Leveraging Invariants for Scalable Verification of RISC-V Cryptography Extensions

Kim Fahrni
ETH Zürich
Zürich, Switzerland
kfahrni@ethz.ch

Katharina Ceesay-Seitz
ETH Zürich
Zürich, Switzerland
kceesay@ethz.ch

Denis Zuppiger
ETH Zürich
Zürich, Switzerland
zdenis@ethz.ch

Kaveh Razavi
ETH Zürich
Zürich, Switzerland
kaveh@ethz.ch

Abstract

RISC-V has recently ratified a vector cryptography extension. Exhaustive formal verification of hardware designs that implement this extension is crucial for security. However, scaling verification for designs with such large bit-widths is challenging. We present the first formal verification of Marian, an open-source implementation of the RISC-V vector cryptography extensions. We show that proof modularization enables us to obtain an unbounded proof for Marian. Together with our systematic invariant identification, we reach a speedup of 2.7×. Our evaluation shows that invariants that assert properties of counters, such as bounds or directions, or handshakes are particularly effective in improving the verification times. We show the generalizability and reusability of our invariant identification methodology by formally verifying another custom implementation of RISC-V vector cryptography extensions. During verification, we found a violation that turned out to be a flaw in the specification, which is now being updated.

CCS Concepts

• **Hardware** → **Equivalence checking**; **Model checking**; • **Security and privacy** → *Cryptography*.

Keywords

Formal verification, invariants, hardware security, accelerators.

ACM Reference Format:
Kim Fahrni, Katharina Ceesay-Seitz, Denis Zuppiger, and Kaveh Razavi. 2026. Leveraging Invariants for Scalable Verification of RISC-V Cryptography Extensions. In *63rd ACM/IEEE Design Automation Conference (DAC '26), July 26–29, 2026, Long Beach, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3770743.3804151

1 Introduction

With the ever-increasing number of security vulnerabilities in software, the trend goes towards implementing critical components such as cryptographic algorithms in hardware. Due to their deterministic state space, hardware designs can, in principle, be exhaustively verified. However, formally verifying complex designs like cryptographic accelerators faces scalability challenges in terms of proof time and engineering effort [33]. In this paper, we present the first formal verification of a RISC-V vector cryptography extension implementation, Marian [36]. Using our generic invariant identification methodology, combined with modularization, we obtain unbounded proofs.

Cryptographic hardware. Cryptographic algorithms are often implemented in hardware to provide higher security assurance and to speed up calculations [34]. Proving their correct implementation is indispensable for security. Verification of such cryptographic hardware often relies on designs written in domain-specific languages [12, 20] or custom hand-written specifications and tooling [2, 34]. RISC-V has recently introduced a *vector* cryptography Instruction Set Architecture (ISA) extension [27] in addition to the scalar one. Given the open nature of the RISC-V ISA, hardware implementations are increasingly available to the public, enabling the formal verification of such implementations to be carried out in the open for the first time. However, finding a scalable methodology that can verify complex data-path designs with large bit-widths, like a vector cryptography accelerator written in a standard Hardware Description Language (HDL), remains an open challenge.

Formal hardware verification. Hardware model checking is gaining popularity despite requiring substantial human effort and facing scalability challenges [32]. To reduce the human effort, recent formal data-path verification tools allow the verification of a hardware design against a high-level C/C++ specification [7]. We find that the official RISC-V ISA simulator, Spike, contains suitable C++ code that we can use as a reference model for verification of vector cryptography accelerators. Yet, proof scalability remains challenging. A common strategy to achieve proof convergence is the addition of helper invariants that aid the model checker in finding a proof [7, 25]. These invariants are seldom published since they are often highly design-specific, and hardware designs were for a long time mostly closed source for commercial reasons [26]. As such, the field currently lacks a generic methodology on how to identify and evaluate the effectiveness of invariants.

Scaling the verification of cryptographic accelerators. Leveraging standard tooling [7], we verify the first open-source available RISC-V vector cryptography accelerator, Marian [36], against Spike's C++ implementation. Due to the complexity of the design, the proofs initially do not converge. To solve the scalability challenges, we modularize the proof and develop a generic invariant identification methodology, suspecting that invariants over signals with large bit-width or large cone of influence will be most effective. Our evaluation methodology reveals that this is not always the case, finding invariants on counters and handshake signals having the largest influence on proof bounds. We find that proof modularization allows us to obtain an unbounded proof and together with our discovered invariants we achieve a proof speedup of 2.7×. We evaluate the generalizability of our methodology by applying it to a

DAC '26, Long Beach, CA, USA

ACM ISBN 979-8-4007-2254-7/2026/07
https://doi.org/10.1145/3770743.3804151

second, custom implementation of a part of the RISC-V vector cryptography extension and analyze similarities. Using our invariant identification methodology, its proof converges.

