heRVé: towards a formally verified RISC-V processor with security mechanisms
Work in Progress

Anonymous authors

No Institute Given

Abstract. Ultimately, the safety of systems is contingent on the security of the hardware platforms they operate on. To achieve the highest degree of confidence in the hardware's security, specifically processors, we advocate the use of formal methods.
In this paper, we build upon the work by Baty *et al.*, which introduces a RISC-V processor enhanced with a shadow stack [3]. This processor is designed in Kôika, a Hardware Description Language embedded into the Rocq proof assistant, and includes formal security proofs for the implemented shadow stack. We enhance this processor to incorporate support for traps, various privilege levels, and physical memory protection (PMP). Additionally, we automate the proofs of the shadow stack by relying on a SMT solver, improving both scalability and maintenance. Our processor is sufficiently rich to run Zephyr, a small real-time operating system for embedded processors.

Keywords: Formal verification · Hardware Security.

1 Introduction

The security of systems depends on the security of all layers, from user software to hardware, through operating systems. For critical systems, we need high confidence in the correctness and security of all these layers, which we can achieve with the use of formal methods.

Formal verification has already been applied to the security and correctness of software [10, 5] and operating systems [9, 7]. In this paper, we focus on the micro-architectural description of hardware.

We build upon the Kôika Hardware Description Language (HDL). This language allows to describe a high-level view of a circuit, and to compile it into Verilog, a more traditional HDL. From there, the standard set of synthesis tools can be applied to target FPGA boards. The compilation from Kôika to Verilog is formally verified, meaning that the behaviour of the Verilog circuit is guaranteed to be equivalent to that of the Kôika circuit.

In previous work, Baty *et al.* enriched a simple RISC-V processor with a shadow stack, which is a security mechanism enforcing backward-edge control flow integrity, i.e. the integrity of return addresses. Together with the implementation of the shadow stack in the pipeline of the processor, Baty *et al.* proved

the correctness of this mechanism. These proofs are however tedious to write and very fragile to changes in the processor.

In this work, we have two objectives. First, we aim to enrich this RISC-V processor with more security mechanisms, to make it more realistic and be able to run a small operating system on it. Second, we want to make the proofs easier to write and maintain.

Our contributions can be summarized as follows:

- We add support for various mechanisms into the existing processor, which we now call heRVé. More specifically, we add support for *a)* Control and Status Registers (CSR); *b)* traps (interrupts and exceptions); *c)* several privilege levels (M and U); *d)* and support for PMP (Physical Memory Protection). With these additions, the heRVé processor can run the Zephyr OS.
- The proofs of the shadow stack are now handed to a SMT solver for better scalability and maintainability.

The remainder of this article is structured as follows. First, Section 2 gives some background about the Kôika language, and the RISC-V processor written in this language. Then, Section 3 presents our modified RISC-V processor, which we call heRVé, with various security mechanisms. In Section 4, we show that heRVé can run the Zephyr RTOS. Section 5 explains how we make proofs about heRVé, by relying on a SMT solver. Finally, Section 6 discusses related work and Section 7 concludes.

2 Background

2.1 The Kôika language

Kôika [4] is a Hardware Description Language (HDL) embedded in the Rocq proof assistant. Unlike more traditional HDLs (e.g. Verilog), Kôika features the notion of atomic rules, that describe different parts of the hardware circuit. Rules manipulate registers and execute concurrently. Conflicts (e.g. when two rules attempt to write to the same register simultaneously) are resolved by ordering the rules in a so-called *schedule*. Kôika circuits can be compiled into Verilog code, through the use of a formally verified compiler. Therefore, the semantics of the compiled Verilog is the same as the semantics of the Kôika circuit. Consequently, all the properties we prove about Kôika circuits also hold for the Verilog code.

