Borrowed Capabilities: Flexibly Enforcing Revocation on a Capability Architecture

Thijs Vercammen
KU Leuven

Thomas Van Strydonck
KU Leuven

Dominique Devriese
Vrije Universiteit Brussel

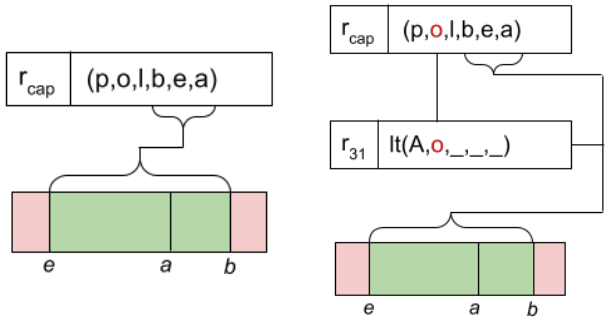
***Abstract*—Capability machine ISAs offer a security primitive called capabilities: unforgeable tokens that represent authority over memory, and the authority to invoke other components. To efficiently implement temporary authority delegation, i.e. capabilities which can be revoked between distrusting components, ISA extensions have been proposed and/or implemented, based on forms of garbage collection, local capabilities or linear capabilities. Each of these mechanisms has specific advantages and disadvantages. In this paper, we explore borrowed capabilities: a novel mechanism that combines some of the advantages of linear capabilities without requiring linear treatment of revocable pointers. Inspired by borrowing in substructural type systems like Rust, the idea is to reinterpret capability sealing as a form of borrowing. The seals behave as a form of lifetime identifier and can be matched against a separate lifetime token, which represents the lifetime. This short paper offers a first exploration of the idea, by defining an extension of the CHERI-RISC-V ISA with borrowed capabilities and a variety of features like mutable and immutable borrows, fractional lifetimes and reborrows. We extend the LLVM assembler for CHERI and experiment with example programs.**

***Index Terms*—capabilities, ownership, revocation, CHERI, Sail, Rust**

1. Introduction

In any application where multiple stakeholders manipulate resources (file system handles, sockets, stack frames, ...), it is important to correctly manage access to them. One essential aspect of this process is *revocation*; repealing the access an untrusted stakeholder has to a resource we own. In this work, we are specifically interested in mechanisms for revocation on *capability machines*; architectures that implement capabilities as a security primitive.

Hardware capabilities are unforgeable pointers that represent authority explicitly. They carry permissions, and grant their owner access to a specific memory region. Fig. 1a illustrates how a capability (p, o, l, b, e, a) in register $\mathsf{r}_{\mathrm{cap}}$ carries permissions p (e.g. RW) over an area $[b, e)$ of memory, currently pointing to address a (l denotes the linearity of the capability and o the otype, further explained below). In recent years, these capabilities have received renewed attention, largely thanks to the CHERI capability machine [1]. Capability machines are particularly well-suited to enforcing spatial safety properties, and thereby prevent e.g. buffer overflows. On the other hand, temporal safety properties, including revocation, are a less obvious fit to the capability model, and therefore harder to enforce *efficiently* in practice [2].



(a) A regular capability *cap* carries its own authority over memory.

(b) A borrowed capability only carries authority over memory if register 31 contains a matching lifetime token.

Figure 1. Representation of authority carried over memory by capabilities.

Prior work has attempted to solve the revocation issue in several ways. CHERIvoke [3] and Cornucopia [4] present a method of revoking capabilities by modifying the system's memory allocator and periodically sweeping memory to remove capabilities pointing to memory that has been freed. While this approach to temporal safety could be tailored to different scenarios than just memory allocation, it includes the memory allocator into the TCB, and could be prohibitively expensive if the memory sweep is required often (e.g. after each call to an adversary). Two other approaches each use a different type of capabilities, respectively called *local* and *linear* capabilities, to implement a type of revocation that does not rely on a memory sweep or a software TCB, and could perform better in the aforementioned scenarios.

