N} \mid \mathsf{loc}\ (\mathsf{Loc}\ L) \mid \mathsf{undef}$$

Val is parameterized over a list of typing namespaces $L \in$ StoreTy where StoreTy is an alias for List Ns. L essentially indicates the *shape* of the run-time store, Store, which will be defined later. This is because in ImpR, which lacks dynamic allocation, the store only increases and decreases per function transition as a stack frame is allocated and freed accordingly. In other words, L *shadows*

[2]This is a fine point in the C semantics which we slightly diverge from as explained in Section III.

$$\text{unit}\ \frac{}{S \models \mathsf{unit} \Rightarrow \mathsf{unit}} \qquad \mathbb{N}\ \frac{n \in \mathbb{N}}{S \models \mathsf{nat}\ n \Rightarrow \mathsf{nat}\ n}$$

$$\text{binOp}\ \frac{f \in \mathbb{N} \to \mathbb{N} \to \mathbb{N} \qquad n, m \in \mathbb{N} \\ S \models a \Rightarrow \mathsf{nat}\ n \qquad S \models b \Rightarrow \mathsf{nat}\ m}{S \models \mathsf{bOp}\ f\ a\ b \Rightarrow \mathsf{nat}\ (f\ n\ m)}$$

$$\text{var}\ \frac{vr \in \text{arg} \uplus \text{local}}{S \models \mathsf{var}\ vr \Rightarrow \mathsf{get}\ S\ (\mathsf{length}\ L), vr}$$

$$\text{addrOf}\ \frac{vr \in \text{arg} \uplus \text{local}}{S \models \&vr \Rightarrow \mathsf{loc}\ (\mathsf{length}\ L), vr}$$

$$\text{deref}\ \frac{poi \in \mathsf{Loc}\ (\Gamma :: L) \qquad S \models e \Rightarrow \mathsf{loc}\ poi}{S \models {*e} \Rightarrow \mathsf{get}\ S\ poi}$$

Fig. 8: Evaluation of expressions in the ideal semantics. Assume $S \in \mathsf{Store}\ (\Gamma :: L)$.

the run-time behavior of a program: each time a function with namespace Γ is called L will be extended to $\Gamma :: L$ and a stack frame will be automatically allocated.

There are the obvious value constructors unit and nat while undef denotes an uninitialized Store entry. Pointers are represented by the type Loc L.

$$\mathsf{Loc}\ (L \in \mathsf{StoreTy}) := (\exists b \in \mathbb{N}.\ b < \mathsf{length}\ L) \times (\text{arg} \uplus \text{local})$$
$$\mathsf{base} \in \mathsf{Loc}\ L \to \mathbb{N}, \ \ \mathsf{off} \in \mathsf{Loc}\ L \to (\text{arg} \uplus \text{local})$$

An instance of Loc L is a **valid** runtime pointer, meaning a pointer that falls within the bounds of a run-time store with shape L. Validity is guaranteed by its dependent nature: The base of the pointer is smaller than the length of parameter L. We are now able to define Store, where Pair L is pair of values Val L. A Store is a slightly weird cons list indexed by a StoreTy:

$$\mathsf{Store}\ [] := []$$
$$\mathsf{Store}\ (\Gamma :: L) := \mathsf{Pair}\ (\Gamma :: L) \times \mathsf{Store}\ L$$

The definition of Store is key to the safety and correctness of the semantics. A Store indexed by an empty StoreTy is empty itself. Extending a Store L takes a Pair [3] of values Val $(\Gamma :: L)$ (argument and local variable of a stack frame) to produce a Store $(\Gamma :: L)$. This construction ensures that a Store L only contains **valid** values under L.

We are now ready to introduce our ideal semantics starting with the evaluation of expressions in Figure 8. We define the relation $S \models e \Rightarrow$ v, read as "Given Store S, Expr e evaluates to Val v". Function get $S\ l$ retrieves the value at location l from store S [4]. The more interesting rules are variable access, address-off and dereferencing. A var expression evaluates to the value stored in the top of the store, an address-of expression evaluates to a *location* pointing at the top of the store and evaluation of the dereference operation depends on the operand evaluating to a valid location.

Before introducing the small-step operational semantics for commands, let us consider what it is that we aim to achieve through them, namely the safe handling of references to local variables. There is no ambiguity as to what is considered safe in C/C++ [7], [8] in that regard: assignments such as in Figure 7 are not undefined behavior on their own but lead to one only when the unsafe pointers are dereferenced after the end of the function. This is especially problematic to reason about and crucially it is hard to enforce at the low-level. At the same time common best practice principles strongly advice against such assignments regardless of the outcome.

We also believe that prohibiting unsafe assignments altogether is a step towards the right direction as it answers the above shortcomings while arguably not undermining the expressiveness of ImpR in a perceptible way. Consequently, if "safe" means to adhere to the above principle, then the ideal semantics should get stuck when evaluating commands like the one in Figure 7 and proceed normally otherwise. Furthermore they should not allow any possibilities for undefined behaviors due to invalid pointers to local variables in general.

To achieve the above notion of safety of the ideal semantics we first need to express the condition that differentiates between good and evil assignments. All assignments where the right side operand evaluates to a nat, unit or undef are trivially safe [5]. Unsafe circumstances may occur only when the right side operand evaluates to a loc value. Reflecting back to the example in Figure 7, the reference to the local variable k is being assigned to pointer r with strictly longer lifetime. Should the lifetime of r be shorter or equal to k there would be no issue. Binary predicate $\mathsf{Up} \subseteq (\mathsf{Loc}\ L) \times (\mathsf{Val}\ L)$ captures this notion (showing only the non-trivial loc case):

$$\frac{l_1, l_2 \in \mathsf{Loc}\ L \qquad \mathsf{base}\ l_1\ \geq\ \mathsf{base}\ l_2}{l_1, \mathsf{loc}\ l_2 \in \mathsf{Up}}$$

The key intuition here is that the base of a Loc fully determines the lifetime of a location. For example, a base of zero means that the object it points to will live for the duration of the program while a base of length L means that the object will be the first to be deallocated. Note that Store satisfies Up for all locations:

[3] Hence off in Loc chooses left or right.

[4] Indexing in a Store begins at the tail end of the inductive definition.

[5] The programmer may deliberately attempt to dereference or pass an undef reference but such cases are not in scope of this paper.