2.2 The Intermediate Representation for Reasoning

Direct proofs on Kôika circuits are impractical (high memory consumption, hard-to-control computation tactics in Rocq). Hence, Baty *et al.* introduced an Intermediate Representation for Reasoning (IRR) [3]. This representation basically unfolds all the rules in the Kôika circuit, and gives the value of each register of the circuit after one clock cycle. This value is expressed as an expression referring to the values of registers before this clock cycle. In particular, all the conflict detection and resolution is encoded into these expressions.

2.3 A RISC-V processor in Kôika

Kôika's original authors developed a simple RV32I processor [4]. This processor is a simple in-order 4-stage processor with a classical Fetch-Decode-Execute-Writeback pipeline. Baty *et al.* enhance this processor with a simple shadow stack: when a call instruction is in the Execute stage of the processor, a copy of the return address is pushed onto this shadow stack; when a ret instruction is in the Execute stage, the return address is checked against the shadow stack. Violations (unmatched ret, shadow stack overflow, return address mismatch) result in halting the processor (for lack of a better signalling system in this simple processor). Also, in the absence of shadow stack violation, the processor runs as it would without the shadow stack. All these properties of the shadow stack are formally proven, using the compilation of Kôika to IRR and manual proofs on this IRR.

3 The heRVé processor

We extend the simple RV32I processor from Baty *et al.* [3] with support for control and status registers, traps, privilege levels and PMP. All these changes amount to about 1500 lines of Coq code.

3.1 Control and Status Registers (CSR)

The Control and Status Registers (CSR) are registers introduced in the RISC-V specification that allow to control the behaviour of the processor. A number of CSRs (`mstatus`, `mtvec`, `mcause`, `mepc`) are related to trap handling and are discussed below. We also have performance counters: `mcycle` counts the number of cycles since boot, `minstret` counts the number of retired instructions.

3.2 Traps

Traps are divided into *exceptions* – synchronous events triggered by the execution of an instruction (e.g. illegal instruction, misaligned memory access, or environment calls) – and *interrupts* – asynchronous events, generally generated by external devices or timers.

Timer. We implement a timer in our processor, following SiFive's CLINT interface, with two registers `mtime` and `mtimecmp` mapped in memory at addresses `0x0200bff8` and `0x02004000` respectively. The `mtime` register is incremented at a constant rate. As soon as the value in `mtime` exceeds the value in `mtimecmp`, a timer interrupt is raised.

Exceptions. We raise exceptions for the following conditions : `ecall`/`ebreak` instructions for environment calls (system calls) and breakpoints, misaligned memory accesses (instruction or data), illegal instruction. We also modified the shadow stack so that a specific exception is raised in case of a violation, instead of halting the processor. As we will see later, we also raise an exception when a memory access is forbidden by the PMP.

Trap Handling Mechanism. As soon as a trap occurs, the pipeline is flushed and the processor starts handling the trap. The processor saves the execution context and transfers control to a trap handler in machine mode. This backup is performed as follows:

1. The program counter (PC) is saved in the `mepc` (*exception program counter*) register;
2. In the `mstatus` register, bit `MIE` (*Machine Interrupts Enabled*) is cleared to disable interrupts, and its previous value in saved in `MPIE` (*Machine Previous Interrupts Enabled*). Additionally, the current privilege level is saved in the `MPP` field;
3. The cause of the trap is stored in the `mcause` register;
4. The `mtval` register is updated with additional information, such as the faulting address or instruction, when relevant;
5. The next PC is defined according to the `mtvec` register, which can specify a direct or vector trap handler.
6. the processor switches to machine mode

Returning from a Trap. The `mret` instruction resumes normal execution after a trap has been processed. This instruction restores the previous privilege level from `mstatus.MPP`; re-enables interrupts by restoring `mstatus.MIE` from `mstatus.MPIE`; and sets the PC to the value stored in `mepc`.

3.3 Privilege levels

The heRVé processor supports two privilege levels, **Machine mode** (M) and **User mode** (U), as defined by the RISC-V privileged architecture [12]. The **Supervisor mode** (S) and **Hypervisor mode** (H) are not yet supported. However, the basic processor logic and privilege management infrastructure have been designed to allow the future addition of these modes with minimal changes.