Contributions. In summary, we make the following contributions:

- We formally verify an open-source vector cryptography accelerator [36] for the first time, making use of an industrial C++ to Register Transfer Level (RTL) equivalence checker [7].
- We devise a methodology for identifying and evaluating invariants for proof speedup and evaluate it on the first open-source RISC-V vector cryptography accelerator, Marian [36] and a custom implementation of some of the RISC-V vector cryptography instructions.
- We find a mismatch between the Marian implementation and the Spike model, revealing a mistake in the ratified RISC-V vector cryptography extension *specification*, which is now being updated due to our finding.

Open sourcing. We open source our scripts and evaluation results as much as licenses allow at [14].

2 Background & Motivation

We first provide background on the RISC-V vector cryptography extension and its first full implementation, Marian [36]. Then, we discuss formal verification challenges.

2.1 RISC-V vector cryptography extension

The RISC-V vector cryptography extension [27] defines vector instruction sets for accelerating cryptographic algorithms: `zvknha/b` for SHA-2 (Secure Hashing Algorithm), `zvksh` for ShangMi 3, `zvksed` for ShangMi 4, `zvkg` for GCM (Galois/Counter Mode), `zvkned` for AES (Advanced Encryption Standard), and `zvkb` for basic bit-manipulation operations. Marian [15, 36] is the first open-source implementation of this extension. It integrates a *Crypto* unit into Ara [23]. Ara is a RISC-V vector processing unit with multiple parallel 'lanes', each containing its own registers and arithmetic units. Ara's sequencer distributes tasks and data across lanes. Marian interfaces with the sequencer to obtain instruction information and with the lanes to read and write operands. Its crypto unit implements all vector cryptography extensions except `zvkb`, which is integrated into Ara existing arithmetic logic unit.

Figure 1 visualizes Marian. Marian must process vector cryptography operands of up to 256 bits, but Ara only provides vector registers up to 64 bits in size and restricts the per-cycle register transfer bandwidth of each lane to three registers in parallel. Marian therefore assembles operand fragments from multiple register groups in the 'operand collector' stage before forwarding them to the 'execution units' for cryptographic operations. The 'write-back' stage splits the results into smaller elements for storage in the lanes.

Any deviation from the specified behavior in a cryptographic accelerator risks confidentiality or integrity violations. Thus, the first research question we address is:

> **Research question 1.** How can we guarantee the correctness of a vector cryptography accelerator?

We answer this question in Section 3 by proposing a methodology based on formal verification.

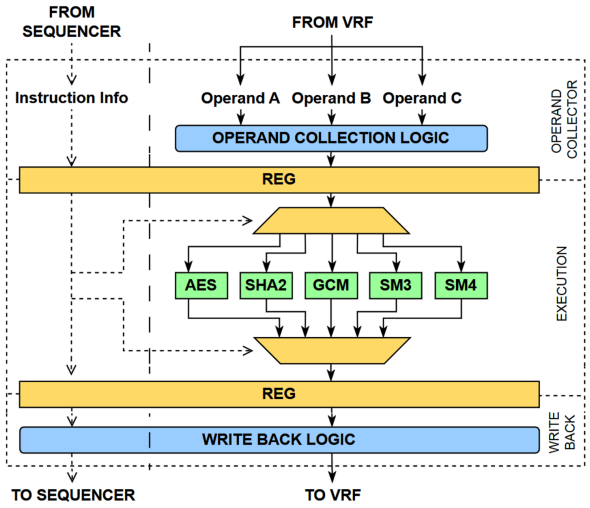

Figure 1: Marian's crypto unit architectural diagram [36]

2.2 Formal verification

Formal verification [31] mathematically proves a design's compliance with its specification for all input–state combinations [1]. The most common approach is to use SystemVerilog Assertions (SVA) to specify properties and prove them with a model checker [5, 32].

Equivalence checking. Sequential equivalence checking proves that untimed C/C++ models match their timed RTL implementations [7, 35]. Jasper C2RTL [7] maps RTL ports and internal signals to a model, allowing the use of SVAs to prove their equivalence.

Proof bound. The proof bound denotes the number of RTL clock cycles for which an assertion is proven. Only unbounded proofs guarantee full compliance. In practice, proofs often suffer from state-space explosion [31, 33]. Certain elements such as FIFOs, counters, or large bit-widths are known to increase complexity [8, 11].

Invariants. Model checking algorithms can use results of already proven assertions to speed up the proof of others. Verification engineers therefore add design-specific helper invariants which prove fast and are then converted into assumptions, reducing the reachable state space for remaining proofs [6]. Such invariants are rarely published as they are highly design-specific [26]. There is a lack of systematic methodologies on how to find the invariants, which leads us to the next research question:

> **Research question 2.** How can we systematically identify invariants for arithmetic units with large input operands?

We answer this question by defining an invariant identification methodology in Section 4 and we evaluate it in Section 5.

3 Verification Strategy

In this section, we describe our verification strategy for Marian. Our goal is to prove that the calculation's performed by Marian's crypto unit match the RISC-V vector cryptography extension specification [27] at all times.

Selecting a reference model. For the base RISC-V ISA, RTL designs can be proven against the official Sail specification [24, 28].

