Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process

Kyndylan Nienhuis*, Alexandre Joannou*, Thomas Bauereiss*, Anthony Fox†, Michael Roe*, Brian Campbell‡, Matthew Naylor*, Robert M. Norton*, Simon W. Moore*, Peter G. Neumann§, Ian Stark‡, Robert N. M. Watson*, and Peter Sewell*

*University of Cambridge †ARM Limited ‡University of Edinburgh §SRI International

***Abstract*—The root causes of many security vulnerabilities include a pernicious combination of two problems, often regarded as inescapable aspects of computing. First, the protection mechanisms provided by the mainstream processor architecture and C/C++ language abstractions, dating back to the 1970s and before, provide only coarse-grain virtual-memory-based protection. Second, mainstream system engineering relies almost exclusively on test-and-debug methods, with (at best) prose specifications. These methods have historically sufficed commercially for much of the computer industry, but they fail to prevent large numbers of exploitable bugs, and the security problems that this causes are becoming ever more acute.**

In this paper we show how more rigorous engineering methods can be applied to the development of a new security-enhanced processor architecture, with its accompanying hardware implementation and software stack. We use formal models of the complete instruction-set architecture (ISA) at the heart of the design and engineering process, both in lightweight ways that support and improve normal engineering practice – as documentation, in emulators used as a test oracle for hardware and for running software, and for test generation – and for formal verification. We formalise key intended security properties of the design, and establish that these hold with mechanised proof. This is for the same complete ISA models (complete enough to boot operating systems), without idealisation.

We do this for CHERI, an architecture with *hardware capabilities* that supports fine-grained memory protection and scalable secure compartmentalisation, while offering a smooth adoption path for existing software. CHERI is a maturing research architecture, developed since 2010, with work now underway on an Arm industrial prototype to explore its possible adoption in mass-market commercial processors. The rigorous engineering work described here has been an integral part of its development to date, enabling more rapid and confident experimentation, and boosting confidence in the design.

I. Introduction

Despite decades of research, memory safety bugs are still responsible for many security vulnerabilities [1]. Microsoft estimates that 70% of the vulnerabilities they have patched between 2006 and 2018 are caused by memory safety issues [2], MITRE considers classic buffer overflows as the third most dangerous software error [3], and high-profile memory-safety bugs such as Heartbleed [4] have become common.

There are two fundamental problems here. First, mainstream hardware architectures and C/C++ language abstractions provide only coarse-grained memory protection, via the memory management unit (MMU). This is hard to change: the mass of legacy C/C++ code makes it infeasible to migrate everything to a type-safe language, or to radically change hardware architectures, but introducing fine-grained memory protection in software, e.g. with bounds-checking, is often too inefficient.

Second, mainstream systems are typically developed using test-and-debug engineering methods. While this often suffices to build systems that are sufficiently functionally correct under normal use, it fails to build secure systems: it is easy to miss a small mistake that manifests itself only in a corner case, but attackers will actively try to find these, and one small bug can compromise the entire system.

CHERI is an ongoing research project that addresses the first problem with hardware support for fine-grained memory protection and scalable software compartmentalisation, aiming to provide practically deployable performance and compatibility [5]–[7]. CHERI achieves this by extending commodity architectures with new security mechanisms, and adapting a conventional software stack to make use of these.

This paper addresses the second problem: we show how more rigorous engineering methods can be used to improve assurance and complement traditional methods, using the CHERI project as a whole as a testbench for this. These include both lightweight methods – formal specification and testing methods that provide engineering and assurance benefits for hardware and software engineering without the challenges of full formal verification – and more heavyweight machine-checked proof, establishing very high confidence that the architecture design provides specific security properties.

A. The CHERI Context

The CHERI design is based on two principles. The *principle of least privilege* [8] says that each part of a program should run only with the permissions it needs to function. For example, a conventional C/C++ program implicitly uses permission to its entire memory region for accesses via a pointer, making it vulnerable to buffer overflows, but in CHERI it can be limited to the permission to access the pointed-to object. On a larger scale, the JavaScript execution engine of a browser does not need access to the encryption keys used in SSL connections, so by restricting its permissions one can ensure that even a compromised JavaScript engine cannot access these keys. The *principle of intentional use* states that when a program performs an action, it should explicitly state which permissions it uses to authorise that action. This helps avoid the confused deputy problem [9]. For example, an operating system (OS) may need to hold permission to access the entirety of a user process (e.g. for paging), but when that process makes a `read()` system call, passing a pointer to a user-space buffer, CHERI lets the relevant OS code to use the more restricted permission to just that buffer. The CHERI design involves no lookup or search for permissions, which is important both to embody this principle and for performance.

These principles are realised in CHERI with *hardware capabilities*: to access memory, one needs to possess a capability that *authorises* that access. A capability augments the usual virtual address of a language-level pointer with bounds that specify the memory region it relates to, and with permission bits that specify what kind of actions it can authorise. To distinguish capabilities from other data, CHERI uses *tagged memory*, one bit for each capability-sized and aligned unit of memory. Valid capabilities have their tag set, and if they are overwritten with a non-capability, their tag is cleared, rendering them safely unusable. CHERI introduces new instructions that can manipulate a capability without clearing its tag: to change its virtual address, shrink its bounds, decrease its permissions, or copy it. It is crucial to the design that no instruction can grow the bounds of a capability, or add permissions. To enable software compartmentalisation, CHERI provides additional mechanisms for mutually untrusting compartments to communicate.

The CHERI design has been elaborated initially as CHERI-MIPS, extending 64-bit MIPS [10] with capabilities. Our CHERI-MIPS instruction-set architecture (ISA) definition is the central design artefact [6]. As usual, the ISA defines the hardware/software interface: the programmer-visible machine state and instruction behaviour that software must be programmed above; it abstracts from hardware-implementation (microarchitectural) specifics, e.g. of pipelines, cache hierarchies, etc. Then there is a Bluespec [11] FPGA hardware implementation of the architecture, and a software stack above it, adapting LLVM [12] and FreeBSD [13] to CHERI-MIPS. All this has involved extensive work on the interaction between the capability system and systems aspects of memory management (static and dynamic linking, process creation, context switching, page swapping and signal delivery) [14]; on the overhead of compiling pointers to capabilities [5], [15]; on compartmentalisation of legacy software [16]; and on the performance overhead of tagged memory [17] and protection-domain switches [18]. The underlying ideas are portable, not MIPS-specific, and work is underway on experimental academic RISC-V and industrial Arm versions – the latter in a major project involving Arm and the UK Government [19], [20] to produce prototype silicon and a development board.

B. The Problems with Traditional Engineering Methods

CHERI-MIPS was initially developed using traditional engineering methods: the security properties that the architecture is intended to enforce were described in prose; the architecture was described in prose and pseudocode; and the hardware implementation of the architecture was validated with hand-written tests. This led to the following problems, often accepted as inescapable aspects of normal practice.

First, the prose and pseudocode architecture description was not executable as a test oracle, to check the hardware implementation behaviour against, or as an emulator, to run software above. Hardware validation tests therefore required manual curation of their intended outcomes, so tests could not be automatically generated, and it was not possible to rerun software tests after changes in the architecture until the hardware implementation was updated.

Second, while the designers had strong intuitions about the security properties the architecture should provide, their prose statements were inevitably less precise than they could be, and omitted crucial details. This led to unnecessary confusion in internal discussions of (for example) the concepts of capability provenance and monotonicity, central to the design, and made it harder to explain these to others.

Third, it was very difficult to assess whether the architecture actually provided these intended security properties. The properties involve quantification over all executions of arbitrary code, e.g. to show that (without a domain transition) the reachable capabilities can never be increased, and that an isolated compartment cannot affect other memory. Testing cannot provide assurance that these hold. Moreover, architecture descriptions are large and complex: the CHERI ISA description is around 7k lines, and involves intricate manipulation of capabilities and of the architected register state (see Fig. 4 for just one instruction). For a conventional architecture, to check, for example, that user code cannot bypass the MMU, it would be easy to check that all user-mode memory accesses go via address translation, but the CHERI security enforcement is intertwined with the computation and use of memory addresses, user/system permissions, and other aspects of the ISA; its security properties depend on subtle invariants of the ISA specification as a whole. A mis-specification of the interaction between capabilities and user/system modes, exceptions, address translation, or any of the 180-odd instructions could potentially break its security protection. This means that informal review of the ISA specification also cannot provide assurance that the properties hold – as confirmed by the subtle bugs found by our proof work (§VI). At the same time, CHERI's correctness is crucially important: its security protection will come under direct attack, and one mistake could be enough to cancel out all the protection that CHERI offers. Moreover, if the architecture specification is flawed, containing some security vulnerability, then any conforming hardware implementation will inherit that vulnerability. All this makes machine-checked statements and proofs of the intended security properties essential.

C. Contributions

We address these problems with three contributions.

1) Lightweight rigorous engineering (§III): We show that prose-and-pseudocode ISA descriptions can be replaced in the architecture design process with rigorous definitions, which are at once clearly readable, executable as test oracles, and support automatic test generation. Importantly, developing such formal definitions for CHERI-MIPS did not require any formal background, so the researchers and engineers who would otherwise write a prose/pseudocode architecture document could write and own them. At the same time, expressing the specifications in terms of instruction description languages, L3 [21] and (more recently) Sail [22], that automatically generate theorem-prover definitions, ensured that the ISA definitions are mathematically rigorous, with no ambiguity. Formal specification alone is widely understood to bring low-cost clarity benefits, even before any proof work is undertaken. Here, our (more unusual) emphasis on specifications that are executable as test oracles [23] goes beyond that, and was essential for the integration of formal specification with traditional engineering. In turn, that ensured the specification was a conventionally well-validated "live artifact": the actual ISA design, rather than a (possibly divergent) definition produced after the fact. This ensured that the following properties and proof are about the actual design.

2) Stating architectural security properties (§IV): We show how the intended security properties of the architecture can be precisely and formally stated, in terms of the above definitions of instruction behaviour. Crucially, we express these properties in relatively simple terms, using basic operations over sets and quantified formulas, so understanding them does not require extensive formal background. This makes it possible to unambiguously communicate the guarantees that CHERI-MIPS offers to the relevant audiences: to software users of the architecture, so they understand the limits of its protection; to hardware engineers, so their implementations do not accidentally break said guarantees; and among the authors of the architecture themselves, so they can ensure that the guarantees are as intended.

The properties we define for CHERI-MIPS (1) capture the memory access intentions of the instructions, (2) capture a reachable-capability monotonicity property, that arbitrary code, if given some initial permissions, cannot increase those during its execution (up to the point of any domain transition); (3) capture the property that arbitrary code, if not initially given permission to access particular system registers and memory regions, leaves them invariant as that code is executed; and (4) capture the guarantees one has (and the required assumptions) when executing an untrusted subprogram within a controlled isolation boundary. Our properties cover executions running in user or kernel mode, and can therefore be used to reason about compartmentalised kernels.

3) Formal proofs of security properties (§V): Finally, we show how to mathematically prove these security properties, with machine-checked proof, in a way that is scalable to the entire ISA and that can be integrated in the ISA design process. This gives high confidence that they hold, which is not achievable with testing. Novel proof automation and automatic generation of lemmas were essential to make this scale to the entire ISA specification, and to make the proof robust in the face of design evolution.