Local capabilities are capabilities in CHERI that do not have the *global* permission set. Capabilities without this permission can only be stored to so-called *write-local* memory (defined through a different permission), providing a useful primitive for software to build upon. Software can limit the propagation of local capabilities by limiting access to write-local memory, and can implement revocation by clearing all write-local memory. Skorstengaard et al. introduce a new calling convention using local capabilities [5] to enforce revocation on the stack. Georges et al. have proposed a new type of capability, called uninitialized capabilities [6] to prevent the necessity of clearing the write-local memory in this stack setting.

Linear capabilities are capabilities that cannot be duplicated. Consequently, when one party passes a linear capability to another, it cannot retain a copy, effectively revoking the capability. They are discussed in the CHERI ISA specification, but have not (yet) been implemented [7]. Van Strydonck et al. used linear capabilities to implement revocation in a fully abstract compiler from separation-logic-verified C code to a capability machine [8], and Skorstengaard et al. used them to enforce well-bracketed control flow and stack encapsulation [9]. Because linear capabilities cannot be copied at all, they are a very restrictive way of revoking capabilities. For example, they cannot be used to provide multiple parties with simultaneous read-only access, or in a multi-threaded setting.

In this paper we present another way of revoking capabilities, through the use of a new type of capability called *borrowed capability*. Borrowed capabilities expand upon linear capabilities and offer a more flexible approach to revocation, without requiring clearing regions of memory like local capabilities do. They allow software to define scopes, called *lifetimes*, in the form of *lifetime tokens* and bind capabilities to those tokens, requiring that the lifetime token is present when dereferencing the capability. Fig. 1b illustrates this concept, where an alive (A) lifetime token $(\text{A}, o, \_, \_, \_)$ (where the _ represents a don't care value for fields we will explain later) needs to be present in register 31 to allow loading or storing through a capability (p, o, l, b, e, a), where o now represents the lifetime. The revocation problem then shifts to the revocation of lifetimes as opposed to the revocation of individual capabilities. This gives software the option to avoid the restrictive copy prohibition associated with linear capabilities, or the restrictions on storing local capabilities in non-write-local memory, while still being able to do fine grained revocation of capabilities. Our work should be considered as an exploration of the design space of borrowed capabilities rather than a complete design; section 6 discusses a few open questions and design decisions that we still aim to resolve. Nevertheless, we think our work on borrowed capabilities is mature enough to contribute to the CHERI ecosystem.

Borrowed capabilities bear a strong resemblance to substructural type systems such as the ownership and borrowing system in Rust. These type systems often allow either mutation or aliasing, but not both at the same time in order to prevent data races. With borrowed capabilities we support this behavior by introducing two types of borrowing: mutable and immutable borrows. One potential line of future research would hence be a secure compiler that expresses high-level ownership-related concepts in terms of borrowed capabilities at the target level. Section 6 discusses how our design can be slightly simplified, in case we do not care about "aliasing XOR mutation", but rather, simply desire flexible revocation.

In summary, our contributions are as follows:

- A proposal for *borrowed capabilities*, a new type of capability to allow for efficient, yet flexible, revocation, combining some of the advantages of local and linear capabilities.
- A design for an ISA extension of CHERI-RISC-V to implement borrowed capabilities and express ownership, borrowing, lifetime fractions, and reborrowing capabilities.
- An implementation of said design in the Sail ISA description language, to illustrate its feasibility.
- An extension of the assembler of the LLVM compiler to allow assembling programs using the new instructions for borrowed capabilities.

Our Sail implementation has been made public and is available at https://github.com/exolyte/sail-cheri-riscv.

2. CHERI

CHERI, which stands for Capability Hardware Enhanced RISC Instructions, is an architecture neutral hardware capability protection model with implementations in MIPS, RISC-V, ARM and x86 [7]. CHERI guarantees strong spatial memory protection. Crucially, CHERI enforces capability monotonicity; instructions can only maintain or decrease the authority capabilities grant, never increase them.

The object type (o or $otype$) field is almost exclusively used to implement sealed *code-data pairs*. These are pairs of capabilities with the same otype value, where one capability points to executable code and the other to data to be used by that code. Code-data pairs are intended to be used to transfer execution to another security domain, and hence constitute an implementation of *object capabilities*. Additionally, 16 fixed, reserved otype values are used to identify certain other types of capabilities. All capabilities with an otype value that is different from a specific *otype_unsealed* value (which we take to be 0 here) are considered sealed, which prevents them from being directly modified or used. As shown in fig. 1b, in this paper we will use the otype field to represent a lifetime.

