End-to-End Formal Verification of a RISC-V Processor Extended with Capability Pointers

Dapeng Gao
University of Oxford

Tom Melham
University of Oxford

***Abstract*—Capability Hardware Enhanced RISC Instructions (CHERI) extend conventional ISAs with *capabilities* that can enable fine-grained memory protection and scalable software compartmentalisation. CHERI-RISC-V is an extended version of the RISC-V ISA with support for CHERI, and Flute is an open-source 64-bit RISC-V processor with a five-stage, in-order pipeline. This case study presents the formal verification of CHERI-Flute, a modified version of Flute that implements CHERI-RISC-V, against the Sail CHERI-RISC-V specification. To the best of our knowledge, this is the first extensive formal verification of a CHERI-enabled processor.**

We first translated relevant portions of the Sail CHERI-RISC-V specification to SystemVerilog Assertions. Then we formulated and proved four classes of end-to-end correctness properties about CHERI-Flute, covering the CHERI instructions and certain liveness properties about the entire processor. None of these results are routine—they all rely on novel proof engineering methodologies that extract microarchitectural invariants to serve as lemmas for the end-to-end proofs.

This work exposed several previously-unknown bugs in CHERI-Flute, most of which occur in the implementation of sophisticated combinational logic for certain CHERI instructions.

I. Introduction

Despite decades of hardening and mitigation efforts—such as stack protection, garbage collection, and virtualisation—memory safety issues remain a common and dangerous source of security vulnerabilities. A 2019 report by Microsoft [1] states that '70% of the vulnerabilities addressed through a security update each year continue to be memory safety issues'. The root cause of this phenomenon is the pervasive use of an unsafe memory model for interpreting the C programming language [2]. This model can be traced back to the PDP-11 and presumes that memory is simply a linear array of individually addressable bytes. This has induced a number of deeply ingrained assumptions about pointer behaviour that go beyond what is guaranteed by the C specification and rely only on 'implementation-defined behaviour'.

The Capability Hardware Enhanced RISC Instructions (CHERI) project offers an alternative model that provides better memory safety [3]. Its main features include a new machine representation of C pointers called *capabilities*, and extensions to existing instruction set architectures (ISA) that enable the secure manipulation of capabilities. For intuitive understanding, capabilities can be regarded as traditional pointers with extra properties that make them more like object references in a memory-managed language, such as Java. On one hand, this model continues to support limited arithmetic operations on capabilities that, for example, allow a loop to iterate through an array by repeatedly incrementing a capability. On the other hand, it makes it impossible to construct arbitrary capabilities that can be dereferenced—a significant departure from the usual 'unsafe' understanding of the C programming language.

Well-developed ISAs that integrate capabilities include CHERI-RISC-V and CHERI-MIPS [4], which are extended from RISC-V and MIPS. Rigorous engineering techniques have been used extensively in their development [5]. Specifically, Sail [6] specifications of these CHERI ISAs exist that give a precise and executable definition to each instruction.

This case study explores the formal verification of an open source implementation of CHERI-RISC-V. Flute is a 64-bit RISC-V processor with a five-stage, in-order pipeline [7] released by Bluespec Inc. in late 2018. Researchers at Cambridge University have extended Flute with support for CHERI-RISC-V [8], and this extended implementation, named CHERI-Flute, was our verification target.

A. Contributions

We have verified several classes of properties for CHERI-Flute using the JasperGold formal verification environment [9]. The scope of our verification comprises the correct execution of all 80-plus CHERI instructions as well as certain liveness properties for the processor as a whole. Our proof does not cover the existing RISC-V instructions, which do not involve capabilities. Formal verification methodologies for these instructions are well-established and so they are not of central interest in this case study.

To the best of our knowledge, this is the first extensive formal verification of a CHERI processor implementation. Our aim in this paper is to make the methodology accessible for future verification projects on novel architectures, including ones that target capability hardware. All our verification code is available open-source [10].

We have deliberately taken an end-to-end approach. That is, properties are proved for the entire core, as opposed to individual components such as the individual execution units. In CHERI-Flute, the hardware that deals with capabilities is novel, complex, and distributed across the pipeline stages. Our end-to-end approach avoids the necessity to isolate this hardware and characterise its environment.

Our verification results all rely on novel proof engineering methodologies that extract microarchitectural invariants to serve as lemmas for the end-to-end proofs. Some of these

https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_10

This article is licensed under a Creative Commons Attribution 4.0 International License

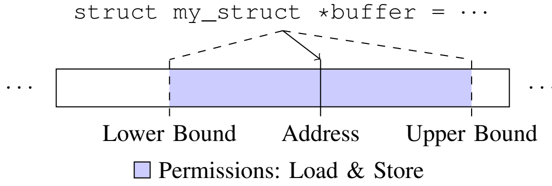

Fig. 1. A typical pointer represented by a capability

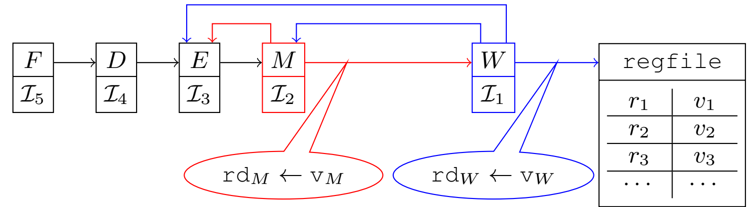

Fig. 2. Pipeline of Flute, including forwarding paths

invariants are of interest in themselves. For example, one of them shows that the core can never create a malformed capability—an important consistency invariant.

This case study exposed several previously-unknown bugs in the implementation of CHERI-Flute, which have all been reported to and confirmed by the designers [11], [12], [13]. Most of these bugs occur in the implementation of sophisticated bit manipulation logic for CHERI-related instructions, demonstrating the effectiveness of formal verification in catching subtle bugs in a novel processor design. In some cases, we have been able to provide verified bugfixes to the designers.

II. Background to Capability Architecture

CHERI extends ISAs with a new hardware representation for pointers and new instructions for manipulating them. See [4] for its full specification and [14] for a high-level summary of the large research effort surrounding CHERI.

Instead of using 32- or 64-bit integers to represent pointers, CHERI uses a richer representation called *capabilities* that can be stored in *capability registers* in the core or in *capability-sized* and *capability-aligned words* in the memory. The program counter, which usually holds integer addresses, is replaced by the *program counter capability* (`pcc`).