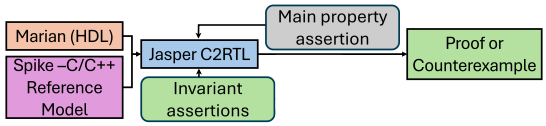

Figure 2: We prove Marian's HDL [15] against the Spike C++ model [29] using a sequential equivalence checker (Jasper C2RTL [7]), and speed up verification with invariants.

However, Sail does not yet support the *vector* cryptography extension, and its scalar crypto instructions differ from the vector ones, preventing direct comparison. Implementing our own model would risk specification errors. Since Spike [29], the former official ISA simulator maintained by RISC-V International, already supports the vector cryptography extension, we use it as reference model.

Verification flow. We choose to perform sequential equivalence checking between Marian's RTL implementation [15] and the Spike model written in C++. Figure 2 visualizes the workflow. We define the following main property that we aim to prove:

> **Main property:** When providing the same inputs to the Spike model and Marian, and Marian signals calculation completion, the output of the crypto unit is equivalent to Spike's output.

3.1 Solving the scalability challenge

When attempting to verify Marian without using any complexity reducing methods, the proof of the main property stalls after around 2 hours, reaching at most 8 clock cycles. To obtain unbounded proofs, we employ various strategies that we discuss next.

Input constraints. First, we limit the input state space of Marian to valid inputs from Ara only. Marian expects valid instructions from Ara whenever the corresponding 'valid' flag at Marian's input is set. Since verifying Ara [23] is out of scope of this work, we constrain Marian's instruction bus to receive valid instructions whenever the flag is set. Specifically, we constraine the instruction information to be limited to an operation implemented by Marian's crypto unit and the element-group width and size to values specified in the RISC-V standard [27], and the crypto unit's enable flag to be high. We additionally apply a fairness constraint [24], ensuring that once a valid instruction is issued, valid operands arrive within 10 cycles, preventing scenarios where Marian waits indefinitely on operands.

Invariants. To speed up the proofs we want to identify invariants that reduce the state space as much as possible when used as assumptions. By examining the code, it is difficult to tell which invariant will have a higher or lower impact, in particular, for verification engineers with little experience or little familiarity with the code. To overcome this challenge, we develop a systematic invariant identification methodology, detailed in Section 4.

Modularization. A common method to reduce complexity is modularization [19, 39]. We modularize the proof of our main property by verifying each Marian pipeline stage separately. The *operand-collection stage* assembles operands and forwards them unchanged. We prove this via an SVA assertion using a standard model checker. We verify the *execution stage*, which performs the computations, is with C2RTL against Spike [29]. Finally, we check the *write-back stage* with an SVA assertion, ensuring correct disassembly of execution results into target registers. Results are shown in Section 5.

Grouping	Category Name	Abbreviation	Bit-width
Data	Regular Data	D	32-256
	Datapath Control	DC	1-4
	Pipeline	P	1-256
State	Finite State Machine	FSM	2-8
	Counter	C	4-16
Control	Internal Control	IC	1-4
	Handshake	H	1

Table 1: Invariant Categories

Valid modularization. Input constraints on submodules must allow all valid values of outputs of modules they read from. This is given as we only constrain inputs that are passed through other modules unmodified and are derived from design inputs.

4 Invariant Identification

We present our methodology for identifying effective invariants for new designs. We initially hypothesize that the bit size and cone of influence of the signals referenced by the invariants would primarily contribute to their impact. The cone of influence for a signal refers to the number of state elements that have an influence on the state of the observed signal. Based on this hypothesis, we define invariant categories.

Invariant categories. We define 7 generic invariant categories, listed in Table 1, based on functionalities and signals that are common in hardware designs. We assign each invariant exclusively to the category where it most fits. Table 1 displays the average bit-width of signals for invariants in each category in accordance with our hypothesis that invariants that regard wider signals have a larger impact on proof speedup. These average bit-widths are composed both from the definitions of such signals (H, D, DC, IC) and from averages we determine based on signals in arithmetic unit designs (C, FSM, P).

4.1 Invariant Categories

4.1.1 Data invariants.

Regular Data Invariants relate to signals carrying data values and assert that these values are correct, either through arithmetic relations or consistency with a reference model. Data elements in arithmetic units are generally wider signals than the rest and are the only part of the design where significant calculations are performed consistently.

Datapath Control Invariants relate to signals that control the flow of primary data and those holding the data values themselves. We distinguish them from regular data invariants because they involve both control and data signals, yet remain fundamentally tied to the datapath rather than general control logic. For example, an invariant may verify correct multiplexer routing or input selection.

Pipeline Invariants relate to data and control signals crossing pipelined stages. As these invariants require more temporal reasoning than regular data invariants, they form a separate category. An example invariant ensures that signal values propagate correctly between pipeline stages across at least one cycle.

4.1.2 State invariants.

Finite State Machine (FSM) Invariants relate to signals implementing state logic within a module. We define this invariant category, because FSM states and derived signals strongly influence the module's current operation. An example invariant checks that a module transitions from its current state to the correct next state.
Counter Invariants relate to signals implementing counter functionality. We define this category because counters are easily distinguishable by their well-defined behavior. An example invariant constrains counter bounds or verifies the counting direction.