The linear bit l determines whether a capability is linear. When the linear bit is set to 1, the capability cannot be duplicated in any way. This means that the source capability in load and store instructions, move instructions and rights update instructions is invalidated or overwritten, to ensure move semantics. When the linear bit is set to 0, no such restrictions apply.

3. Borrowed Capabilities

The motivation for borrowed capabilities stems from the lack of flexibility of linear capabilities, and the memory clearing overhead of local capabilities. In the design of borrowed capabilities we introduce a way to break linearity in a controlled manner.

Linear capabilities themselves still represent unique ownership of a resource, but can be *borrowed*, transforming them into borrowed capabilities instead, and allowing for linearity to be broken temporarily, and e.g. read-only ownership to be shared between multiple stakeholders. Separate linear tokens, called *lifetime tokens*, define unforgeable lifetimes that linear capabilities are bound to during this borrowing process. Borrowed capabilities can only be dereferenced while a live (as opposed to dead), matching lifetime token is present in register 31. Ending the lifetime corresponds to revocation, without requiring all borrowed capabilities to be returned; killing a lifetime revokes all authority of borrowed capabilities bound to

$$
\begin{array}{lll}
a & \in \text{Addr} & \triangleq [0, \text{AddrMax}] \\
p & \in \text{Perm} & \triangleq \text{O} \mid \text{RO} \mid \text{RX} \mid \text{RW} \mid \text{RWX} \\
o & \in \text{Otype} & \triangleq [0, \text{OtypeMax}] \\
l & \in \text{Linear} & \triangleq 0 \mid 1 \\
c & \in \text{Cap} & \triangleq \{(p, o, l, b, e, a) \\
 & & \quad \mid b, e, a \in \text{Addr}\} \\
\hline
s & \in \text{State} & \triangleq \text{A} \mid \text{D} \\
f & \in \text{Fraction} & \triangleq [0, \text{FractionMax}] \\
lt & \in \text{LifetimeToken} & \triangleq \{\text{lt}(s, lid, cid, pid, f) \\
 & & \quad \mid lid, cid, pid \in \text{Otype}\} \\
idx & \in \text{Index} & \triangleq [0, \text{IndexMax}] \\
it & \in \text{IndexToken} & \triangleq \{\text{it}(lid, idx) \mid lid \in \text{Otype}\} \\
bt & \in \text{BorrowTable} & \triangleq \text{Index} \rightharpoonup \text{Cap} \\
\hline
r & \in \text{RegName} & \\
\varphi & \in \text{ExecConf} & \triangleq \text{Reg} \times \text{Mem}
\end{array}
$$

$$
\begin{array}{lll}
i & ::= & \ldots \mid \texttt{CCreateToken}\ r\ r \mid \texttt{CKillToken}\ r\ r \mid \\
 & & \texttt{CUnlockToken}\ r\ r\ r \mid \texttt{CSplitLT}\ r\ r \mid \\
 & & \texttt{CMergeLT}\ r\ r\ r \mid \texttt{CBorrowImmut}\ r\ r\ r \mid \\
 & & \texttt{CBorrowMut}\ r\ r\ r \mid \texttt{CRetrieveIndex}\ r\ r\ r
\end{array}
$$

Figure 2. Machine words, machine state and added instructions.

that lifetime. Similarly to the situation in Rust, we provide for two types of borrowing; mutable and immutable borrowing, corresponding to the cases where we are interested in respectively aliasing and mutation.

To describe our design more precisely, we define some basic syntactic constructs for our capability machine in fig. 2. The top section defines pre-existing notions for capabilities and their fields while the middle section defines the constructs we add which we will explain throughout this section. The bottom section lists the state of our machine, consisting of memory and registers, and lists the new instructions we add. The semantics of the new instructions are shown in fig. 3.