A capability, illustrated in Fig. 1, contains additional information compared to a traditional pointer, most notably including the following.

Validity Tag. A 1-bit tag that indicates whether the capability is valid. Such a tag is associated with 'each location that can hold a capability—whether a capability register or a capability-sized, capability-aligned word of memory' and it 'tracks capability validity for the value stored at that location' [4]. When a location that can hold a capability is *untagged*, its contents are simply data and hence do not grant any privilege.

Permissions. A bitmask that controls what the capability can be used for, such as loading or storing from the memory, or setting `pcc` to execute code.

Bounds. A capability with a set of permissions is not by default authorised to exercise them at all addresses. Instead, the capability also encodes a range of addresses within which it may exercise its permissions.

CHERI instructions operate on capabilities in accordance to security principles such as *privilege minimisation*, *monotonicity*, and *provenance*; these are enforced by checking the Validity Tag, Permissions, Bounds, and other information attached to capabilities [4]. For example, only a valid capability, with permission to load, and whose address is within its bounds, can be used to load from that memory address. Otherwise, the processor traps and potentially causes the program to crash. The checks performed by each CHERI instruction are known as its *guard conditions*, and the correctness of their hardware implementation is crucial to the security protections provided by CHERI.

III. Basics of CHERI-RISC-V

CHERI-RISC-V extends the RISC-V ISA with support for CHERI [4]. This case study treats its 64-bit variant.

A. Compression of Capabilities

When stored in memory, capabilities are represented in a compressed format [4], [15]. A compressed capability in 64-bit CHERI-RISC-V takes 128 bits (plus an out-of-band validity tag bit)—twice as many bits as a traditional pointer. In the capability registers of the core, however, they are represented in a decompressed format that occupies even more bits. Decompression and compression are done transparently when they are moved between memory and the core.

Capability compression is lossy. That is, there exist decompressed capabilities that do not correspond to any compressed capability. These decompressed capabilities are termed *unrepresentable*. Such a capability poses a significant problem if it appears in the core, since there is no well-defined way to store it to the memory—as that would require compressing the capability first. Part of our verification is to show that unrepresentable capabilities can never be created by the processor.

B. Sail CHERI-RISC-V Instruction Specification

The definition of each CHERI instruction in the Sail CHERI-RISC-V specification [16] roughly takes the form of Algorithm 1. An instruction can retire either *unsuccessfully*, due to violations of one of its guard conditions, or *successfully*, after modifying the architectural state of the processor. As will be seen in Section V-A, the distinction between successful and unsuccessful retirement is central to the way we specify instruction correctness in this work.

IV. Flute and CHERI-Flute

Flute [7] is a 64-bit RISC-V processor with a five-stage, in-order pipeline designed for low- to medium-end applications. The processor is designed in Bluespec SystemVerilog (BSV) and has been synthesised and tested on Xilinx FPGAs.

Flute has the basic pipelined microarchitecture commonly found in computer architecture textbooks [17], featuring a

Algorithm 1: Typical CHERI instruction specification

if $\neg$*guard condition 1* **then retire** FAIL(*TagViolation*);
else if $\neg$*guard condition 2* **then retire**
  FAIL(*PermitLoadViolation*);
...
else if $\neg$*guard condition 12* **then retire**
  FAIL(*LengthViolation*);
else
  modify architectural state;
  retire SUCCESS;
end

Fetch (F), a Decode (D), an Execute (E), a Memory (M), and a Write-back (W) stage. It also comes with forwarding mechanisms to make the pipeline more efficient. The register file (`regfile`) consists of 32 general-purpose registers $r_0, \ldots, r_{31}$, where r_0 is hardwired to zero.

Fig. 2 illustrates the pipeline of Flute with its stages occupied by instructions $\mathcal{I}_1, \ldots, \mathcal{I}_5$. Outgoing paths from stage M and W, including forwarding paths, are highlighted in red and blue respectively. These paths carry information about pending updates to the register file: the pending update in stage W writes the value $\mathtt{v}_W$ into register $\mathtt{rd}_W$, and the pending update in stage M writes the value $\mathtt{v}_M$ into register $\mathtt{rd}_M$.

To articulate properties, we define two subscripted register files: $\mathtt{regfile}_M$, which contains the contents of `regfile` after committing the pending update in stage W, and $\mathtt{regfile}_E$, which contains the contents of `regfile` after committing the pending updates in both stages W and M, in that order. The subscripted versions are essentially what the register file appears to be to stages M and E after forwarded values are taken into account. Hence their subscripts.

A. CHERI-Flute

CHERI-Flute [18] extends Flute with support for CHERI-RISC-V. We sketch here the main relevant changes.

First, the registers are widened to become hybrid registers that can be used as both integer and capability registers. Second, most of the computation supporting the CHERI instructions—calculating bounds, incrementing addresses, and so on—is implemented within the ALU located in stage E. Finally, circuitry is added to stage M that partially checks whether any CHERI instruction passing through it violates the instruction's guard conditions. The rest of the checks are performed earlier by the ALU. While these checks could in principle all be placed in the ALU, this would cause unacceptably long delays in stage E for certain instructions. Hence they are spread across stages E and M instead.

V. Formulating Correctness

Our formal verification flow is driven by JasperGold. The design is first compiled into SystemVerilog using the open-source `bsc` compiler and then imported into JasperGold. This pre-compilation is necessary because JasperGold cannot read the Bluespec SystemVerilog source of CHERI-Flute directly.

The specification for correctness, which in our case is the Sail CHERI-RISC-V specification, also needs to be mapped into properties—written as SystemVerilog Assertions (SVA)—about the compiled SystemVerilog design. Tooling does not exist to achieve this automatically, so for this case study we manually translated those portions of the Sail specification necessary for the verification effort into SVA. This yielded more than 1000 lines of data structures and functions of SystemVerilog and almost 100 correctness properties in SVA. As these properties are about a compiled design, a certain amount of 'reverse engineering' was needed to identify the relevant signal names.

A. The Instruction Specification Framework

A RISC-V processor is simple enough to formulate correctness of its instructions in the classical, direct way that will be familiar from many examples in the literature.