4.1.3 Control invariants.

Internal Control Invariants relate to internal control flow, e.g., flags used for inter-module communication or enabling/disabling functionality. We define this category to capture the significant impact such control signals have on design behavior. An example invariant checks that queue full/empty flags are set correctly.
Handshake Invariants relate to acknowledgements, valid/ready pairs, and other typically 1-bit wide signals exchanged between internal modules or at design interfaces. We define this category because handshakes coordinate data movement between modules. An example invariant asserts that the design correctly signals completion of a computation.

4.2 Evaluation methodology

Candidate invariant identification. For each category we design and test candidate invariants, which are invariant assertions not yet proven to hold for unbounded time. When a candidate holds, we use it as assumption to reduce the main property's search space. To evaluate a candidate's effectiveness, we require metrics that measure its impact on proof speed or bound. In Section 5 we define these metrics and collect them while proving the main property under all permutations of invariant subsets. When a category proves impactful for a given design type, we prioritize identifying candidates within that category.
Testruns. A testrun refers to one evaluation run, with individual verification attempts called experiments. Since testing all subsets of invariants can be infeasible, we group invariants by category. Every invariant is active in 50% of experiments since we test all group combinations. The number of experiments (E) depends on the number of groups (G), calculated as $E = 2^G - 1$ (excluding the case with no active groups). The total testrun time is $T = E*(D+C)$, where D is the duration of a single experiment set by the verification engineer according to the time needed for the verification to either stall or reach an unbounded proof when manually running a proof attempt and C is the startup time for the verification tools, typically negligible relative to D. We design reusable tools to automatically generate a verification script for each subset combination of these invariant groups.
Invariant identification for proof convergence. When the main property has not yet converged, measuring candidate invariants reveals which are impactful and which ones only slow down the proof. This helps identify design complexity not evident to the verification engineer and guides focus for further invariant identification. We demonstrate this on Marian in Section 5.1.
Regression speedup. When unbounded proofs are already obtainable, the methodology identifies invariants that most improve proof time and determines optimal combinations for faster regression runs. We demonstrate this in Sections 5.2 and 6.

Invariant group	Nr.	C	M1[s]	M2[%]	M3	M4
Main property	1	-	-	0	**7.16**	0
Counter set/reset	6	C	0.57	100	∞	**154**
Result handshakes	1	H	5.5	100	∞	**140**
Intermediate data	8	D	0.8	25	94	13.75
FSM state	3	FSM	0.07	85	173	39.67
Pipeline checks	3	P	78	0.02	101	0
Datapath control	2	DC	88	1	∞	123

Table 2: Marian proof speedup for selected invariants. Nr. = number of invariants, C = Category, M1 = Average time to unbounded proof [s] for measured invariant, M2 = Success Chance (% times unbounded proof was reached for the invariants), M3 = Average bound of invariant, M4 = Influence on bound. ∞ signifies an unbounded proof was reached every single experiment for these invariants.

5 Verification of Marian

We first discuss the results from our verification of Marian [15]. Then, we present our strategy to evaluate our invariants for improving proof convergence. Finally, we discuss the results.
Testbed. Our measurements are performed on an Intel Xeon server with 3.4 GHz, 60 logical cores and 1.25 TB RAM. For formal verification, we use Jasper Gold 24.09 [7] and for our tooling we use Python 3.6.8 [17]. We use Spike commit 51b32b4 [30] and we verified Marian's most recent version, commit e107560 [16].

5.1 Invariants for complete Marian

We manually identified 38 candidate invariants for Marian and categorize them according to Section 4.1. Following Section 4.2, 6 groups of invariants lead to 63 experiments, i.e., different combinations of active invariant groups. With a timeout per experiment of 1 hour (chosen, because verification stalls by that point), the total runtime is around 64 hours, including the overhead of C2RTL's compilation. Table 2 shows metrics for the best invariants per category, measured on the entire crypto unit. Results are averages of multiple ("Nr.") invariants. We discuss the metrics next.
(M1) Average time to unbounded proof. This metric measures the time it takes for the invariant to reach an unbounded proof within the timeout. The average for many invariants is below one second, although variance was high and in some cases some invariant proofs take more than 45 minutes.
(M2) Success chance. The fraction of experiments an unbounded proof was reached for these invariants, relative to the total number of times when the invariants were active.
(M3) Average bound. The proof bounds on average over all experiments combined, excluding the experiments where an unbounded proof was achieved. Only invariants that *always* reached an unbounded proof within the timeout have an infinite average bound.
(M4) Influence on bound. This metric measures if an invariant has positively impacted other invariants. It is calculated by the number of times another invariant reached a higher bound than average while the measured one was active.