The current privilege level is stored in a micro-architectural register that is not accessible or modifiable by software through standard register reads or writes. This design choice is intended to prevent unauthorized privileges escalation and while maintaining simple implementation. Transitions between privilege levels are controlled by the processor's trap and exception logic.

Transitions to a lower privilege level (from M-mode to U-mode) are performed by updating the `mstatus.MPP` (Machine Previous Privilege) field and then executing the `mret` (machine return) instruction. All exceptions and traps are handled in M-mode, ensuring that sensitive operations such as system calls (`ecall`) or illegal instruction traps are always processed in the most privileged context.

The RISC-V specification defines access rights to Control and Status Registers (CSRs) based on the current privilege level. Each CSR address range is associated with a minimum required privilege, as summarized in Table 1. For every CSR read or write, the processor checks that the current privilege level is sufficient for that CSR access, and raises an `IllegalInstruction` exception otherwise.

Address Range	Category	Accessible By
0x000–0x0FF	User-level CSR	User, Machine
0x300–0x3FF	Machine-level CSR	Machine only
0xC00–0xC1F	Read-only counters	User, Machine

Table 1. CSR address ranges and required privilege levels.

3.4 PMP

Physical Memory Protection (PMP) is a hardware mechanism in the RISC-V privileged architecture that enables fine-grained control over memory access permissions, independently of the virtual memory system. PMP is essential for enforcing isolation between privilege levels and for supporting secure execution environments.

PMP Entry Structure and Configuration. Each PMP entry is defined by a configuration byte (`pmpcfgN`) and an address register (`pmpaddrN`). The configuration byte encodes the access permissions (Read, Write, Execute), the address-matching mode (see Table 2), and a lock bit. The address register specifies the boundary of the protected region. Our implementation supports all 4 modes of the PMP.

A1:A0	Mode	Description	Supported
00	OFF	Entry disabled	Yes
01	TOR	Top of Range	Yes
10	NA4	Naturally aligned 4-byte region	Yes
11	NAPOT	Naturally aligned power-of-two region	Yes

Table 2. PMP address-matching modes and their support in our implementation.

Bitfield Layout and Semantics The PMP configuration byte (`pmpcfgN`) is a compact 8-bit register that precisely encodes the access control policy for a memory region. Its structure is illustrated in Figure 1. Each bit serves a distinct purpose, enabling fine-grained and hardware-enforced memory protection:

– **L (Lock, bit 7):** This bit is for establishing a secure execution environment. When the L bit is set, the corresponding PMP entry becomes immutable: neither the configuration nor the address register can be modified by any software, including machine-mode code, until the next hardware reset. Typically, the L bit is set by trusted boot code after initializing the PMP regions, thereby guaranteeing that even the most privileged software cannot alter the memory protection until a full system reboot.

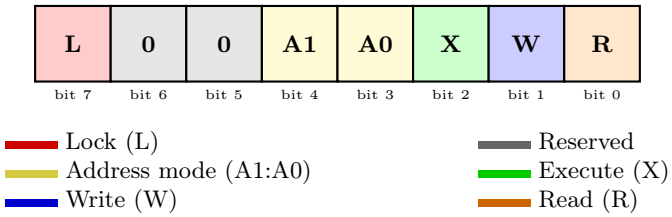

Fig. 1. Format of a PMP configuration byte (`pmpcfgX`).