Lemma 1 (Monotonicity of lifetimes in a Store)

$$\forall L, S \in \mathit{Store}\ L, l \in \mathit{Loc}\ L \implies l, \mathit{get}\ S\ l \in \mathit{Up}$$

Proof. By induction on the structure of S. □

As a result, to update a Store at location l with v requires $l, v \in \mathsf{Up}$.

An execution state in the ideal semantics is an element of State, which is a triple of a Store, a list of commands and a list of optional references to store return values. The three of them together compose the activation record of a running program.

$$\begin{aligned}&\mathsf{State}\ (X \in \mathsf{Exports})\ (L \in \mathsf{StoreTy}) := \\ &(\mathsf{Store}\ L) \times (\mathsf{CmdStore}\ X\ L) \times (\mathsf{Ret}\ L)\end{aligned}$$

We additionally define projection functions stack, cmds and ret respectively.

Finally we introduce the ideal small-step operational semantics of ImpR. The ternary relation $P \models St_1 \longrightarrow St_2$ is interpreted as "Under program P, State St_1 evaluates to State St_2". The full semantics can be found in Figure 9 but the cases most relevant to the issue are the assignment and return-and-store (*ret+*). For assignments, the l-expression l has to evaluate to a pointer value p and the r-expression r has to evaluate to v so that $\mathsf{Up}\ p\ v$. The witness for Up is then supplied to update. The case for return-and-store is similar; the added complexity comes from the deallocation of the stack frame upon return. Note that return behaves as a skip command when encountered in the middle of a function.

IV. Capability semantics and their shortcomings

Following up on the ideal semantics are the state-of-the-art yet less-than-ideal capability machine semantics. They are in essence a condensed, simplified version of the semantics of a real-world capability machine like CHERI [3], focusing on the handling of local capabilities. An equivalent description would be that they are the extended capability semantics only without the locality extensions, a fact that outlines their purpose: to showcase their limitations compared to the ideal machine and prepare the ground for the extended semantics in Section V.

The operation of a capability machine revolves around the manipulation of capabilities, unforgeable pointers that grant access to memory. Typically a capability would consist of a lower and an upper bound for accessing a specific subset of the address space, but for our language we need only consider capabilities to access a two-cell of a stack. In other words, we need a base and a left-or-right choice. Note the absence of a lifetime counter, just like in CHERI [6].

$$\mathsf{Addr} := \mathbb{N} \times (\mathrm{arg} \uplus \mathrm{local})$$

$$\mathsf{base} \in \mathsf{Addr} \to \mathbb{N},\ \ \mathsf{off} \in \mathsf{Addr} \to (\mathrm{arg} \uplus \mathrm{local})$$

[6] Recall that in CHERI the only distinction is between local and global capabilities.

$$\text{skip}\ \frac{}{P \models S, (\mathsf{return}\ e\ ;\ c) :: C, R \longrightarrow S, c :: C, R}$$

$$\text{seq}\ \frac{P \models S, c_1 :: C, R \longrightarrow S', c_2 :: C, R}{P \models S, (c_1\ ;\ c) :: C, R \longrightarrow S', (c_2\ ;\ c) :: C, R}$$

$$\text{cZ}\ \frac{\mathit{cmd} = \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \qquad S \models e \Rightarrow \mathsf{nat}\ 0}{P \models S, \mathit{cmd} :: C, R \longrightarrow S, c_2 :: C, R}$$

$$\text{cN}\ \frac{\begin{array}{c}\mathit{cmd} = \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \\ n \in \mathbb{N} \qquad n \neq 0 \qquad S \models e \Rightarrow \mathsf{nat}\ n\end{array}}{P \models S, \mathit{cmd} :: C, R \longrightarrow S, c_1 :: C, R}$$

$$\text{assign}\ \frac{\begin{array}{c}S \models l \Rightarrow \mathsf{loc}\ p \qquad S \models r \Rightarrow v \qquad c = (l := r) \\ \mathit{prw} \in \mathsf{Up}\ p\ v \qquad S' = \mathsf{update}\ S\ p\ v\ \mathit{prw}\end{array}}{P \models S, c :: C, R \longrightarrow S', (\mathsf{return}\ \mathsf{unit}) :: C, R}$$

$$\text{ret}\ \frac{}{P \models s :: S, (\mathsf{return}\ e) :: C, \textsf{-} :: R \longrightarrow S, C, R}$$

$$\text{ret+}\ \frac{\begin{array}{c}s :: S \models e \Rightarrow v \qquad \mathit{prw} \in \mathsf{Up}\ p\ v \\ S' = \mathsf{update}\ S\ p\ v\ \mathit{prw}\end{array}}{P \models s :: S, (\mathsf{return}\ e) :: C, p :: R \longrightarrow S', C, R}$$

$$\text{call}\ \frac{\begin{array}{c}\mathit{cmd} = \mathsf{body}\ (\mathsf{getDef}\ P\ \mathit{decl}) \qquad S \models \mathit{arg} \Rightarrow v \\ S' = (v, \mathsf{undef}) :: S \\ C' = \mathit{cmd} :: c :: C \qquad\qquad R' = \textsf{-} :: R\end{array}}{P \models S, (c \leftarrow \mathit{decl}\ a\ r\ ;\ c) :: C, R \longrightarrow S', C', R'}$$

$$\text{call+}\ \frac{\begin{array}{c}\mathit{cmd} = \mathsf{body}\ (\mathsf{getDef}\ P\ \mathit{decl}) \qquad S \models \mathit{arg} \Rightarrow v \\ S \models r \Rightarrow \mathsf{loc}\ p \qquad S' = (v, \mathsf{undef}) :: S \\ C' = \mathit{cmd} :: c :: C \qquad\qquad R' = p :: R\end{array}}{P \models S, (c \leftarrow \mathit{decl}\ a\ r\ ;\ c) :: C, R \longrightarrow S', C', R'}$$

Fig. 9: Evaluation of commands in the ideal semantics. Assume $X \in \mathsf{Exports}$, $P \in \mathsf{Prog}\ X$ and S, C, R as a Store, $\mathsf{CmdStore}$, Ret parameterized by a suitable $L \in \mathsf{StoreTy}$.