Stage	Operand collection	Execution units	Writeback unit
Proof time, all	⊙ (8c)	⊙ (8c)	⊙ (8c)
Proof time, all + i	⊙ **(16c)**	⊙ **(16c)**	⊙ **(16c)**
Proof time, m1	**1273** (∞)	**6350** (∞)	**12** (∞)
Proof time, m2	1273 (∞)	**4676** (∞)	12 (∞)
Proof time, m2 + i	1273 (∞)	**1516** (∞)	12 (∞)

Table 3: Final times [s] for Marian's crytpo unit. Proofs for 'all' modules progress only 8 cycles (8c), improving to 16 cycles (16c) with invariants (⊙ means 1 hour timeout reached). m1 = modularization per stage, reaching unbounded proofs (∞). m2 (+ i) adds modularization of the execution unit (+ invariants), leading to a speedup of 1.28× (2.7×) compared to m1.

Results for the main property. Without any invariants, the main property did not reach an unbounded proof within the timeout (1 hour) per experiment, reaching a bound of 8 cycles and stalling. Using all invariants, we reach an even lower average bound of 7.16 cycles. However, the *highest bound reached for the main property in an experiment was 16 cycles* when only invariants from groups C, H, FSM, P were active. Thus, some invariants have brought a clear improvement, *improving the bound of the main property by 100% compared to the average bound.* The invariants on internal control (IC) yielded no improvements.

Counter and handshake invariants. Invariants for Counters (C) and Handshakes (H) had the largest influence on bound (M4). Datapath control invariants (DC) ranked third, with a still considerable, but lower, influence on bound. FSM invariants ranked lower, despite themselves reaching a proof fastest, highlighting that the influence of counter and handshake invariants is not a mere artifact of their fast convergence.

5.2 Modularization

As shown in Table 3, initially, verification of the entire Marian crypto unit timed out (all). Invariants improved the proof bound by 100% (all + i). With modularization, as discussed in Section 3.1, we reached unbounded proofs without invariants (m1). Further modularization of the execution unit (m2), as discussed below (Table 4), leads to a **speedup of 1.28×** for the crypto unit. Invariants lead to an additional **speedup of 2.13×** (m2 + i), with the **total speedup (from m1 to m2 + i) being 2.73×**.

Execution units modularization. We modularize the execution units per instruction and add the subset of the invariants that constrain the state space of signals used within the execution units. These are of handshake and datapath control categories. For the results in Table 4, we verified the execution units with and without invariants, and then each execution unit separately as well, applying the same invariants across all groups. Summing the isolated-group runtimes gives 4676s, **a speedup of 1.34×** compared to the 6350s required when all instructions were active—indicating that modularization is effective even without invariants. Invariants further improved performance for every group, with the largest **speedup of 6.25×** for the Shang–Mi 4 instructions (3168s to 507s). Invariants brought a total **speedup of 3×** to the modularized verification of the execution unit, as seen in the "Sum" entry in Table 4.

Module	Instruction count	Runtime [s] with invariants	Runtime [s] no invariants
Combined	**24**	**6066**	**6350**
SHA256	3	90	123
SM3	2	14	67
SM4	4	507	3168
AES	13	904	1316
GCM	2	1.4	1.5
Sum	**24**	**1516**	**4676**

Table 4: Runtime comparison of proving all instructions (Combined) on Marian together, compared to the individual runtime when proving each execution unit separately and the sum of these individual runs in (Sum).

Takeaway 1. Proving the execution unit submodules individually leads to a proof speedup of 1.34×. Invariants brought a further speedup of 3×.

5.3 Vulnerabilities

Our main property was initially violated for ShangMi 4 instructions. Based on manual calculations following the specification, we initially assumed it to be an error within Spike [29] and reported it as such. However, it turned out to be an issue within the RISC-V vector cryptography specification, where endianness of the ShangMi 4 vector crypto instructions is not defined clearly [27]. Due to our finding, the RISC-V specification will have to be updated [37, 38].

6 Generalizability

To test the generalizability of our invariant identification methodology, we introduce VCRYPTU, our implementation of the Zvknha RISC-V vector cryptography sub-extension, implementing SHA-256 instructions. We chose this representative sub-extension due to its wide operands and to ensure complete implementation of the control flow that a vector cryptography extension requires. We apply the same verification methodology and prove the same property as we did for Marian for VCRYPTU.

6.1 Design of VCRYPTU

VCRYPTU is also integrated into Ara [23], but we make different microarchitectural choices: while Marian is pipelined, VCRYPTU is not. Rather than interfacing with lanes, VCRYPTU embeds a crypto unit in every lane alongside other vector arithmetic units. Because each VCRYPTU instance can access only its own lane's registers, it must fetch operand parts sequentially, slowing operand collection down, but enabling high parallelism across lanes. Table 5 compares the operand collection, execution and writeback components needed for SHA-2 instructions as elaborated by Jasper for Marian and VCRYPTU.

6.2 Verification of VCRYPTU

Without invariants, verification of VCRYPTU times out. Using the methodology from Section 4, we identify 24 invariants. With all 24 active, we obtain an *unbounded proof of VCRYPTU within 25 seconds.*

Design info	**Marian**		**VCRYPTU**	
SHA-2 only	Signals	Bits	Signals	Bits
Flops	278	3178	46	719
Latches	25	1056	13	388
Gates	7557	190989	1491	22428
Nets	9020	-	1748	-

Table 5: Design information about Marian and VCRYPTU.

