

# End-to-end Mechanized Proof of a JIT-accelerated eBPF Virtual Machine for IoT

Shenghao Yuan<sup>1,3</sup>, Frédéric Besson<sup>2</sup>, and Jean-Pierre Talpin<sup>1</sup>



<sup>1</sup> Inria, France ([firstname.lastname@inria.fr](mailto:firstname.lastname@inria.fr))

<sup>2</sup> Inria, Univ Rennes, CNRS, IRISA

<sup>3</sup> State Key Laboratory of Blockchain and Data Security, Zhejiang University, China ([shenghaoyuan0928@163.com](mailto:shenghaoyuan0928@163.com))



**Abstract.** Modern operating systems have adopted Berkeley Packet Filters (BPF) as a mechanism to extend kernel functionalities dynamically, *e.g.*, Linux’s eBPF or RIOT’s rBPF. The just-in-time (JIT) compilation of eBPF introduced in Linux eBPF for performance has however led to numerous critical issues. Instead, RIOT’s rBPF uses a slower but memory-isolating interpreter (a virtual machine) which implements a defensive semantics of BPF; and therefore trades performance for security. To increase performance without sacrificing security, this paper presents a fully verified JIT implementation for RIOT’s rBPF, consisting of: i/ an end-to-end refinement workflow to both proving the JIT correct from an abstract specification and by deriving a verified concrete C implementation; ii/ a symbolic CompCert interpreter for executing `jited` binary code; iii/ a verified JIT compiler for rBPF; iv/ a verified hybrid rBPF virtual machine. Our core contribution is, to the best of our knowledge, the first and fully verified rBPF JIT compiler with correctness guarantees from high-level specification to low-level implementation. Benchmarks on microcontrollers hosting the RIOT operating system demonstrate significant performance improvements over the existing implementations of rBPF, even in worst-case application scenarios.

**Keywords:** mechanized proof · virtual machines · JIT compiler · eBPF.

## 1 Introduction

Kernel extensibility is the capability of an operating system to extend its core functionalities with privileged services at runtime. It is an essential operating system feature for mainframes, PCs, phones but also smaller devices of the Internet of Things (IoT). Berkeley Packet Filters (BPF) was originally introduced [22,23] to provide such extensions for Unix-BSD systems (*e.g.* network packets filtering, cryptographic protocols, tools such as `tcpdump`, etc). BPF is an assembly language that defines a virtual RISC-like instruction set architecture (ISA). BPF scripts are executed in kernel to parameterize or extend privileged network stacks. For devices like PCs, servers and routers, the Linux community adopted the concept of BPF and extended it to provide ways to run custom in-kernel virtualized code, hooked as "plugins" to various services, and for

many other purposes beyond packet filtering [9]. The ISA of Linux’s extended BPF (eBPF) is derived from the 64-bit RISC-V instruction set. It features a sophisticated verifier [27], to statically analyze eBPF binary instructions, and an interpreter/just-in-time (JIT) compiler, to execute eBPF binaries on varieties of 64/32-bit architectures, *e.g.*, x86, ARM and RISC-V.

The correctness of the eBPF VM and/or JIT is critical for the integrity of the Linux kernel. Bugs in their implementations have led to security vulnerabilities, *e.g.*, allowing execution of arbitrary code within the kernel context [16]. For high-assurance of correctness, researchers have successfully applied verification methods to eBPF JITs *e.g.*, [28,34,35], to find and fix previously unknown bugs. The eBPF instruction set is also used at the lowest end of the spectrum of the IoT, on low-power and resource-constrained devices using micro-controller units (MCUs) such as ARM Cortex-M, running smaller and resource-frugal operating systems, such as the RIOT operating system [2]. Recent work has extended the RIOT micro-kernel runtime with rBPF [38], a 64-bit register-based VM, using fixed-size 64-bit instructions and a reduced ISA derived from eBPF. This extension provides so-called femto-containers [39]: the capability for RIOT to run privileged services, compiled as BPF binaries, each run in a sandboxed VM.

However, low-power IoT devices that run RIOT rarely support hardware memory protections and they cannot afford the resource demands of an online verifier to detect possibly faulty scripts. Instead, MCU-class femto-containers implement a defensive semantics which checks dynamically the preconditions of each instruction before executing it. This ensures the safety and isolation of the eBPF script. Previous works [37,39] have tackled the challenges of implementing a fully-verified and memory isolating VM for RIOT.

