Towards a Theoretician's CHERI (working note)

Thomas Bauereiss, Kyndylan Nienhuis, Peter Sewell

December 12, 2018

1 Introduction

CHERI is an extension for Instruction Set Architectures (ISAs) that adds capability primitives to the underlying base architecture, designed to enable fine-grained memory protection and software compartmentalisation [6, 7]. CHERI adds capability registers, new instructions for manipulating capabilities, and tagged memory that can be used to load and store capabilities.

Such security mechanisms are hard to get right, and even small mistakes in their design or their integration with the base ISA can compromise the desired security properties. Hence, it is crucial to rigorously justify that these mechanisms are sound and cannot be circumvented. In recent work, Nienhuis has formalised key properties of the CHERI-MIPS architecture in a mathematically precise way [3], using a formal model of the instruction semantics specified in the L3 language. The properties have been verified using the interactive theorem prover Isabelle/HOL, with the proofs fully machine-checked.

The CHERI extensions are intended to also be applicable to other architectures, not just CHERI-MIPS. In particular, there is work underway exploring how they can be adapted to, variously, ARMv8-A, ARMv8-M, and RISC-V, and x86 is also discussed as a potential target in the CHERI ISA specification [6]. We would like to formalise and verify security properties of CHERI adaptions of some of those ISAs, analogous to the CHERI-MIPS proof, and also address more involved properties, using the methods of Skorstengaard et al. [5]. However, to help manage the complexity of dealing with real ISAs, to avoid duplication of effort in repeating proof work for multiple architectures, and to help clarify the fundamental properties that the CHERI design embodies, we intend to factor out common parts of the proofs via an abstraction layer that captures essential properties of the capability mechanisms in CHERI architectures. Note that we aim for an *abstraction*, that captures properties that hold in a precise mathematical sense of each full CHERI architecture, not merely an *idealisation*, that could be expressed as a simplified ISA that embodies some CHERI features but has no precise relationship to the full CHERI ISA behavioural specifications. This abstraction layer needs to support proofs of high-level properties, e.g. isolation properties between software components, and it has to be simple enough to keep the proof that each concrete ISA refines the abstract model tractable.

There are different possible choices for such an ISA abstraction. The existing proofs about CHERI-MIPS [3] are centered around abstract actions that represent basic operations involving capabilities: StoreCap, LoadCap, RestrictCap, SealCap, UnsealCap, InvokeCap, StoreData, LoadData. Each of these is associated with properties that capture their intended behaviour. For example, when restricting a capability, the resulting capability must not grant more permissions than the original; when unsealing, the capability used to unseal must have the `Unseal` permission, and so on. These properties become proof obligations when showing that the instructions of the concrete CHERI-MIPS ISA only perform allowed capability actions, and they can then be used to prove higher-level properties.

It is tempting to further factor these actions out into a stand-alone abstraction of CHERI, where the actions become labels of a transition system with an abstract state, and the valid transitions are those satisfying the above properties. However, there is not a one-to-one correspondence between abstract actions and concrete instructions in the ISA in general. In CHERI-MIPS in particular, an ISA instruction can perform multiple capability actions. For example, the CJALR instruction ("Jump and link capability register") not only loads the capability in the source register into the branch delay slot of the PCC register (which holds the program counter capability used for instruction fetching), but also increments and saves the current PCC to the destination register; both of these actions are modelled as instances of RestrictCap. This granularity mismatch (unusual in the sense that abstract actions are more fine-grained than concrete instructions) complicates the proof of abstraction and the transfer of properties between the layers. For example, a temporal property of the form "X until Y" might hold in an abstract trace, but not in a corresponding concrete trace if the first abstract state where Y holds does not correspond to any concrete state.

In [4], Skorstengaard et al. use an abstraction that is much closer to the (idealised) original ISA used there. It has the same instructions as the original, but adds an explicit representation and handling of the stack, such that the desired properties of well-bracketed control flow and local state encapsulation hold by construction. This "overlay semantics" is then shown to be indistinguishable by an adversary from the original, and therefore the desired properties hold for the original as well. In our setting, however, there is not only one concrete ISA, but we want to simultaneously abstract several different CHERI ISAs with different underlying base architectures. Hence, our model has to be significantly more abstract than any single one of the concrete ISAs, which again makes it harder to prove a refinement relation between them and the abstract model.