Our semantics distinguish between plain values and capabilities, which is considered standard in capability machines. The two additional constructors, unit and undef, are there for convenience.

$$\mathsf{Cval} := \mathsf{unit} \mid \mathsf{nat}\ \mathbb{N} \mid \mathsf{cap}\ \mathsf{Addr} \mid \mathsf{undef}$$

Instead of a Store, we define $\mathsf{Memory}\ L$ as a list of Cval of size $\mathsf{length}\ L$, where $L \in \mathsf{StoreTy}$ works in a similar fashion to the ideal semantics introduced in Section III, meaning that it keeps track of the size of the execution call stack.

$$\text{unit}\ \frac{}{M \Vdash \mathsf{unit} \Rightarrow \mathsf{unit}} \qquad \mathbb{N}\ \frac{n \in \mathbb{N}}{M \Vdash \mathsf{nat}\ n \Rightarrow \mathsf{nat}\ n}$$

$$\text{binOp}\ \frac{\begin{array}{cc} f \in \mathbb{N} \to \mathbb{N} \to \mathbb{N} & n, m \in \mathbb{N} \\ M \Vdash a \Rightarrow \mathsf{nat}\ n & M \Vdash b \Rightarrow \mathsf{nat}\ m \end{array}}{M \Vdash \mathsf{bOp}\ f\ a\ b \Rightarrow \mathsf{nat}\ (f\ n\ m)}$$

$$\text{var}\ \frac{\begin{array}{c} w \in \mathsf{length}\ L < \mathsf{length}\ (\Gamma :: L) \\ vr \in \mathrm{arg} \uplus \mathrm{local} \end{array}}{M \Vdash \mathsf{var}\ vr \Rightarrow \mathsf{get}\ M\ ((\mathsf{length}\ L), vr)\ w}$$

$$\text{addrOf}\ \frac{vr \in \mathrm{arg} \uplus \mathrm{local}}{M \Vdash \&vr \Rightarrow \mathsf{cap}\ (\mathsf{length}\ L), vr}$$

$$\text{deref}\ \frac{\begin{array}{c} w \in \mathsf{base}\ poi < \mathsf{length}\ (\Gamma :: L) \\ M \Vdash e \Rightarrow \mathsf{loc}\ poi \end{array}}{M \Vdash {*}e \Rightarrow \mathsf{get}\ M\ poi\ w}$$

Fig. 10: Evaluation of expressions in the capability semantics. Assume $M \in \mathsf{Memory}\ (\Gamma :: L)$.

$$\text{skip}\ \frac{}{P \Vdash M, (\mathsf{return}\ e\ ;\ c) :: C, R \longrightarrow M, c :: C, R}$$

$$\text{seq}\ \frac{P \Vdash M, c_1 :: C, R \longrightarrow M, c_2 :: C, R}{P \Vdash M, (c_1\ ;\ c) :: C, R \longrightarrow M, (c_2\ ;\ c) :: C, R}$$

$$\text{cZ}\ \frac{cmd = \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \qquad M \Vdash e \Rightarrow \mathsf{nat}\ 0}{P \Vdash M, cmd :: C, R \longrightarrow M, c_2 :: C, R}$$

$$\text{cN}\ \frac{\begin{array}{c} cmd = \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \\ n \in \mathbb{N} \qquad n \neq 0 \qquad M \Vdash e \Rightarrow \mathsf{nat}\ n \end{array}}{P \Vdash M, cmd :: C, R \longrightarrow M, c_1 :: C, R}$$

$$\text{assign}\ \frac{\begin{array}{cc} M \Vdash l \Rightarrow \mathsf{cap}\ p & M \Vdash r \Rightarrow v \\ c = (l := r) & w \in \mathsf{base}\ p < \mathsf{length}\ (\Gamma :: L) \\ \multicolumn{2}{c}{M' = \mathsf{update}\ M\ p\ v\ w} \end{array}}{P \Vdash M, c :: C, R \longrightarrow M', (\mathsf{return}\ \mathsf{unit}) :: C, R}$$

$$\text{ret}\ \frac{}{P \Vdash m :: M, (\mathsf{return}\ e) :: C, \text{-} :: R \longrightarrow M, C, R}$$

$$\text{ret+}\ \frac{\begin{array}{cc} m :: M \Vdash e \Rightarrow v & w \in \mathsf{base}\ p < \mathsf{length}\ (\Gamma :: L) \\ v \in \mathsf{!Cap} & M' = \mathsf{update}\ M\ p\ v\ w \end{array}}{P \Vdash m :: M, (\mathsf{return}\ e) :: C, p :: R \longrightarrow M', C, R}$$

$$\text{call}\ \frac{\begin{array}{cc} cmd = \mathsf{body}\ (\mathsf{getDef}\ P\ decl) & M \Vdash arg \Rightarrow v \\ v \in \mathsf{!Cap} & M' = (v, \mathsf{undef}) :: M \\ C' = cmd :: c :: C & R' = \text{-} :: R \end{array}}{P \Vdash M, (\mathsf{c}{\leftarrow}\ decl\ a\ r\ ;\ c) :: C, R \longrightarrow M', C', R'}$$

$$\text{call+}\ \frac{\begin{array}{ccc} \multicolumn{2}{c}{cmd = \mathsf{body}\ (\mathsf{getDef}\ P\ decl)} & M \Vdash arg \Rightarrow v \\ \multicolumn{2}{c}{M \Vdash r \Rightarrow \mathsf{loc}\ p} & M' = (v, \mathsf{undef}) :: M \\ v \in \mathsf{!Cap} & C' = cmd :: c :: C & R' = p :: R \end{array}}{P \Vdash M, (\mathsf{c}{\leftarrow}\ decl\ a\ r\ ;\ c) :: C, R \longrightarrow M', C', R'}$$

Fig. 11: Evaluation of commands in the capability semantics. Assume $X \in \mathsf{Exports}$, $P \in \mathsf{Prog}\ X$ and M, C, R as a Memory, CmdStore, Cret parameterized by a suitable $L \in \mathsf{StoreTy}$.

The contrast between the dependently typed Store and the non-dependent Memory illustrates the conflicting design goals between an ideal machine versus a more realistic capability machine. This perspective is further elaborated by the expression evaluation relation in Figure 10.