While previous works focused on trust (verified fault-isolation) and frugality (minimal footprint), the challenge addressed in this paper is to boost performance while maintaining the highest degree of reliability, security, and frugality. For that purpose, we extend the existing fault-isolating VM, namely CertBPF [37], with a JIT compiler that comes with mechanized correctness proofs. Our goal is to improve efficiency while providing certified security guarantees *i.e.*, the integrity of the host device, even in the presence of malicious code. In this aim, we present a *hybridly* accelerated virtual machine (HAVM), the first rBPF virtual machine that locally Just-In-Time compiles and executes sequences of safe instructions (either exempt from, or subject to benign, runtime checks), while behaving as a virtual machine for the most sensitive memory transactions which, if `jited`, would require multiple runtime checks ruining any performance or resources benefit. JIT acceleration highly improves the efficiency of rBPF femto-containers, but may potentially introduce subtle errors due to the sheer complexity of designing a JIT compiler. We exhaustively waive such risks by a correctness theorem mechanically verified using the Coq proof assistant [5].

## 1.1 Challenges

Typically, a JIT compiler manipulates three different programming or intermediate languages: it translates bytecode (the *source language*) to specific machine

code (the *target languages*), and it is usually implemented in a low-level system language such as C (the *host language*) for space efficiency and performance. Developing a JIT compiler of high-assurance hence poses major challenges.

*JIT design is error-prone.* JIT compilers are more complex than ahead of time compilers. The translations of instructions and protocols they perform are error-prone : 1/ they perform transformations of architecture-dependent information at binary level, different instruction encoding formats and specific calling conventions and 2/ the host C language of the compiler eases low-level memory management mistakes, *e.g.*, *array-out-of-bound*. Unsurprisingly, many eBPF *JIT-related* vulnerabilities have been reported to the Linux community, regarding kernel execution of arbitrary code [12] or confidentiality [14].

*Formalizing a JIT compiler is challenging.* Ahead-of-Time compilers usually output assembly code by relying on separate assembler and linker to produce machine code (plus runtime libraries in, *e.g.*, Rust, OCaml). JIT compilers produce machine code directly, exposing vital semantics-level gaps: the host (C) language, source, and target (ARM) binary have different semantics along with specific calling convention. Existing compiler verification works, *e.g.*, CompCert, provide semantics from C to assembly languages but not to binary level currently. One cannot directly reuse a CompCert backend assembly semantics to formalize the target binary semantics of a JIT compiler as it does not conform its calling convention.

*End-to-end verification gap.* Our JIT compiler is intended to run within the RIOT operating system kernel on resource-limited micro-controller devices. In that context, the provision of a formally verified JIT model with a high-level specification is not enough: there is still a verification gap to produce a verified low-level C implementation from that abstract specification. The CompCert verification workflow is not suitable to that aim as it extracts OCaml code and depends on unverified OCaml runtime libraries, assembler and linker. Other JIT verification approaches, such as Jitterbug [28], suffer from this very same verification gap: the high-level specification is verified, not its extracted compiler.

## 1.2 Contributions

In this paper, we address these challenges by presenting the first end-to-end refinement workflow for application to the verified extraction of equivalent C implementations of a hybrid virtual machine embedding a JIT compiler. Specifically, we make the following contributions:

*End-to-end Refinement Methodology.* We propose an end-to-end refinement methodology that i/ *Horizontally, from source to target*, formally verifies a JIT compiler’s correctness in Coq using the standard CompCert simulation framework, and ii/ *Vertically, from Gallina model to C code*, formally extracts an equivalent, optimized and executable C implementation from its own JIT specification in Gallina (the functional language embedded in Coq). A strength of our

proof methodology lies in the capacity of extracting a verified C model from the standard compiler verification workflow in Coq, from specification to executable (end-to-end).

*Symbolic CompCert ARM Interpreter.* We extend the standard CompCert ARM backend with symbolic execution, for the purpose of reusing the existing CompCert calling convention to support binary code execution. This extension allows our new CompCert ARM interpreter to correctly interpret (`jit`) binary code while ensuring the preservation of the ARM calling convention.

*A Verified JIT Compiler for rBPF.* We design a JIT compiler translating rBPF Arithmetic and Logic (ALU) instructions into binary code. To implement our end-to-end approach, we prove a semantics preservation theorem between the source transition system (rBPF) and the target transitive semantics (rBPF with `jit` code), and extract a verified C implementation of the JIT compiler.