We represent the fields of lifetime tokens as a tuple of the form $\text{lt}(s, lid, cid, pid, f)$ representing the state, lifetime id, child id, parent id and fraction fields. The state field indicates whether the lifetime associated with the lifetime token is alive (A) or dead (D). The first case of the `CCreateToken` instruction in fig. 3 creates a normal standalone alive lifetime token. The second case will be explained later in this section. Alive lifetime tokens are linear and can be used to borrow new capabilities, and dereference existing borrowed capabilities when placed in a specific register. A lifetime token can be irreversibly killed with the `CKillToken` instruction. This changes the state to dead which results in the lifetime token losing its linearity. In this state, a lifetime token cannot be used anymore to borrow capabilities or dereference borrowed capabilities, but it does serve as a proof of the lifetime's end that can be freely copied and passed around. The lifetime id field holds the lifetime id associated with the token. This is a unique id that is used to match a borrowed capability with its corresponding lifetime token. The fraction, parent id and child id fields will be discussed later in this section.

A lifetime token can be used to borrow a capability through the `CBorrowImmut` or `CBorrowMut` instructions, for immutable and mutable borrows respectively. What happens in these borrow operations is that the capability that gets borrowed is stored in a table that we call the *borrow table* (bt). The source register is overwritten with the borrowed capability. This borrowed capability points to the same region of memory as the original capability but it differs in a few ways. First, the borrowed capability holds the lifetime id of the lifetime it was borrowed under in its otype field. This is necessary in order to check whether the used lifetime token matches the borrowed capability when dereferencing it. Second, depending on whether the original capability was immutably or mutably borrowed, the borrowed capability might have differing permissions. Immutably borrowed capabilities lose their linearity as well as their write permissions if they were present on the original capability. This weakens the original linear capability's exclusive access to a resource, but does so in a controlled manner, namely for the duration of the lifetime. This is sufficient to prevent simultaneous write accesses while allowing multiple references with read access. This behavior allows programming languages with substructural type systems to be mapped to borrowed capabilities. Mutably borrowed capabilities keep their linearity and write access. While it may seem that this behavior does not have advantages over regular linear capabilities, this is not the case. Callees only need to return lifetime tokens to their callers, not the borrowed capabilities themselves. This allows callers to hand out multiple capabilities borrowed under the same lifetime to a callee and revoke all of them at once by receiving the lifetime token back, and ending the lifetime. Some issues arise when the structure that a borrowed capability points to contains references in the form of capabilities itself. We will address these issues further in section 6.

As previously mentioned, when a capability is borrowed, the original capability is stored in the borrow table, to be retrieved after the lifetime ends. In order to retrieve capabilities from the borrow table, we introduce linear *index tokens* that can be traded for the original capability in the borrow table through the `CRetrieveToken` instruction. We represent index tokens as a tuple of the form $\text{it}(lid, idx)$, keeping track of the lifetime id lid under which the capability stored at index idx was borrowed. Index tokens are produced by the previously described borrow instructions. The `CRetrieveToken` instruction requires both the index token as well as a dead corresponding lifetime token that acts as proof that the borrowed capabilities cannot be used anymore. This system allows for mutual distrust between a callee and their caller since the callee has a guarantee that its caller cannot access the original capability while it owns the living lifetime token, and the caller has the guarantee that the callee can no longer access the borrowed capability when it has a dead lifetime token.

As we have explained, one of the disadvantages of linear capabilities is their restrictiveness with regards to simultaneous access. With the design of borrowed capabilities, this restrictiveness shifts from capabilities to lifetime tokens. This is why we introduce lifetime fractions.

To allow dereferencing immutable borrows in multiple threads concurrently, we allow lifetime tokens to have a fraction f and be split using the `CSplitToken` in-