An alternative is to formulate an abstraction of CHERI in a more axiomatic way, in terms of predicates that all possible executions of instructions have to satisfy. For example, in the proof of the fundamental theorem in [5], such predicates are derived as an intermediate step, summarizing the specific ways in which an instruction may change memory and register contents: a register may either remain unchanged, or be updated either by writing a non-capability value, or copying a capability from another register, or restricting the capability stored in the register, etc. These simple predicates are easy to check for the concrete ISA, yet they have been chosen to be strong enough to allow proving the desired higher-level properties.

In this note, we also follow an axiomatic approach, but the axioms we formulate below assume a somewhat different style of instruction semantics. Instead of using relations between global machine states, we model the set of possible behaviours of an instruction as a set of *event traces*, where each event in a trace represents an effect such as writing a specific value to a register, reading a specific value from a register, loading a data value from memory, etc. This expresses the *instruction-local* semantics of instructions in isolation, factored out from the inter-instruction register and memory semantics.

This style of semantics also has the advantage that it does not commit to a sequential model, where instructions execute atomically and sequentially and the final state of one instruction becomes the initial state of the next one. Such a sequential model is not adequate for multiprocessors with concurrent hardware threads. Instruction-local traces, however, can be interleaved; together with a relaxed memory model specifying which interleavings are allowed we can obtain a concurrent model. This is the style of semantics used in our operational models for relaxed-memory architectural concurrent behaviour.

The Sail tool [1, 2] can generate theorem prover definitions of ISA models with this style of semantics, and substantial and well-validated Sail models exist for several architectures, including CHERI-MIPS, RISC-V, and ARMv8-A. Sail models for smaller (user-mode) fragments of IBM Power and x86 also exist, albeit currently in an older version of Sail.

```
datatype 'regval event =
  | E_read_reg register_name 'regval
  | E_write_reg register_name 'regval
  | E_read_mem read_kind addr nat (memory_byte list)
  | E_read_memt read_kind addr nat (memory_byte list) bitU
  | E_write_ea write_kind addr nat
  | E_write_mem write_kind addr nat (memory_byte list)
  | E_write_memt write_kind addr nat (memory_byte list) bitU
  | E_excl_res bool
  | E_barrier barrier_kind
  | E_footprint
  | E_choose string bool
  | E_print string
```

Figure 1: Event datatype in Isabelle/HOL

2 Instruction-local semantics

Figure 1 lists the events used in our model (as an Isabelle/HOL datatype), including memory and register reads and writes. Reading from memory is represented using either an `E_read_memt` event, for reading a value together with a tag, or `E_read_mem`, for reading just a value; having these two kinds of events allows us to distinguish whether the instruction *intended* to read a tagged value, i.e., a capability, or not, which is useful for formulating axioms about memory reads. When storing to memory, we likewise allow capability sized and aligned writes to explicitly provide a tag bit together with a value (`E_write_memt`), or just write a plain value (`E_write_mem`), in which case the system semantics will clear the appropriate tag bit(s) in memory. Memory stores must be preceded by an announcement of the address (`E_write_ea`), so that, in a concurrent model, program-order succeeding instructions can be informed about the write address as early as it is known. There are also other events needed for concurrent models that can be ignored here: `E_excl_res` for the result of a store-exclusive; `E_barrier` for memory barriers; `E_footprint` for the rare instructions that need dynamically recalculated register footprints; `E_choose` for nondeterministic choice, and `E_print` for printing debugging information when executing Sail models.

Figure 2 shows the semantics for a sample family of CHERI-MIPS instructions, for loads via an explicitly given capability register, as a clause of the Sail `execute` function. For example, the instruction `CLW rd,cb,offset`, which reads 4 bytes from the addresses of capability `cb + offset` into target register `rd`, will execute this with `width=4`, `linked=false`, and `signext=false`.

The Sail code involves several auxiliary functions: `checkCP2usable`, `readCapRegDDC`, `raise_c2_exception`, `wordWidthBytes`, etc. Some of these are pure, while others do a variety of register and memory reads and writes. Leaving aside many auxiliary checks and exception cases, the main events are:

- In Line 3, the `readCapRegDDC` function contains a read of the capability value of the `cb` register. In the instruction-local traces of the instruction, this will give rise to an `E_read_reg` event with an arbitrary value. The whole-system semantics will then cut down to those traces in which this arbitrary value is constrained to be equal to that of the most recent program-order predecessor write to this register.

