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

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 propel 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 \texttt{getData} or \texttt{sortInt} are exported by a different sandbox than \texttt{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 known 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

\begin{verbatim}
void aFun(void) {
    int ar[100];
    getData(ar, 100);
    sortInt(ar, 100);
}
\end{verbatim}

Fig. 1: A C function calling external functions \texttt{getData} and \texttt{sortInt} to mutate local array \texttt{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.
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 \texttt{h} assigns argument pointer \texttt{p} an address that, after \texttt{h} returns, points at unallocated memory. When function \texttt{victim} is called and depending on the compiler and stack allocator, \texttt{q} might be pointing at the return address of \texttt{victim}. Thus any assignment compromises the execution flow of \texttt{victim}. Interestingly, this example works both as an exploitable bug in single-sandbox situations or as a security violation in a sandboxed environment: \texttt{main} and \texttt{f} can also be seen as malicious sandboxes that perform an attack on \texttt{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 formalize \textit{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.

---

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{image.png}
\caption{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.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{image2.png}
\caption{An example of an exploitable single-sandbox bug or an attack in a sandboxed environment. Function \texttt{victim} may end up overwriting its own return address.}
\end{figure}

\begin{verbatim}
void \texttt{f(int** p)} {
    int x;
    *p = &x; \textit{/* Unsafe assignment */}
}
void \texttt{main()} {
    int *q;
    \texttt{f(&q); /* q points at unused stack memory */}
    \texttt{h(q);}
}
void \texttt{victim(int* q)} {
    *q = 0; \textit{/* May overwrite own return address */}
}
\end{verbatim}
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 $a \in \text{Set}$ to denote that variable $a$ belongs in set $\text{Set}$. 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:

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

We shall be using “×” 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 $a \triangleright L$ 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, $\text{Ty}$, as

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

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

$$\text{Ns} := \text{Ty} \times \text{Ty}$$

$$\text{TypeOf} \in \text{Ns} \times (\text{arg} \triangleright \text{local}) \rightarrow \text{Ty}$$

$\text{Ns}$ 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 $(\text{arg} \triangleright \text{local}) \cong \text{Bool}$ when choosing between local variable and argument.

The set of expressions $\text{Expr}$ in ImpR is listed in Figure 5. It is parameterized over a local typing environment $\Gamma \in \text{Ns}$ and indexed by a type $\text{Ty}$. 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 ($\text{var}$) and dereferencing (*). It is worth noting that the underlying type system is exactly analogous to the scoping/tying rules of C\footnote{Not considering type-casting.}; 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 $\text{Expr}$ they are parameterized over a local typing environment $\Gamma \in \text{Ns}$ and indexed by $\text{Ty}$. There is however an extra parameter in $X \in \text{Exports}$, an alias for List Decl, which is defined as follows:

$$\text{Decl} := \text{Ty} \times \text{Ty}$$

$$a, r \in \text{Ns} \rightarrow \text{Ty}$$
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 `rv`. Consequently, `Exports` is a list of callable functions. Going back to the `Cmd` datatype, the parameter `X ∈ 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.

\[
\text{FunDef} \ (X ∈ Exports) :=
(t ∈ Ty) \times (n ∈ Ns) \times (Cmd \ X \ n \ t)
\]

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

\[
\text{DefList} \ (X ∈ Exports) \ (Y ∈ Exports) :=
[∀ d \triangleright_1 Y. \exists! fd ∈ \text{FunDef} \ X. \text{rd} \ [i | \ d]
\]

The judgment `fd \ [i | \ d` where `fd ∈ \text{FunDef} X` and `d ∈ 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 with a body.