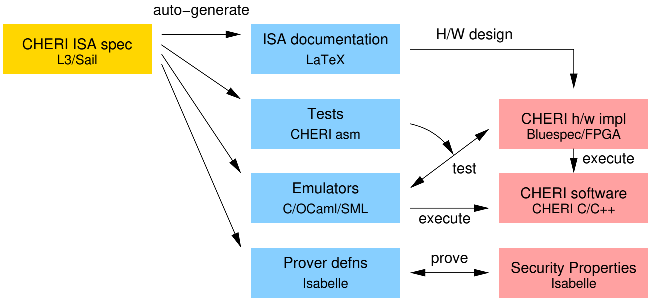
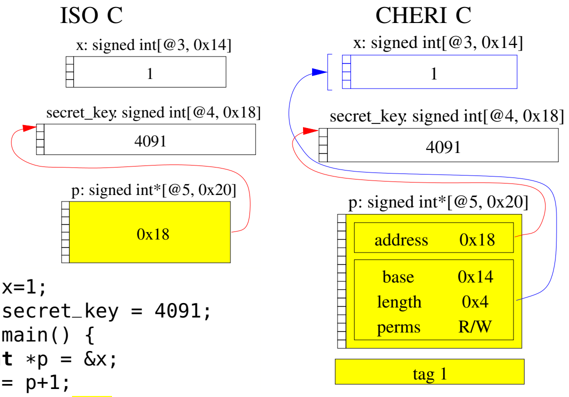


Fig. 1. The main artifacts of the CHERI engineering process. Those in the central column are all automatically generated from the L3/Sail formal ISA specifications. The CHERI hardware design is tested against the generated emulators, using both auto-generated and (not shown) manually written tests. The CHERI software stack, including adaptions of Clang and FreeBSD, is developed by running above the generated emulators, the hardware, and (not shown) a QEMU emulator. The security properties are stated and proved in terms of the automatically generated Isabelle version of the ISA specification.

Fig. 1 illustrates the main artifacts of the CHERI engineering process. We have done this for CHERI-MIPS, but the same problems exist for other security architecture features, e.g. Intel SGX and ARM TrustZone, and we believe similar solutions could be applied. CHERI started off with a hardware/software co-design approach, which is already unusual but is necessary for improving security at the architectural interface. Here we show the benefits of a three-way *hardware/software/formal-semantics* co-design approach.

D. Non-goals and Limitations

Our focus here is on the definition and proof of specific security properties of the CHERI *architecture*: the specification of the hardware/software interface. This is an essential step in increasing confidence in CHERI-MIPS, but we have not, of course, proved that CHERI as a whole "is secure" (which, unqualified, has no precise meaning). Our work has helped validate the CHERI-MIPS hardware implementation, but it does not prove the hardware implementation correct with respect to the architecture, or prove correctness or security properties of system software above the architecture (though it is a necessary precondition for any such proofs). Our security properties only address behaviour that is visible at the architectural abstraction. As usual, this abstracts from timing behaviour and power consumption, so our properties cannot talk about possible side-channel information flow via these. How to manage side channels remains an open research area, but the isolation properties that we establish are certainly necessary, even if not sufficient, for whole-system security. There is ongoing non-formal work exploring side-channels w.r.t. CHERI [24]. Moreover, the architecture is an intention-



```
int x=1;
int secret_key = 4091;
int main() {
  int *p = &x;
  p = p+1;
  int y = *p ;
  printf("%d\n",y); }
```

Fig. 2. Fine-grain memory protection, comparing ISO C with CHERI C. In ISO C the (flawed) program on the left has undefined behaviour, but in practice can leak the secret key. In CHERI C the same program has defined behaviour and will always trap with a hardware exception, because the address used for the `*p` access is not within the footprint of the capability.

ally loose specification, to admit implementation variation, and therefore our properties also cannot talk about *architecturally visible* information flow: a (compromised or adversarial) hardware implementation could leak information while conforming to the architecture by exploiting this looseness. Our properties do exclude architecturally visible *capability* flow, which is likewise necessary, but not sufficient, for whole-system security. Finally, the semantics only covers the uniprocessor case. The capability system is largely orthogonal to concurrency, but lifting the reasoning to a realistic relaxed memory model would add significant complication.

II. Background: CHERI

CHERI aims to prevent or mitigate many security vulnerabilities. It does so by extending commodity instruction-set architectures with a capability system, enabling fine-grained memory protection and scalable software compartmentalisation, while offering a practical gradual adoption path for existing software. In this section we explain the main aspects of the CHERI capability system, with two examples. CHERI also supports a range of other use-cases with additional features that we cannot explain here; for details of those see [6].

A. *Fine-grained Memory Protection*

As highlighted at the start, many security vulnerabilities involve memory safety violations arising from coding errors in unsafe languages. The CHERI hardware extensions make it possible to implement unsafe languages, notably C and C++, in ways that enforce spatial memory safety – thus preventing or mitigating many vulnerabilities – while aiming to keep performance and code-porting costs acceptable.

For example, consider the C program in Fig. 2. This declares a `secret_key`, but the programmer does not intend that the `main()` function access that, let alone leak it. However, a coding error introduced code that creates a pointer `p` to another global, `x`, increments `p`, and dereferences it with `*p`. In ISO C this program has undefined behaviour, because the `*p` access is to a location outside the `x` allocation, meaning that conventional C implementations can assume that programs do not contain such accesses (implementations are not required to trap or otherwise detect the error). In practice, however, the machine representation of a pointer is typically just an integer address, and a conventional implementation will typically output whatever is next to `x` in the memory layout, which here can be, and often is, `secret_key`. The left of Fig. 2 illustrates one such execution, with `x` allocated at 0x14 and `secret_key` at 0x18. After the increment, the value of `p` is just 0x18; a conventional C compiler will generate a simple machine load instruction for the read `*p`, and a conventional processor implementation will simply load from that address – turning that coding error into a security leak.

In CHERI, however, one can compile C or C++ source to represent pointer values with capabilities instead of integer addresses, as shown on the right of Fig. 2. On a 64-bit CHERI architecture, the value of `p` is a 256-bit or 128-bit *capability*, together with a 1-bit validity tag. The 256-bit format includes a 64-bit virtual address, base, and length; permission bits, to execute, load, and store data, to load, store, seal, and unseal capabilities, to invoke sealed capabilities, and to store local capabilities; an is-sealed bit; and a 24-bit object type, used in sealing, unsealing, and invoking. The 128-bit format [15] compresses this, exploiting the redundancy from allocation alignment (the proofs later in this paper are about the 256-bit version). In this example, the base and length, identifying the memory region that the capability is allowed to access, are those of the original `x` allocation. C pointer arithmetic is compiled to instructions that change capability virtual addresses, leaving their allowed memory regions unchanged. All this means the hardware can do an efficient access-time check, at the `*p` dereference, that the capability *authorises* the memory access: namely that the access is within the capability's memory region and the type of access is allowed by its permissions. The details of this check are part of the ISA definition, and specific to each operation – for example, Fig. 4 shows the check for an integer load via a capability register, and Fig. 5 shows the check for sealing a capability. If the check fails, the hardware deterministically traps. So far, this is similar to software fat-pointer designs [25]–[27], but CHERI capabilities are protected from accidental or malicious modification: an additional *tag* bit, one per capability-sized and aligned region of memory, keeps track of whether that memory holds a valid capability. The hardware preserves the tag if valid capability instructions are used, but clears it otherwise (e.g., if individual bytes are written); tags are not independently addressable. The ISA is designed so that no instruction can increase the access rights of a capability. The hardware provides a universal capability at start-up, and the OS, linker, compiler-generated code, and language runtime allocators gradually construct smaller capabilities; in this example, the linker constructs precise capabilities for the globals. Capabilities are also used to protect non-source pointers, including code pointers such as PC values and return addresses.

This example illustrates the principles of least privilege and intentional use: the program as a whole has the permission to load the secret key, but it was not the programmer's intention to do this when dereferencing `p`. In CHERI C this intention is preserved and the capability which is the value of `p` has the least privilege required to access `x`, so it is impossible to load the key through that memory access.

Porting legacy software to CHERI is eased by the fact that source changes are needed only in rare cases, e.g. where code manipulates the representation bytes of pointers explicitly. This example requires none, just a re-compile. In other work we have ported FreeBSD user-space to CHERI and analysed the changes required [14], [28]; we have also ported other software, including WebKit. Porting does involve an ABI change, as pointer sizes change, but one can also compile in a hybrid mode, with only selected pointers represented with capabilities, or encapsulate legacy code compiled in the normal way by running it with default data and code capabilities.

The above provides spatial but not temporal memory safety, but the fact that CHERI pointers and integers can be reliably distinguished creates new possibilities for temporal enforcement, currently being investigated.

B. *Software Compartmentalisation*

At a larger granularity, and especially when running untrusted binaries, or code for which the porting or performance costs of CHERI C (modest though they are) are not acceptable, software compartmentalisation can help to make a system more robust against attacks by mitigating the effects of successful exploits. Conventional systems do this with operating-system processes and hypervisor virtual machines, using hardware memory-management-unit (MMU) protection, managed by trusted OS or hypervisor code, to enforce memory isolation (or confinement) between arbitrary untrusted binaries. However, MMU protection scales poorly with larger numbers of compartments and operates only at page granularity, limiting its applicability.

CHERI supports scalable compartmentalisation by extending the core capability system described above with additional mechanisms for controlled communication between mutually untrusting compartments. Memory isolation is achieved with the same mechanism as above: if a compartment does not possess (and is not passed) a capability with permission to access a region of memory, CHERI guarantees that it cannot access that region. Controlled communication between compartments is achieved as follows. A compartment can use a `CSeal` instruction to *seal* a capability with an *object type* (a 24-bit field within the 256-bit capability), which makes it temporarily unable to authorise memory accesses. Concretely, sealing just sets the *sealed* bit and the object type within the capability (see Fig. 5 in the Appendix). A pair of sealed capabilities, one for code and one for data, both with the same object type, can be used to transition to another domain. CHERI supports secure domain transition with an instruction to *invoke* the pair of sealed capabilities: this jumps directly to the address of the sealed code capability, and atomically unseals both the code and the data capability (see `CCallFast`, in Fig. 6 in the Appendix). The authority of a sealed capability can also be restored by *unsealing* it.

For example (Fig. 3), *libcheri* within CheriBSD [14] provides an interface for sandboxed compartments to perform I/O operations. Upon initialisation, a sandbox receives a sealed code capability c_C to a particular entry point within *libcheri*, and a sealed data capability c_D to the local data of *libcheri*, both sealed with some particular object type o. These sealed capabilities do not authorise direct access to code or data of *libcheri*, but a sandbox can invoke them to jump to that specific entry point. When doing so, it can also pass a capability c_B to a specific buffer of its own local data, e.g. containing a string to be printed. After the jump, *libcheri* is in control, and can access its own data using the now-unsealed data capability. The sealing and unsealing of capabilities is itself capability-controlled; for the above to be secure it is essential that sandboxes are set up without the capability to unseal object type o. Theorem 11 shows, for example, that if *libcheri* and the sandbox are set up without other capabilities to each other's memory regions, then during the execution of the sandbox it cannot change code or data of *libcheri* or vice versa, with the exception of explicitly shared buffers.