- In Line 33 (in the non-load-linked case), the `MEMr_wrapper` contains a memory read of the appropriate `size` bytes from the physical address `pAddr` calculated, with address translation, in Lines 14–24. This will give rise to an `E_read_mem` event with an arbitrary value.

- In Line 34 the `wGPR` function contains a register write of the resulting value (sign extended

```
function clause execute (CLoad(rd, cb, rt, offset, signext, width, linked)) =
{
  checkCP2usable();
  cb_val = readCapRegDDC(cb);
  if not (cb_val.tag) then
    raise_c2_exception(CapEx_TagViolation, cb)
  else if (cb_val.sealed) then
    raise_c2_exception(CapEx_SealViolation, cb)
  else if not (cb_val.permit_load) then
    raise_c2_exception(CapEx_PermitLoadViolation, cb)
  else
    {
      let 'size   = wordWidthBytes(width);
      let cursor  = getCapCursor(cb_val);
      let vAddr   = (cursor + unsigned(rGPR(rt)) + size*signed(offset)) % pow2(64);
      let vAddr64 = to_bits(64, vAddr);
      if ((vAddr + size) > getCapTop(cb_val)) then
        raise_c2_exception(CapEx_LengthViolation, cb)
      else if (vAddr < getCapBase(cb_val)) then
        raise_c2_exception(CapEx_LengthViolation, cb)
      else if not (isAddressAligned(vAddr64, width)) then
        SignalExceptionBadAddr(AdEL, vAddr64)
      else
        {
          pAddr      = (TLBTranslate(vAddr64, LoadData));
          memResult : bits(64) = if (linked) then
              {
                CP0LLBit  = 0b1;
                CP0LLAddr = pAddr;
                extendLoad(MEMr_reserve_wrapper(pAddr, size), signext)
              }
            else
              extendLoad(MEMr_wrapper(pAddr, size), signext);
          wGPR(rd) = memResult;
        }
    }
}
```

Figure 2: Sail execution semantics for a sample CHERI-MIPS Instruction

if required) into the general-purpose register `rd`. This will give rise to an `E_write_reg` event with that value.

The instruction-local semantics of the `CLW rd,cb,offset` instruction is thus roughly the set of traces of the form:

$$[\texttt{E\_read\_reg}(\texttt{cb}, c),\ \texttt{E\_read\_mem}(\texttt{Read\_plain}, addr, 4, c'),\ \texttt{E\_write\_reg}(\texttt{rd}, c'')]$$

(still ignoring exceptional cases, which add many more traces) where c is an arbitrary capability value with load permission, with a virtual address (and 4-byte footprint) within its bounds; c' is an arbitrary value; and c'' is the appropriately sign-extended value.

3 Axiomatic model

Our axioms place constraints on what instructions are allowed to do with capabilities and which capabilities are required in order to perform certain operations. We formulate different axioms for different basic actions, i.e. writing a capability to a register, storing a capability to memory, and storing or loading data to or from memory. For example, when an instruction stores a capability to memory, then that capability must be derivable from capabilities that have been legally read by the instruction before. These axioms are parametric in some ISA details, such as the concrete representation of capabilities or the relation between virtual and physical memory addresses. Given an instantiation of these parameters, we can say that an ISA implements our CHERI abstraction if, for any instruction of the ISA, any local event trace possibly generated by that instruction satisfies all of our axioms. We aim for these axioms to be minimal in the sense that they place just enough constraints on instructions to rule out behaviour that would violate the security properties that we are interested in, without constraining the design space for CHERI architectures more than necessary.

We begin formalising these predicates by defining some auxiliary notions, starting with the set of capabilities that are available at a certain point in the execution of an instruction. An instruction may load capabilities from memory if there is another capability available that authorises that load (we formulate this as a separate constraint below). Capabilities may also be read from registers, if those registers are accessible. This includes general purpose registers, but typically excludes certain privileged registers, unless access to the latter is enabled by a permission bit in the PCC capability. For example, the KCC register in CHERI-MIPS contains a capability that is meant to allow executing an exception handler in kernel mode. Hence, this register is not supposed to be accessed by user space processes directly, as that would allow direct access to kernel memory. Which registers are privileged is architecture-specific; for CHERI-MIPS, they include registers such as KCC, KDC, EPCC, or CapCause.