Evaluation of unit, nat and bOp are largely similar to the ideal semantics. The address-of case outlines the unforgeability of capabilities: a sandbox/function only has access to two capabilities; everything else must be propagated. The get function used in the case of a var or * expression is different than the one in the ideal semantics in that it requires the address given to be within the bounds of $M \in \mathsf{Memory}\ L$ or, consequently, less than length L. This is simply a common bounds check performed by the processor during memory access. Note that the requirement is satisfied automatically for var.

$$\begin{aligned} &\mathsf{CapState}\ (X \in \mathsf{Exports})\ (L \in \mathsf{StoreTy}) := \\ &(\mathsf{Memory}\ L) \times (\mathsf{CmdStore}\ X\ L) \times (\mathsf{Cret}\ L) \end{aligned}$$

The notion of state in the capability semantics, CapState, is very similar to the one in the ideal semantics: a triple of a run-time store, a stack of return addresses and a list of optional references for the return values. Once again, we define projection functions mem, cmd and ret.

The small-step operational semantics of the capability machine are presented in Figure 11. Similarly to the ideal semantics, we introduce the ternary relation $P \Vdash Ct_1 \longrightarrow Ct_2$ which is read as "Under program P, CapState Ct_1 evaluates to CapState Ct_2".

The interesting cases are function invocation and the assignment command. With regards to the latter, preconditions $M \Vdash l \Rightarrow \mathsf{cap}\ a$ and $M \Vdash r \Rightarrow v$ ensure that the left and right side expressions evaluate while the witness for $\mathsf{base}\ p < \mathsf{length}\ (\Gamma :: L)$ ensures that the address is in bounds. Unlike the ideal semantics, the lack of any lifetime information means there can be no additional constraints on the values. This is bound to be problematic but, lacking the required primitives, it is dealt with an artifice in lieu of a proper solution which is evident in the function invocation rules, *call* and *call+*.

Almost all of the preconditions when invoking functions are similar between the two semantics except one predicate on Cval found in both rules: !Cap, which disallows passing capabilities altogether. This restriction mirrors the CHERI processor, where passing local capabilities during cross-domain function calls is prohibited by default.

At this point it is important to briefly mention the automatic memory management present in the capability semantics. Looking back at the *call* case, the Memory in the new CapState is $(v\ \text{-}\ \mathsf{undef})\quad M$. The semantics automatically allocate a two-cell stack frame for the callee, update the instruction pointer and set the return value location accordingly. The reverse takes place during function return. In other words the capability semantics include a built-in notion of the stack instead of using registers explicitly.

We believe that this simplification is sensible in that it hides low-level operations that are not directly involved in the solution nor are part of the problem. The underlying assumptions are also straightforward: the individual stack frames must be disjoint and properly initialized at each entry point. Disjointedness of the stack region for each sandbox is guaranteed by default in CHERI but there is no automatic cleaning of the stack at exit or entry. We discuss this and other implementation questions in Section VI.

A. The problems

At this point we have already hinted at the problematic cases but we have yet to pinpoint the precise points of failure. In this section we proceed to formally verify that the capability semantics cannot simulate the ideal semantics presented in Section III. Doing so provides meaningful insight as to why exactly the capability semantics fail and paves the way for the solution in Section V. For the purposes of the proof we first define a relation $St \sim Ct$ where St : State and Ct : CapState and then show that it is not a bisimulation:

Theorem 2 (Falsity of forwards simulation)

$$\neg(\forall P, St_1, St_2, Ct_1.\ St_1 \sim Ct_1 \times P \models St_1 \longrightarrow St_2 \\ \rightarrow \exists Ct_2.\ St_2 \sim Ct_2 \times P \Vdash Ct_1 \longrightarrow Ct_2)$$

Theorem 3 (Falsity of backwards simulation)

$$\neg(\forall P, Ct_1, Ct_2, St_1.\ St_1 \sim Ct_1 \times P \Vdash Ct_1 \longrightarrow Ct_2 \\ \rightarrow \exists St_2.\ St_2 \sim Ct_2 \times P \models Ct_1 \longrightarrow Ct_2)$$

Relation $St \sim Ct$ is meant to describe similarity between the sets of states, State and CapState so naturally the relation is defined component-wise:

$$S, C, R \sim M, C, R' \Longleftrightarrow (S \mid\!\rightsquigarrow\!\mid M) \times (R \rightleftharpoons R')$$

$St \sim Ct$ iff their stores are related, their stack of result locations are also related and they have the same stack of return addresses. The store relation $S \mid\!\rightsquigarrow\!\mid M$ and result location relation $R \rightleftharpoons R'$ are structural "lifts" on the value relation $\rightsquigarrow$.

$$\forall\ L \in \mathsf{StoreTy}, v \in \mathsf{Val}\ L, cv \in \mathsf{Cval}.\ v \rightsquigarrow cv := \\ \mathsf{unit} \rightsquigarrow \mathsf{unit} \mid \mathsf{nat}\ n \rightsquigarrow \mathsf{nat}\ n \mid \mathsf{undef} \rightsquigarrow \mathsf{undef} \mid \\ \mathsf{loc}\ base, off \rightsquigarrow \mathsf{cap}\ base, off$$

The value relation is straightforward. Non-pointer values are related to their non-capability counterparts while pointers are related to capabilities that give access to the same cell at the same address. Note that the dependently typed Loc ensures that $base < \mathsf{length}\ L$.

Proving Theorem 2 and Theorem 3 is a matter of constructing two counterexample programs along with execution states for the semantics. The latter provides little insight so we will focus on the former and in particular on the functions where the erroneous situations occur. The first counterexample is fully reproducible in CHERI while the second one requires lifting of the capability restriction when invoking sandboxes.

```
Unit bFun(Ref Unit);

Unit aFun(Unit arg){
  Unit local;

  bFun(&local) ; return unit
}
```

Fig. 12: Passing a pointer as argument in ImpR.

```
Unit aFun(Ref (Ref Unit) arg){
  Unit local;

  var arg := &local ; return unit
}
```

Fig. 13: An unsafe assignment in ImpR.

The counterexamples can be found in Figure 12 and Figure 13, where we used slightly relaxed syntax for readability. For Theorem 2, all it takes is to invoke a function which accepts a pointer as an argument. In this instance the ideal semantics evaluate normally but the capability semantics get stuck. For Theorem 3, we need to evaluate a dangerous assignment which the capability semantics can not protect from. The underlying CapState is fabricated: it cannot occur under normal execution due to the !Cap restriction on arguments. In other words, without the !Cap restriction, the capability semantics are not safe yet not sufficiently permissive with it.