i	$[\![i]\!](\varphi)$	Conditions
`CCreateToken` r_1 r_2	$\varphi[\text{reg}.r_1 \mapsto w_1, \text{reg}.r_2 \mapsto w_2]$	if $\varphi.\text{reg}(r_2) == \text{lt}(D,0,0,0,0)$ then $w_1 = \text{lt}(A, lid, 0, 0, 0)$ and $w_2 = \text{lt}(D,0,0,0,0)$ else if $\varphi.\text{reg}(r_2) == \text{lt}(A, pid, 0, ppid, f)$ then $w_1 = \text{lt}(A, lid, 0, pid, 0)$ and $w_2 = \text{lt}(A, pid, lid, ppid, f)$
`CKillToken` r_1 r_2	$\varphi[\text{reg}.r_1 \mapsto w_1, \text{reg}.r_2 \mapsto 0]$	$\varphi.\text{reg}(r_2) = \text{lt}(A, lid, 0, pid, f_{max})$ and $w_1 = \text{lt}(D, lid, 0, pid, f_{max})$
`CUnlockToken` r_1 r_2 r_3	$\varphi[\text{reg}.r_1 \mapsto w_1, \text{reg}.r_2 \mapsto 0]$	$\varphi.\text{reg}(r_2) = \text{lt}(A, lid, cid, pid, f)$ and $\varphi.\text{reg}(r_3) = \text{lt}(D, cid, \_, \_, \_)$ and $w_1 = \text{lt}(A, lid, 0, pid, f)$
`CSplitLT` r_1 r_2	$\varphi[\text{reg}.r_1 \mapsto w_1, \text{reg}.r_2 \mapsto w_2]$	$\varphi.\text{reg}(r_1) = \text{lt}(A, lid, cid, pid, f)$ and $w_1 = \text{lt}(A, lid, cid, pid, f+1)$ and $w_2 = \text{lt}(A, lid, cid, pid, f+1)$
`CMergeLT` r_1 r_2 r_3	$\varphi[\text{reg}.r_1 \mapsto w_1, \text{reg}.r_2 \mapsto 0, \text{reg}.r_3 \mapsto 0]$	$\varphi.\text{reg}(r_2) = \text{lt}(A, lid, cid, pid, f)$ and $\varphi.\text{reg}(r_3) = \text{lt}(A, lid, cid, pid, f)$ and $w_1 = \text{lt}(A, lid, cid, pid, f-1)$
`CBorrowImmut` r_1 r_2 r_3	$\varphi[\text{reg}.r_1 \mapsto w_1, \text{reg}.r_2 \mapsto w_2, \text{bt}.idx \mapsto \varphi.\text{reg}(r_2)]$	$\varphi.\text{reg}(r_2) = (p, o, 1, b, e, a)$ and $\varphi.\text{reg}(r_3) = \text{lt}(A, lid, \_, pid, \_)$ and if $o == 0$ *or* $o == pid$ then $w_1 = \text{it}(lid, idx)$ and $w_2 = (\text{RO}, lid, 0, b, e, a)$
`CBorrowMut` r_1 r_2 r_3	$\varphi[\text{reg}.r_1 \mapsto w_1, \text{reg}.r_2 \mapsto w_2, \text{bt}.idx \mapsto \varphi.\text{reg}(r_2)]$	$\varphi.\text{reg}(r_2) = (p, o, 1, b, e, a)$ and $\varphi.\text{reg}(r_3) = \text{lt}(A, lid, \_, pid, \_)$ and if $o == 0$ *or* $o == pid$ then $w_1 = \text{it}(lid, idx)$ and $w_2 = (\text{RW}, lid, 1, b, e, a)$
`CRetrieveIndex` r_1 r_2 r_3	$\varphi[\text{reg}.r_1 \mapsto w_1, \text{reg}.r_2 \mapsto 0, \text{bt}.idx \mapsto 0]$	$\varphi.\text{reg}(r_2) = \text{it}(lid, idx)$ and $\varphi.\text{reg}(r_3) = \text{lt}(D, lid, \_, \_, \_)$ and $w_1 = \varphi.\text{bt}(idx)$

Figure 3. Operational semantics for essential cases in our novel instructions.

struction. Fractions are represented as an integer where a fraction of 0 signifies the full lifetime token. Splitting a lifetime token with fraction f results in two copies of the lifetime token with respective fractions f_1 and f_2, such that $f_1 = f_2 = f + 1$. Both fractions can still be used to borrow capabilities or dereference borrowed capabilities. Merging identical lifetime tokens is possible with the `CMergeToken` instruction. To prevent dead and alive lifetime tokens of the same lifetime id being present at the same time, only unfractured lifetime tokens can be killed. Once different threads or adversaries using the fractions of the lifetime token have completed their work, they can return the lifetime fractions which can then be merged again to the full lifetime token, which can then be killed.