The register reads and memory loads that an execution of an instruction performs are reflected in its event trace, as described in Section 2. Let t be such a trace of a single instruction execution, and let i with $0 \leq i < \mathit{length}(t)$ be an index representing a point in (local) time during this execution of the instruction. Formally, a capability c is read from a register r at index i of t if the event at that position in the trace has the form $\texttt{E\_read\_reg}(r, c)$; similarly for loaded capabilities from memory. We consider system register access to be permitted at index i of t if a tagged and unsealed capability with the `Access_System_Registers` permission has been read from PCC before i (i.e., a some index $j < t$). We define the set of "available capabilities at index i of a trace t" as the set of tagged capabilities that have either been

- loaded from memory before i, or
- read from a register r before i, such that, if r is privileged, then system register access is permitted at index i of t.

This captures the set of capabilities that an instruction may use at a given point. We further constrain it in Axiom 4 below, which requires that capabilities are only loaded from memory if there is another capability already available that authorises the load.

Note that our axioms always consider available capabilities w.r.t. a local trace of a *single* instruction. In a global execution with multiple instructions, each of those may only use the capabilities available in its local trace.

In order to give an upper bound on what instructions may do with capabilities, we define a notion that captures the reflexive-transitive closure of a set of capabilities under the capability manipulations that CHERI allows. This includes the restriction of permissions and memory regions of capabilities, as well as sealing and unsealing, provided that another capability is available that authorises the sealing or unsealing operation. Taking the transitive closure here gives more flexibility than is currently used by existing CHERI implementations, which only perform up to one of those manipulations, but it leaves room for future architectures with more complex capability instructions. Formally, we define the capabilities "derivable from a set of capabilities C" inductively as the set of tagged capabilities c such that

- $c \in C$, or
- c can be derived from another capability c' derivable from C via a restriction operation (so that c has at most the permission bits of c' set, the memory region of c is included in or equal to that of c', and both c and c' are tagged and unsealed), or
- there are capabilities c' and c'' derivable from C such that c' is sealed, c'' authorises unsealing of c', and c is the result of unsealing c' using c'', or
- there are capabilities c' and c'' derivable from C such that c' is unsealed, c'' authorises sealing of c', and c is the result of sealing c' using c''.

This is used in the following axiom, stating that a capability that is stored to memory by an instruction must be derivable from a capability that has been legally read (by that instruction) beforehand:

Axiom 1. *If a tagged capability c is stored to memory at index i of a local trace t of an instruction, then c is derivable from capabilities that are available at index i of t.*

Note that the notion of capability derivation as defined above does not include invocation of capabilities (in a CCall), so the above axiom does not allow storing capabilities to memory that are generated by a CCall and are not derivable from available capabilities via regular unsealing. This is necessary because a CCall involves a non-monotonic crossing of protection domains, and the unsealed code and data capabilities are only intended to be used for executing the jump by writing them to the PC and default data capability registers, and nothing else.

We specify constraints about these jumps in an axiom about register writes. For specifying CCalls, we define the auxiliary notion of an invokable capability pair, consisting of a code and a data capability with matching object type:

Definition 1. An ordered pair of capabilities (cc, cd) is invokable iff

- both are tagged, sealed, have the same object type, and have the `CCall` permission,
- cc has the `Execute` permission and its current pointer is within its memory region, and
- cd does not have the `Execute` permission.

The specification of CCalls and exception handling also has to refer to relevant registers such as the PC and invoked data capability registers, whose names are architecture-specific (in CHERI-MIPS, they are called PCC and IDC, respectively). We again model these as parameters of the abstraction. With these considerations in mind, we formulate an axiom about register writes as follows.

Axiom 2. *If a tagged capability c is written to a register r at index i of a local trace t of an instruction, then one of the following holds:*

- *c is derivable from the available capabilities at index i of t, or*
- *t raises an exception, c has been read from KCC before i (at some index $j < i$ of t), and r is PCC and is not read after i (at any index $k > i$ of t), or*
- *t invokes the capability pair (cc, cd), where cc and cd are derivable from the available capabilities at i, (cc, cd) is invokable, and either*
  - *r is PCC, c is the unsealed version of cc, and r is not read after i, or*
  - *r is IDC, c is the unsealed version of cd, and r is not read after i.*

The newly installed PCC (or IDC) capability is intended for use only by the *following* instructions, after the domain switch has occurred. This means we have to make sure that the current instruction cannot (mis)use these capabilities if it is not supposed to have them available, in particular the sealed arguments of a CCall or the privileged KCC capability in user mode. In order to ensure this, the above axiom requires that PCC and IDC are not read after the invoked capabilities have been written. In combination with the other axioms and our definition of "available capabilities", which only includes capabilities that have been read by the instruction, this forbids the instruction to derive capabilities from these invoked capabilities or use their privileges, if they are not available to the instruction already.