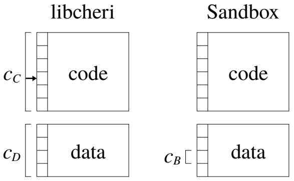


Fig. 3. A sandbox and *libcheri* sharing only restricted or sealed capabilities

In some compartmentalisation scenarios one trusts compartments to not leak their own capabilities, but in others one wants to prevent (malicious or compromised) compartments from cooperating. To support that, CHERI defines separate permissions for storing/loading data and for storing/loading capabilities. For example, to allow two compartments to communicate plain data while preventing them from exchanging their capabilities, one could give them authority to load and store data (only) to a shared region. Additionally, CHERI has a mechanism to allow some capabilities to be shared but not others: capabilities can be flagged as *local* (as opposed to global). To store local capabilities one needs an additional permission. By setting up compartments with just the authority to read capabilities, and to store global capabilities to a shared region, they cannot exchange their local capabilities.

These mechanisms can be used for secure encapsulation at various scales, from protection of individual C++ objects that call each other's methods, to vulnerability-prone compression or media code libraries, to whole processes within the same address space, protected from each other using CHERI alone rather than with MMU protection.

III. Lightweight Rigorous Engineering

Formal specifications, in security and elsewhere, are often introduced solely to support mechanised proofs, and the skills and techniques this needs often makes formal specification divorced from normal engineering practice. For example, much (though not all) of the large literature on security protocol verification only addresses abstract models of protocols, disconnected from their actual implementations, and its techniques are often not accessible to the engineers who code those.

In contrast, in bringing rigorous techniques to bear on the CHERI secure architecture design, an important goal was to support the security, architecture, and operating systems researchers and developers involved, who are not theorem-prover experts, by fitting in with and complementing their normal engineering practice – while simultaneously enabling formal statement and proof of security properties of their actual architecture design, not just some idealised model thereof. We aimed to improve their engineering practice in multiple lightweight ways, with immediate benefits long before the formal proof was complete.

Industrial processor architecture specifications, including AMD64, IBM POWER, Intel 64, MIPS, RISC-V, and SPARC, usually define their envelopes of programmer-visible allowed behaviour with documents containing a mix of prose and pseudocode [29]–[33]. These are typically multi-thousand-page books containing masses of detail, about instruction behaviour, encodings, address translation, interrupts, etc. They are not computational artefacts, so vendors typically also develop internal “golden” reference models to use as oracles for hardware testing, often in conventional programming languages such as C++. Arm, exceptionally, have recently transitioned to a machine-processed pseudocode, so what is in their manual can actually be tested against [34].

A reference model could be written in almost any language, but a rigorous architecture specification should be clear enough to also use as readable documentation of instruction behaviour. For this, it is desirable to use as simple a language as possible, close in appearance to the imperative pseudocode common in industrial architectures, with intentionally limited expressiveness compared to general-purpose languages such as C or C++. Many such Instruction Definition Languages (IDLs) have been developed [35]. Then, for use in testing, it has to be possible to execute the definitions efficiently enough, and to support proof, the IDL must have a straightforward semantics that can be mapped directly into the input languages of theorem provers.

For CHERI, we started off in 2011 with traditional pseudocode descriptions, together with experimental formal modelling (unpublished) of key instructions in PVS [36]. In 2014, starting the work reported on in this paper, we began work on complete formal models in the L3 [21] IDL, and more recently in Sail [22], partly developed with CHERI in mind.

L3 and Sail are both strongly typed, first-order imperative language that aim to be accessible to engineers without a formal background. To illustrate this, Fig. 4 shows the L3 specification of the CHERI-MIPS `CLB rd, rt, offset(cb)` instruction, to load a byte via the capability in register `cb`. This takes a capability from `cb`, calculates an effective address (Line 15) by summing its virtual address, the value in register `rt`, and the sign-extended `offset`, uses that to load a byte from memory (Line 29), and writes the appropriate value (possibly sign-extended) back into `rd`. The instruction checks several conditions, e.g. (Lines 5–10) that the capability has its tag set, is not sealed, and has permission to load, and (Lines 23–27) that the address is within the bounds of the capability. Together, these define what it means for a capability to authorise this load. If any of the conditions fail, the instruction raises a hardware exception.

```
 1 define CLoad (rd::reg, cb::reg, rt::reg, offset::bits(8),
 2              s::bits(1), t::bits(2)) =
 3  if not CP0.Status.CU2 then
 4    SignalCP2UnusableException
 5  else if not getTag(CAPR(cb)) then
 6    SignalCapException(capExcTag,cb)
 7  else if getSealed(CAPR(cb)) then
 8    SignalCapException(capExcSeal,cb)
 9  else if not getPerms(CAPR(cb)).Permit_Load then
10    SignalCapException(capExcPermLoad,cb)
11  else {
12    cap_cb = CAPR(cb);
13    cursor = getBase(cap_cb) + getOffset(cap_cb);
14    extOff = (([offset<7>]::bits(1))^3:offset) << [t];
15    addr = cursor + GPR(rt) + SignExtend(extOff);
16    var size; var access; var bytesel = '000';
17    match t {
18      case 0 => {
19        size <- 1;
20        access <- BYTE;
21        bytesel <- addr<2:0> ?? BigEndianCPU^3 }
22      [...OTHER CASES ELIDED, FOR 2,4,8-BYTE LOADS...] };
23    if ('0':addr) + ('0':size) >+
24       ('0':getBase(cap_cb)) + ('0':getLength(cap_cb)) then
25      SignalCapException(capExcLength,cb)
26    else if addr <+ getBase(cap_cb) then
27      SignalCapException(capExcLength,cb)
28    else {
29      data = LoadMemoryCap(access, true, [addr], false);
30      when not exceptionSignalled do {
31        data_list = [data]::bool list;
32        bottom = ([bytesel]::nat)*8;
33        top = ([bytesel]::nat)*8 + ([size]::nat)*8 - 1;
34        final_data = data_list<top:bottom>;
35        if s == 0 then
36          GPR(rd) <- [ZeroExtendBitString(64, final_data)]
37        else GPR(rd) <- [SignExtendBitString(64, final_data)]
38  } } }
```

Fig. 4. The L3 specification of the `CLB rd, rt, offset(cb)` *Load Integer via Capability Register* instruction, and variants.

The complete model includes everything that is necessary to boot an operating system, including exceptions, the translation lookaside buffer (TLB), and the programmable interrupt controller (PIC). The Sail specification is broadly similar; Sail differs from L3 mainly in providing a rich but decidable type system, with lightweight dependent types to let computed bitvector lengths be statically checked. The L3 and Sail versions are each around 7k lines of specification.

A. *Using the Models as Design Documents*

Our first lightweight use of these rigorous models is as improved design documentation. L3 and Sail parse and type-check their input, immediately catching errors that are easy to make in non-mechanised pseudocode specifications, and they allow no ambiguity. For example, Fig. 4 makes clear exactly what checks are done, which exceptions will be flagged if they fail, and the priority among these. The Sail specifications of each instruction are now (2019) included verbatim in the CHERI architecture document [6], replacing earlier informal pseudocode that had to be maintained separately.

B. Using the Models as Oracles for Hardware Testing

The second lightweight use is as oracles for testing our Bluespec FPGA hardware implementations of CHERI processors against. L3 and Sail both automatically generate emulators from ISA models, variously in SML, OCaml, and C. For CHERI-MIPS these run at 300–400 KIPS, which is fast enough to boot FreeBSD in around four minutes, and to run many hardware tests. Being able to automatically re-run such tests against the architecture specification in a continuous integration environment has been invaluable, as both the architecture and the hardware implementations have been developed in parallel.

The simplicity of the L3 and Sail IDLs enabled Computer Architecture and Security researchers to directly edit and own the CHERI models, and to automatically see the benefits in the continuous integration setup. In turn, this assisted in updating the other artifacts in the project (FPGA implementation, software unit tests, etc.), which ultimately led to the adoption of this approach by the whole research team.

Furthermore, executable specifications make it easier to experiment with design alternatives, as one can compute their behaviours without needing micro-architectural implementations. For example, early exploration of compression schemes for CHERI capabilities and their potential architectural impacts were explored in the CHERI L3 model. We also extended the models with some microarchitectural details, such as a cache hierarchy, to rapidly explore potential uses of CHERI capabilities within the memory subsystem. This is also an appealing feature to a Computer Architecture researcher.

We also wrote a traditional QEMU [37] CHERI emulator, by hand. This was useful, with 100x the speed of the L3 model, and support for many devices, but it was in C, in a not particularly readable style, and was quite error prone.

The L3 model and the original prose ISA specification do have some intentional subtle differences, to ease comparison of hardware-implementation and L3 model traces in a few cases where the former is non-deterministic. For example, the `TLBWR` instruction architecturally writes a random TLB entry, but our hardware implementation writes a certain entry based on a counter, and the L3 model follows that.

C. Using the Models for Software Bring-up

Being a whole-system project, CHERI involves extensive hardware and software work by different sub-teams. This means that when something goes wrong, it may not be clear whether the hardware is failing to meet the spec, or the software is making an invalid assumption. A big benefit of the architecture specification being executable is that it can help answer this question. For example, when first bringing up multicore CHERI on FPGA, FreeBSD was getting stuck late in boot. It seemed most likely that this was a hardware issue in the new cache coherency mechanism, but we could use our recently-developed L3 model to reproduce the problem, suggesting a software issue. After exploring the trace, we identified a kernel bug in which the programmable interrupt controller was being mapped to an incorrect address, preventing inter-processor interrupts. Simply narrowing down the source of the problem saved many futile hours of painful hardware debugging. We now routinely bring up new software, e.g. CHERI compiler support, above the formal models.

D. Using the Models for Test Generation

Our initial hardware development relied on a manually (and painfully) written test suite. With authoritative formal models, it is no longer necessary to manually specify the intended outcomes of tests, as one can simply compare hardware vs model running arbitrary code, so it becomes possible to autogenerate tests. To generate random sequences of instructions that achieve good coverage in the presence of a large number of security tests in the capability instructions, it was important to control which instructions could generate a processor exception and why. We used a combination of symbolic execution of the L3 specification and automatic constraint solving [38] to find an initial processor state where only the chosen instruction would fault. A few thousand tests generated in this way were sufficient to cover almost all of the instruction behaviour in the specification. These revealed discrepancies between the L3, the hardware, and the QEMU simulator, including bugs in the modelling and simulation of delay slots and exceptions (one in the upstream QEMU MIPS), and a security-relevant bug in the hardware. This automatic technique was easily adapted to changes as the ISA was developed.

IV. Stating Architectural Security Properties

Formal security properties have two main benefits over prose properties. First, prose properties are prone to ambiguities, which may lead to security vulnerabilities if users, designers, and implementers misunderstand each other. Formalisation helps by forcing one to identify and resolve these ambiguities. Second, it is hard to establish that prose properties actually hold, as they are not susceptible to either experimental validation (by testing or model-checking) or mathematical proof. It is possible to formally state properties about L3 or Sail specifications because these can be automatically exported to theorem prover definitions: variously Isabelle/HOL [39], HOL4 [40] and/or Coq [41].

To illustrate the first benefit, we identify ambiguities in the prose definition of a fundamental property of CHERI's capability system, namely *capability monotonicity*. The prose documentation defines this as the property that "new capabilities must be derived from existing capabilities only via valid manipulations that may narrow (but never broaden) rights ascribed to the original capability" [42, §2.3.4]. But what constitutes broadening the rights of a capability? Broadening its bounds and increasing its permissions are given as examples, but does unsealing a capability also broaden its rights?

This is left unclear. Furthermore, the documentation states that "controlled violation of monotonicity can be achieved via the exception delivery mechanism [. . .] and also by the CCall instruction", without specifying what "controlled violation" means. It continues with "monotonicity allows reasoning about the set of reachable rights for executing code, as they are limited to the rights in any capability registers, and inductively, the set of any rights reachable from those capabilities". This property describes an upper bound of the rights that (untrusted) code can use if we allow it to execute arbitrary instructions. This upper bound is defined as the rights that are transitively reachable from the capabilities in the capability registers. However, the documentation does not define when a right is reachable from a capability, so one cannot know exactly what this upper bound is.