*A Verified Hybrid Virtual Machine for rBPF.* We introduce HAVM, a Hybridly jit-Accelerated VM. HAVM can switch between (*verified*) interpreted, runtime-costly, defensive memory bound checks (for load-store operations) and fully *verified* JIT-compiled code (for arithmetic operations). The verified C model of HAVM is derived from its abstract semantics by our end-to-end workflow.

**Plan** The rest of the paper is organized as follows: [Section 2](#) provides background on CompCert. [Section 3](#) outlines our end-to-end refinement workflow and introduces the application to produce both a verified JIT compiler and VM in C. [Section 4](#) defines our symbolic CompCert ARM interpreter. [Section 5](#) introduces our JIT design and applies our workflow to produce a verified C implementation of the JIT with semantics preservation. [Section 6](#) presents the complete HAVM mixing with a hardware ARM interpreter, a rBPF interpreter, and an interface function allowing them to interleave execution. [Section 7](#) case-studies the performance of our generated VM implementation in comparison to all existing VMs. [Section 8](#) reports our lessons learned, [Section 9](#) discusses related works and [Section 10](#) concludes.

## 2 Preliminaries

CompCert [17] is a C compiler that is both programmed and proven correct using the Coq proof assistant. It compiles C programs into assembly code *e.g.*, ARM. The compiler is structured into passes using several intermediate languages. Each intermediate language is equipped with an operational semantics defined by a labelled transition system denoted as  $E \vdash st \xrightarrow{t} st'$ . It represents one execution step from machine state  $st$  to machine state  $st'$  in some environment  $E$ . The trace  $t$  denotes the observable events generated by the execution step.

Each pass is proven to preserve observational equivalence of programs using a *simulation* relation. CompCert employs two types of simulations: forward

simulation (*i.e.*, every behaviour of the source program is also a behaviour of the compiled program) and backward simulation. CompCert proves most of its passes using forward simulation because it is easier to reason with. It uses a *forward to backward* lemma to construct a backward simulation from a forward one. The composition of all the simulation lemmas for the individual compiler passes forms the semantic preservation theorem:

**Theorem 1 (Semantic Preservation).** *Suppose that  $tp \in T$  is the result of the successful compilation of the program  $p \in S$ . If  $bh$  is a behaviour of  $tp$  ( $bh \in \llbracket tp \rrbracket^T$ ) then there exists a behaviour  $bh'$  such that  $bh'$  is a behaviour of  $p$  ( $bh' \in \llbracket p \rrbracket^S$ ) and  $bh'$  improves  $bh$ , *i.e.*:*

$$\forall p \ tp \ bh, \text{ compcert } p = [tp] \rightarrow bh \in \llbracket tp \rrbracket^T \rightarrow \exists bh', bh' \in \llbracket p \rrbracket^S \wedge bh' \subseteq bh$$

$bh' \subseteq bh$  if either,  $bh'$  is equal to  $bh$ , or  $bh'$  is an undefined behaviour replaced by a defined behaviour in  $bh$ . CompCert returns an option-typed object:  $[tp]$  denotes success with result  $tp$ , and  $\emptyset$  denotes failure.

The memory model and data structures (representation of values) are shared across all the intermediate languages of CompCert [18,19]. CompCert defines machine integers with different sizes, *e.g.*, *int* for 32-bit words and *int64* for 64-bits long integers. A value  $v \in val$  can either be a 32-bit  $Vint(i_{32})$  or 64-bit  $Vlong(i_{64})$  machine integer, a pointer  $Vptr(b, o)$  to a block  $b$  and offset  $o$ , a floating-point number, or the undefined value  $Vundef$ . A CompCert memory  $m$  consists of a collection of partitioned arrays. Each array has a fixed size and is identified by an uninterpreted block  $b \in block$ .

In addition to CompCert, our project employs the same Gallina-to-C transpiler  $\partial x$  as the verified virtual machine presented in [37].  $\partial x$  is an unverified translator that was developed to design the verified PIP proto-kernel [13] in Coq. It transpiles a monadic (imperative) Gallina source definition to CompCert C code of identical structure and terms. We chose to reuse  $\partial x$  for its practicality (traceability) and to reuse the same translation validation methodology as [37].