V. Extended capability semantics

It is now clear that our capability semantics from Section IV are unable to manage objects of varying lifetimes in a hierarchy of stack frames, resulting in cases where a semantically safe program is unable to evaluate properly.

The problem can be traced down to the definition of a capability.

$$\mathsf{Cval} := \mathsf{unit} \mid \mathsf{nat}\ \mathbb{N} \mid \mathsf{cap}\ \mathsf{Addr} \mid \mathsf{undef}$$

With a capability value consisting only of an address, the semantics are essentially agnostic with regards to object lifetimes. It is sensible to expect that a solution should augment capabilities with extra information that represent the lifetimes of each object. Moreover, we expect the semantics to be able to compare these lifetimes and evaluate accordingly. For that reason we decided to add an extra lifetime counter to the cap constructor.

$$\mathsf{Cval} := \mathsf{unit} \mid \mathsf{nat}\ \mathbb{N} \mid \mathsf{cap}\ \mathsf{Addr}\ \mathbb{N} \mid \mathsf{undef}$$

The expression evaluation relation is also slightly altered, namely in the case of the & operator.

$$\text{addrOf}\ \frac{vr \in \mathrm{arg} \uplus \mathrm{local} \qquad len = \mathsf{length}\ L}{M \Vdash \&vr \Rightarrow \mathsf{cap}\ len, vr, len}$$

This rule outlines the manner which the semantics utilize the new bits of information. Upon creation, a capability is assigned a lifetime. The semantics are aware of the status of the allocations by the shape of the M : Memory involved. So at any moment a capability being created always points at the current, topmost stack frame and it should have the shortest life among all live objects. There is more than one way to achieve this but we decided to use the reverse ordering, so that the most long-lived capability has a counter of zero, while the length of Memory is attributed to the most ephemeral. Due to the simplicity of the addressing mode addresses and lifetimes coincide, a property that will play a crucial role in proving correctness.

Having established a comparison scheme between lifetimes, we may now define the safety notion for assignments in the extended semantics, the predicate $\mathsf{Safe} \subseteq \mathbb{N} \times \mathsf{Cval}$ (omitting trivially safe unit, nat and undef cases).

$$\text{safeCap}\ \frac{\begin{array}{cc} vr \in \mathrm{arg} \uplus \mathrm{local} & n \le n' \\ b \in \mathbb{N} & c = \mathsf{cap}\ b, vr, n \end{array}}{n', c \in \mathsf{Safe}}$$

This definition is largely analogous to Up from the ideal semantics. The conclusion essentially states that any capability with a lifetime of n' will outlast c. Moving on, we update the operational rule for assignments to evaluate only when the operands *left* and *right* are Safe and finally lift the !Cap constraint for function calls, thus concluding the extended capability semantics.

$$\text{assign}\ \frac{\begin{array}{cc} M \Vdash l \Rightarrow \mathsf{cap}\ base, off, n & M \Vdash r \Rightarrow v \\ c = (l := r) & w \in base < \mathsf{length}\ (\Gamma :: L) \\ n, v \in \mathsf{Safe} & M' = \mathsf{update}\ M\ p\ v\ w \end{array}}{P \Vdash M, c :: C, R \longrightarrow M', (\mathsf{return\ unit}) :: C, R}$$

$$\text{call}\ \frac{\begin{array}{cc} cmd = \mathsf{body}\ (\mathsf{getDef}\ P\ decl) & M \Vdash arg \Rightarrow v \\ v \in \mathsf{!Cap} & M' = (v, \mathsf{undef}) :: M \\ C' = cmd :: c :: C & R' = \text{-} :: R \end{array}}{P \Vdash M, (c \leftarrow decl\ a\ r\ ;\ c) :: C, R \longrightarrow M', C', R'}$$

A. Full abstraction

We were confident that our extensions from Section V adequately captured the essence of stack objects and their interactions and that they worked as intended. The next challenging step was to formally prove our intuitions. In this section we shall focus on the important lemmas and key insights of the proof beginning with the value relation $\rightsquigarrow$, a definition that markedly diverges from the one in Section IV-A.

Definition 1 (Value relation, extended semantics)

$$\forall\ L \in \mathit{StoreTy}, v \in \mathit{Val}\ L, cv \in \mathit{Cval}.\ v \rightsquigarrow cv :=$$
$$\mathit{unit} \rightsquigarrow \mathit{unit} \mid \mathit{nat}\ n \rightsquigarrow \mathit{nat}\ n \mid \mathit{undef} \rightsquigarrow \mathit{undef} \mid$$
$$\mathit{loc}\ base, off \rightsquigarrow \mathit{cap}\ base, off, base$$

The difference is unsurprisingly in the capability case and specifically on the lifetime counter. In the ideal semantics there is no explicit lifetime counter; the lifetime always coincides with the base of a Loc. This is not generally true for our extended capability machine as the base of an Addr is not connected to the lifetime counter in any way. What this relation establishes is a *correct*, *sane configuration* for the capability machine. It is the simplest possible configuration and one that is already respected by the expression evaluation rules in the extended semantics as evident by the addrOf rule: the base being equal to the lifetime counter. We encode this property as predicate $\mathsf{Sane} \subseteq \mathsf{Cval}$, which is free for related values.

Lemma 4 (Related values are sane)

$$\forall v', v.\ v' \rightsquigarrow v \rightarrow v \in \mathit{Sane}$$

Proof. Trivially by Definition 1. □

The value relation gives rise to a crucial lemma regarding evaluating expressions under the extended capability semantics when the Memory involved is related to a Store.

Lemma 5 (Sanity of expression evaluation when $S \mid\rightsquigarrow\mid M$)

$$\forall S, M, e, v.\ S \mid\rightsquigarrow\mid M \times M \Vdash e \Rightarrow v \rightarrow v \in \mathit{Sane}$$

Proof. By case-splitting on the evaluation relation combined with Lemma 4 expanded on M. □

So evaluating any expression in the extended capability semantics when the underlying Memory is related to a Store results to a value v such as $v \in$ Sane. In a way property $S \mathrel{|\rightsquigarrow|} M$ acts like a "safety net" that enforces the sanity of the values found in M which eventually leads to the sanity of the result.