To illustrate the second benefit we discuss a security property that describes how CHERI's capability system can be used to protect a reference monitor from untrusted code [42, §9.4]. The property describes the guarantees and the assumptions under which they hold in great detail, but originally the property was verified only by a high-level paper proof outline. During our work we discovered that the CHERI ISA does not satisfy the property, not because of a bug in the ISA, but because of a mistake in the definition of the security property: the property mistakenly states that after a domain transition to the reference monitor, the reference monitor does not have permission to its own memory anymore. While it would be easy to fix the mistake in the prose definition, it would remain difficult to validate whether the property would then be correct, and whether the CHERI ISA would satisfy it.

To show that it is possible to formally define and prove security properties over a production scale architecture, in the remainder of this section we formally define the following properties about CHERI, and formally prove (in Isabelle/HOL) that the CHERI ISA satisfies them in §V.

- We define an order over capabilities, capturing when the authority of one capability is contained in the authority of another capability (§IV-A). This order clarifies what "broadening the rights of a capability" should mean in the prose definition of capability monotonicity.
- We define an abstraction of CHERI-MIPS with abstract actions for each type of memory access and capability manipulation, capturing the intentions of CHERI-MIPS instructions by mapping them onto these actions (§IV-B). For each action, we state under what conditions it can be performed, and what effects it has. Amongst other things, this precisely states the effects of instructions that can broaden the rights of a capability, clarifying what "controlled violation of capability monotonicity" should mean. It also states properties that have no prose counterparts in the CHERI documentation, but that are nonetheless crucial to the capability system.
- We characterise which capabilities a (potentially compromised) compartment could access or construct if it is allowed to execute arbitrary code, and we state related properties about which part of the memory and which registers the compartment can overwrite (§IV-C). This captures the "reachable rights for executing code".
- Turning from properties about the capability system itself to use-cases thereof, we state what assumptions need to be satisfied in order to isolate a compartment from the rest of the program, and we state which guarantees CHERI-MIPS then offers (§IV-D). This property is inspired by the reference monitor example discussed above.

When formalising security properties one should consider which mathematical concepts to use to express them: more sophisticated mathematics can let one state properties closer to one's intention, or more elegantly, but it can also make them less accessible. Here, we spell out properties in terms of concrete traces of the ISA model, for accessibility.

A. The Order Over the Authority of Capabilities

Recall from §II that a capability can authorise memory accesses and the sealing/unsealing of other capabilities, if it has a tag, is unsealed itself, its permissions allows the type of access or operation (e.g. load, store, execute, seal, unseal), and its memory region contains the footprint of the access or the object type of the sealed/to be sealed capability. Also recall that invoking a pair of sealed capabilities sets the program counter to the address of the code capability and atomically unseals both capabilities.

Here we define an order, $\leq$, that captures when one capability has at least as much authority as another capability. It is based on the following observations. The authorities of sealed and unsealed capabilities are incomparable even if they have the same bounds and permissions: sealed capabilities cannot authorise memory accesses or operations on other capabilities, while unsealed capabilities cannot be invoked. We also observe that invalid capabilities have no authority; sealed capabilities are immutable while they stay sealed; unsealed capabilities can be restricted by shrinking their bounds or removing their permissions; and the virtual address of unsealed capabilities can be changed to any value without affecting the authority. This leads to the following.

Definition 1 (Order over capabilities)**.** We say $cap \leq cap'$ if either cap is invalid (Line 2 below), or cap and cap' are equal (Line 3), or both capabilities are valid and unsealed (Lines 4 and 5) and: the bounds of cap is contained in the bounds of cap' (Line 6), the permissions of cap are less then or equal to those of cap' (Line 7) and similarly for the user permissions (Line 8), their object types agree (Line 9), and their reserved bits agree (Line 10). Note that Lines 4–10 do not constrain the virtual addresses. As usual in Isabelle and in functional languages, we write function application just with juxtaposition, e.g. IsSealed cap is just the IsSealed function (returning a boolean) applied to cap.

```
$\mathrm{cap} \le \mathrm{cap}'$ is defined as
  not Tag cap
  or $\mathrm{cap} = \mathrm{cap}'$
  or Tag cap and Tag $\mathrm{cap}'$
     and not IsSealed cap and not IsSealed $\mathrm{cap}'$
     and CapBounds cap $\subseteq$ CapBounds $\mathrm{cap}'$
     and Perms cap $\le_{\mathrm{bitwise}}$ Perms $\mathrm{cap}'$
     and UPerms cap $\le_{\mathrm{bitwise}}$ UPerms $\mathrm{cap}'$
     and ObjectType cap $=$ ObjectType $\mathrm{cap}'$
     and Reserved cap $=$ Reserved $\mathrm{cap}'$
```

This order is reflexive and transitive (a preorder). It is not antisymmetric: if cap and cap' are valid, unsealed, and differ only by their virtual addresses, we can have $cap \le cap'$, $cap' \le cap$, and $cap \ne cap'$. The preorder is also not total: if cap and cap' are respectively the sealed and unsealed version of the same capability, then $cap \not\le cap'$ and $cap' \not\le cap$.

B. Capturing the Intention of Instructions

We now define properties about the effects of a single execution step, abstracting from the detailed behaviour of CHERI-MIPS instructions defined by the L3 (or Sail) model. Our first goal is to capture the principle of intentional use: for example, if the intention of the execution step is to load data using the authority of the capability in register 2, then our properties should forbid this execution step if that does not have enough authority, even if capabilities in other registers would be able to authorise the load. We capture the intentions of each instruction by mapping them onto abstract actions: we define an abstract action for each kind of memory access (loading data, storing data, loading capabilities, and storing capabilities), one for each kind of capability manipulation (restricting, sealing, unsealing, and invoking it), and one for hardware exceptions. Abstract actions contain some extra information, for example the register index of the capability that is used as authority (if applicable). By mapping the 180-odd CHERI-MIPS instructions onto these nine actions we abstract away from many details but retain the ability to define different security properties for different intentions.

Our second goal is that the properties defined in this subsection are strong enough to imply the properties in §IV-C and §IV-D. To achieve this we define an invariant about address translation, and properties that describe what happens when a certain abstract action is *not* intended, for example if the instruction does not intend to store anything to an address a, then the memory at a should remain unchanged.

We first define the non-domain-crossing abstract actions.

- *LoadDataAction* has parameters $auth$, the register of the capability that is used as authority, a, the physical address of the data, and l, the length of the data that is loaded. *StoreDataAction* is the analogue for stores.
- *LoadCapAction* has parameters $auth$, the register of the capability that is used as authority, a, the physical address of the capability that is loaded, and r, the destination register. *StoreCapAction* is the analogue for storing capabilities, except here r is the source register and a the physical address of the destination.
- *RestrictCapAction* has parameters r, the source register, and r', the destination register where a restricted version of the source is copied to.
- *SealCapAction* has parameters $auth$, the register of the capability that is used as authority, r, the source register, and r', the destination register where a sealed version of the source is copied to. *UnsealCapAction* is the analogue for unsealing capabilities.

The following actions yield the execution to another domain:

- *RaiseException* has no parameters.
- *InvokeCapability* has parameters r, the register of the code capability, and r', the register of the data capability that is invoked.

An instruction intention can either be a single action that yields the execution, or a set of actions that do not (e.g. for the `CJALR` "jump and link capability register" instruction, which manipulates two capabilities in one execution step):

- *SwitchDomain* has parameter a, an action that yields the execution to another domain,
- *KeepDomain* has parameter $actions$, a set of actions that continue the execution in same domain.

Mapping instructions onto the abstraction is mostly straightforward. For example, the `CSeal` instruction executed with parameters (cd, cs, ct) maps to

$$\mathit{KeepDomain}\ \{\mathit{SealCapAction}\ ct\ cs\ cd\}.$$

Instructions that access memory are less straightforward, as their parameters refer to virtual memory, while the parameters of abstract actions refer to physical memory. When mapping these instructions we therefore translate the addresses. Since CHERI-MIPS instructions only access memory from at most one page, these translated addresses form a contiguous region of physical memory. Furthermore, instructions that load capabilities map to both a *LoadCapAction* and a *LoadDataAction* because they can indirectly be used to load data (for example, by loading data into a capability register and inspecting the fields of the capability). Similarly, instructions that store capabilities map to a *StoreCapAction* and a *StoreDataAction*.

1) Defining security properties for the abstraction: For each abstract action we define a property that states the prerequisites and effects of that action. These are properties of an arbitrary CHERI-MIPS ISA semantics *sem*, which we later prove hold of the actual semantics. The property about restricting capabilities is the simplest, requiring only that the resulting capability is less than or equal to the original:

Property 2 (Restricting capabilities)**.** An ISA semantics *sem* satisfies this property if Lines 2–6 below hold. Assume s is a valid machine state (Line 3). This is a technical assumption: the type *state* can express things that are not supported by CHERI-MIPS, e.g. little-endian mode, so we need to require that s does not. Then consider an execution step from state s to s' that keeps the domain and performs a number of actions (Line 4) and assume that *RestrictCapAction* r r' is one of those (Line 5). The property then requires that the capability in the destination register r' in the resulting state s' is less than or equal (in the §IV-A order) to the capability in the source register r in the original state s (Line 6).

```
1 RestrictCapProp sem is defined as
2   for all s s′ actions r r′.
3   if StateIsValid s
4     and (KeepDomain actions, s′) ∈ sem s
5     and RestrictCapAction r r′ ∈ actions
6   then CapReg s′ r′ ≤ CapReg s r
```

The next property states that to store data one needs a valid, unsealed capability with the *PermitStore* permission, and the physical addresses stored to should correspond to virtual addresses that lie within the bounds of the capability:

Property 3 (Storing data)**.** A semantics sem satisfies this property if the following holds. Assume s is a valid state (Line 3), consider an execution step from state s to s' that keeps the domain and performs a number of actions (Line 4), and assume that $StoreDataAction\ auth\ a\ ln$ is one of those (Line 5). The property then requires the capability used as authority has a tag (Line 6), is unsealed (Line 7), and has *PermitStore* permission (Line 8); that the length ln of the stored segment is non-zero (Line 9); that the addresses of the segment are all translations of virtual addresses that the capability has authority to (Line 10); and that the tag at address a has been stripped, or the capability at that address remained unchanged (Line 13). The latter case allows the action to behave as a no-op, relevant e.g. for stores to UART devices.

```
1  StoreDataProp sem is defined as
2    for all s s′ actions auth a ln.
3    if StateIsValid s
4      and (KeepDomain actions, s′) ∈ sem s
5      and StoreDataAction auth a ln ∈ actions
6    then Tag (CapReg s auth)
7         and not IsSealed (CapReg s auth)
8         and PermitStore (CapReg s auth)
9         and ln ≠ 0
10        and MemSegment a ln
11   ⊆ TranslateAddresses (CapBounds (CapReg s auth))
12                              Store s
13        and not Tag (MemCap s′ (GetCapAddress a))
14            or MemCap s′ (GetCapAddress a) =
15               MemCap s (GetCapAddress a)
```

The property for storing capabilities is similar: it assumes that $StoreCapAction\ auth\ r\ a$ is one of the performed actions, and requires the capability used as authority has a tag, is unsealed, and has both the *PermitStore* and the *PermitStoreCapability* permissions; that the addresses of the capability-sized segment starting at a are all translations of virtual addresses that the capability has authority to; and that the capability at address a in the resulting state s' equals the capability in register r in state s.