Care must be taken when borrowing borrowed capabilities, a so-called "reborrow" operation. Borrowed capabilities cannot be allowed to be reborrowed under just any lifetime. This would make it possible to reborrow a capability under a lifetime that is longer than the original borrow which breaks all guarantees provided by revocation. To make reborrows possible, we introduce lifetime hierarchies. With this system, lifetimes can be created as sublifetimes of existing lifetimes by providing a fraction of the desired parent lifetime to the `CCreateToken` operation as shown in the second case in fig. 3. This sets the child id field on the parent to the lifetime id of the newly created lifetime and sets the parent id field of the newly created lifetime token to the lifetime id of the parent. Lifetime tokens cannot be killed while they have a child id set, but don't lose any of their other functionality. This ensures that sublifetimes cannot last longer than their parent lifetime while still keeping the parent lifetime available. In order to remove a child from a parent lifetime token, a dead lifetime token with the lifetime id of the child id field on the parent token is needed. These two tokens can then be used in the `CUnlockToken` instruction which clears the child id field on the parent token. The parent token can then be killed or receive a new sublifetime.

Lifetime hierarchies allow for safe reborrowing through the normal borrow operations `CBorrowImmut` and `CBorrowMut`. The main requirement for reborrowing borrowed capabilities is that the parent id on the used lifetime token matches the lifetime id of the capability that is being reborrowed. This ensures that a reborrow happens under a sublifetime and thus that the reborrow has a shorter lifetime than the original borrow. One issue with lifetime hierarchies is that a lifetime token can only have one parent, which makes it impossible to reborrow multiple capabilities with different lifetimes under the same lifetime. We will address this further in section 6.

4. Design Implementation

In this section, we discuss how we implemented the design from the last section as an extension of the Sail model for CHERI-RISC-V. Sail is an instruction set architecture (ISA) specification language that can be used to formally describe the semantics of the instructions of an ISA [10]. Sail models have a variety of additional uses such as generating documentation, deriving an emulator and formal reasoning about the ISA. The implementation provides us with an exact description of the semantics of new instructions as well as an emulator that we can use to run test assembly programs. Because Sail does not generate an assembler out of the box, we also extended CHERI's fork of the LLVM compiler to support our newly added instructions. This extension has the added advantage of providing a means to generate well-formed

binaries that we can run in the Sail emulator. Because no implementation of linear capabilities for the Sail model existed, we had to implement linear capability support before starting on borrowed capabilities. We will not expand further on our implementation of linear capabilities.

The first issue with mapping the design of borrowed capabilities to the CHERI architecture is the representation of lifetime and index tokens. CHERI assumes all capabilities satisfy a single, specific layout and does not offer a layout for tokens that hold different information. Because we require a number of fields that are not present in the current capability layout, we reinterpret the layout of capabilities when they are interpreted as lifetime or index tokens. To keep the interpretation of a capability simple, we simply reinterpret and split up fields that are present in regular capabilities. In the long run, it would be interesting for the CHERI architecture in general to support more than one capability format, for different functionality.

For both tokens, we keep the permissions and otype fields as they are in regular capabilities. We use the otype field to identify lifetime or index tokens by setting their value to one of the 16 reserved otype values that were previously mentioned. For lifetime tokens, we use one of the capability bounds fields to store the lifetime fraction and split up the address field to store the lifetime id, parent id and child id values that are 18 bits each. For index tokens, we split up the address field to store the index and lifetime id values. The width of the index field is proportional to the size of the borrow table and was chosen to be 16 bits in our prototype, but can be expanded or shrunk as needed. As mentioned in section 3 we use the otype field to store the lifetime id in borrowed capabilities. Because this use of the otype field clashes with the already present use case of code-data pairs, we reserve the lower half of the otype space for borrowed capabilities. This means that generated lifetime id's can only come from this range of values. An alternative to this scheme would be to add a borrowed bit to capabilities which would then be used as an indication for how to interpret the otype field. In order to create unique lifetime ids we introduce a special lifetime counter register that is read from and incremented each time a new lifetime token is created.