Let α be an abstraction function that maps each microarchitectural state of CHERI-Flute to a CHERI-RISC-V architectural state. Write $s \xrightarrow{\mathcal{I}} s'$ to mean that a CHERI-Flute processor retires instruction $\mathcal{I}$ and thereby transitions from microarchitectural state s to microarchitectural state s'. Similarly, write $S \xrightarrow{\mathcal{I}} S'$ to mean that, according to the CHERI-RISC-V specification, executing instruction $\mathcal{I}$ alters the architectural state S to architectural state S'. Note that both transition relations are deterministic.

Now for the implementation of an instruction $\mathcal{I}$ to conform to specification, we require that

$$\forall s\, s'.\; s \xrightarrow{\mathcal{I}} s' \implies \alpha(s) \xrightarrow{\mathcal{I}} \alpha(s') \tag{1}$$

where s ranges over the reachable microarchitectural states of CHERI-Flute. The reachability of s is, of course, crucial; this is further discussed in Section VI-B.

Now the formulation Prop. (1) faces a significant practical challenge. A CHERI instruction can be retired either successfully or unsuccessfully—and, in the latter case, there are sometimes more than a dozen ways in which it can fail. So formulating correctness as in Prop. (1) will require a full specification of what the processor's behaviour, and the resulting architectural state, should be for each kind of failure. This would be ideal, but also greatly increases the effort of formulating the required properties.

We therefore formulate a weaker notion of correctness that greatly simplifies the properties, albeit at the cost of a less comprehensive verification. Define two checkmarked relations as follows. For any instruction $\mathcal{I}$ and microarchitectural states s and s', the relation $s \xrightarrow{\mathcal{I}\checkmark} s'$ holds iff $s \xrightarrow{\mathcal{I}} s'$ and instruction $\mathcal{I}$ is retired successfully. And for any instruction $\mathcal{I}$ and architectural states S and S', the relation $S \xrightarrow{\mathcal{I}\checkmark} S'$ holds iff $S \xrightarrow{\mathcal{I}} S'$ and all instruction $\mathcal{I}$'s guard conditions are met.

Now, consider the property expressed by the proposition

$$\forall s\, s'.\; s \xrightarrow{\mathcal{I}\checkmark} s' \implies \alpha(s) \xrightarrow{\mathcal{I}\checkmark} \alpha(s') \tag{2}$$

which says that any *successful* retirement of instruction $\mathcal{I}$ occurs in compliance with the specification. Proving the stronger

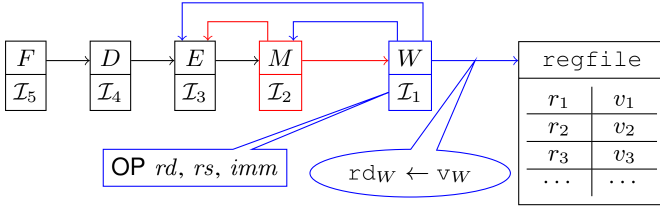

Fig. 3. Microarchitectural state with register-only instruction

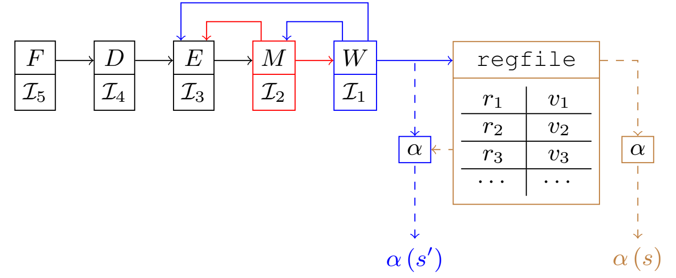

Fig. 4. Microarchitectural state with state abstractions

condition Prop. (1) shows the processor complies with the full specification indicated by Algorithm 1, which has numerous branches leading to different types of failures. Prop. (2) is a weaker condition but greatly simplifies the properties.

This simplified property cannot detect a faulty processor with incorrect *unsuccessful* retirement. That is, a processor that correctly prevents a certain CHERI instruction that violates its guard conditions from being retired at the end of the pipeline, but which nontheless produces an incorrect processor state according to the CHERI RISC-V specification. The property will, however, still detect processors with incorrect *successful* retirement. That is, processors that produce the wrong architectural state upon a CHERI instruction being retired the end of the pipeline, or processors that retire a CHERI instruction at the end of the pipeline that violates its guard conditions. This ensures that none of the security guarantees offered by CHERI is compromised. To see this, suppose for contradiction that Prop. (2) is true for some faulty processor which *incorrectly retires successfully* some instruction $\mathcal{I}$, i.e., there exist s and s' such that the relation $s \xrightarrow{\mathcal{I}\checkmark} s'$ holds but some of instruction $\mathcal{I}$'s guard conditions are not met. Consequently, by Prop. (2), the relation $\alpha(s) \xrightarrow{\mathcal{I}\checkmark} \alpha(s')$ also holds. But this implies that all of instruction $\mathcal{I}$'s guard *are* are met, which contradicts the assumption. Section IX discusses ways to relatively easily obtain properties that reflect the stronger specification.

B. Expressing Specifications as Properties

For mechanised formal verification in JasperGold, it is of course necessary to articulate the intent of the abstract correctness condition described by Prop. (2) as a group of SystemVerilog expressions. In practice, this means

(i) characterising the microarchitectural states s and s' for which $s \xrightarrow{\mathcal{I}\checkmark} s'$ holds, and
(ii) defining the mapping α for at least microarchitectural states s and s' where $s \xrightarrow{\mathcal{I}\checkmark} s'$ does hold.

Note that expressing (i) means characterising when the instruction $\mathcal{I}$ *has retired successfully*. One of the contributions of our methodology is to observe that this can be tied to the detection of certain microarchitectural states. Note also that (ii) is much simpler than having also to define the architectural states resulting from every kind of unsuccessful retirement.

In practice, we have developed these properties in separate groups for each of three distinct classes of instructions that share common structure. The sections that follow explain these. In the actual proof code, a systematic scheme of 'property templates' is employed to makes it easy to create and manage almost 100 properties without having to maintain multiple copies of boilerplate code. It also allowed us to quickly implement and validate proof engineering ideas for a large batch of properties, improving research efficiency.

C. Register-Only CHERI Instructions

A register-only CHERI instruction computes a function of its operands and writes a result into a given register, causing a trap if any of its guard conditions is not met.