Value sanity is not the only desirable property that the ideal semantics satisfy for free. Recall that a Store contains valid values by definition: a pointer value (poi) saved in a Store always points to a cell in the Store. We introduce relation Valid $\subseteq$ Memory $\times$ Cval to denote that a $c \in$ Cval is either a literal value or an in-bounds address (omitting trivial cases).

$$\text{validCap}\frac{vr \in \text{arg} \uplus \text{local} \qquad b \mathrel{|\in|} M \qquad n \in \mathbb{N} \qquad v = \mathsf{cap}\ b, vr, n}{M, v \in \mathsf{Valid}}$$

Validity is also automatic for related values.

Lemma 6 (Related values are valid)

$$\forall L, v' \in \mathit{Val}\ L, M \in \mathit{Memory}\ L, v.\ v' \rightsquigarrow v \rightarrow M, v \in \mathit{Valid}$$

Proof. Trivially by Definition 1. ☐

We may now prove the validity lemma analogous to Lemma 5.

Lemma 7 (Validity of evaluation when $S \mathrel{|\rightsquigarrow|} M$)

$$\forall S, M, e, v.\ S \mathrel{|\rightsquigarrow|} M \times M \Vdash e \Rightarrow v \rightarrow M, v \in \mathit{Valid}$$

Proof. Similar to Lemma 5. By case-splitting the evaluation relation combined with Lemma 6 expanded on M. ☐

Again, it is the extra $S \mathrel{|\rightsquigarrow|} M$ that guarantees validity for the evaluation of expressions. Sanity and validity combine with great potency:

Lemma 8 (Inversion of Cval)

$$\forall L, M \in \mathit{Memory}\ L, v \in \mathit{Sane}, (M, v) \in \mathit{Valid}.$$
$$\exists v' \in \mathit{Val}\ L.\ v' \rightsquigarrow v$$

Proof. By Definition 1. ☐

Lemma 8 shows that that sanity and validity are the bridging point between the ideal semantics and the extended capability semantics. The abstract jump from the more sophisticated, dependently-typed machine to the real-world processor via $v \rightsquigarrow c$ preserves the two properties while the other direction requires them. In fact, it can be shown that the set of values $v \in$ Val L is precisely the set of $c \in$ Cval with $c \in$ Sane and $M, c \in$ Valid for some $M \in$ Memory L, albeit not necessarily for the subsequent proofs.

The above lemmas outline the true conceptual relation between the ideal semantics and our extended capability machine. They are also the major components behind the proof of correctness, its crux. As an intermediate step, we need a bisimulation-like result for the evaluation of expressions.

Lemma 9 (Relation $|\rightsquigarrow|$ is bisimulation-like)

$$\forall S, M, v, e.\ S \mathrel{|\rightsquigarrow|} M \times S \models e \Rightarrow v \rightarrow \exists v'.\ M \Vdash e \Rightarrow v'$$

$$\forall S, M, v, e.\ S \mathrel{|\rightsquigarrow|} M \times M \Vdash e \Rightarrow v \rightarrow \exists v'.\ S \models e \Rightarrow v'$$

Proof. By induction on the structure of the expression evaluation relation on the ideal semantics and the extended capability semantics respectively. ☐

Our notion of correctness is similar to the one used in Section IV-A and is synonymous to $\sim$ being a bisimulation.

Theorem 10 (Relation $\sim$ is a bisimulation)

$$\forall P, St_1, St_2, Ct_1.\ St_1 \sim Ct_1 \times P \models St_1 \longrightarrow St_2$$
$$\rightarrow \exists Ct_2.\ St_2 \sim Ct_2 \times P \Vdash Ct_1 \longrightarrow Ct_2$$

$$\forall P, Ct_1, Ct_2, St_1.\ St_1 \sim Ct_1 \times P \Vdash Ct_1 \longrightarrow Ct_2$$
$$\rightarrow \exists St_2.\ St_2 \sim Ct_2 \times P \models Ct_1 \longrightarrow Ct_2$$

Proof. By induction on the structure of the respective command evaluation relation. ☐

The proofs themselves are not entirely straightforward and utilize a number of additional lemmata that are of little intellectual interest. For that reason as well as the size and complexity of the proof terms we omit them from the paper but offer them online for the intrepid reader [7].

Full abstraction of the identity compiler is a straightforward corollary from Theorem 10. We start with the notion of contextual equivalence, which is a standard Morris-style [18] definition:

Definition 2 (Contextual equivalence)

$$\forall d, X, f_1, f_2 \in \mathit{FunDef}\ (d :: X), f_1 \mathrel{|i|} d, f_2 \mathrel{|i|} d.$$
$$f_1 \equiv^{i/c}_{ctx} f_2 \iff$$
$$(\forall ctx.\ (ctx \mid f_1 \mid d) \Downarrow_{i/c} \iff (ctx \mid f_2 \mid d) \Downarrow_{i/c})$$

Where $P \Downarrow_{i/c}$ denotes that $P \in$ Prog terminates.

Definition 3 (Termination in the ideal semantics)

$$P \Downarrow_i \iff P \models (\mathit{undef}, \mathit{undef}) :: [], \mathit{main} :: [], \mathit{nothing} :: []$$
$$\longrightarrow * [], [], []$$

And similarly for the capability semantics. Relation $P \models St_1 \longrightarrow * St_2$ is the reflexive, transitive closure of the command evaluation relation and main is by convention the body of the second function definition in P. Intuitively, termination is equivalent to evaluation from an initial state to the final, empty state. Finally:

Theorem 11 (Full abstraction)

$$\forall d, X, f_1, f_2 \in \mathit{FunDef}\ (d :: X), f_1 \mathrel{|i|} d, f_2 \mathrel{|i|} d.$$
$$f_1 \equiv^{i}_{ctx} f_2 \iff f_1 \equiv^{c}_{ctx} f_2$$

Proof. By Theorem 10, because the two underlying pairs of initial and terminal states belong in $\sim$. ☐

[7] https://github.com/solidsnk/cap-extensions.git

VI. Implementation notes

Our extended capability model is not restricted to a specific capability machine but was mainly inspired by CHERI and it is thus the first platform to consider when discussing potential implementation targets. Indeed, in this section we aim to elaborate why CHERI is a solid choice as a platform. At the same time, we shall explain why the various simplifications in our capability model implies the presence of machinery, software or hardware, which largely or fully exists already in CHERI.