### 3 A Workflow for End-to-End Refinement

This section presents an overview of our methodology to prove the correctness of a virtual machine which dynamically compiles, at load time, a subset of the instructions. Informally, the end-to-end correctness guarantee of the virtual machine can be phrased as follows. Suppose that a source code  $s$  executes according to the small-step operational semantics and returns a value  $v$ . The virtual machine just-in-time compiles a subset of the source instructions of  $s$  into binary code and, therefore, generates a *compound* program  $t$  composed of original instructions augmented with calls to binary code. The virtual machine then executes the program  $t$  and returns the exact same value  $v$ .

In the following, we explain the high-level structure of the proof and how to get a rigorously end-to-end formal guarantee for a virtual machine written in the C language using the Coq proof assistant and the CompCert compiler.

### 3.1 Methodology

At high-level, the methodology can be explained using T-diagrams [6]. The T-diagram of [Figure 1a](#) depicts a compiler which given as input a source program

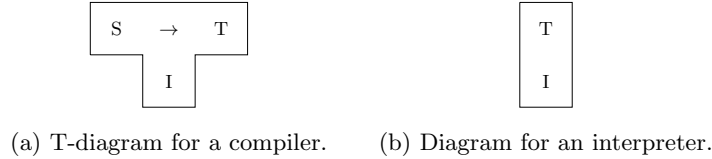


Fig. 1: T-diagrams.

$s \in S$ , generates a target program  $t \in T$  and is implemented using the implementation language  $I$ . We will also make use of the diagram of [Figure 1b](#) which depicts an interpreter for the language  $T$  implemented in the language  $I$ . As our methodology is formally grounded each diagram comes with a soundness proof. In particular, each language is equipped with a formal semantics and a compiler diagram comes with a semantic preservation theorem similar to [Theorem 1](#).

**JIT compiler structure** The JIT compiler and its proof follow the structure of the diagram of [Figure 2a](#). To begin with, we write a compiler for the source language  $S$  to the target language  $T$  using the Gallina language, written  $G$ , of the Coq proof assistant. We also prove a semantics preservation theorem guaranteeing the correctness of the compiler. To get an executable compiler outside the proof assistant, the usual approach is to perform program extraction [20,32] to a functional language. However, functional languages require a sophisticated runtime *e.g.* a garbage collector, that is not compatible with our constrained resources. Instead, we perform a rewrite of the compiler to a tiny subset of Gallina ( $G_0$ ). Though the transformation is systematic, it is manual as indicated in [Figure 2a](#) by the implementation language  $H$  which stands for *Human*. In that case, that associated correctness is that both compilers compute the exact same

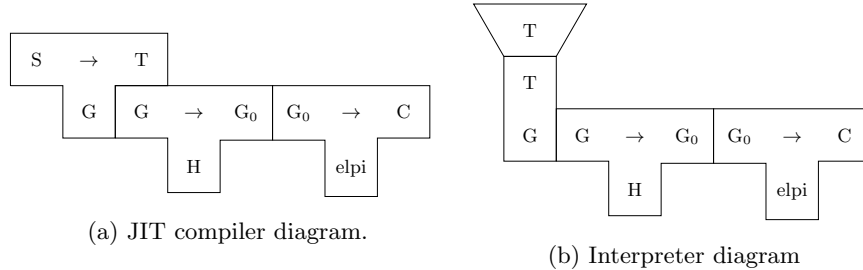


Fig. 2: Virtual machine diagrams.

output program. However, the compiler using the language  $G$  (Gallina without restriction) is designed as a composition of passes. This simplifies the semantic preservation proof but constructs intermediate functional data-structures. The compiler restricted to  $G_0$  is using more imperative data-structures (using an explicit state-monad) and is using a more direct generation of binary code avoiding intermediate data-structures thus using resources that are compatible with our resource constrained environment. The last step consists in using the  $\partial x$  Coq plugin, written in elpi [8], which converts the language  $G_0$  into  $C$ . As  $\partial x$  is not proof generating, we perform a manual but systematic translation validation step with respect to the formal semantics of CompCert showing that both the  $G_0$  program and the generated  $C$  code compute the same result.