Recall from Section V-B that two expressions are needed to formulate the required correctness properties. To express (i), consider Fig. 3, which shows the microarchitectural state when some register-only instruction $\mathcal{I}_1$ is in stage W. Denote this state by s and the state right after instruction $\mathcal{I}_1$ is retired by s'. Since stage W is at the end of the pipeline, any instruction reaching stage W is retired at the end of the current cycle. Moreover, any instruction reaching stage W can no longer cause traps, so it is bound to be retired successfully. Conversely, if a register-only instruction is retired successfully, then it must have been in stage W just before its retirement. So $s \xrightarrow{\mathcal{I}_1\checkmark} s'$ and (i) can be expressed simply by checking whether the given instruction is in stage W.

To express (ii), consider Fig. 4, which illustrates the microarchitectural state of CHERI-Flute in some state s that is about to successfully retire instruction $\mathcal{I}_1$ and enter state s', i.e., $s \xrightarrow{\mathcal{I}\checkmark} s'$. Hence $\alpha(s)$ and $\alpha(s')$ must give the architectural states right before and after instruction $\mathcal{I}_1$ is retired. Then observe that

- $\alpha(s)$ can be obtained directly from the current register file, $\mathtt{pcc}$, etc., and
- $\alpha(s')$ can be obtained by combining the current register file, $\mathtt{pcc}$, etc. with the pending updates contained in the output of stage W,

so (ii) can be expressed as a function of state s.

Given formulations of expressions (i) and (ii), the SVA property for a register-only instruction with register addresses rd and rs, and immediate data imm will say that if stage W contains an instruction with opcode OP, then

- $\mathtt{rd}_W = rd$,
- $\mathtt{v}_W = result_{\mathsf{OP}}(\mathtt{regfile}[rs], imm)$, and
- $guard_{\mathsf{OP}}(\mathtt{regfile}[rs], imm)$.

Where $result_{\mathsf{OP}}$ and $guard_{\mathsf{OP}}$ are SystemVerilog functions translated from the Sail specification of the instruction with opcode OP that compute its write-back result and guard conditions respectively.

D. Branching CHERI Instructions

A branching CHERI instruction redirects the control flow and (optionally) saves the return address in a given register. Of course, it also has guard conditions to ensure that the updated `pcc` has the right Bounds and Permissions. This creates an opportunity to decompose what a branching instruction does into two operations: checking its guard conditions and (optionally) saving the return address, and (conditionally or unconditionally) redirecting the control flow.

The first of these is just what a register-only instruction does, so we can simply reuse the property template developed in Section V-C. So the rest of this section is devoted to formulating the correctness properties about the second operation.

First, it is necessary to briefly explain how the control flow is managed in CHERI-Flute. Initially, stage F fetches an instruction from `fetch_addr` and predicts the address of the next instruction using the branch predictor. This predicted address (`pred_addr`) is *by default* used as the next `fetch_addr`, and it is also passed along the pipeline with the *currently* fetched instruction until it reaches stage E, where the ALU computes the correct address of the next instruction (`next_addr`). The processor then compares the computed `next_addr` with the `pred_addr` it received. If the two addresses do not match, then a branch misprediction has occurred, and stage F has been fetching the wrong instructions and passing them along the pipeline. To rectify this, `fetch_addr` is set to `next_addr`, and all pipeline stages prior to stage E are flushed. Otherwise, if the branch prediction has been correct, no flushing is needed and `fetch_addr` is updated in the default way.

Fig. 5 shows the microarchitectural state when some branching instruction $\mathcal{I}_3$ is in stage E. To formulate the correctness properties about control flow redirection, the framework developed in Section V-A is slightly generalised. Specifically, if a branching instruction $\mathcal{I}$ is in stage E and a branch misprediction has occurred, then instruction $\mathcal{I}$ is now considered 'about to be retired successfully' insofar as control flow redirection is concerned, and it is now considered to have been 'retired successfully' after `fetch_addr` is set to `next_addr`. This gives the expression (i) discussed in Section V-B. As for expression (ii), the architectural states of the processor right before and after some branching instruction is retired successfully are taken from the values of `fetch_addr` before and after that instruction is retired successfully, respectively.

E. Memory CHERI Instructions

A memory CHERI instruction loads from or stores to the memory using the capability (directly or indirectly) specified by its operands, causing a trap if any of its guard conditions is not met. What a memory instruction does can be decomposed into two operations: checking its guard conditions, and loading from or storing to the memory.

The correctness properties about the first operation can be formulated simply by reusing the property template developed in Section V-C. Hence this section focuses on formulating the correctness properties about the second operation.

CHERI-Flute is connected to the memory hierarchy through an interface consisting of several input and output ports, which must be properly used in order for the memory to function correctly. As with register-only instructions, a memory instruction $\mathcal{I}$ is about to be retired successfully when it is in stage W, after having sent and fulfilled its request to the memory in stage M. Thus, the correctness property should assert that before $\mathcal{I}$ is retired successfully, when it was in stage M, the memory interface had been properly used to fulfil what the specification requires of it. In our proof, SVA sequences are used to precisely specify the exact sequence of events that must have taken place when instruction $\mathcal{I}$ was in stage M.

Fig. 6 and Fig. 7 show how a memory *load* instruction $\mathcal{I}_2$ is moved from stage M to stage W and becomes ready to be retired successfully. The correctness property checks that

- a new memory request was not sent before the previous request had been fulfilled,
- a memory exception did not occur,
- the value returned from the memory when $\mathcal{I}_2$ was in stage M was decompressed correctly (if it was a capability) and used in the pending update to the register file, and
- the content of the pending update remains stable as $\mathcal{I}_2$ is moved from stage M to stage W.

The correctness properties about memory *store* instructions are highly similar and thus omitted here.

F. Processor Liveness

All correctness properties discussed so far are safety properties. Our verification also tackled the important issue of processor liveness—demonstrating that the processor does not freeze so that the pipeline never progresses.

Of course, there are challenges when dealing with liveness. First, it is usually very difficult to prove liveness properties in practice, and there is no such thing as a bounded proof for liveness that can at least give some confidence. Second, even if a liveness property is proved, there is still no guarantee about *when* the desirable event will occur, which is not ideal when performance is critical. Third, a necessary condition for a processor to exhibit liveness is the correct behaviour of the external components connected to it. For example, if the memory never fulfils a load request, then the processor might wait indefinitely for a response, stalling the pipeline. This can be ruled out by assuming certain fairness constraints about the external components, but these can of course potentially be violated unless they are themselves verified.