The CHERI ecosystem is composed by the CHERI Instruction Set Architecture [9] and its prototype implementation, the compiler suite and the CheriBSD operating system [3]. As mentioned in Section I, capabilities in CHERI are either global or local. Disallowing the passing of local capabilities between sandboxes is not a policy enforced by the ISA, but by kernel-space handlers of CheriBSD. This 1-bit information flow model is essentially a restricted form of our n-bit model: global capabilities are similar to values in the form of cap a 0 and may be propagated freely while local capabilities cannot. Furthermore, storing a local capability using a global capability is prohibited by default due to the *store local* bit set to zero (although it can be turned on with sufficient privileges). This is consistent with our extended capability model as regions referenced by global capabilities will outlive those of local capabilities.

A. Locality bits

It is thus sensible that the first step towards implementing our extensions would be to increase the locality levels, which is a point of contention as n bits would support up to 2^n levels. Modern Linux and Windows systems set the stack size to 8 and 1 megabytes respectively. Assuming an average frame size of 128 bytes leads to a maximum of 65536 and 8192 frames respectively. Stack exhaustion limits are typically very conservative and not meant to be reached under normal usage so 16 bits for 65536 levels should be more than enough. At the same time a sandbox in CHERI might comprise an entire library instead of a function, thus reducing the demand for levels. Regardless, it is unclear what the ideal least number of bits is.

Capabilities in version 6 of the CHERI Instruction Set Architecture come in two formats: 256-bit and 128-bit [9]. 256-bit capabilities offer a 20-bit field for user-defined permissions on capabilities and an extra 8 unused bits, a high enough number for a potential implementation. On the other hand, there is no such luxury for 128-bit capabilities as just 4 bits are offered for user-defined permissions and 2 unused. There are 2 additional bits in the exponent that are always set to zero and the current information flow bits, *global* and *store local*, for a potential grand total of 10 bits, a number which might not guarantee stable execution for large processes. However, a new capability compression scheme for CHERI [19] increased the number of unused bits in 128-bit capabilities from 2 to 7 which could allow for an implementation without the need to add more bits or reduce the address space.

Using these spare bits for encoding locality levels in capabilities, the CHERI hardware would enforce that capabilities cannot be stored in memory with a locality level indicating a longer lifetime than their own. In kernel mode, these restrictions would not apply, so that the trusted stack manager, which handles domain transitions, could set up the stack pointer appropriately. Note also that we do not require the equivalent of store-local permissions, essentially because all our capabilities are implicitly store-local at their locality level. We plan to further investigate potential implementations for both 128-bit and 256-bit capabilities in future work.

The extensions would also require slight changes to the microcode of the various *Store Capability Register* instructions [9] to meet the safety requirements of our model (for example the Safe predicate from Section V). As mentioned earlier, there is already a policy to prohibit unsafe interactions between local and global capabilities based on the *store local* bit so extending this to support more levels should be simple and not amount to any perceptible overhead. Additionally, lifting the local capability restriction between sandbox invocations should also be trivial as it is a matter of modifying the kernel-space handlers.

One important scenario to consider is the edge case where all of the locality bits are exhausted, one that our model does not account for since the stack depth is potentially infinite. Should the counter wrap to zero then all new capabilities would be automatically considered of maximum lifetime, which is clearly untrue. The safest solution would be to throw an exception and stop the offending process, at the expense of potentially breaking safe programs. Fixing the counter at the maximum value can be a viable choice as long as extra measures [6] are taken to ensure correct interaction.

B. Stack management

Our capability machine model assumes a built-in notion of stack where the semantics of call and return take care of allocating and deallocating stack frames. CHERI on the other hand manages the stack using a combination of registers, kernel-space routines and the user-space sandbox library **libcheri** [3]. More specifically, libcheri will allocate the stack region for each sandbox *class* while the kernel-space routines keep track of a *trusted stack*, which acts as a call stack for sandboxes. Each activation record in the trusted stack consists of data, stack and code segment capabilities that are saved from and loaded to registers when calling into and exiting from a sandbox.

The capability semantics presented in Section IV and Section V essentially simplify the above steps and embed them in the evaluation rules. Indeed, what our semantics require is partly what CHERI already provides: that the stack for each sandbox is private and disjoint except for the explicit sharing via capability arguments. The embedding, however, is not perfect for two reasons: our semantics require automatic allocation and deallocation of stack frames upon entry or exit as well as everything to be initialized to undef. In CHERI this would translate to including a simple stack allocator in the handlers which also initializes every new region accordingly. Recent work on efficient tagged memory suggests this can be done efficiently [20].

If automatic initialization is unfeasible in CHERI, we anticipate that it is still possible to achieve a weakened correctness result within our model by altering the value relation $v \rightsquigarrow cv$ (and consequently the state relation $St \sim Ct$) so that an undef value in the ideal semantics is related to *any* value in the extended capability semantics. This would not guarantee preservation or reflection of contextual equivalence, yet it would protect against rudimentary attacks.

Using our extensions as a mitigation for *stack-based use-after-free* cases in a single-sandbox environment would not directly involve the sandboxing infrastructure. Conversely, it would require slight modifications to the CHERI compiler so that each function's stack frame is correctly assigned a locality value. It would also imply the presence of a hardware flag that allows user-space code to change the locality bit of a capability.

VII. Conclusion

Capabilities are a very promising hardware mechanism to efficiently enforce spatial memory safety for C programs. But it is less obvious how to enforce temporal memory safety efficiently. This paper proposes a dynamic mechanism for ensuring that capabilities pointing to stack-allocated memory can only flow upward on the stack, towards more recent activation records that will have a shorter lifetime than the memory these capabilities are pointing to, thus enforcing temporal safety for stack-allocated memory. We provided a machine-checked proof of this property by showing full abstraction of the identity compiler between an idealized semantics that checks temporal safety and the new efficient run-time checking semantics, and we briefly discussed how our mechanism can be implemented as an extension to CHERI.

VIII. Related work

The main contribution of this paper is a generalization of the current treatment of the stack in CheriBSD, as described by Watson et al. [3], adding temporal safety for stack references. Like us, they use separate, per-component stacks, a trusted stack manager in the kernel and local capabilities, but as explained before, do not allow passing stack references between components. Another treatment of stack references on a capability machine was envisioned in early work on CHERI [21] and has recently been proven secure by Skorstengaard et al. [6]. In this approach, there is no trusted stack manager and a single stack shared between all components, but the downside compared to our work is that the lack of locality levels restrict information flow and stack clearing is more extensive. Another, more recent proposal, by the same authors, manages to avoid the stack clearing by replacing local capabilities with a hypothetical new type of linear capabilities [22].