**Execution of JIT Compiled Code** The proofs of the JIT compiler are performed over the small-step operation semantics of the target language which is the combination of source semantics for the non-JIT compiled instruction and the semantics of the binary code. The diagram of Figure 2b shows how to derive an executable  $C$  Virtual Machine for this language. To execute a program  $\boxed{T}$  in language  $T$ , we program in Gallina ( $G$ ) an interpreter for the language  $T$ . As we explain in Section 4, the interpreter is equipped with a sub-interpreter for executing binary code. Here, the proof is that if the interpreter terminates without exhausting its allocated execution steps, it computes the same result as the small-step semantics. To get an executable  $C$  code, we follow a similar methodology: express the interpreter in restricted Gallina  $G_0$  and run the  $\partial x$  tool to get a  $C$  program. What may be puzzling is how the Gallina semantics of binary code may be compiled into  $C$  code. This indeed requires some substantial work. What we do is to augment the semantics of all the intermediate language of CompCert with a so-called `builtin` which embeds the semantics of our Gallina interpreter for executing binary code. Eventually, we show that this semantics coincide with the existing semantics of CompCert assembly augmented to fetch and decode instructions from memory.

**Terminology** In the following, we call *horizontal* refinement a proof related to a Gallina program  $p \in G$  and *vertical* refinement a proof related to lower-level programs  $p \in G_0$  or  $p \in C$ .

### 3.2 Application to a rBPF Virtual Machine

We instantiate our approach to derive a verified  $C$  implementation of a VM for rBPF enhanced with a JIT compiler. As we target a 32-bit ARM architecture, we consider the rBPF variant operating on 32 bit registers. The JIT compiler is invoked at load time and translates to ARM code straight-line sequence of arithmetic and logic (ALU) instructions. The rationale is that these are the part of the code for which we can expect a substantial speedup as the rBPF registers can be mapped to ARM registers and an ALU instructions is mapped to a short sequence of ALU ARM instructions. For memory operations, rBPF

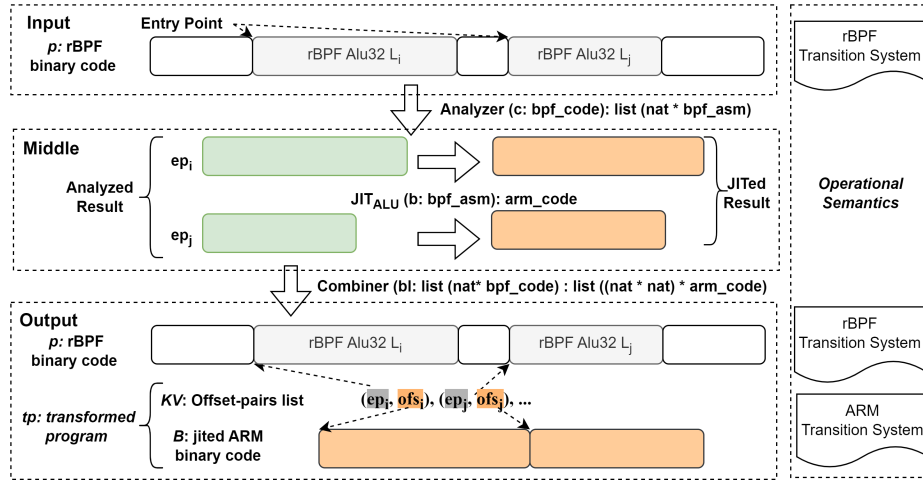


Fig. 3: *JITCompiler* Structure (left) and related Semantics (right).

implements a costly dynamic defensive semantics which consists in iterating over a list of allowed memory regions and checking that the memory address is correctly aligned and respect the access rights. Yet, a partial JIT compiler does not simplify the verification task as the VM needs to ensure the interoperability between streams of rBPF and ARM instructions while reasoning at high-level using formal models.

*JIT Compiler Structure.* The structure of the JIT compiler is illustrated in Figure 3. As hinted in the previous Section, the JIT compiler is not monolithic but made of three passes. The *Analyzer* pass identifies sequences of rBPF ALU instructions and disassemble them. The core of the JIT compiler is  $JIT_{ALU}$  (see Section 5) which translates a list of rBPF instructions into the corresponding ARM code. Eventually, *Combiner* collects all the binary instructions into a single array  $B$  and generates another array  $KV$  such that  $KV[i] = ofs$  if  $i$  is the entry point of a sequence of rBPF ALU instructions  $si$  and  $B[ofs]$  is the start of the binary ARM code corresponding to  $si$ .