The properties for loading data and loading capabilities are analogues of the above. The fact that the capability at the right of Fig. 2 in §II-A cannot be used to load the secret key is a consequence of the property about loading data. For the other abstract actions we also define properties that describes the necessary authority and the effects of the action, and we define two security properties that are not directly linked to performing a specific action, but that are linked to the execute and access-system-register permissions. These are in the appendix.

2) Defining invariants of the execution step: There are two invariants capturing what happens if certain abstract actions are *not* performed. The first requires that if no action specifies that anything was stored to address a, then the memory at a remains unchanged. The second requires that if no action has capability register r as its destination register, the capability at r remains unchanged. Their formal definitions are given in the appendix. Then there is an invariant that requires that the address translation function remains unchanged during execution steps from s to s' that do not raise an exception and where the PCC of s does not have the permission to access system registers.

3) Defining capability derivations: The CHERI documentation describes informal properties about *capability derivations*. For example, *capability provenance* states that valid capabilities can only be "derived from" other valid capabilities [42, §2.3.1], and capability monotonicity states that new capabilities must be "derived from" existing capabilities via manipulations that do not broaden the rights of the capability. But what "derive" precisely means is not defined.

We can now precisely define what derivations are, in terms of the abstract actions an instruction maps to. For example, if an instruction maps to $RestrictCapAction\ r\ r'$, we say the capability in register r' in the resulting state is derived from the capability in r in the original state. Capability provenance can then be formalised by stating that the derived capability can only be valid if the capability it is derived from is valid, and capability monotonicity can be formalised by stating that for all actions except $SealCapAction$, $UnsealCapAction$, $InvokeCapability$, and $RaiseException$ the derived capability is less than or equal to the capability it is derived from.

Capability provenance and monotonicity capture properties about derivations that should certainly hold and that help explain the capability system. Nevertheless, in our proofs we did not find them useful as independent properties: whenever we needed to relate a derived capability to the original capability, we also needed to know that the capability that authorised the derivation had enough authority and, in the case of non-monotonic derivations, what the resulting capability could be, which is not captured by these. In other words, the proof forced us to identify more fundamental properties about the design.

C. Characterising Reachable Capabilities

We now characterise which capabilities a (potentially untrusted) compartment can access or construct if it is allowed to execute arbitrary code. This is a fundamental property for compartmentalisation, as it allows reasoning about which memory or system registers the compartment can access, whether it can delegate its own capabilities to other compartments, and which addresses in other compartments it can jump to. CHERI supports many compartmentalisation scenarios, for example compartments that can communicate via a region of shared memory, or that can only communicate via another compartment, or that share the same code but work on isolated data, or have isolated code but share their data. Much of what compartmentalisation means is common to all these.

Our definition of reachable capabilities depends on the capabilities in the current state. The definition is inductive because reaching a capability can make other capabilities reachable. For example, a capability in a register might authorise loading another capability from memory, and that capability might be able to authorise unsealing a capability, etc.

Definition 4. The set of reachable capabilities in a state s are inductively defined by the following rules.

- The base case: if a register r is always accessible (either a normal register, or the special registers DDC or TLSC) and the capability cap in that register is valid, then cap is reachable.
- If cap is a reachable, unsealed capability with the *PermitLoadCapability* permission, a is a physical address that is a translation of a virtual address within the bounds of cap, and the capability cap' at address a is valid, then cap' is reachable.
- If cap is a reachable capability, and cap' is a valid capability less than or equal to cap, then cap' is reachable.
- If cap is a reachable, unsealed capability, and $sealer$ is a reachable, unsealed capability with the *PermitSeal* permission, and the object type t lies within its bounds, then the capability that is the result of sealing cap with object type t is reachable.
- If cap is a reachable, sealed capability, and $unsealer$ is a reachable, unsealed capability with the *PermitUnseal* permission, and the object type of cap lies within its bounds, then the capability that is the result of unsealing cap is reachable.

The property that justifies the name "reachable capabilities" says that, until the execution is yielded to another domain, the set of reachable capabilities is monotonic. To avoid confusion with capability monotonicity (above), we call that property *intra-instruction capability monotonicity* and the property we define here *reachable capability monotonicity*:

Property 5 (Reachable capability monotonicity)**.** An ISA semantics sem satisfies this if the following holds. Assume s is a valid state (Line 3) and the access-system-register permission is not reachable from s (Line 4). Consider an execution trace from s to s' (Line 6) and assume it is intra-domain (no instruction in the trace yields the execution to another domain, Line 7). Then the property requires that the capabilities reachable in s' were already reachable in s.

```
1 MonotonicityReachableCaps sem is defined as
2   for all s s′ trace.
3   if StateIsValid s
4     and not SystemRegisterAccess
5                 (ReachablePermissions s)
6     and s′ ∈ FutureStates sem s trace
7     and IntraDomainTrace trace
8   then ReachableCaps s′ ⊆ ReachableCaps s
```

From reachable capability monotonicity we can derive properties about which memory a compartment can overwrite:

Property 6 (Intra-domain memory invariant)**.** If each reachable capability in s either does not have the *PermitStore* and *PermitStoreCapability* permissions, or does not contain an address within its bounds that translates to a; the execution trace from s to s' is intra-domain; and s is valid and in user mode; then the memory at a in s' is the same as in s.

The memory does not change during execution steps that yield the execution, so we define the same property for traces that are intra-domain except for the last step, which yields the execution to another domain. For the invariance of memory tags and special registers we define similar properties.

D. Isolating a User-space Compartment

Finally, we consider a simple compartmentalisation scenario, where a compartment is isolated from the rest of the program. Isolation here means that the compartment can only access its own region of memory, it cannot access any special registers, and when it yields the execution it can jump only to a restricted set of addresses. CHERI only guarantees this if the compartment is set up correctly. The definition below gives the top-level statement, referring to some definitions we define later.

Property 7 (Compartment isolation)**.** An ISA semantics sem satisfies this property if the following holds. Assume s is a state that is correctly set up (Line 3); we define this precisely in Definition 17. Consider an execution trace consisting of a prefix $trace$ that is intra-domain, meaning none of its steps yield the execution (Line 4), and a final $step$ that does yield the execution (Line 5). Let s' be the state after the latter (Line 6). Then the property requires that the isolation guarantees hold in s' (Line 7). We define what these guarantees entail in Definition 8.

```
1 CompartmentIsolation sem is defined as
2   for all addrs types s s′ trace step.
3   if IsolatedState addrs types s
4     and IntraDomainTrace trace
5     and SwitchesDomain step
6     and s′ ∈ FutureStates sem s (trace; step)
7   then IsolationGuarantees addrs types s s′
```

The predicate *IsolatedState* that defines when a state is correctly set up is defined in detail in Definition 17 in the appendix. The intuition is that the capabilities the compartment can reach (according to Definition 4) do not have authority outside the region of memory that we grant to the compartment, and that the compartment does not have permission to unseal the capabilities of other protection domains (cf. the example in §II-B). Below we define the predicate *IsolationGuarantees* that defines the guarantees that CHERI offers.

Definition 8 (Isolation guarantees)**.** Let *addrs* be the set of virtual addresses that we grant to the compartment, *types* the set of object types we grant to the compartment, and s and s' arbitrary states. The following describes the guarantees that one expects if a compartment starts in s and has yielded the execution in s'.

```
1  IsolationGuarantees addrs types s s′ is defined as
2    Base (PCC s′) + PC s′
3    ∈ ExceptionPCs ∪ InvokableAddresses addrs s
4    and for all a.
5        if not a ∈ TranslateAddresses addrs Store s
6        then MemData s′ a = MemData s a
7             and MemTag s′ (GetCapAddress a) =
8                 MemTag s (GetCapAddress a)
9    and for all r.
10       if r ≠ 0 and r ≠ 1 and r ≠ 31
11       then SpecialCapReg s′ r = SpecialCapReg s r
```

Lines 2–3 limit the exit points of the compartment. They state that the address of the next instruction in s' (given by the base of the PCC plus the PC) is either an exception handler entry address (CHERI-MIPS has a fixed set of these) or an address that one of the invokable capabilities point to.

Lines 4–8 state that the memory that is not granted to the compartment remained unchanged. More precisely, if the physical address a is not a translation of a virtual address in *addrs* (Line 5), then the memory value at a in s' is still the same as it was in s (Line 6), and the tag of the 32-byte region that a belongs to is still the same (Lines 7–8).

Finally, Lines 9–11 state that the special registers stayed the same, except for registers 0, 1, and 31 (DDC, TLSC, and EPCC). The DDC and TLSC are always accessible, while the EPCC is overwritten if the compartment raises a hardware exception.

V. Proving the Architectural Security Properties

Mathematical proof can give much higher confidence than traditional testing-based methods, because it considers every possible corner case, not just those exercised by the tests. This is especially important for security properties. However, there are three challenges in proving security properties for production-scale architectures. The first is that architectures contain many low-level details that are easy to miss, so checking a paper proof by hand would be unusably error-prone. To solve this we *mechanised* all our proofs in Isabelle [39], an interactive theorem prover. Such tools (Isabelle, Coq, HOL4, and others) let one write proof scripts – instructions for how to construct a proof, combining automated and manual reasoning – and the machine checks that they do construct valid proofs. To minimise the trusted computing base (TCB) Isabelle has an LCF-style inference kernel [43]: proof scripts generate a series of simple inference steps that are checked by a small kernel. Because we use the Isabelle export of the L3 model as the definition of CHERI-MIPS, our TCB consists only of this L3-to-Isabelle translation and the Isabelle kernel. This gives very high assurance that the proved statements are indeed true in the L3 definition of CHERI-MIPS (the user also has to read and understand the statements, of course).

The second challenge is the scale of the proof development. Production-scale architecture specifications are large, and any part of the architecture could potentially break security properties, even if it does not directly interact with the security mechanisms. For example, in CHERI-MIPS, the majority of the 180-odd instructions do not interact with capabilities, but could still break the properties of §IV. To solve this challenge we developed automated proof methods, tailored to L3 specifications, that reduce the need for manual proof scripts. We used Eisbach [44], an extension of Isabelle's proof language, for these. Our custom tactics can automatically prove the security properties for most CHERI-MIPS instructions that do not directly interact with the capability mechanisms, and significantly simplify the proofs for the others.

The third challenge is that architectures keep evolving. As a research architecture, CHERI has evolved rapidly, but industrial architectures such as Intel 64/IA-32 and ARMv8-A do too, with new versions every six months or so. It would be infeasible to continuously re-check whether a manual proof still holds for updated versions of the architecture, but automated theorem provers can do this automatically, and will point out any places where the proof fails. Our automatic proof tactics are somewhat resilient to changes in the model. To further reduce the effort needed to change our proofs we use python scripts to generate the statements and proofs of many lemmas. The Isabelle LCF-style kernel means that these scripts are not part of our TCB.

We finished the proof of our first variant of monotonicity of reachable capabilities in October 2016 and since then have rerun the proof regularly on new versions of the L3 model. In our experience the effort needed to adapt our proofs to changes in the ISA were reasonable. Some changes caused us to invent new properties, for example the introduction of capability invocation; and other changes involved refactoring properties and proofs, for example when capability registers were split into normal and special registers; but for most changes we only needed to update the scripts that auto-generate the majority of our proofs. This did need prover expertise, though.