Note that the axioms so far do not forbid an instruction to access privileged registers if it does not have permission to do so. The axioms so far only constrain the *usage* of capabilities read from privileged registers without permission: Since those are not included in the "available capabilities" by definition, an instruction is not allowed to use them in regular capability operations, and Axiom 2 only allows the value of KCC to be copied to PCC for exception handling. The following axiom ensures that this is the only special case by forbiding instructions to read privileged registers without permission in all other cases.

Axiom 3. *If a register r is read at index i of a local trace t of an instruction and r is privileged, then either system register access is permitted at index i of t, or r is KCC and t raises an exception.*

Memory accesses in CHERI are guarded by capabilities; the virtual address range of each load and store has to be within the memory region of an available capability with the corresponding permission. Formalising this in terms of effect traces is somewhat complicated by the fact that the events in those traces carry physical addresses, while capabilities talk about virtual addresses. However, the traces also contain the events belonging to address *translation*. This includes the reading of translation table entries from memory, which can be used to reconstruct the relevant mappings of virtual to physical addresses from the trace. We capture this architecture-specific mapping as a parameter $translate\_address$, a partial function that takes a trace, an index, a flag indicating a load or a store, and a virtual address, and may return a physical address. Note that, if the memory accesses of address translation itself are not guarded by capabilities, we have to exclude them from the axioms below; we model this using a parameter $translation\_tables$ that maps a trace to a set of physical addresses that belong to the translation and should be excluded from the capability requirements.

Axiom 4. *If a data value val of size sz is loaded from a physical memory address $paddr$ at index i of a local trace t of an instruction and $paddr$ is not in $translation\_tables(t)$, then there are c' and $vaddr$ such that*

- *c' is a tagged and unsealed capability with the* `Load` *permission available at index i of t,*

- *vaddr gets mapped to paddr by* $translate\_address(Load, i, t)$,
- *the address range from vaddr up to and including* $vaddr + sz - 1$ *is in the memory region of* c',
- *if the load operation is a tagged memory load, then it is aligned to the tag granularity and* c' *has the* `Load_Capability` *permission.*

An analogous axiom constrains memory stores:

Axiom 5. *If a data value val of size sz is stored to the physical memory address paddr at index i of a local trace t of an instruction and paddr is not in* $translation\_tables(t)$*, then there are* c' *and vaddr such that*

- c' *is a tagged and unsealed capability with the* `Store` *permission available at index i of t,*
- *vaddr gets mapped to paddr by* $translate\_address(Store, i, t)$,
- *the address range from vaddr up to and including* $vaddr + sz - 1$ *is in the memory region of* c',
- *if the store operation is a tagged memory store, then it is aligned to the tag granularity and* c' *has the* `Store_Capability` *permission; additionally, if val is a tagged local capability, then* c' *has the* `Store_Local_Capability` *permission.*

Note that address translation is trusted in this model. In particular, even if we subject all events belonging to address translations to the same constraints as the other events (i.e. guarding them with capabilities), address translation is still trusted to not modify the mapping of virtual to physical addresses in a way that circumvents the capability mechanism (for example, by changing a page table entry for a virtual address range of a userspace process to point into physical memory belonging to the kernel). Hence, the security guarantees of the CHERI mechanisms depend on properties of the virtual memory mechanism, e.g. the invariance of address translation in user space, possibly with additional preconditions. In [3], such an invariant is proved separately, and then used together with the capability abstraction to derive the high-level result.

Our axioms focus on one instruction execution at a time. When considering sequences of instructions, the axioms must still hold for each instruction individually, to avoid capabilities read by one instruction to wrongly be considered available for following instructions, which might be in a different protection domain.

In addition to the semantics of the instructions themselves, our Sail specifications also model instruction *fetching*, so event traces can be obtained for that aspect as well, and subjected to the same axioms. There is one difference in the requirements that CHERI places on data loads and instruction fetches, however: the latter require the `Execute` instead of the `Load` permission. Hence, we have to add a corresponding variant of Axiom 4, which by itself is only meant to constrain data loads.