*Horizontal Refinement: JIT Correctness.* The proof of our JIT compiler follows the structure of a standard CompCert compiler proof, with the difference that the target language is made of both rBPF instructions and binary ARM code. This multi-language semantics requires the calling-conventions of ARM to ensure the interoperability of the rBPF semantics and ARM semantics.

*Vertical Refinement: Verified JIT and HAVM.* The goal of vertical refinement is to extract a verified C  $JITCompiler_C$  from its Gallina model  $JITCompiler$  and generate a VM  $HAVM_C$ . One challenge is to ensure that the C program is a valid refinement of the Gallina program. It appears, however, that calling



some in-memory ARM code from a C program has no defined semantics in CompCert. To tackle the issue, we augment the semantics of CompCert with a defensive symbolic ARM semantics.

## 4 Symbolic CompCert ARM Interpreter

The current standard CompCert backend defines various assembly languages, *e.g.*, ARM, along with their formal semantics. Unfortunately, it cannot be reused for our JIT compiler because *JITCompiler* requires the binary-level semantics of ARM. Additionally, the calling convention of the `jited` code exceeds the capability of the existing CompCert ARM semantics.

To address this issue, we firstly define an ARM decoding function to link the ARM semantics from binary-level to CompCert assembly-level. Subsequently, we introduce a symbolic CompCert ARM semantics that lifts the ARM instruction semantics and the calling convention into a symbolic form. This new CompCert backend employs symbolic execution to interpret binary ARM instructions, and symbolic values allow to i/ initialise ARM registers when switching from C to binary; ii/ define an executable semantics of ARM capable of simulating (and verifying) the calling conventions. Yet, during the assembly code generation pass of CompCert, the symbolic ARM semantics is switched to the concrete CompCert ARM semantics.

*ARM Decode.* We implement a decoding function in Gallina that translates binary ARM instructions to standard CompCert assembly ARM instructions. We also define an encode function embedded in the JIT process, and prove that this ‘decode-encode’ pair is consistent.

**Lemma 1 (Decode-Encode Consistency).**  $\forall i, \text{decode}(\text{encode } i) = [i]$

*ARM Calling Convention.* When interpreting (‘calling’) a list of `jited` binary code, the ARM calling conventions need to be preserved: i/ the caller must save the value of argument registers ( $r_0 - r_3$ ), and ii/ the callee must save the value of ( $r_4 - r_{11}$ ). For efficiency purposes, we stipulate that:

- callee-saved registers must be dynamically preserved by the `jited` binary code, as it may not modify all registers during one procedure call.
- all caller-saved registers are statically preserved by our ARM backend.

We also allocate a stack frame to implement the calling convention before binary execution, and verify that all ARM callee-saved registers in the final register state have been reset to their initial values, relying on a symbolic execution technique.

*Symbolic Execution.* The register map `SReg` is symbolic: each register `sr` is either an abstract value or a concrete value bound to an actual ARM register `r`.

$$\text{SReg} \ni sr ::= \text{abstract}(r) \mid \text{concrete}(v)$$

All initial registers `init_rs` have abstract values, *e.g.*,  $\text{SReg}[r_0] = \text{abstract}(r_0)$ . The concrete values of registers are inserted by running the `jited` code.

*CompCert ARM Interpreter.* We design a symbolic variant of the CompCert ARM interpreter that utilizes the existing CompCert ARM transition function to execute user-specific ARM binary code. We first introduce the initial and final states of the interpreter, then explain how the interpreter works.

We define a function `init_state` to create a new ARM environment for interpreting binary code. It first copies values from the arguments list `args` to caller-saved registers of the symbolic register map `init_rs`, according to the function's signature `sig`. Then, `init_state` allocates a new memory block `stk` in CompCert memory with a fixed stack size `sz`. It stores the previous stack pointer `sp` at position `pos` in `stk` and updates the stack pointer with the start address of this block. Finally, it stores the return address, *i.e.*, the next address of the old `pc`, to `r14`. Since the first argument always points to the location of the `jited` binary code to be executed, `init_state` also assigns the program counter `pc` with the first argument value `r0`.

```

init_state(sig, args, sz, pos, m) =
  match alloc_arguments(sig, args, init_rs) with
  |  $\emptyset \Rightarrow \emptyset$ 
  |  $[rs] \Rightarrow$  match alloc_frame(sz, pos, rs, m) with
    |  $\emptyset \Rightarrow \emptyset$ 
    |  $[(rs', m')] \Rightarrow [(rs' \{ r_{14} \leftarrow \mathbf{abstract}(pc) + 1, pc \leftarrow rs'[r_0] \}, m')]$ 