\[
\text{return} \ rv \in \text{Expr} \ Γ \ t
\]
\[
\text{loc} \in \text{Expr} \ Γ \ (\text{Ref} \ t) \quad v \in \text{Expr} \ Γ \ t
\]
\[
\text{loc} := v \in \text{Cmd} \ X \ Γ \ Unit
\]
\[
c \in \text{Cmd} \ X \ Γ \ t_1 \quad d \in \text{Cmd} \ X \ Γ \ t_2
\]
\[
c \triangleright_1 d \in \text{Cmd} \ X \ Γ \ t_2
\]
\[
\text{cond} \in \text{Expr} \ Γ \text{Nat}
\]
\[
c \in \text{Cmd} \ X \ Γ \ t \quad d \in \text{Cmd} \ X \ Γ \ t
\]
\[
\text{if} \ \text{cond} \ \text{then} \ c \ \text{else} \ d \in \text{Cmd} \ X \ Γ \ t
\]
\[
\text{decl} \in \text{Decl} \quad \text{decl} \triangleright_1 X \quad \ a \in \text{Expr} \ Γ \ (a \ \text{decl})
\]
\[
\text{call} \ i \ a \in \text{Cmd} \ X \ Γ \ Unit
\]
\[
\text{decl} \in \text{Decl} \quad \text{decl} \triangleright_1 X \quad \ a \in \text{Expr} \ Γ \ (a \ \text{decl}) \quad \text{rv} \in \text{Expr} \ Γ \ (\text{Ref} \ (r \ \text{decl})
\]
\[
c \leftarrow i \ a \ r \in \text{Cmd} \ X \ Γ \ Unit
\]

Fig. 6: Commands in ImpR. The first four rules apply for all `X ∈ Exports` and `Γ ∈ Ns`.

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 ∈ Exports` implementing `Y ∈ Exports`. A program `Prog` is a `DefList` that interfaces with and implements the same `X ∈ 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 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 ∈ FunDef` completes `ctx ∈ 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.

\[
\text{Val} \ (L ∈ \text{StoreTy}) := \text{unit} | \text{nat} | \text{Loc} \ (\text{Loc} \ L) | \text{undef}
\]

`Val` is parameterized over a list of typing namespaces `L ∈ 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` 2This is a fine point in the C semantics which we slightly diverge from as explained in Section III.
the run-time behavior of a program: each time a function
with namespace \( \Gamma \) 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 undefined denotes an uninitialized Store entry. Pointers
are represented by the type Loc \( L \).

\[
\text{Loc } (L \in \text{StoreTy}) := (\exists b \in \mathbb{N}, b < \text{length } L) \times (\text{arg } \cup \text{ local})
\]

\[
\text{base } \in \text{Loc } L \rightarrow \mathbb{N}, \text{ off } \in \text{Loc } L \rightarrow (\text{arg } \cup \text{ local})
\]

An instance of \( \text{Loc } L \) 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:

\[
\text{Store } [[]] := []
\]

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

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

\( ^3 \)Hence off in Loc chooses left or right.

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
undefined 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
Up \( \subseteq (\text{Loc } L) \times (\text{Val } L) \) captures this notion (showing
only the non-trivial loc case):

\[
\begin{align*}
l_1, l_2 \in \text{Loc } L & \quad \text{base } l_1 \geq \text{base } l_2 \quad (l_1, l_2 \in \text{Up})
\end{align*}
\]

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:

\( ^4 \)Indexing in a Store begins at the tail end of the inductive definition.

\( ^5 \)The programmer may deliberately attempt to dereference or pass
an undefined reference but such cases are not in scope of this paper.
Lemma 1 (Monotonicity of lifetimes in a Store)

\[ \forall L, S \in \text{Store} \ L, l \in \text{Loc} \ L \implies l, \text{get S l} \in \text{Up} \]

Proof. By induction on the structure of S.

As a result, to update a Store at location l with v requires \( l, v \in \text{Up} \).

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.

State \((X \in \text{Exports}) \ (L \in \text{StoreTy}) := \)

\((\text{Store} \ L) \times (\text{CmdStore} \ X \ L) \times (\text{Ret} \ L)\)

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

Finally we introduce the ideal small-step operational semantics of ImpR. The ternary relation \( P \models S t_1 \implies S t_2 \) 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 \( \text{Up} \ p \ v \). The witness for \text{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 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.

\[
\begin{align*}
\text{Addr} & := \mathbb{N} \times (\text{arg} \cup \text{local}) \\
\text{base} & \in \text{Addr} \rightarrow \mathbb{N}, \ \text{off} \in \text{Addr} \rightarrow (\text{arg} \cup \text{local})
\end{align*}
\]

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

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

\[
\begin{align*}
\text{Cval} & := \text{unit} \mid \text{nat} \ \mathbb{N} \mid \text{capAddr} \mid \text{undef}
\end{align*}
\]

Instead of a Store, we define Memory \( L \) as a list of Cval of size \text{length} \( L \), where \( L \in \text{StoreTy} \) 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.
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 unforgiveable 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 \( M \in Memory \) 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.

\[
\text{CapState} \ (X \in \text{Exports}) 
\quad (L \in \text{StoreTy}) := \\
\quad (\text{Memory} \ L) \times (\text{CmdStore} \ X \ L) \times (\text{Cret} \ L)
\]

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 \( P \vdash C_{t_1} \rightarrow C_{t_2} \) which is read as “Under program \( P \), CapState \( C_{t_1} \) evaluates to CapState \( C_{t_2} \).”