4 Reasoning above the model

Our axioms are designed to support proofs of fundamental security properties of CHERI architectures. We have formalised our model in the interactive theorem prover Isabelle/HOL and proved a monotonicity property for reachable capabilities: given an ISA that satisfies our axioms, executing an arbitrary instruction does not increase the set of reachable capabilities, if the instruction does not raise an exception or jump to another protection domain by invoking capabilities.

Our proof is with respect to a sequential model that is defined in terms of a global state that holds the values of registers, memory locations, and tag bits. It sequentially executes instructions

according to their local semantics. A read event is allowed to be produced in a given state if the value it contains matches the value of the corresponding register or memory location in that state, while write events update the state accordingly.

We extend the model by state-based variants of trace-based parameters, e.g. *translate_address*. The reachable capabilities are defined similarly to available capabilities in Section 3, but with respect to a state instead of a trace, and we also include derivable capabilities. Formally, we define the set of capabilities reachable in a state s inductively as the capabilities stored in accessible registers in s, or in memory locations in s that are within the memory region of a reachable capability with the `Load_Capability` permission, or derivable from a reachable capability.

We show that, for any instruction-local trace t of an arbitrary instruction that can be executed in a state s resulting in a new state s', if

- no exception is raised in t,
- no capability pair is invoked in t,
- address translation stays invariant along t, and
- the in-memory translation tables in s do not contain tagged capabilities,

then the set of capabilities reachable in s' is a subset of the capabilities reachable in s.

Invariance of address translation ensures that the instruction cannot re-map a reachable virtual memory region to physical memory with additional capabilities. Since address translation is architecture-specific, it is not covered by our axioms and has to be reasoned about separately. In CHERI-MIPS, invariance of address translation always holds in user space, whereas in ARMv8-A, it only holds if the in-memory translation tables are set up suitably by the operating system.

The final assumption about capabilities in translation tables is due to the fact that our axioms do not cover translation table lookups, again in order to avoid having to model architecture-specific details of address translation. If one wants to allow capabilities in translation tables, then these lookups should be subject to the same capability checks as regular memory loads.

We plan to continue proving high-level properties above our model, e.g. isolation properties between software compartments. We also plan to go in the other direction and show that concrete CHERI ISAs satisfy our axioms.

5 Conclusion

We have discussed an axiomatic model of essential properties of the capability features in CHERI architectures. Its requirements are designed to be both easy to verify for concrete ISAs and to enable abstract proofs of high-level properties. In particular, we have used the axioms to show a monotonicity property for the capabilities reachable within a compartment. This can be used to rule out unauthorised privilege escalations by arbitrary code running in a compartment. It has to be noted, though, that there are desirable properties in CHERI-based systems for which our abstraction is not sufficient. Consider, for example, operating system code that is initially started with an all-powerful capability, and suppose that we want to prove that this code only passes sufficiently restricted capabilities to user-space processes so that those cannot write to kernel memory. This requires detailed reasoning about the exact capability restrictions performed, including calculations of bounds and pointers using instructions of the base ISA. Hence, such a proof has to include ISA details that we specifically want to abstract away from, so the kernel code will have to be reasoned about in more detail outside of our model. However, our model can be used to reason about arbitrary code running in user-space processes, complementing the verification of kernel code.

References

[1] The Sail ISA semantics specification language, 2018. http://www.cl.cam.ac.uk/~pes20/sail/.

[2] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. In *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages*, January 2019. Proc. ACM Program. Lang. 3, POPL, Article 71.

[3] Kyndylan Nienhuis, Alexandre Joannou, and Peter Sewell. Proving security properties of CHERI-MIPS. In *Proc. Automated Reasoning Workshop*, page 18, April 2018. Two-page abstract. http://www.cl.cam.ac.uk/~pes20/cheri/cheri-arw18.pdf.

[4] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. StkTokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities. Submitted. Available at http://cs.au.dk/~lask/papers/stktokens-wbcf-lse.pdf.

[5] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. Reasoning about a machine with local capabilities – provably safe stack and return pointer management. In *ESOP 2018: 27th European Symposium on Programming, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018*, pages 475–501, 2018.

[6] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, David Chisnall, Brooks Davis, Nathaniel Wesley Filardo, Alexandre Joannou, Ben Laurie, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alex Richardson, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 7). Technical Report UCAM-CL-TR-927, University of Cambridge, Computer Laboratory, 2018.

[7] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI capability model: revisiting RISC in an age of risk. In *ISCA '14: Proceeding of the 41st International Symposium on Computer Architecture*, pages 457–468, Piscataway, NJ, USA, 2014. IEEE Press.