Other approaches to protecting the stack have been proposed, often by relying on some form of special hardware. On protected module architectures, every component's stack can be placed inside an enclave's private memory [23]–[25], [25]. In such an approach, sharing stack references across modules is also problematic.

On commodity hardware, many approaches have been proposed to add temporal memory safety for stack references. For space reasons, we cannot discuss all the work in this category, but refer to Nagarakatte et al. [26] for an overview. Instead of relying on special hardware primitives, these approaches somehow maintain metadata about allocated memory and inject additional checks during compilation. As a result, they usually come with relatively high overhead in terms of both memory usage and performance.

References

[1] N. P. Carter, S. W. Keckler, and W. J. Dally, "Hardware support for fast capability-based addressing," in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VI. New York, NY, USA: ACM, 1994, pp. 319–327. [Online]. Available: http://doi.acm.org/10.1145/195473.195579

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

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

[4] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. M. Watson, and P. Sewell, "Into the depths of c: Elaborating the de facto standards," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. New York, NY, USA: ACM, 2016, pp. 1–15. [Online]. Available: http://doi.acm.org/10.1145/2908080.2908081

[5] S. Tsampas, A. El-Korashy, M. Patrignani, D. Devriese, D. Garg, and F. Piessens, "Towards automatic compartmentalization of c programs on capability machines," in *Workshop on Foundations of Computer Security 2017*, 2017, pp. 1–14.

[6] L. Skorstengaard, D. Devriese, and L. Birkedal, "Reasoning about a machine with local capabilities," in *Programming Languages and Systems*, A. Ahmed, Ed. Cham: Springer International Publishing, 2018, pp. 475–501.

[7] ISO, *ISO/IEC 9899:2018 Information technology — Programming languages — C*, Jun. 2018. [Online]. Available: https://www.iso.org/standard/74528.html

[8] ——, *ISO/IEC 14882:2017 Information technology — Programming languages — C++*, 5th ed., Dec. 2017. [Online]. Available: https://www.iso.org/standard/68564.html

[9] R. N. Watson, P. G. Neumann, J. Woodruff, M. Roe, J. Anderson, J. Baldwin, D. Chisnall, B. Davis, A. Joannou, B. Laurie, S. W. Moore *et al.*, "Capability hardware enhanced risc instructions: Cheri instruction-set architecture (version 6)," University of Cambridge, Computer Laboratory, Tech. Rep., 2017.

[10] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, “The cheri capability model: Revisiting risc in an age of risk,” in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA ’14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 457–468. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665740

[11] L. Szekeres, M. Payer, T. Wei, and D. Song, “Sok: Eternal war in memory,” in *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, 2013, pp. 48–62. [Online]. Available: https://doi.org/10.1109/SP.2013.13

[12] J. Feist, “Finding the needle in the heap : combining binary analysis techniques to trigger use-after-free. (analyses de code binaire pour la détection et le déclenchement de use-after-free),” Ph.D. dissertation, Grenoble Alpes University, France, 2017. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01681707

[13] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, “Sok: Sanitizing for security,” *CoRR*, vol. abs/1806.04355, 2018. [Online]. Available: http://arxiv.org/abs/1806.04355

[14] “CVE-2015-1730.” Available from MITRE, CVE-ID CVE-2015-1730., Jun. 9 2015. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1730

[15] “CVE-2017-7756.” Available from MITRE, CVE-ID CVE-2017-7756., Jun. 11 2018. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7756

[16] U. Norell, “Dependently typed programming in agda,” in *Proceedings of the 6th International Conference on Advanced Functional Programming*, ser. AFP’08. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 230–266. [Online]. Available: http://dl.acm.org/citation.cfm?id=1813347.1813352

[17] C. Bach Poulsen, A. Rouvoet, A. Tolmach, R. Krebbers, and E. Visser, “Intrinsically-typed definitional interpreters for imperative languages,” *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 16:1–16:34, Dec. 2017. [Online]. Available: http://doi.acm.org/10.1145/3158104

[18] J. H. Morris Jr, “Lambda-calculus models of programming languages.” Ph.D. dissertation, Massachusetts Institute of Technology, 1969.

[19] J. Woodruff, A. Joannou, H. Xia, B. Davis, P. G. Neumann, R. N. M. Watson, S. Moore, A. Fox, R. Norton, D. Chisnall, and A. Fox, “Cheri concentrate: Practical compressed capabilities,” *IEEE Transactions on Computers*, pp. 1–1, 2019.

[20] A. Joannou, J. Woodruff, R. Kovacsics, S. W. Moore, A. Bradbury, H. Xia, R. N. M. Watson, D. Chisnall, M. Roe, B. Davis, E. Napierala, J. Baldwin, K. Gudka, P. G. Neumann, A. Mazzinghi, A. Richardson, S. Son, and A. T. Markettos, “Efficient Tagged Memory,” in *IEEE International Conference on Computer Design (ICCD)*. IEEE, Nov. 2017.

[21] R. N. Watson, P. G. Neumann, J. Woodruff, J. Anderson, R. Anderson, N. Dave, B. Laurie, S. W. Moore, S. J. Murdoch, P. Paeps *et al.*, “Cheri: a research platform deconflating hardware virtualization and protection,” in *Workshop paper, Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE 2012)*, 2012.

[22] L. Skorstengaard, D. Devriese, and L. Birkedal, “StkTokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities,” *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019.

[23] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens, “Secure Compilation to Protected Module Architectures,” *ACM Trans. Program. Lang. Syst.*, vol. 37, no. 2, Apr. 2015.

[24] M. Patrignani, D. Devriese, and F. Piessens, “On Modular and Fully Abstract Compilation,” in *Computer Security Foundations*. IEEE, 2016.

[25] P. Agten, R. Strackx, B. Jacobs, and F. Piessens, “Secure Compilation to Modern Processors,” in *Computer Security Foundations*, Jun. 2012, pp. 171–185.

[26] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, “CETS: Compiler Enforced Temporal Safety for C,” in *International Symposium on Memory Management*, ser. ISMM ’10. ACM, 2010.