Group	C	M2	**M5**	M6
Main property	-	0.25	0	**256**
V1, Counter upper bounds	C	1	1105.1	**256**
V2, Counter relational bounds	C	**0.5**	1105.1	**256**
V3, Vector length flags	DC	1	1074.3	128
V4, Counter step size	C	1	596.9	128

Table 6: Measurements of VCRYPTU. C = Category, M2 = Success chance, M5 = Time saved on main property [s], M6 = Success points.

We evaluate these invariants using the methodology in Section 4.2, running 1023 experiments across 10 groups for finer-grained analysis, with a 3-minute timeout per experiment.

Additional metrics. To assess invariant impact on proof speed in cases where multiple invariant combinations already yield unbounded proofs, we introduce metrics (M5) and (M6) discussed next. To ensure that the invariants indeed contribute to state-space reduction for the main property, we only consider cases where the invariant proves *before* the main property.

(M5) Time saved on main property. This metric measures by how much the main property proves faster than its average speed when the evaluated invariant is active. This metric is similar to (M4), but focuses on the influence on the main property instead of the bound of invariants.

(M6) Success points. This metric counts experiments in which both the invariant and the main property reach a proof.

Results. Table 6 summarizes the results and ranks the top four groups by how much time they save for the main property. Only two invariant groups V1 and V2 in Table 6 achieve 256 Success Points (M6). Each of these are active in 512 experiments and their overlap is exactly the 256 experiments where their proof succeeds. Since V1 alone is insufficient and V2 succeeds only 50% of the time, V2 enables a proof only when V1 is active. Thus, the main property depends on V2 and V2 depends on V1. Both groups constrain design counters: V1 enforces an upper bound on a counter, while V2 asserts that one counter is never larger than the other. These counters track processed and received vector elements. V3 also slightly improves proof time. Starting from V4 the improvements decline.

Common results for Marian and VCRYPTU. Our analysis suggests that invariants about counters and handshakes have higher impact than other types of invariants in both implementations, regardless of size of the signals being constrained by the invariant.

We conclude that we can systematically identify invariants for arithmetic units by focusing our efforts on these types of invariants.

Takeaway 2. Invariants on counters and handshakes were consistently the most impactful at improving the speed and bound of the main property for both Marian and VCRYPTU.

7 Related Work

Verifying cryptographic hardware. Cryptographic hardware is typically verified by simulation test cases that for example compare against test vectors taken from standards [20, 36]. Some formal approaches that verify AES implementations are known [3, 20, 34, 42]. Intel uses a proprietary tool based on Symbolic Trajectory Evaluation for proving an AES hardware implementation against a custom specification [34]. Cryptol [12] is a language that is designed for cryptography algorithms that can be partially directly synthesized to hardware. Software Analysis Workbench, has been used to verify a RISC-V cryptographic extension implementation of AES, written in Bluespec SystemVerilog, against multiple Cryptol specifications that were proven to be equivalent amongst each other [20]. The functional language Haskell has been used to develop specifications and verify them against an implementation in Haskell's HDL Lava [3, 4]. These methods are not applicable to verifying existing designs implemented in standard HDLs.

Invariants. Recently, manually identified invariants for functional verification of a CPU against the base RISC-V ISA were published [24]. Raia et al. verify a tone generator with C2RTL and map intermediate result RTL signals to the C model for proof speedup [25]. For Marian, such invariants brought no benefit. Probably, because the execution units are stateless. Automatically generated invariants over information flow tracking state signals were used to speed up verification of microarchitectural control-flow integrity of a RISC-V scalar cryptography implementation [9]. These are not reusable for functional verification. Environment invariants were synthesized to aid case splitting per instruction when equivalence checking an AES implementation against an instruction level specification developed by the authors [42]. These invariants constrain the symbolic initial state per instruction to be arbitrary, but valid. Fadiheh et al. use similar invariants for constant-time verification [13]. Our proofs start in the reset state and thus don't require such invariants. Several automated invariant mining methods have been proposed [10, 18, 21, 22, 40, 41, 43]. These works focus on the correctness of the discovered invariants. The invariants are not evaluated for proof speedup. Evaluating them with our evaluation strategy is an interesting direction for future work.

8 Conclusion

We formally verify Marian [15], the first open source implementation of the vector cryptography extension for RISC-V and identify a mistake within the RISC-V ratified vector cryptography specification. To scale the verification, we design a systematic invariant identification methodology and evaluate it with two implementations, showing its generality. We find that invariants about state-defining signals such as counters and handshakes are the most impactful. Combining effective invariants and modularization, we achieve proof convergence of Marian, with invariants bringing up to 6.25× speedup per module, and a speedup of 2.7× in total.

9 Acknowledgments

This work was supported by a Qualcomm Innovation Fellowship and by the Swiss State Secretariat for Education, Research and Innovation under contract number MB22.00057 (ERC-StG PROMISE).