- **Reserved (bits 6–5):** These bits are reserved for future use and must always be written as zero. Any nonzero value is considered illegal and may result in unpredictable behavior.
- **A1:A0 (bits 4–3):** These two bits select the address-matching mode for the region (see Table 2). The mode determines how the associated address register(s) define the protected memory range.
- **X (Execute, bit 2):** This bit controls instruction fetch permissions. If X is set, code execution (i.e., instruction fetches) is permitted from the protected region for the relevant privilege level. If X is clear, any attempt to fetch instructions from the region will result in an instruction fault.
- **W (Write, bit 1):** This bit governs write access. When W is set, store operations (writes) to the region are allowed. If W is clear, any attempt to write data to the region will be blocked, and an exception will be raised.
- **R (Read, bit 0):** This bit manages read access. If R is set, load operations (reads) from the region are permitted. If R is clear, any attempt to read data from the region will be denied, and an exception will be raised.

Addressing and Region Semantics. In TOR mode, each PMP entry i defines a protected region as the half-open interval $[\texttt{pmpaddr}_{i-1} \ll 2, \texttt{pmpaddr}_i \ll 2)$. The addresses are shifted left by two bits, enabling 34-bit physical addresses. The first region always starts at address zero, i.e., $[0, \texttt{pmpaddr}_0 \ll 2)$.

Example Configuration For example, to define two regions:

- Region 0: [`0x0000`, `0x1FFF`] with execute, write, and read permissions (XWR)
- Region 1: [`0x2000`, `0x2FFF`] with read-only permission (R)
- All other addresses: no access restrictions (default M-mode only)

one would set:

- `pmpaddr0 = 0x2000`
  `pmp0cfg = 0b00001111 (0x0F)` (TOR mode, X/W/R enabled)
- `pmpaddr1 = 0x3000`
  `pmp1cfg = 0b00001001 (0x09)` (TOR mode, R only)
- `pmpaddr2, pmpaddr3 = 0` (unused)
- `pmp2cfg = pmp3cfg = 0x00` (entries disabled)

Thus, the 32-bit configuration register `pmpcfg0` is `0x0000090F`.

Runtime Enforcement. On every memory access (instruction fetch, load, or store), the processor needs to determine whether the access is allowed. The PMP entries are scanned in order of increasing index (priority), and the first matching region determines the access rights. If the access falls within a region, the permissions (R/W/X) are checked against the requested operation and the current privilege level. If the access does not match any region, only M-mode is allowed to proceed.

The PMP configuration and address registers are mapped in the CSR address space from `0x3A0` to `0x3EF`, and are only writable in M-mode, provided the L bit is not set.

Validation and Testing. We wrote a number of simple test files, exercising various types of PMP entries (TOR, NA4, NAPOT) and attempting accesses from both machine mode and user mode.

4 Running Zephyr

Zephyr is an open-source real-time operating system (RTOS) designed for resource constrained embedded systems [13]. It features a modular kernel, a large device driver infrastructure and a rich set of APIs. Zephyr supports a wide range of architectures, including RISC-V, and is used as a reference platform for hardware validation. In the following, we adapt Zephyr's `qemu_riscv32` configuration to our heRVé processor.

UART Integration. A primary adaptation concerned the Universal Asynchronous Receiver/Transmitter (UART) peripheral. Unlike the standard QEMU RISC-V platform, which maps its UART at address `0x10000000`, heRVé exposes its UART at `0x40000000`. To this end, we have developed a minimal custom UART driver for Zephyr and customized the device tree. An alias is also defined to ensure that Zephyr selects this UART as it default system console. This allows us to use Zephyr's standard output functions `printf` and `printk`.

Timer and Interrupts. Timer management also had to be adapted. In RISC-V's standard privileged architecture, the `mtime` and `mtimecmp` registers are memory-mapped. We initially implemented these registers as custom CSRs in heRVé. We then map these registers in memory at the addresses expected by Zephyr. The Platform-Level Interrupt Controller (PLIC) is currently disabled in our configuration, as our processor does not yet implement external interrupt support. All interrupt management is therefore handled via the machine timer.

Instruction Set and Privilege Configuration. HeRVé only implements the RV32I base integer instruction set, without the M (multiplication/division) or C (compressed) extensions. Zephyr is accordingly configured to generate and execute only supported instructions.