To support borrowed capabilities, we had to modify a number of existing CHERI instructions. The first class of these instructions are the load and store instructions, which were modified to check for a lifetime token with a specific lifetime id in a specific register when dereferencing a borrowed capabilities. The second class are the instructions that deal with sealed capabilities, because using the otype field for lifetime ids results in borrowed capabilities being a special-cased form of sealed capabilities. This means that the instructions that assume that all sealed capabilities are code-data pairs need to be modified. The last class of instructions are the instructions that manipulate the address field of a capability. We modified these instructions to make it possible to change the address field on a sealed borrowed capability. This makes it possible to dereference every part of a borrowed capability that points to a data structure such as an array or struct. Of course we also added the instructions described in section 3.

5. Example Program

Fig. 4 shows a simple snippet of an assembly program that illustrates the usage of the new instructions, annotated with Rust code that matches this assembly. The first two lines load an integer to integer register `x1` and store that integer in the memory pointed to by the capability in capability register `c2`. This is somewhat similar to creating a new variable, x, and assigning it a value. The next four lines start by creating a lifetime token, which is roughly similar to opening a new scope. The lifetime token is then used to mutably borrow x with the borrowed capability being placed in register `c2` and the resulting index token being placed in `c4`. The following two instructions store a new value to memory using the borrowed capability. This is allowed, because the lifetime token is present in the lifetime register `c31`. In the next five lines, a new lifetime token is created as the child lifetime of the previous lifetime token and used to immutably reborrow the mutable borrow. The program then uses the immutable reborrow and the lifetime token in `c31` to load the value to `x7` after which it closes the scope by killing the lifetime token. The next four lines start by using the index token and dead lifetime token to retrieve the mutable borrow from the borrow table. Following that, the child lifetime is removed from the first lifetime token, which allows it to be killed and used together with the index token in `c4` to retrieve the original capability, corresponding to variable x.

```
1  li              x1 0x5
2  sw.cap          x1 0(c2)     #let mut
3                               #x = 5;
4
5  CCreateToken    c31 c0       #{
6  CBorrowMut      c4 c2 c31    #y = &mut x;
7  li              x1 0x6
8  sw.cap          x1 0(c2)     #*y = 6;
9
10 CMove           c30 c31
11 CCreateToken    c31 c30      #{
12 CBorrowImmut    c5 c2 c31    #z = &y;
13 lw.cap          x7 0(c2)     #temp = 6;
14 CKillToken      c31 c31      #}
15
16 CRetrieveIndex  c5 c5 c31
17 CUnlockToken    c30 c31
18 CKillToken      c30 c30      #}
19 CRetrieveIndex  c4 c4 c30
```

Figure 4. An assembly program illustrating the proposed ISA extension.

6. Discussion

This section discusses remaining challenges in our current design, which we aim to resolve in future work.

Multiple output registers. The biggest obstacle to implementing borrowed capabilities in microarchitecture is the requirement for some new instructions to write to multiple registers, which can be expensive in hardware. This cost was already present in the implementation of linear capabilities themselves [9]. Sometimes an input register simply needs to be cleared, such as the index token in `CRetrieveIndex`. In these cases it suffices to require that the output register is equal to the input

register, resulting in the input register being overwritten. However, in other cases multiple register writes cannot be avoided; e.g. the borrow instructions need to write both an index token and the borrowed capability.

No "read XOR write". In case we are not interested in preventing simultaneous read and write accesses to borrowed capabilities, we can combine (im)mutable borrows into one general borrow operation, which provides a duplicable capability that preserves write-access. A caller of this borrow operation can then still choose to only share read-access with an adversary, by restricting the permissions of the borrowed capability. This scheme also permits borrowing non-linear capabilities, as the recipient no longer requires the guarantee that no aliases of a mutable borrow or writable aliases of an immutable borrow exist elsewhere.

Dynamic lifetime dependencies. As mentioned in section 3, an issue with lifetime hierarchies is that a lifetime token can only have a single parent. To solve this, we have come up with a different scheme for reborrowing that we have not yet fully explored, called *dynamic lifetimes*, which is inspired by the lifetime calculus in RustBelt [11]. This involves borrowing a fraction of the parent lifetime under the child lifetime, and thereby storing this fraction in the borrow table. Since the fraction of the parent that is needed to kill it can only be retrieved from the borrow table with the help of the dead child lifetime token, it is ensured that the parents' lifetime is longer than the child's. This scheme allows any lifetime token to dynamically become a child of another lifetime token. Reborrowing a capability with the parent's lifetime under the child's lifetime would then be possible by providing the reborrowing instruction with the borrowed lifetime fraction, as dynamic proof of relationship between the parent and child lifetime. Dynamic lifetimes would make the lifetime system more flexible at the cost of extra stores to the borrow table, and the complexity of managing the borrowed lifetime token fractions.