All this made it possible to integrate our proofs in the CHERI-MIPS design process.

Theorem 9. The L3 model of the CHERI-MIPS ISA satisfies the properties defined or mentioned in §IV-B, namely properties about execution, loading and storing data, loading, storing, restricting, sealing, unsealing and invoking capabilities, accessing system registers, and exceptions; and invariants about memory values, tags, valid states, and address translation.

Theorem 10. Any semantics that satisfies the security properties defined in §IV-B, satisfies *reachable capability monotonicity* (Property 5), the *intra-domain memory invariant* (Property 6), and the other properties mentioned in §IV-C.

Together with Theorem 9 this shows that CHERI-MIPS satisfies the properties we defined in §IV-C.

Theorem 11. Any semantics that satisfies the security properties defined in §IV-B, satisfies the property about compartment isolation (Property 7).

The entire proof development is 33k lines of Isabelle/HOL, of which 16k lines are generated. Checking the proof in Isabelle takes 25 minutes on a 16GB Intel i7-2600. Inventing and proving the properties took around three person-years, a significant effort, but only a small fraction of the roughly 120 academic person-years for the CHERI project as a whole.

The proofs consider predictable and unpredictable behaviour separately. In the latter case, the execution step does not change the memory or any capability registers, and the proof is straightforward. In the former case, the semantics of CHERI-MIPS is deterministic and can be expressed in a state monad. We express our properties as Hoare triples [45] over this monad and use proof tactics to compute preconditions that imply the postcondition. This is complicated by the fact that the L3 model uses many auxiliary definitions that in principle can read and modify any part of the state. The core of our custom proof tactics is a tactic that decides whether two auxiliary definitions commute, for example if they do not write to the same part of the state, and neither reads the part of the state that the other writes to.

VI. Bugs Found by Proof Work

The point of our proof is to provide *assurance* that the properties hold of the CHERI-MIPS architecture, not merely to find some bugs, but, unsurprisingly, we did find some along the way. CHERI-MIPS was already reasonably mature when we started the proof, so these are not very numerous – but each could lead to security vulnerabilities, and it is instructive to see what can remain, even in a carefully considered and reviewed design, without proof.

- The `CLC` instruction loads a capability *cap* if the capability *cap′* that is used as authority has the *PermitLoadCapability* permission. If *cap′* does not have that permission, the instruction still loads the byte representation of *cap*, but without its tag. This does not violate our property about loading capabilities, but it does violate our property about loading data, as `CLC` loads from the memory without checking the *PermitLoad* permission. This bug was in the architecture document and the L3 model.
- Legacy MIPS stores allowed writing one byte past the region of memory the code had permission to, and, if the code had access to the end of the address space, stores could write to the beginning of the address space. (In L3)
- In some cases, unaligned MIPS loads allowed loading from a region of memory without permission. (In the L3 model and the Bluespec hardware implementation.)
- The `CBuildCap` instruction created a capability with the wrong base. (In L3)
- Exception return (`ERET`) could access a system register (the EPCC) without permission. (In L3)
- The `CCallFast` instruction, which invokes capabilities, exposed the unsealed code capability, breaking isolation between compartments that can invoke each other's capabilities. (In L3)

We also found counter-intuitive behaviour that led to the discovery of a vulnerability in CheriBSD, allowing a leak of an unsealed data capability. Throwing an exception just after performing a "`CCallFast`" gave the exception handler access to the unsealed data capability. By registering a signal handler to deal with segfaults and triggering a segfault in the delay slot of `CCallFast`, the signal handler could obtain the unsealed data capability of another protection domain and use it to access memory. One could conceivably fix this in CheriBSD, but correct code would be harder to write and understand, so we removed the `CCallFast` delay slot.

VII. Transition from L3 to Sail

In another aspect of formal engineering maintenance, we are in the process of shifting our ISA specifications from L3 to Sail [22], the design of which has been informed by our experience with L3. Sail generates emulators in OCaml and C, as well as theorem prover definitions for Isabelle/HOL, HOL4, and Coq, and Sail definitions can be integrated with multicore relaxed memory models. Sail models include CHERI-MIPS (ported from the L3 model and included in the CHERI architecture document [6]), a complete ISA semantics for ARMv8-A (automatically derived from the Arm-internal definition), and new hand-written models for RISC-V and CHERI-RISC-V. In ongoing and future work (not part of the contribution of this paper) we aim to *uniformly* prove security properties of multiple realisations of CHERI for different base architectures; we have already established a version of intra-instruction capability monotonicity for Sail CHERI-MIPS.

VIII. Related Work

In terms of formalising full-scale ISA definitions, using them in mainstream engineering, and proving security properties of them, the closest related work is Reid et al.'s within Arm, shifting essentially the entire ARMv8-M and ARMv8-A sequential ISA specifications from pseudocode to machine-readable definitions, which are now used for documentation and hardware verification [34], [46], [47]. These specifications are 10x or more larger, and much more complex, than CHERI-MIPS. For ARMv8-M, he formalised 59 properties about the ISA, based on prose statements in the architecture document, and used an SMT model-checking approach to verify that they hold [48]. Some of these properties are security-relevant, but they are much more specific than the whole-architecture properties we consider, which are strong enough to prove a use case correct (Property 7). On the other hand, his SMT approach is largely automated, while our proofs require theorem-proving expertise. In earlier work we proved correctness of an abstraction of address translation w.r.t. the Sail version of this ARMv8-A model [22].

Schwarz and Dam [49] use the HOL4 interactive theorem prover to verify noninterference properties of MIPS and a fragment of ARMv7, showing that the contents of privileged registers do not have observable effects during user-mode execution. These properties are certainly necessary, but are not by themselves strong enough to prove the correctness of use cases.

Turning to security properties for capability systems, the closest related work is by Skorstengaard et al. [50]. Our properties give an upper bound on what (untrusted) code can do until it yields the execution, but their *capability safety* result also allows reasoning about intermingled trusted and untrusted execution. They use it to prove that a certain calling convention guarantees control-flow correctness and encapsulation of local state. This is a stronger result than ours, but it is w.r.t. an idealised capability machine, inspired by CHERI but much simpler, rather than a complete ISA; their security properties are defined in terms of a step-indexed Kripke logical relation, which is hard to understand for practitioners; and their proofs are not mechanised. Ideally one would combine the two.

De Amorim et al. [51], [52] prove that their PUMP [53], [54] architecture, supporting multiple hardware security policies, correctly implements a memory safety policy. This is mechanised, but for an idealised PUMP, not a full ISA.

To reason about confidentiality in x86 SGX enclave programs, Sinha et al. [55] extend BAP [56] with a model of the SGX instructions, mapped into BoogiePL [57], but they do not discuss validation of this model, or its relationship with the complex x86 system semantics.

Ferraiuolo et al. [58] describe a RISC-V-based processor implementation that enforces secure information flow, including controlled timing channels, established using a security-typed hardware description language. Zagieboylo et al. [59] discuss an ISA for this and non-mechanised proofs, albeit somewhat idealised.

There is a very extensive literature discussing capabilities, access control, and information flow control in general, both informally and formally, dating back to the 1960s. We situate CHERI in that context in [6, Ch. 11]; space precludes detailed discussion here. In the context of software capability systems, Doerrie [60, Ch. 12] illustrates the need for proof mechanisation, identifying flaws in an earlier pen-and-paper proof of confinement for an idealised capability machine [61]. Their "proof that the potential access of the system is attenuating" is broadly similar to our Property 5 *Reachable capability monotonicity*. Murray et al. [62] give a mechanised proof of information flow properties for seL4.

Leaving security aside, there is extensive work on formal ISA modelling for hardware verification, e.g. for x86 in ACL2 [63], [64], in Coq [65], for RISC-V [66], and for Arm [34].

Acknowledgements

We thank Wes Filardo and Prashanth Mundkur for comments, and all the members of the CHERI team for their work on the project as a whole.

This work was supported by EPSRC programme grant EP/K008528/1 (REMS: Rigorous Engineering for Mainstream Systems). This work was supported by a Gates studentship (Nienhuis). This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement 789108, ELVER). This work was supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contracts FA8750-10-C-0237 (CTSRD), HR0011-18-C-0016 (ECATS), and FA8650-18-C-7809 (CIFV). The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government. Approved for public release; distribution is unlimited.

Appendix

A. Stating More Architectural Security Properties

In §IV we defined the properties about restricting capabilities (Property 2) and storing data (Property 3). Here we define some of the additional properties needed, all of which have been established by our proof (Theorem 9).

The property about unsealing capabilities is stated below:

Property 12 (Unsealing capabilities)**.** A semantics *sem* satisfies this property if the following holds. Assume s is a valid state (Line 3), consider an execution step from s to s' that keeps the domain and performs a number of actions (Line 4), and assume that *UnsealCapAction auth* r r' is one of those actions (Line 5). The property then requires the capability that is used as authority has a tag (Line 6), is unsealed (Line 7), and has the *PermitUnseal* permission (Line 8); that the object type of the capability that is being unsealed lies within the bounds of the capability that is used as authority (Line 9); that the capability that is being unsealed was sealed in state s (Line 11); and that the capability in the destination register r' in the resulting state s' is less than or equal to the unsealed version of the original capability (Line 12).

```
1  UnsealCapProp sem is defined as
2    for all s s′ actions auth r r′.
3    if StateIsValid s
4      and (KeepDomain actions, s′) ∈ sem s
5      and UnsealCapAction auth r r′ ∈ actions
6    then Tag (CapReg s auth)
7         and not IsSealed (CapReg s auth)
8         and PermitUnseal (CapReg s auth)
9         and UCast (ObjectType (NormalCapReg s r))
10              ∈ CapBounds (CapReg s auth)
11        and IsSealed (NormalCapReg s r)
12        and NormalCapReg s′ r′
13              ≤ NormalCapReg s r with
14                  IsSealed ← False, ObjectType ← 0
```

The property about sealing capabilities is the analogue of the above. Then there are two security properties that are

```
define COP2 > CHERICOP2 > CSeal (cd::reg, cs::reg, ct::reg) =
   if not CP0.Status.CU2 then
      SignalCP2UnusableException
   else if not getTag(CAPR(cs)) then
      SignalCapException(capExcTag,cs)
   else if not getTag(CAPR(ct)) then
      SignalCapException(capExcTag,ct)
   else if getSealed(CAPR(cs)) then
      SignalCapException(capExcSeal,cs)
   else if getSealed(CAPR(ct)) then
      SignalCapException(capExcSeal,ct)
   else if not getPerms(CAPR(ct)).Permit_Seal then
      SignalCapException(capExcPermSeal,ct)
   else if getOffset(CAPR(ct)) >=+ getLength(CAPR(ct)) then
      SignalCapException(capExcLength,ct)
   else if (getBase(CAPR(ct)) + getOffset(CAPR(ct))) >=+ [2**OTYPEWIDTH] then
      SignalCapException(capExcLength,ct)
   else if not canRepSeal (CAPR(cs), true) then
      SignalCapException(capExcInexact,cs)
   else
   {
      var new_cap = CAPR(cs);
      new_cap <- setSealed(new_cap, true);
      new_cap <- setType(new_cap, (getBase(CAPR(ct)) + getOffset(CAPR(ct)))<(OTYPEWIDTH-1):0>);
      CAPR(cd) <- new_cap
   }
```