Experiments. So far, we run only very simple examples from Zephyr, namely the `hello_world` and `hello_world_user` programs. Although simple, these examples validate that our processor switches to user mode when necessary, that our UART driver and our timer work correctly. Attempting to read the timer value (the `mtime` register) from user mode results in a PMP load address fault, as one would have expected.

5 Proving properties of heRVé

We reuse the Intermediate Representation for Reasoning (IRR) from Baty's work [2]. However, we completely automate the tedious and fragile work of writing proofs directly in Rocq. Instead, we translate both the IRR corresponding to a cycle in the processor, and the property to be verified, into a SMT problem. We then rely on a SMT solver (in our experiments, Microsoft's Z3) to prove our properties; or disprove them, with a counterexample.

Most properties we prove are about a single cycle of the processor: given hypotheses P on the initial state of registers, prove that after one clock cycle, the registers are in a state that satisfy Q. The SMT problem is constructed as follows:

1. we introduce one SMT variable for each register of the processor;
2. we introduce one SMT variable for each variable in the IRR;
3. for each variable in the IRR, we assert that it is equal to the expression contained in the IRR;
4. we assert our hypotheses (P) about the initial state of registers;
5. we assert the negation of Q on the final state of registers.

If the conclusion Q is true, the SMT solver should fail to find a model which satisfy P but not Q, so we expect the SMT solver to answer `unsat`. If the SMT solver finds a model, it is a counter-example for our property, which we can use to either fix the processor, or the property of interest.

We proved again the properties of the shadow stack, as was done by Baty *et al.* earlier. The performance of these proofs is reported in Table 3. The proofs are generally faster to check than the old Rocq proofs, but more importantly they automatically adapt when changes are made to the processor. This means that when developing all the extensions described in earlier sections, we did not have to change the proof itself, but only check that the properties still hold. The SMT problems contain more than 60000 variables (corresponding to IRR variables), and as many assertions to associate each variable with its value (an expression depending on other variables, or initial values of registers at the beginning of the cycle).

6 Related Work

There are other approaches for formal verification of hardware designs. The authors of Kôika initially developed the Kami language [6]. This language is also

Proof	Lines in [3]	Time in [3]	Time now
Shadow Stack underflow	166	70s	6.8s
Shadow Stack overflow	218	150s	1.9s
Shadow Stack violation	812	190s	205s

Table 3. Performance of SMT proofs.

developed in Rocq and can be compiled to Verilog. The language is larger than Kôika and is not actively maintained anymore. Additionally, Kôika allows to reason at the cycle-accurate level. Vericert [8] is a High-Level Synthesis (HLS) tool that is formally verified, i.e. a verified compiler from C to Verilog. In our case, we want to have fine control over how the hardware circuit will look like, so we prefer to use a lower-level approach with Kôika. In the CakeML project, Lööw *et al.* [11] propose a verified Verilog compiler, using Isabelle/HOL.

Several proof assistants can rely on SMT solvers. Isabelle/HOL and Lean, for example, interact with SMT solvers natively. For the Rocq proof assistant, SMTCoq [1] is an attempt at translating a Rocq goal into a SMT problem, have a SMT solver build a proof for our problem, and translate the SMT proof back into Rocq. This works well for the original use case (theories of arithmetic, uninterpreted functions), but is not applicable in our case (theory of bitvectors).

7 Conclusion

We presented heRVé, a RV32I processor with support for CSRs, traps, machine and user privilege levels and physical memory protection (PMP). This processor can run the ZephyrOS with all these extensions.

We introduced an hybrid method to perform proofs. While the processor and its specification are expressed inside the Rocq proof assistant, the proofs themselves are delegated to SMT solvers for efficiency and maintainability.

In the future, we plan to enhance our processor with new security mechanisms (e.g. a memory management unit, a cache). We also aim to prove more about this processor, both for functional – does this processor implement the RISC-V ISA? – and security properties. For example, we could start by proving that PMP entries indeed prevent arbitrary memory accesses, and that these entries cannot be modified even in machine mode once their L bit is set.