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

Fig. 11: Evaluation of commands in the capability semantics. Assume \( X \in \text{Exports} \), \( P \in \text{Prog} \ X \) and \( M, C, R \) as a Memory, CmdStore, Cret parameterized by a suitable \( L \in \text{StoreTy} \).
invocation rules, \textit{call} and \textit{call+}.

Almost all of the preconditions when invoking functions are similar between the two semantics except one predicate on \textit{Cval} found in both rules: \textit{!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 \textit{call} case, the \textit{Memory} in the new \textit{CapState} is \((v - \text{undef}) M\). 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.

\subsection{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 \(ST : \text{State}\) and \(CT : \text{CapState}\) and then show that it is not a bisimulation:

\begin{center}
\begin{verbatim}
Unit bFun(Ref Unit); Unit aFun(Unit arg){
  Unit local;
  bFun(&local) ; return unit
}
\end{verbatim}
\end{center}

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

\begin{center}
\begin{verbatim}
Unit aFun(Ref (Ref Unit) arg){
  Unit local;
  var arg := &local ; return unit
}
\end{verbatim}
\end{center}

Fig. 13: An unsafe assignment in ImpR.

\begin{center}
\begin{verbatim}
Unit bFun(Ref Unit);
Unit aFun(Unit arg){
  Unit local;
  bFun(&local) ; return unit
}
\end{verbatim}
\end{center}

location relation \(R \Rightarrow R’\) are structural “lifts” on the value relation \(\rightsquigarrow\).

\begin{quote}
\begin{center}
\begin{verbatim}
∀ L ∈ StoreTy, v ∈ Val L, cv ∈ Cval. v ⇞ cv :=
  unit ⇞ unit | nat n ⇞ nat n | undef ⇞ undef |
  loc base, off ⇞ cap base, off
\end{verbatim}
\end{center}
\end{quote}

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 \textit{Loc} ensures that \(base < \text{length} L\).

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.

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 \textit{CapState} is fabricated: it cannot occur under normal execution due to the \textit{!Cap} restriction on arguments. In other words, without the \textit{!Cap} restriction, the capability semantics are not safe yet not sufficiently permissive with it.

\section{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.

\[
C\text{val} := \text{unit} \mid \text{nat} \mid \text{cap Addr} \mid \text{undef}
\]

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 \text{cap} constructor.

\[
C\text{val} := \text{unit} \mid \text{nat} \mid \text{cap Addr} \mid \text{undef}
\]

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

\[
\text{addrOf} \quad \text{vr} \in \text{arg} \uplus \text{local} \quad \text{len} = \text{length} \quad \Gamma \vdash \& \text{vr} \Rightarrow \text{cap len, vr, len}
\]

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 configuration and one that is already respected by 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 \text{base} being equal to the lifetime counter. We encode this property as predicate $\text{Sane} \subseteq C\text{val}$, which is free for related values.

**Lemma 4** (Related values are sane)

\[
\forall v, v' : v \rightsquigarrow v \in \text{Sane}
\]

**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 $S \rightsquigarrow M$)

\[
\forall S, M, e, v : S \rightsquigarrow M \times M \vdash e \Rightarrow v \rightarrow v \in \text{Sane}
\]

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

\[
M \models l \Rightarrow \text{cap base, off, n} \quad M \models r \Rightarrow v \\
\text{c} = (l := r) \quad w \in \text{base} \triangleleft \text{length} \quad (F :: L) \\
n, v \in \text{Safe} \quad M' = \text{update} \quad M \quad p \quad v \quad w \\
P \models M, c :: C, R \rightarrow M', (\text{return unit}) :: C, R \\
\text{cmd} = \text{body} (\text{getDef P decl}) \quad M \models \text{arg} \Rightarrow v \\
v \in !\text{Cap} \quad M' = (v, \text{undef}) :: M \\
C' = \text{cmd} :: c :: C \quad R' = \text{r} :: R \\
P \models M, (\text{c} \leftarrow \text{decl a r} ; c) :: C, R \rightarrow M', C', R'
\]

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)

\[
\forall L \in \text{StoreTy}, v \in \text{Val L}, cv \in C\text{val}. \quad v \rightsquigarrow cv := \\
\text{unit} \rightsquigarrow \text{unit} \mid \text{nat} \rightsquigarrow \text{nat} \mid \text{undef} \rightsquigarrow \text{undef} \\
\text{loc base, off} \rightsquigarrow \text{cap base, off, base}
\]

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 $\text{Sane} \subseteq \text{Cval}$, which is free for related values.