Fig. 5. The L3 specification of the `CSeal cd, cs, ct` *Seal a Capability* instruction. Capability register `cs` is sealed with otype `ct.base+ct.offset` and the result placed in `cd`. `ct` must grant `Permit_Seal` permission, and the new otype of `cd` must be between `ct.base` and `ct.base+ct.length-1`.

```
define COP2 > CHERICOP2 > CCallFast (cs::reg, cb::reg) =
   if not CP0.Status.CU2 then
      SignalCP2UnusableException
   else if not getTag(CAPR(cs)) then
      SignalCapException(capExcTag,cs)
   else if not getTag(CAPR(cb)) then
      SignalCapException(capExcTag,cb)
   else if not getSealed(CAPR(cs)) then
      SignalCapException(capExcSeal,cs)
   else if not getSealed(CAPR(cb)) then
      SignalCapException(capExcSeal,cb)
   else if getType(CAPR(cs)) <> getType(CAPR(cb)) then
      SignalCapException(capExcType,cs)
   else if not getPerms(CAPR(cs)).Permit_CCall then
      SignalCapException(capExcPermCCall,cs)
   else if not getPerms(CAPR(cb)).Permit_CCall then
      SignalCapException(capExcPermCCall,cb)
   else if not getPerms(CAPR(cs)).Permit_Execute then
      SignalCapException(capExcPermExe,cs)
   else if getPerms(CAPR(cb)).Permit_Execute then
      SignalCapException(capExcPermExe,cb)
   else if getOffset(CAPR(cs)) >=+ getLength(CAPR(cs)) then
      SignalCapException(capExcLength,cs)
   else
   {
      CheckBranch;
      BranchDelayPCC <- Some(getOffset(CAPR(cs)), setType(setSealed(CAPR(cs), false),0));
      IDC <- setType(setSealed(CAPR(cb), false), 0)
   }
```

Fig. 6. The L3 specification of the `CCallFast cs, cb` instruction. The new `PCC` and `IDC` are unsealed versions of `cs` and `cb`.

not linked to performing a specific action. The first describes the authority the PCC must have if an action was performed without raising an exception:

Property 13 (Executing)**.** A semantics sem satisfies this property if the following holds. Assume s is a valid state (Line 3), and consider an execution step from s to s' (Line 4) that did not raise an exception (Line 5). The property then requires that the PCC has a tag (Line 6), is unsealed (Line 7), has the *PermitExecute* permission (Line 8), and that the address of the next instruction lies within its bounds (Line 9).

```
1 ExecuteProp sem is defined as
2   for all s s′ step.
3   if StateIsValid s
4      and (step, s′) ∈ sem s
5      and step ≠ SwitchDomain RaiseException
6   then Tag (PCC s)
7          and not IsSealed (PCC s)
8          and PermitExecute (PCC s)
9          and Base (PCC s) + PC s ∈ CapBounds (PCC s)
```

The second requires that if r is a special register (except the DDC or TLSC) that is the parameter of a performed action, then the PCC must have permission to access system registers (the DDC and TLSC are accessible without that permission):

Property 14 (System register access)**.** A semantics sem satisfies this property if the following holds. Assume s is a valid state (Line 3). Consider an execution step from s to s' that keeps the domain and performs a number of actions (Line 4). Let r be a register that is not equal to 0 or 1 (respectively the DDC and TLSC, Line 5) and that is a parameter of one of the performed actions (Line 6). The property then requires that the PCC has the permission to access system registers (Line 9).

```
1 SystemRegisterProp sem is defined as
2   for all s s′ actions r.
3   if StateIsValid s
4      and (KeepDomain actions, s′) ∈ sem s
5      and r ≠ 0 and r ≠ 1
6      and exists action.
7            action ∈ actions
8            and r ∈ CapDerivationRegisters action
9   then PermitAccessSystemRegisters (PCC s)
```

Then there are two invariants about what happens when certain actions are not taken. The auxiliary function *CapDerivationTargets* returns the target locations of an action. For example, the target of *StoreCapAction auth r a* is the address a and the target of *SealCapAction auth r r′* is the register r'. The first invariant states that the capability at location loc remains invariant of none of the performed actions has loc as their target:

Property 15 (Capability invariant)**.** A semantics sem satisfies this property if the following holds. Assume s is a valid state (Line 3), consider an execution step from s to s' that keeps the domain and performs a number of actions (Line 4), and assume that none of these actions has the location loc as their target (Line 5). The property then requires that the capability at location *loc* in the resulting state equals the capability at *loc* in the original state (Line 8).

```
1 CapabilityInvariant sem is defined as
2   for all s $s'$ actions loc.
3   if StateIsValid s
4     and (KeepDomain actions, $s'$) $\in$ sem s
5     and not exists action.
6               action $\in$ actions
7               and loc $\in$ CapDerivationTargets action
8   then Cap $s'$ loc $=$ Cap s loc
```

The second invariant is similar: if none of the performed actions has the address a as their target, then the value at a in the memory remains invariant. This property is more granular than the capability invariant. For example, if an action changes one byte in the byte representation of a capability, this invariant requires that the other 31 bytes of the representation remain unchanged even though the capability (as a whole) changes.

Then there are two properties about actions that yield the execution. The first describes capability invocation:

Property 16 (Invoking capabilities)**.** A semantics *sem* satisfies this property if the following holds.

```
1  InvokeCapProp sem is defined as
2    for all s $s'$ r $r'$.
3    if StateIsValid s
4      and (SwitchDomain (InvokeCapability r $r'$), $s'$)
5          $\in$ sem s
6    then let codeCap $=$ NormalCapReg s r in
7         let dataCap $=$ NormalCapReg s $r'$ in
8         Tag codeCap and Tag dataCap
9         and IsSealed codeCap and IsSealed dataCap
10        and PermitCCall codeCap
11        and PermitCCall dataCap
12        and PermitExecute codeCap
13        and not PermitExecute dataCap
14        and ObjectType codeCap $=$ ObjectType dataCap
15        and PC $s'$ $=$ Offset codeCap
16        and PCC $s'$ $=$ codeCap with IsSealed $\leftarrow$ False,
17                                 ObjectType $\leftarrow$ 0
18        and BranchDelay $s'$ $=$ None
19        and BranchDelayPCC $s'$ $=$ None
20        and IDC $s'$ $=$ dataCap with IsSealed $\leftarrow$ False,
21                                ObjectType $\leftarrow$ 0
22        and for all cb.
23            if cb $\neq$ 26
24            then NormalCapReg $s'$ cb $=$
25                    NormalCapReg s cb
26        and for all cb. SpecialCapReg $s'$ cb $=$
27                        SpecialCapReg s cb
28        and for all a. Mem $s'$ a $=$ Mem s a
```

Assume s is a valid state (Line 3) and consider an execution step from s to s' that invokes a pair of a code and a data capability (Line 4). The property then requires that both code and data capability are valid (Line 8), sealed (Line 9), have the permission to be invoked (Line 10–11), the code capability has the permission to execute (Line 12), but the data capability does not (Line 13), and both capabilities have the same object type (Line 14). Furthermore, it requires that the offset of the code capability is copied to the PC (Line 15), the unsealed code capability is copied to the PCC (Line 16), the MIPS branch slots are cleared (Line 18–19), the unsealed data capability is copied to the IDC (Line 20), all normal registers except register 26 (the IDC) remain unchanged (Line 22), all special registers remain unchanged (Line 26), and the memory remains unchanged (Line 28).

The second describes the effects of a hardware exception: if an execution step from a valid state s to s' raises an exception, then the exception flag is set, the address of the next instruction is one of a fixed set of exception entry addresses, the KCC is copied to the PCC, all the normal capability registers remain unchanged, the PCC is copied to the EPCC (unless the exception flag was already set in s in which case the EPCC remains unchanged), the special registers other than EPCC remain unchanged, the memory remains unchanged, and the MIPS branch slots are cleared.

Finally, we define two invariants. The first simply says that if s is a valid state and there is an execution step from s to s', then s' is also a valid state. The second says that if an execution step from s to s' did not raise an exception and the PCC in s does not have permission to access system registers, then the translation of any address a is the same in s as in s'.

Moving to the compartmentalisation scenario described in §IV-D we define in detail when a state is correctly set up.

Definition 17 (Isolation assumptions)**.** Let *addrs* be the set of virtual addresses that we grant to the compartment, *types* the set of object types we grant to the compartment, and s an arbitrary state. Our aim is to require that each unsealed capability the compartment can reach (according to Definition 4) does not have authority outside the virtual addresses *addrs* and types *types* that we granted to the compartment. Because the inductive definition of reachable capabilities is difficult to work with, we overapproximate the authority of unsealed, reachable capabilities as follows. First, consider the set *grantedCaps* of capabilities that we granted to the compartment, consisting of the program counter capability (PCC), the capability in the MIPS branch slot (if there is one), capabilities in normal capability registers, capabilities in the special registers 0 and 1 (DDC and TLSC), and the capabilities in the physical region of memory that correspond to the virtual addresses *addrs*. From that set, we take each capability whose authority can be used by the compartment, namely each capability that has a tag, and that is either unsealed, or sealed with an object type in *types*. The resulting set, *grantedAuth*, is the overapproximation (the following lines do not claim that, but we prove it as part of Theorem 11).

Line 2 requires that capabilities in *grantedAuth* do not have the permission to access system registers; Line 3 requires that the bounds of capabilities in *grantedAuth* with permission to access memory are contained in *addrs*; and Line 4 requires that capabilities in $grantedAuth$ with permission to seal or unseal have authority only to object types in $types$. Line 5 is an assumption on the capabilities that the compartment can invoke. The compartment can invoke any capability in $grantedCaps$ that has a tag and that has the $PermitCCall$ permission. The assumption requires that these capabilities are sealed and that their object type is *not* contained in $types$, which ensures that the compartment cannot directly use the authority of these capabilities. Line 6 below requires that $addrs$ is capability aligned. This means that for every a and a' within the same capability (only differing in their last 5 bits) either both or neither are in $addrs$. Finally, Line 7 requires that s is a valid state.

```
1 IsolatedState addrs types s is defined as
2   NoSystemRegisterAccess addrs types s
3   and ContainedCapBounds addrs types s
4   and ContainedObjectTypes addrs types s
5   and InvokableCapsNotUsable addrs types s
6   and CapabilityAligned addrs
7   and StateIsValid s
```

References

[1] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 48–62.

[2] M. Miller, "Trends, challenge, and shifts in software vulnerability mitigation," https://github.com/Microsoft/MSRC-Security-Research/tree/master/presentations/2019_02_BlueHatIL, Februari 2019, Microsoft Security Response Center.

[3] B. Martin, M. Brown, A. Paller, D. Kirby, and S. Christey, "2011 CWE/SANS top 25 most dangerous software errors," *Common Weakness Enumeration*, 2011.

[4] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey *et al.*, "The matter of Heartbleed," in *Proceedings of the 2014 conference on Internet Measurement*. ACM, 2014, pp. 475–488.

[5] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI capability model: Revisiting RISC in an age of risk," in *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, 2014, pp. 457–468.

[6] R. N. Watson, P. G. Neumann, J. Woodruff, M. Roe, H. Almatary, J. Anderson, J. Baldwin, D. Chisnall, B. Davis, N. W. Filardo, 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 7)," University of Cambridge, Computer Laboratory, Tech. Rep., 2019. [Online]. Available: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-927.pdf

[7] "CHERI," https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/.

[8] J. Saltzer, "Protection and the control of information sharing in Multics," *Communications of the ACM*, vol. 17, no. 7, pp. 388–402, July 1974. [Online]. Available: https://multicians.org/saltzer-pacisim.pdf

[9] N. Hardy, "The confused deputy (or why capabilities might have been invented)," *ACM SIGOPS Operating Systems Review*, vol. 22, no. 4, pp. 36–38, 1988.