References

[1] 2001. *RTL Formal Verification.* Springer US, Boston, MA, 103–129. doi:10.1007/0-306-47631-2_6

[2] David Basin, Cas Cremers, Jannik Dreier, and Ralf Sasse. 2017. Symbolically analyzing security protocols using tamarin. *ACM SIGLOG News* 4, 4 (Nov. 2017), 19–30. doi:10.1145/3157831.3157835

[3] Abir Bitat and Salah merniz. 2019. Formal verification of cryptographic circuits: A semi-automatic functional approach *(NISS '19).* Association for Computing Machinery, New York, NY, USA, Article 35, 6 pages. doi:10.1145/3320326.3320367

[4] Abir Bitat and Salah Merniz. 2021. Formal verification of pipelined cryptographic circuits: A functional approach. *Informatica* 45, 4 (2021).

[5] Gianpiero Cabodi, Paolo Enrico Camurati, Marco Palena, and Paolo Pasini. 2024. Hardware Model Checking Algorithms and Techniques. *Algorithms* 17, 6 (2024). doi:10.3390/a17060253

[6] Cadence. 2024. Jasper Platform and Formal Property Verification App User Guide. (2024).

[7] Cadence. 2025. *Jasper Formal Verification Platform.* https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification.html Accessed: 2025-11-18.

[8] Katharina Ceesay-Seitz, Hamza Boukabache, and Daniel Perrin. 2020. A Functional Verification Methodology for Highly Parametrizable, Continuously Operating Safety-Critical FPGA Designs: Applied to the CERN RadiatiOn Monitoring Electronics (CROME). In *Computer Safety, Reliability, and Security*, António Casimiro, Frank Ortmeier, Friedemann Bitsch, and Pedro Ferreira (Eds.). Springer International Publishing, Cham, 67–81.

[9] Katharina Ceesay-Seitz, Flavien Solt, and Kaveh Razavi. 2024. μCFI: Formal Verification of Microarchitectural Control-flow Integrity. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security.* https://comsec-files.ethz.ch/papers/mucfi_ccs24.pdf

[10] Po-Hsien Chang and Li-C Wang. 2010. Automatic assertion extraction via sequential data mining of simulation traces. In *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC).* IEEE, 607–612.

[11] A. Darbari and I. Singleton. 2016. https://arxiv.org/abs/1606.02347

[12] Levent Erkök, Magnus Carlsson, and Adam Wick. 2009. Hardware/software coverification of cryptographic algorithms using Cryptol. In *2009 Formal Methods in Computer-Aided Design.* 188–191. doi:10.1109/FMCAD.2009.5351121

[13] Mohammad Rahmani Fadiheh, Alex Wezel, Johannes Müller, Jörg Bormann, Sayak Ray, Jason M Fung, Subhasish Mitra, Dominik Stoffel, and Wolfgang Kunz. 2022. An exhaustive approach to detecting transient execution side channels in RTL designs of processors. *IEEE Trans. Comput.* 72, 1 (2022), 222–235.

[14] Fahrni, Kim and Ceesay-Seitz, Katharina and Zuppiger, Denis and Razavi, Kaveh. 2026. Artifacts of this paper. https://github.com/comsec-group/riscv-vector-crypto-invariants

[15] SoC-Hub Finland. 2025. *Vector-Crypto Subsystem (Marian) for Bow SoC.* https://github.com/soc-hub-fi/Marian Accessed: 2025-04-16.

[16] SoC-Hub Finland. 2025. *Vector-Crypto Subsystem (Marian) for Bow SoC, Version verified.* https://github.com/soc-hub-fi/Marian/commit/e107560ac6a6882db3b3a88c460c60831c3e5cde Accessed: 2025-04-16.

[17] Python Software Foundation. 2026. Python 3.6.8. https://www.python.org/downloads/release/python-368/ Accessed: 2025-11-03.

[18] Sudheendra Hangal, Naveen Chandra, Sridhar Narayanan, and Sandeep Chakravorty. 2005. Iodine: a tool to automatically infer dynamic invariants for hardware designs. In *Proceedings of the 42nd annual Design Automation Conference.* 775–778.

[19] Yonit Kesten and Amir Pnueli. 1998. Modularization and abstraction: The keys to practical formal verification. In *Mathematical Foundations of Computer Science 1998*, Luboš Brim, Jozef Gruska, and Jiří Zlatuška (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 54–71.

[20] Joseph R Kiniry, Daniel M Zimmerman, Robert Dockins, and Rishiyur Nikhil. 2018. A formally verified cryptographic extension to a RISC-V processor. *Computer Architecture Research with RISC-V–CARRV 2018* (2018).

[21] Wenchao Li, Alessandro Forin, and Sanjit A Seshia. 2010. Scalable specification mining for verification and diagnosis. In *Proceedings of the 47th design automation conference.* 755–760.