The problem can be traced down to the definition of a capability.
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 \( v \in \text{Sane} \). In a way property \( S \sim \Rightarrow M \) 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 \( \text{Valid} \subseteq \text{Memory} \times \text{Cval} \) to denote that a \( c \in \text{Cval} \) is either a literal value or an in-bounds address (omitting trivial cases).

\[
\begin{align*}
&vr \in \text{arg} \cup \text{local} \quad b \mid \varepsilon \mid M \\
&\text{validCap} \quad n \in \mathbb{N} \quad v = \text{cap} \ b, vr, n \\
\end{align*}
\]

Validity is also automatic for related values.

**Lemma 6** (Related values are valid)

\[
\forall L, v' \in \text{Val} \ L, M \in \text{Memory} \ L, v, v' \leadsto v \Rightarrow M, v \in \text{Valid}
\]

*Proof.* Trivially by Definition 1.

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

**Lemma 7** (Validity of evaluation when \( S \sim \Rightarrow M \))

\[
\forall S, M, e, v. \ S \sim \Rightarrow M \times M \vdash e \Rightarrow v \Rightarrow M, v \in \text{Valid}
\]

*Proof.* Similar to Lemma 5. By case-splitting the evaluation relation combined with Lemma 6 expanded on \( M \).

Again, it is the extra \( S \sim \Rightarrow M \) that guarantees validity for the evaluation of expressions. Sanity and validity combine with great potency:

**Lemma 8** (Inversion of Cval)

\[
\forall L, M \in \text{Memory} \ L, v \in \text{Sane}, (M, v) \in \text{Valid}.
\]

\[
\exists v' \in \text{Val} \ L, v' \leadsto v
\]

*Proof.* By Definition 1.

Lemma 8 shows that validity is 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 \( v \leadsto c \) preserves the two properties while the other direction requires them. In fact, it can be shown that the set of values \( v \in \text{Val} \ L \) is precisely the set of \( c \in \text{Cval} \) with \( c \in \text{Sane} \) and \( M, c \in \text{Valid} \) for some \( M \in \text{Memory} \ L \), 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 \( \sim \) is bisimulation-like)

\[
\forall S, M, v, e. \ S \sim \Rightarrow M \times S \vdash e \Rightarrow v \Rightarrow \exists v'. \ M \vdash e \Rightarrow v'
\]

\[
\forall S, M, v, e. \ S \sim \Rightarrow M \times M \vdash e \Rightarrow v \Rightarrow \exists v'. \ S \vdash e \Rightarrow v'
\]

*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)

\[
\forall P, St_1, St_2, Ct_1, St_1 \leadsto Ct_1 \times P \vdash St_1 \rightarrow St_2
\]

\[
\Rightarrow \exists Ct_2, St_2 \leadsto Ct_2 \times P \vdash Ct_1 \rightarrow Ct_2
\]

*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.

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)

\[
\forall d, X, f_1, f_2 \in \text{FunDef} \ (d :: X), f_1 \mid i \mid d, f_2 \mid i \mid d.
\]

\[
f_1 \equiv_{ctx} f_2 \iff \ldots
\]

*Where \( P \equiv_{ctx} \) denotes that \( P \in \text{Prog} \) terminates.*

**Definition 3** (Termination in the ideal semantics)

\[
P \downarrow_i \iff P \vdash (\text{undef, undef}) :: [], \text{main} :: [], \text{nothing} :: [] \rightarrow* \text{} [\text{, ,}]
\]

And similarly for the capability semantics. Relation \( P \vdash St_1 \rightarrow* St_2 \) is the reflexive, transitive closure of the command evaluation relation and \text{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)

\[
\forall d, X, f_1, f_2 \in \text{FunDef} \ (d :: X), f_1 \mid i \mid d, f_2 \mid i \mid d.
\]

\[
f_1 \equiv_{ctx} f_2 \iff f_1 \equiv_{ctx} f_2
\]

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

\footnote{\url{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 \( \text{cap a } 0 \) 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 single stack allocators 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], [26]. 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