[10] I. T. LTD, "MIPS® architecture for programmers volume II-A: The MIPS64® instruction set reference manual," 2016.

[11] R. Nikhil, "Bluespec System Verilog: efficient, correct RTL from high level specifications," in *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04.* IEEE, 2004, pp. 69–70.

[12] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004, p. 75.

[13] M. K. McKusick, G. V. Neville-Neil, and R. N. Watson, *The design and implementation of the FreeBSD operating system*. Pearson Education, 2014.

[14] B. Davis, R. N. Watson, A. Richardson, P. G. Neumann, S. W. Moore, J. Baldwin, D. Chisnall, J. Clarke, N. W. Filardo, K. Gudka *et al.*, "CheriABI: Enforcing valid pointer provenance and minimizing pointer privilege in the POSIX C run-time environment," University of Cambridge, Computer Laboratory, Tech. Rep., 2019.

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

[16] K. Gudka, R. N. Watson, J. Anderson, D. Chisnall, B. Davis, B. Laurie, I. Marinos, P. G. Neumann, and A. Richardson, "Clean application compartmentalization with SOAAP," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 1016–1031.

[17] A. Joannou, J. Woodruff, R. Kovacsics, S. W. Moore, A. Bradbury, H. Xia, R. N. Watson, D. Chisnall, M. Roe, B. Davis *et al.*, "Efficient tagged memory," in *Computer Design (ICCD), 2017 IEEE International Conference on*. IEEE, 2017, pp. 641–648.

[18] R. N. Watson, R. M. Norton, J. Woodruff, S. W. Moore, P. G. Neumann, J. Anderson, D. Chisnall, B. Davis, B. Laurie, M. Roe *et al.*, "Fast protection-domain crossing in the CHERI capability-system architecture," *IEEE Micro*, vol. 36, no. 5, pp. 38–49, 2016.

[19] R. Grisenthwaite (SVP, Chief Architect & Fellow, Arm), "A safer digital future, by design," https://www.arm.com/blogs/blueprint/digital-security-by-design, Oct. 2019.

[20] Department for Business, Energy & Industrial Strategy, Department for Digital, Culture, Media & Sport, Home Office, UK Research and Innovation, The Rt Hon Andrea Leadsom MP, and Matt Warman MP, "Confronting cyber threats to businesses and personal data (press release)," https://www.gov.uk/government/news/confronting-cyber-threats-to-businesses-and-personal-data, Oct. 2019.

[21] A. C. Fox, "Directions in ISA specification," in *ITP*, 2012, pp. 338–344. [Online]. Available: https://doi.org/10.1007/978-3-642-32347-8_23

[22] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell, "ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS," in *POPL 2019: Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2019.

[23] S. Bishop, M. Fairbairn, H. Mehnert, M. Norrish, T. Ridge, P. Sewell, M. Smith, and K. Wansbrough, "Engineering with logic: Rigorous test-oracle specification and validation for TCP/IP and the Sockets API," *J. ACM*, vol. 66, no. 1, pp. 1:1–1:77, Dec. 2018. [Online]. Available: http://doi.acm.org/10.1145/3243650

[24] R. N. M. Watson, J. Woodruff, M. Roe, S. W. Moore, and P. G. Neumann, "Capability Hardware Enhanced RISC Instructions (CHERI): Notes on the Meltdown and Spectre Attacks," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-916, Feb. 2018. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-916.pdf

[25] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," in *USENIX Annual Technical Conference, General Track*, 2002, pp. 275–288.

[26] G. C. Necula, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy code," in *ACM SIGPLAN Notices*, vol. 37, no. 1. ACM, 2002, pp. 128–139.

[27] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for C," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 245–258, 2009.

[28] K. Memarian, V. B. F. Gomes, B. Davis, S. Kell, A. Richardson, R. N. M. Watson, and P. Sewell, "Exploring C semantics and pointer provenance," in *Proc. 46th ACM SIGPLAN Symposium on Principles of Programming Languages*, Jan. 2019, proc. ACM Program. Lang. 3, POPL, Article 67. Also available as ISO/IEC JTC1/SC22/WG14 N2311.

[29] AMD, "AMD64 Architecture Programmer's Manual, Volumes 1–5," http://developer.amd.com/resources/developer-guides-manuals/, Mar. 2017, 3178 pages.

[30] IBM, "Power ISA Version 3.0," Nov. 2015, 1246 pages.

[31] Intel Corporation, "Intel 64 and IA-32 Architectures Software Developer's Manual. Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4." https://software.intel.com/en-us/articles/intel-sdm, Jul. 2017, 325462-063US. 4744 pages.

[32] "The RISC-V Instruction Set Manual. Volume I: User-Level ISA; Volume II: Privileged Architecture," https://riscv.org/specifications/, May 2017, 236 pages.

[33] *The SPARC Architecture Manual, Version 9*. SPARC International, Inc., 1994, sAV09R1459912.

[34] A. Reid, R. Chen, A. Deligiannis, D. Gilday, D. Hoyes, W. Keen, A. Pathirane, O. Shepherd, P. Vrabel, and A. Zaidi, "End-to-end verification of processors with ISA-Formal," in *International Conference on Computer Aided Verification*. Springer, 2016, pp. 42–58.

[35] P. Misra and N. Dutt, Eds., *Processor Description Languages*. Morgan Kaufmann, 2008.

[36] "PVS specification and verification system," http://pvs.csl.sri.com/, accessed 2019-07-27.

[37] "QEMU: the FAST! processor emulator," 2017, https://www.qemu.org/.

[38] B. Campbell and I. Stark, "Extracting behaviour from an executable instruction set model," in *Formal Methods in Computer-Aided Design (FMCAD), 2016*. IEEE, 2016, pp. 33–40.

[39] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2012.

[40] M. Gordon and A. Pitts, "The HOL logic and system," in *Real-Time Safety Critical Systems*. Elsevier, 1994, vol. 2, pp. 49–70.

[41] P. Castéran and Y. Bertot, "Interactive theorem proving and program development. Coq'Art: The calculus of inductive constructions." 2004.

[42] R. N. Watson, P. G. Neumann, J. Woodruff, M. Roe, J. Anderson, J. Baldwin, D. Chisnall, B. Davis, A. Joannou, B. Laurie, S. W. Moore, S. J. Murdoch, R. Norton, S. Son, and H. Xia, "Capability hardware enhanced RISC instructions: CHERI instruction-set architecture (version 6)," University of Cambridge, Computer Laboratory, Tech. Rep., 2017. [Online]. Available: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-907.pdf

[43] M. Gordon, R. Milner, and C. Wadsworth, *Edinburgh LCF: a mechanised logic of computation*. Springer-Verlag, 1979.

[44] D. Matichuk, T. Murray, and M. Wenzel, "Eisbach: A proof method language for Isabelle," *Journal of Automated Reasoning*, vol. 56, no. 3, pp. 261–282, 2016.

[45] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[46] A. Reid, "Trustworthy Specifications of Arm v8-A and v8-M system Level Architecture," in *Proceedings of Formal Methods in Computer-Aided Design (FMCAD 2016)*, October 2016, pp. 161–168. [Online]. Available: https://alastairreid.github.io/papers/fmcad2016-trustworthy.pdf

[47] ——, "Defining interfaces between hardware and software: Quality and performance," Ph.D. dissertation, School of Computing Science, University of Glasgow, March 2019.

[48] ——, "Who guards the guards? Formal validation of the Arm v8-M architecture specification," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 88, 2017.

[49] O. Schwarz and M. Dam, "Automatic derivation of platform noninterference properties," in *Software Engineering and Formal Methods - 14th International Conference, SEFM 2016*, ser. Lecture Notes in Computer Science, R. De Nicola and eva Kühn, Eds., vol. 9763. Springer, 2016, pp. 27–44. [Online]. Available: https://doi.org/10.1007/978-3-319-41591-8_3

[50] L. Skorstengaard, D. Devriese, and L. Birkedal, "Reasoning about a machine with local capabilities," in *European Symposium on Programming*. Springer, 2018, pp. 475–501.

[51] A. A. De Amorim, M. Dénes, N. Giannarakis, C. Hriţcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach, "Micro-policies: Formally verified, tag-based security monitors," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 813–830.

[52] A. Azevedo de Amorim, N. Collins, A. DeHon, D. Demange, C. Hriţcu, D. Pichardie, B. C. Pierce, R. Pollack, and A. Tolmach, "A verified information-flow architecture," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14. New York, NY, USA: ACM, 2014, pp. 165–178. [Online]. Available: http://doi.acm.org/10.1145/2535838.2535839

[53] U. Dhawan, C. Hriţcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight Jr, B. C. Pierce, and A. DeHon, "Architectural support for software-defined metadata processing," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1. ACM, 2015, pp. 487–502.

[54] U. Dhawan, N. Vasilakis, R. Rubin, S. Chiricescu, J. M. Smith, T. F. Knight Jr, B. C. Pierce, and A. DeHon, "PUMP: a programmable unit for metadata processing," in *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2014, p. 8.

[55] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani, "Moat: Verifying confidentiality of enclave programs," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 1169–1184.

[56] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, 2011, pp. 463–469. [Online]. Available: https://doi.org/10.1007/978-3-642-22110-1_37

[57] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, 2005, pp. 364–387. [Online]. Available: https://doi.org/10.1007/11804192_17

[58] A. Ferraiuolo, M. Zhao, A. C. Myers, and G. E. Suh, "Hyperflow: A processor architecture for nonmalleable, timing-safe information-flow security," in *25th ACM Conf. on Computer and Communications Security (CCS)*, October 2018. [Online]. Available: http://www.cs.cornell.edu/andru/papers/hyperflow

[59] D. Zagieboylo, G. E. Suh, and A. C. Myers, "Using information flow to design an isa that controls timing channels," in *32nd IEEE Computer Security Foundations Symp. (CSF)*, June 2019. [Online]. Available: http://www.cs.cornell.edu/andru/papers/hyperisa

[60] M. S. Doerrie, "Confidence in confinement: An axiom-free, mechanized verification of confinement in capability-based systems," Ph.D. dissertation, Johns Hopkins University, 2015.

[61] J. S. Shapiro and S. Weber, "Verifying the EROS confinement mechanism," in *Proceeding 2000 IEEE Symposium on Security and Privacy. S&P 2000*. IEEE, 2000, pp. 166–176.

[62] T. C. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, "sel4: From general purpose to a proof of information flow enforcement," in *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, 2013, pp. 415–429. [Online]. Available: https://doi.org/10.1109/SP.2013.35

[63] S. Goel *et al.*, "Formal verification of application and system programs based on a validated x86 ISA model," Ph.D. dissertation, The University of Texas at Austin, 2016.

[64] W. A. Hunt, M. Kaufmann, J. S. Moore, and A. Slobodova, "Industrial hardware and software verification with ACL2," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 375, 2017.

[65] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, "Kami: a platform for high-level parametric hardware specification and its modular verification," *PACMPL*, vol. 1, no. ICFP, pp. 24:1–24:30, 2017. [Online]. Available: https://doi.org/10.1145/3110268

[66] C. Wolf, "End-to-end formal ISA verification of RISC-V processors with riscv-formal," In 7th RISC-V Workshop Proceedings, Nov. 2017, http://www.clifford.at/papers/2017/riscv-formal/.