There is a conventional workaround to the first two problems. Instead of proving the liveness property that 'the pipeline eventually progresses', we derive a *safety* property that 'the pipeline progresses within n cycles' parametrised by n and search for the smallest n (if it exists) for which the safety

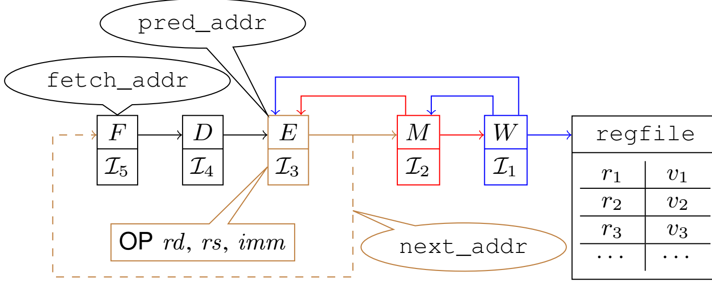

Fig. 5. Microarchitectural state with branching instruction

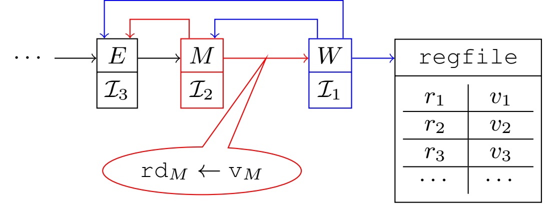

Fig. 6. Microarchitectural state with load instruction in stage M

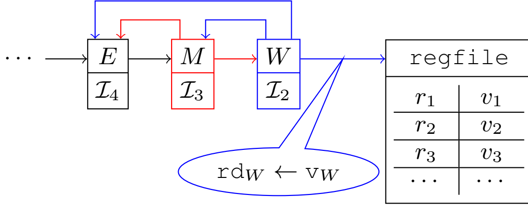

Fig. 7. Microarchitectural state with load instruction in stage W

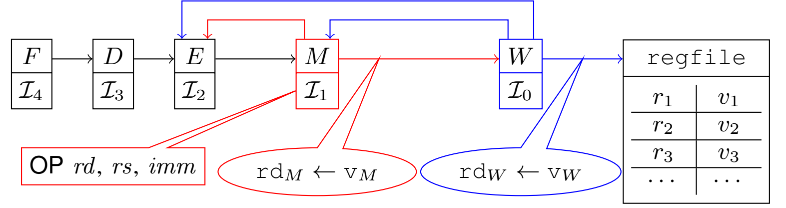

Fig. 8. Microarchitectural state with register-only instruction in stage M

property can be proved. This not only averts the difficulty of proving liveness properties but also generates a concrete bound on when the pipeline progresses.

The derived safety property we proved for CHERI-Flute says that if an instruction enters stage E, then within nine cycles, either a new instruction enters stage E, or the processor enters one of three special states, triggered by particular instructions, that requires it to wait for certain external signals.

This property shows that as long as the processor does not enter one of the special states, new instructions will enter stage E periodically, so the pipeline never freezes. The number 'nine' is the smallest number for which this property can be proved, and the focus on stage E is because certain RISC-V instructions are retired in stage E—i.e. they are never moved into stages M or W. Asserting this property on any stage *prior to* stage E always attracts a counterexample where an instruction is repeatedly issued but never reaches beyond stage E, effectively stalling the subsequent stages.

Of course, the proof of this property relies on several fairness constraints. Most notably, it is assumed that the memory always fulfils a request within *two* cycles. The number 'two' here is arbitrarily chosen, and it is reasonable to conjecture that a different number can be used without making any substantial difference other than perhaps affecting the number 'nine' in the derived safety property.

VI. Proof Engineering

Not all our correctness properties can be proved in a push-button manner. Specifically, those properties about register-only CHERI instructions as well as those about the register-only components of branching and memory CHERI instructions *cannot* be proved straightforwardly. Instead, proof convergence on these properties relies on proof engineering methodologies that are explained in this section.

A. Decomposing the Pipeline

This methodology is called 'decomposing the pipeline' because it enables one to prove some property about a desired instruction when it is in a *later* stage of the pipeline by first proving some lemmas about the instruction when it was in *earlier* stages of the pipeline.

1) The First Lemma: The correctness property shown in Section V-C for any register-only instruction cannot be proved directly in JasperGold. Instead, we prove a structurally identical version of the property that is 'pushed back' one stage in the pipeline, referencing $\mathtt{regfile}_M$ instead of $\mathtt{regfile}$, $\mathtt{rd}_M$ and $\mathtt{V}_M$ instead of $\mathtt{rd}_W$ and $\mathtt{V}_W$, and using a suitably adjusted $guard^M_{\mathsf{OP}}$ function, as we sketch below.

If this version of the property can be proved, then it can be used as a lemma to successfully prove the original correctness property through k-induction [19]. The lemma is a property of a register-only instruction in stage M instead of stage W. Observe that the write-back result of any register-only instruction is computed by the ALU in stage E. Therefore, for any register-only instruction $\mathcal{I}_1$ in stage M with opcode OP as illustrated in Fig. 8, its write-back result must already be available in $\mathtt{v}_M$. This means that we can assert

$$\mathtt{v}_M = result_{\mathsf{OP}}\left(\mathtt{regfile}_M\left[rs\right], imm\right)$$

in the lemma, where the subscripted $\mathtt{regfile}_M$ is used to take into account any forwarded value $\mathtt{v}_W$ from stage W.

Now recall from Section IV-A that checks for guard conditions are spread across stages E and M. Thus, when instruction $\mathcal{I}_1$ reaches stage M, only the checks in stage E have been performed, whereas the checks in stage M are still underway. Therefore, it is incorrect to assert that

$$guard_{\mathsf{OP}}\left(\mathtt{regfile}_M\left[rs\right], imm\right)$$

in the lemma. Rather, the lemma only asserts that the subset of instruction $\mathcal{I}_1$'s guard conditions that are checked in stage E have been met. This subset is given by $guard_{\mathsf{OP}}^{M}$.