Recursive loads. Lastly, a possible concern with borrowed capabilities is their ability to load other capabilities from the memory they point to. This is a problem because the recursively loaded capabilities would not be lifetime-restricted, resulting in the possibility for an adversary to keep the references after the borrow has ended. A possible solution is to automatically borrow any loaded capabilities under the parent capability's lifetime (or a sublifetime thereof). However, this might still lead to simultaneous read and write accesses; it is e.g. possible to load the same mutable reference twice. This problem can be mitigated by making all loaded capabilities lose their write access which is something that the experimental CHERI *Permit_Recursive_Mutable_Load* permission does [7]. Alternatively, to avoid loading the same linear, mutable reference multiple times, the mutable reference in memory would have to be overwritten.

7. Conclusion

We have presented our design for borrowed capabilities, a new alternative for revocation in CHERI, by showcasing the semantics for a CHERI-RISC-V ISA extension, and discussing our Sail implementation of said design. As discussed, the design is in the prototype stage, and details of the design as well as practical usefulness need to be investigated further.

References

[1] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera, "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization," in *IEEE Symposium on Security and Privacy*, 2015, pp. 20–37.

[2] N. Joly, S. ElSherei, and S. Amar, "Security analysis of CHERI ISA," 2020. [Online]. Available: https://github.com/microsoft/MSRC-Security-Research/blob/master/papers/2020/Security%20analysis%20of%20CHERI%20ISA.pdf

[3] H. Xia, J. Woodruff, S. Ainsworth, N. W. Filardo, M. Roe, A. Richardson, P. Rugg, P. G. Neumann, S. W. Moore, R. N. M. Watson, and T. M. Jones, "CHERIvoke: Characterising Pointer Revocation using CHERI Capabilities for Temporal Memory Safety," in *IEEE/ACM International Symposium on Microarchitecture*. ACM, Oct. 2019.

[4] N. W. Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson, "Cornucopia: Temporal Safety for CHERI Heaps," in *IEEE Symposium on Security and Privacy*. IEEE, May 2020.

[5] L. Skorstengaard, D. Devriese, and L. Birkedal, "Reasoning about a machine with local capabilities - provably safe stack and return pointer management," in *European Symposium on Programming*, 2018, pp. 475–501.

[6] A. L. Georges, A. Guéneau, T. Van Strydonck, A. Timany, A. Trieu, S. Huyghebaert, D. Devriese, and L. Birkedal, "Efficient and provable local capability revocation using uninitialized capabilities," *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, pp. 6:1–6:30, Jan. 2021.

[7] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, H. Almatary, J. Anderson, J. Baldwin, G. Barnes, D. Chisnall, J. Clarke, B. Davis, L. Eisen, N. W. Filardo, R. Grisenthwaite, A. Joannou, B. Laurie, A. T. Markettos, S. W. Moore, S. J. Murdoch, K. Nienhuis, R. Norton, A. Richardson, P. Rugg, P. Sewell, S. Son, and H. Xia, "Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8)," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-951, 2020. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-951.html

[8] T. Van Strydonck, F. Piessens, and D. Devriese, "Linear capabilities for fully abstract compilation of separation-logic-verified code," *Proc. ACM Program. Lang.*, vol. ICFP, 2019.

[9] 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.

[10] A. Armstrong, T. Bauereiss, B. Campbell, S. Flur, J. French, K. E. Gray, G. Kerneis, N. Krishnaswami, P. Mundkur, R. Norton-Wright, C. Pulte, A. Reid, P. Sewell, I. Stark, and M. Wassell, "The Sail instruction-set semantics (ISA) specification language," 2013–2019.

[11] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "RustBelt: Securing the foundations of the Rust programming language," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017.