[22] Lingyi Liu, David Sheridan, Viraj Athavale, and Shobha Vasudevan. 2011. Automatic generation of assertions from system level design using data mining. In *Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMPCODE2011).* 191–200. doi:10.1109/MEMCOD.2011.5970526

[23] Matteo Perotti, Matheus Cavalcante, Nils Wistoff, Renzo Andri, Lukas Cavigelli, and Luca Benini. 2022. A "New Ara" for Vector Computing: An Open Source Highly Efficient RISC-V V 1.0 Vector Processor Design. In *2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP).* 43–51. doi:10.1109/ASAP54787.2022.00017

[24] Louis-Emile Ploix, Alasdair Armstrong, Tom Melham, Ray Lin, Haolong Wang, and Anastasia Courtney. 2025. Comprehensive Formal Verification of Observational Correctness for the CHERIoT-Ibex Processor. arXiv:2502.04738 [cs.AR] https://arxiv.org/abs/2502.04738

[25] Gaetano Raia, Gianluca Rigano, David Vincenzoni, and Maurizio Martina. 2025. A Case Study on Formal Equivalence Verification Between a C/C++ Model and Its RTL Design. In *Formal Methods*, Andre Platzer, Kristin Yvonne Rozier, Matteo Pradella, and Matteo Rossi (Eds.). Springer Nature Switzerland, Cham, 373–389.

[26] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. 2016. End-to-End Verification of Processors with ISA-formal. In *CAV '16* (Cham, 2016), Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, 42–58.

[27] RISC-V. [n. d.]. *Vector Crypto Specification Release 1.0.0.* https://github.com/riscv/riscv-crypto/releases/tag/v1.0.0 Accessed: 2025-11-18.

[28] RISC-V. 2025. *RISC-V Sail Model.* https://github.com/riscv/sail-riscv Accessed: 2025-11-18.

[29] RISC-V. 2025. *Spike RISC-V ISA Simulator.* https://github.com/riscv-software-src/riscv-isa-sim Accessed: 2025-11-18.

[30] RISC-V. 2025. *Spike RISC-V ISA Simulator, Version used.* https://github.com/riscv-software-src/riscv-isa-sim/commit/51b32b48257382b2ef859442bad86c68cf2e07cc Accessed: 2025-11-18.

[31] E. Seligman, T. Schubert, and M.V.A.K. Kumar. 2023. *Formal Verification: An Essential Toolkit for Modern VLSI Design.* Elsevier Science. https://books.google.ch/books?id=OljAEAAAQBAJ

[32] Siemens AG. [n. d.]. The 2024 Wilson Research Group Functional Verification Study. https://verificationacademy.com/topics/planning-measurement-and-analysis/2024-siemens-eda-and-wilson-research-group-functional-verification-study/ Accessed: 2025-04-17.

[33] Anna Slobodová. 2007. Challenges for Formal Verification in Industrial Setting. In *Formal Methods: Applications and Technology*, Luboš Brim, Boudewijn Haverkort, Martin Leucker, and Jaco van de Pol (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–22.

[34] Anna Slobodova. 2008. Formal Verification of Hardware Support for Advanced Encryption Standard. In *2008 Formal Methods in Computer-Aided Design.* 1–4. doi:10.1109/FMCAD.2008.ECP.12

[35] Synopsys. 2025. *VC Formal Datapath Validation.* https://www.synopsys.com/verification/static-and-formal-verification/vc-formal/vc-formal-datapath-validation.html Accessed: 2025-11-18.

[36] Thomas Szymkowiak, Endrit Isufi, and Markku-Juhani Saarinen. 2024. Poster: Marian: An Open Source RISC-V Processor with Zvk Vector Cryptography Extensions. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24).* Association for Computing Machinery, New York, NY, USA, 4931–4933. doi:10.1145/3658644.3691394

[37] Various. 2026. RISC-V Instruction Set Manual - Issue 2615. https://github.com/riscv/riscv-isa-manual/issues/2615 Accessed: 2026-03-25.

[38] Various. 2026. Spike issue 1994. https://github.com/riscv-software-src/riscv-isa-sim/issues/1994 Accessed: 2026-03-25.

[39] Muralidaran Vijayaraghavan, Adam Chlipala, Arvind, and Nirav Dave. 2015. Modular Deductive Verification of Multiprocessor Hardware Designs. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 109–127.

[40] Qinhao Wang and Masahiro Fujita. 2022. Template-Based Semi-Formal Approach to Robust Equivalence Checking. *Electronics* 11, 11 (2022). doi:10.3390/electronics11111691

[41] Jiahui Xu and Lana Josipović. 2023. Automatic Inductive Invariant Generation for Scalable Dataflow Circuit Verification. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD).* 1–9. doi:10.1109/ICCAD57390.2023.10323796

[42] Hongce Zhang, Weikun Yang, Grigory Fedyukovich, Aarti Gupta, and Sharad Malik. 2020. Synthesizing Environment Invariants for Modular Hardware Verification. In *Verification, Model Checking, and Abstract Interpretation: 21st International Conference, VMCAI 2020, New Orleans, LA, USA, January 16–21, 2020, Proceedings* (New Orleans, LA, USA). Springer-Verlag, Berlin, Heidelberg, 202–225. doi:10.1007/978-3-030-39322-9_10

[43] Tong Zhang, Daniel Saab, and Jacob A. Abraham. 2017. Automatic Assertion Generation for Simulation, Formal Verification and Emulation. In *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI).* 471–476. doi:10.1109/ISVLSI.2017.88