Given the lemma, the original correctness property can be proved by k-induction. But without it, k-induction is unable to converge because for any value of k, the SAT-solver can always find a trace that violates the inductive hypothesis. Such a trace would begin at an *unreachable* microarchitectural state where the desired instruction is in stage M. It would then *stall* the pipeline during the next $(k-1)$ steps, only moving the desired instruction to stage W at the $(k+1)$-th step, where the inductive hypothesis fails to hold. The pipeline can stall for arbitrarily many cycles in such traces due to the absence of the very fairness constraints that enable the proof of the liveness properties in Section V-F. However, it is unnecessary to add fairness constraints here. Instead, we use the given lemma to prevent the SAT-solver from exploring such unreachable states. And since stage M is immediately prior to stage W, $k = 1$ is sufficient for the proof to converge.

2) The Second Lemma: To actually prove the lemma just explained, the same methodology is simply reapplied. That is, a *second* lemma is used to narrow the space of states in which the desired instruction is in stage E so as to exclude traces that violate the first lemma.

Fortunately, this second lemma is relatively easy to discover, since the only state information contained in stage E is the decoded content of the current instruction in stage E. Thus, the second lemma simply needs to assert that any instruction in stage E is properly decoded, which enables the proof of the first lemma by 1-induction.

Now this second lemma can, in turn, be proved by 1-induction if a similar *third* lemma is proved about stage D. And so on. This chain of lemmas stops, of course, at stage F where the last lemma can be proved directly. In practice, however, since CHERI-Flute's design of stages F and D is relatively simple, we took advantage of one of JasperGold's black-box proof engines to automatically complete the proof.

B. Developing Microarchitectural Invariants

CHERI instructions compute relatively sophisticated functions of their operands. In the Sail specification, these are given by total functions on all decompressed capabilities, including the unrepresentable ones mentioned in Section III-A. But since unrepresentable capabilities pose a significant problem if they appear in the processor, CHERI-Flute is designed so they can never be created by the hardware in the first place. CHERI-Flute is then excused from conformance with the specification for unrepresentable capabilities.

This, of course, leads to the generation of unreachable counterexamples in model checking, so our verification includes a global consistency invariant over the entire processor, showing that only representable capabilities are present. Formulating and proving this invariant was challenging because there are many internal registers in CHERI-Flute's microarchitecture that can influence the architecturally visible registers. A weak invariant that does not cover these internal registers cannot be proved by k-induction since the SAT-solver can always find an unreachable state in which one of these registers contains an unrepresentable capability, which then 'pollutes' one of the architecturally visible registers within the next few cycles.

This challenge was overcome using State-Space Tunnelling, a JasperGold feature that allows the user to prune unreachable portions of the state space when performing k-induction proofs. Essentially, it allows us to specify some k and let the SAT-solver generate a trace of length k that violates the invariant. The user then examines this trace to identify any internal register that causes the violation, and manually strengthens the invariant to include it.

This process repeats until, for some sufficiently large k, no violating trace can be found, at which point proof convergence for the invariant is achieved. In the end, the invariant in our proof was sufficiently strong to be proved by 1-induction.

VII. Results and Evaluation

In this case study, the implementations of all 80-plus CHERI instructions (except a very few not yet implemented) have been subject to formal verification in JasperGold against the correctness properties in Section V through the proof engineering methodologies in Section VI.[1] While the implementations of most instructions were found to satisfy the correctness properties, several were found to be buggy.

The bugs found roughly fell into two categories. The first category are simple coding mistakes: the designer failed to notice details of the specification, or the specification changed after the design was created. These bugs are usually detectable with a moderate amount of scrutiny or simulation testing. The second category are algorithmic errors, typically caused by subtle mistakes in complex pieces of logic. These are much more difficult to uncover, even with the most intensive code review or simulation testing.

- In the `incOffsetFat` function, a bit vector is truncated but subsequent code still uses the old non-truncated value. This can potentially lead to the creation of unrepresentable capabilities for certain inputs.
- Several CSR registers are not initialised to the null capability when the processor is reset.

These two bugs have been confirmed and fixed by the designers [11], [13]. The following have also been confirmed by the designers and fixes are pending:

- The `getTop` function incorrectly truncates the returned value.
- `AUIPCC` incorrectly clears the validity tag of the returned capability for certain inputs.
- `CUnseal` fails to check a permission bit.
- `CCSeal` incorrectly causes the processor to trap for certain inputs.

One final bug illustrates an especially productive collaboration between verification and design: in the `setAddress`

[1]On a 24-core AMD EPYC 7F72 processor, with 256 GB of RAM, the proofs are completed within two hours through parallelisation.

function, the validity tag of the returned capability is cleared incorrectly in a corner case.

This function was originally developed by trial and error using the BlueCheck automated test generation framework [20] and as well as TestRIG, a framework for testing RISC-V processors with random instruction generation [21]. But neither method detected this corner case. The designers' initial patch for the function was buggy because it mishandles another corner case, which was yet again detected by formal verification. Consequently, we redesigned the function from scratch and formally verified its correctness against the specification before it was submitted to and accepted by the designers [12].

A. Bug or Feature?

Two issues belong to an interesting category sometimes encountered in formal verification: a trace violates the specification, but it is unclear whether the hardware should be changed to match the specification or *vice versa*.

The first was that specification requires the `CSetOffset` and `CIncOffset` instructions perform a standard 'representability check' to determine if the capabilities they return are representable. But in CHERI-Flute the `CSetOffset` instruction performs a slightly different, non-standard check optimised for that particular instruction, although the `CIncOffset` instruction uses use the standard check.

So the behaviour of the `CSetOffset` instruction violates the specification, but in a beneficial way. It is therefore up to the designers to decide whether the specification should be changed to incorporate this optimised representability check.

The second was that, when trying to prove the global consistency invariant, we found counterexample traces where memory corruption causes injects corrupted capabilities into the core. Since memory bit-flips do occur in actual hardware, we suggested that the core should perform sanity checks on any capability retrieved from the memory, clearing its validity tag if it is found to be corrupted.

In the end, the designers decided not to add the sanity checks because it may cause even more unexpected behaviour when memory corruption occurs, making the situation more complex to debug. So to make the proof of the global consistency invariant converge, we added an assumption that the memory never returns a corrupted capability.

VIII. Related Work

The correctness of processor cores and their implementation of instructions has been a focus of verification research for decades, going at least back to the pioneering work of Hunt on verifying the FM8501 [22] and FM8502 processors [23]. To verify more complicated, pipelined designs, Burch and Dill devised the flushing abstraction [24], a member an extensive family of formulations of correctness that has expanded to cover even out-of-order designs. Aagaard et al. [25] present a useful framework for classifying these different approaches.