References

1. Armand, M., Faure, G., Grégoire, B., Keller, C., Thery, L., Werner, B.: A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In: Jouannaud, Jean-Pierre, Shao, Zhong (eds.) Lecture notes in computer science - LNCS. Lecture notes in computer science - LNCS, vol. 7086, pp. 135–150. Springer, Kenting, Taiwan (Dec 2011). https://doi.org/10.1007/978-3-642-25379-9_12, https://inria.hal.science/hal-00639130

2. Baty, M.: Formal Specification and Verification of Security Mechanisms for RISC-V Processors. Theses, CentraleSupélec (Dec 2024), https://hal.science/tel-04892968
3. Baty, M., Wilke, P., Hiet, G., Fontaine, A., Trieu, A.: A generic framework to develop and verify security mechanisms at the microarchitectural level: Application to control-flow integrity. In: 36th IEEE Computer Security Foundations Symposium, CSF 2023, Dubrovnik, Croatia, July 10-14, 2023. pp. 372–387. IEEE (2023). https://doi.org/10.1109/CSF57540.2023.00029, https://doi.org/10.1109/CSF57540.2023.00029
4. Bourgeat, T., Pit-Claudel, C., Chlipala, A., Arvind: The essence of bluespec: a core language for rule-based hardware design. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020. pp. 243–257. ACM (2020). https://doi.org/10.1145/3385412.3385965, https://doi.org/10.1145/3385412.3385965
5. Cao, Q., Beringer, L., Gruetter, S., Dodds, J., Appel, A.W.: Vst-floyd: A separation logic tool to verify correctness of C programs. J. Autom. Reason. **61**(1-4), 367–422 (2018). https://doi.org/10.1007/S10817-018-9457-5, https://doi.org/10.1007/s10817-018-9457-5
6. Choi, J., Vijayaraghavan, M., Sherman, B., Chlipala, A., Arvind: Kami: a platform for high-level parametric hardware specification and its modular verification. Proceedings of the ACM on Programming Languages **1**(ICFP), 24:1–24:30 (2017). https://doi.org/10.1145/3110268
7. Gu, R., Shao, Z., Chen, H., Wu, X.N., Kim, J., Sjöberg, V., Costanzo, D.: CertiKOS: An extensible architecture for building certified concurrent os kernels. In: Keeton, K., Roscoe, T. (eds.) 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016. pp. 653–669. USENIX Association (2016), https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu
8. Herklotz, Y., Pollard, J.D., Ramanathan, N., Wickerson, J.: Formal verification of high-level synthesis. Proceedings of the ACM on Programming Languages **5**(OOPSLA), 1–30 (2021). https://doi.org/10.1145/3485494, https://doi.org/10.1145/3485494
9. Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS microkernel. TOCS **32**(1) (2014). https://doi.org/10.1145/2560537
10. Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., Ferdinand, C.: CompCert — a formally verified optimizing compiler. In: Embedded Real Time Software and Systems (2016), https://inria.hal.science/hal-01238879v1
11. Lööw, A., Kumar, R., Tan, Y.K., Myreen, M.O., Norrish, M., Abrahamsson, O., Fox, A.: Verified compilation on a verified processor. In: Programming Language Design and Implementation (PLDI). ACM (2019). https://doi.org/10.1145/3314221.3314622, https://cakeml.org/pldi19.pdf
12. Waterman, A., Asanović, K., Hauser, J.: The RISC-V Instruction Set Manual, Volume II: Privileged Architecture. https://riscv.github.io/riscv-isa-manual/snapshot/privileged/ (2025), online; accessed 16 June 2025
13. Zephyr Project: Zephyr Project Documentation (2025), https://docs.zephyrproject.org/latest/, available at https://docs.zephyrproject.org/latest/