From about the mid 1990s, verification was increasingly adopted in industry to verify critical components of large-scale designs. Notable experiments include Kaivola et al.'s verification of the Pentium 4 floating-point divider [26], Jacobi et al.'s fully automated verification of fused-multiply-add floating-point units [27], Kaivola's methodology for large-scale formal verification of control-intensive circuits [28], and Slobodova's verification of AES hardware support [29]. A landmark achievement in this direction was Kaivola et al.'s work on replacing testing with formal verification for validating the core execution cluster of the Core i7 design [30].

The starting point of our work was Reid et al.'s end-to-end verification of Arm processors [31]. But our approach to verifying properties differs significantly from this work. While the Arm verification uses bounded model checking, we obtained much stronger unbounded proofs of all correctness properties by extracting microarchitectural invariants. Of course, the relative simplicity of RISC-V helped make this possible, but it was also enabled by the complexity management methodologies we explain in this paper.

A landmark in the verification of complex cores is the work by Goel et al. [32] on verifying x86 instructions. This was done using the ACL2 theorem prover in concert with a number of tightly integrated support tools, and achieved an end-to-end verification that encompasses decoding, translation into microcode, traps to microcode ROM, and execution.

There has been related work on verifying processors using Symbolic Quick Error Detection (SQED) and its variants [33], [34], [35]. These methodologies use bounded model checking to find sequence-dependent bugs that violate a self-consistency property, but they are not intended for checking single-instruction bugs where an instruction always produces the wrong result for certain inputs [33]. In contrast, our methodology checks for both types of bugs. Indeed, most, if not all of the bugs we found were single-instruction bugs that could not be uncovered by checking for self-consistency. Instead, a more traditional approach using a formal specification was required.

IX. Conclusions and Prospects

There are several ways in which the present work can be improved and extended.

For this project, we manually translated the Sail specification of CHERI-RISC-V into SVA. It would obviously be preferable to have an automatic translation, and we are investigating some options for this. Apart from the usual benefits of automation, automatic translation could eliminate the pragmatic need to weaken the specification as described in Section V-A. As Sail has been adoped by the RISC-V Foundation for its golden formal model, a flow from Sail to SVA seems highly desirable in any case.

Further work can also be done to address the drawbacks of the liveness properties described in Section V-F. For example, it would be ideal to remove the proof's reliance on fairness constraints that contain arbitrarily chosen numbers. Also, the work can be made more complete by proving liveness properties about pipeline stages subsequent to stage E.

Attempts could be made to verify more complex CHERI-RISC-V processors, such as Toooba [36], where the main challenge will be to formulate correctness properties about

an *out-of-order* microarchitecture. We note, however, that the SystemVerilog functions translated from the Sail specification during the present work can be completely reused when formulating the new correctness properties.

Finally, we mention that in 2019, the UK announced its *Digital Security by Design* programme with £190 million of funding for a set of research projects [37] to 'radically update the foundation of our insecure digital computing infrastructure, by demonstrating that mainstream processor technology ... can be updated to include new security technologies based on the CHERI Architecture' [38]. A cornerstone of the programme is Morello [39], a CHERI-enabled prototype developed by Arm and scheduled for release in late 2021. We hope that this early RISC-V case study provides at least some insights that might eventually apply in the formal verification of Morello.

X. Acknowledgements

We are grateful to members of the CHERI group at Cambridge. Alasdair Armstrong, Alexandre Joannou, Simon Moore, Peter Rugg, Peter Sewell, Robert Watson, and Jonathan Woodruff all kindly provided assistance or comments on this work. Thanks also go to Ziyad Hanna at Cadence and to Joe Stoy at Bluespec, who thoughtfully answered our questions about Bluespec SystemVerilog. This work was funded in part by the UKRI programme on Digital Security by Design (Ref. EP/V000225/1, SCorCH [40]).

References

[1] M. Miller. (2019, February) Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. Presented at the BlueHat IL. [Online]. Available: https://msrnd-cdn-stor.azureedge.net/bluehat/bluehatil/2019/assets/doc/Trends%2C%20Challenges%2C%20and%20Strategic%20Shifts%20in%20the%20Software%20Vulnerability%20Mitigation%20Landscape.pdf

[2] D. Chisnall, C. Rothwell, R. N. M. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. G. Neumann, "Beyond the PDP-11: Architectural support for a memory-safe C abstract machine," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, March 2015, pp. 117–130.

[3] J. Woodruff, R. N. M. 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 *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, June 2014, pp. 457–468.

[4] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, H. Almatary, J. Anderson, J. Baldwin, G. Barnes, D. Chisnall, J. Clarke, B. Davis, L. Eisen, N. W. Filardo, R. Grisenthwaite, A. Joannou, B. Laurie, A. T. Markettos, S. W. Moore, S. J. Murdoch, K. Nienhuis, R. Norton, A. Richardson, P. Rugg, P. Sewell, S. Son, and H. Xia, "Capability hardware enhanced RISC instructions: CHERI instruction-set architecture (version 8)," University of Cambridge Computer Laboratory, Technical Report UCAM-CL-TR-951, October 2020.

[5] K. Nienhuis, A. Joannou, T. Bauereiss, A. Fox, M. Roe, B. Campbell, M. Naylor, R. M. Norton, S. W. Moore, P. G. Neumann, I. Stark, R. N. M. Watson, and P. Sewell, "Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2020, pp. 1003–1020.

[6] A. Armstrong, T. Bauereiss, B. Campbell, S. Flur, K. E. Gray, P. Mundkur, R. M. Norton, C. Pulte, A. Reid, P. Sewell, I. Stark, and M. Wassell, "Detailed models of instruction set architectures: From pseudocode to formal semantics," in *Proceedings of the 25th Automated Reasoning Workshop: Bridging the Gap between Theory and Practice: ARW 2018*. University of Cambridge, April 2018, pp. 23–24.

[7] Bluespec announces flute. [Online]. Available: https://bluespec.com/2018/12/13/bluespec-announces-flute/

[8] CHERI-RISC-V. [Online]. Available: https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-risc-v.html

[9] Jaspergold formal verification platform. [Online]. Available: https://www.cadence.com/ko_KR/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html

[10] Dpgao/FMCAD2021. [Online]. Available: https://github.com/dpgao/FMCAD2021

[11] Use tmpAddr in place of pointer in incOffsetFat following Dapeng Gao's ...CTSRD-CHERI/Cheri-Cap-Lib@508da81. [Online]. Available: https://github.com/CTSRD-CHERI/cheri-cap-lib/commit/508da818baa0d3d103c563c148b6ef2dc4aba057

[12] Use Dapeng's algorithm (adapted from his verified verilog) for ...CTSRD-CHERI/Cheri-Cap-Lib@6e8df02. [Online]. Available: https://github.com/CTSRD-CHERI/cheri-cap-lib/commit/6e8df025326565f146c2fd11d3e2d7fbebec61af

[13] Fix some CSRs with missing reset logic spotted by Dapeng Gao · CTSRD-CHERI/Flute@36de38d. [Online]. Available: https://github.com/CTSRD-CHERI/Flute/commit/36de38d87740baeed6ebbc242a872206d9f0b032

[14] "An Introduction to CHERI," Computer Laboratory, Technical Report UCAM-CL-TR-941, Sep. 2019.

[15] J. Woodruff, A. Joannou, H. Xia, A. Fox, R. M. Norton, D. Chisnall, B. Davis, K. Gudka, N. W. Filardo, A. T. Markettos, M. Roe, P. G. Neumann, R. N. M. Watson, and S. W. Moore, "CHERI Concentrate: Practical Compressed Capabilities," *IEEE Transactions on Computers*, vol. 68, no. 10, pp. 1455–1469, Oct. 2019.

[16] CTSRD-CHERI/sail-cheri-riscv: CHERI-RISC-V model written in Sail. [Online]. Available: https://github.com/CTSRD-CHERI/sail-cheri-riscv

[17] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, third edition ed. Pearson, 2016.

[18] CTSRD-CHERI/Flute: RISC-V CPU, simple 5-stage in-order pipeline, for low-end applications needing MMUs and some performance. [Online]. Available: https://github.com/CTSRD-CHERI/Flute

[19] M. Sheeran, S. Singh, and G. Stålmarck, "Checking Safety Properties Using Induction and a SAT-Solver," in *Formal Methods in Computer-Aided Design*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, vol. 1954, pp. 127–144.

[20] CTSRD-CHERI/bluecheck: A generic test bench written in Bluespec. [Online]. Available: https://github.com/CTSRD-CHERI/bluecheck

[21] CTSRD-CHERI/TestRIG: Testing processors with Random Instruction Generation. [Online]. Available: https://github.com/CTSRD-CHERI/TestRIG

[22] W. A. Hunt, *FM8501: A Verified Microprocessor*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1994, vol. 795.

[23] ——, "Microprocessor design verification," *Journal of Automated Reasoning*, vol. 5, no. 4, pp. 429–460, Dec. 1989.

[24] J. R. Burch and D. L. Dill, "Automatic verification of pipelined microprocessor control," in *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, vol. 818, pp. 68–80.

[25] M. D. Aagaard, B. Cook, N. A. Day, and R. B. Jones, "A Framework for Microprocessor Correctness Statements," in *Correct Hardware Design and Verification Methods*. Springer Berlin Heidelberg, 2001, vol. 2144, pp. 433–448.

[26] R. Kaivola and K. Kohatsu, "Proof engineering in the large: Formal verification of pentium 4 floating-point divider," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 4, no. 3, pp. 323–334, May 2003.

[27] C. Jacobi, Kai Weber, V. Paruthi, and J. Baumgartner, "Automatic Formal Verification of Fused-Multiply-Add FPUs," in *Design, Automation and Test in Europe*. Munich, Germany: IEEE, 2005, pp. 1298–1303.

[28] R. Kaivola, "Formal verification of Pentium components with symbolic simulation and inductive invariants," in *Computer Aided Verification*. Springer Berlin Heidelberg, 2005, vol. 3576, pp. 170–184.

[29] A. Slobodova, "Formal Verification of Hardware Support for Advanced Encryption Standard," in *2008 Formal Methods in Computer-Aided Design*. IEEE, Nov. 2008, pp. 1–4.

[30] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, and A. Naik, "Replacing testing with formal verification in Intel core I7 processor execution engine validation," in *Computer Aided Verification*. Springer Berlin Heidelberg, 2009, vol. 5643, pp. 414–429.

[31] 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 ARM Processors with ISA-Formal," in *Computer Aided Verification*. Springer International Publishing, 2016, vol. 9780, pp. 42–58.

[32] S. Goel, A. Slobodová, R. Sumners, and S. Swords, "Verifying x86 instruction implementations," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*. ACM, 2020, pp. 47–60.

[33] D. Lin, E. Singh, C. Barrett, and S. Mitra, "A structured approach to post-silicon validation and debug using symbolic quick error detection," in *2015 IEEE International Test Conference (ITC)*. IEEE, pp. 1–10. [Online]. Available: http://ieeexplore.ieee.org/document/7342397/

[34] E. Singh, K. Devarajegowda, S. Simon, R. Schnieder, K. Ganesan, M. Fadiheh, D. Stoffel, W. Kunz, C. Barrett, W. Ecker, and S. Mitra, "Symbolic QED Pre-silicon Verification for Automotive Microcontroller Cores: Industrial Case Study," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, pp. 1000–1005. [Online]. Available: https://ieeexplore.ieee.org/document/8715271/

[35] F. Lonsing, K. Ganesan, M. Mann, S. S. Nuthakki, E. Singh, M. Srouji, Y. Yang, S. Mitra, and C. Barrett, "Unlocking the Power of Formal Hardware Verification with CoSA and Symbolic QED: Invited Paper," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, pp. 1–8. [Online]. Available: https://ieeexplore.ieee.org/document/8942096/

[36] CTSRD-CHERI/Toooba: RISC-V Core; superscalar, out-of-order, multi-core capable; based on RISCY-OOO from MIT. [Online]. Available: https://github.com/CTSRD-CHERI/Toooba

[37] Digital security by design challenge – UKRI. [Online]. Available: https://www.ukri.org/our-work/our-main-funds/industrial-strategy-challenge-fund/artificial-intelligence-and-data-economy/digital-security-by-design-challenge/

[38] Department of Computer Science and Technology – CHERI: The Digital Security by Design (DSbD) Initiative. [Online]. Available: https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/dsbd.html

[39] Department of Computer Science and Technology – CHERI: The Arm Morello Board. [Online]. Available: https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-morello.html

[40] SCorCH: Secure code for capability hardware. [Online]. Available: https://scorch-project.github.io