```

We then define a Boolean predicate `is_final_state` to describe a well-formed final state of the `jited` code.

```

is_final_state(rs : SReg) : bool = rs[pc] == abstract(r14) &&
  rs[sp] == abstract(sp) && ( $\forall i. 4 \leq i \leq 11 \rightarrow rs[r_i] == \mathbf{abstract}(r_i)$ )

```

The predicate `is_final_state` stipulates that 1/ `pc` should hold the return address stored in `r14`; 2/ The stack pointer is restored; 3/ All callee-saved registers should have their initial values.

The symbolic CompCert ARM interpreter is defined as follows:

```

bin_exec(fuel, sig, args, sz, pos, m) =
  match init_state(sig, args, sz, pos, m) with
  |  $\emptyset \Rightarrow \emptyset$ 
  |  $[(rs', m')] \Rightarrow$  bin_interp(fuel, rs', m')

```

Where the parameters include: 1/ `fuel`, ensuring the termination of its recursive call to `bin_interp`. 2/ `sig`, the signature of the arguments used by the input ARM binary code. 3/ `args`, the arguments list. 4/ `sz`, the size of the allocated stack frame. 5/ `pos`, the position of the old stack pointer in the new stack frame. 6/ `m`, the CompCert memory.

First, `bin_exec` uses `init_state` to create a proper ARM environment, including the initialized ARM register map  $rs'$  and the new memory  $m'$ . It then calls `bin_interp` recursively to interpret ARM binary code until it reaches the final state. It either returns  $r_0$ 's value or exhausts  $fuel$ . Each iteration of `find_instr` fetches the instruction at the program counter  $pc$  and decodes it. If its binary instruction decodes successfully, `bin_interp` then calls the symbolic ARM transition function `symbolic_transf` to execute it and proceeds to the next instruction, if no errors occur.

```

bin_interp(fuel, rs, m) =
  if is_final_state(rs) then [(rs[r0], m)] else if fuel == 0 then ∅
  else match find_instr(rs[pc], m) with
    | ∅ ⇒ ∅
    | [ins] ⇒ match symbolic_transf(ins, rs, m) with
      | ∅ ⇒ ∅
      | [(rs', m')] ⇒ bin_interp(fuel - 1, rs', m')

```

We have integrated this symbolic ARM backend into the CompCert environment and proven that it is compatible with the standard CompCert ARM semantics. This interpreter also provides an equivalent built-in C function ‘`bin_exec`’: the CompCert “builtins” mechanism ensures that the semantics preservation theorem still holds between the Gallina function `bin_exec` and its built-in ‘`bin_exec`’.

## 5 A verified Just-In-Time Compiler for rBPF

Our JIT compiler is exclusively designed to translate rBPF ALU instructions into target binary code. The compiler structure is shown in [Figure 3](#). This section highlights the  $\text{JIT}_{\text{ALU}}$  translation as the other two are straightforward. We then detail the end-to-end refinement verification process introduced to prove this JIT compiler correct.

### 5.1 JIT Design

*High-level Intuition.*  $\text{JIT}_{\text{ALU}}$  translates a list of rBPF ALU instructions into a list of ARM binary code. As depicted in [Figure 4](#), the target `jited` binary list has a specific linear structure: i/ The Head part copies  $r_1$ 's to  $r_{12}$  as the following stages may override  $r_1$ ; ii/ The dotted part is made of the following stages: *Spilling* copies ARM registers on the stack, *Load* transfers register values from rBPF to ARM, and *Core* performs the arithmetic computation operating on ARM registers that is equivalent to the behaviour of the source rBPF ALU list; iii/ The subsequent part *Store* updates registers from ARM to rBPF, and *Reloading* pulls stack values into ARM registers; iv/ The *Tail* part frees the current stack frame and branches to the return address.

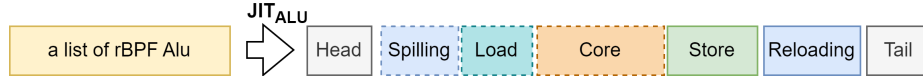


Fig. 4: Structure of jited code.

The *Load* and *Store* stages perform interactions between ARM registers and rBPF registers for consistency after executing the `jited` code, while the *Spilling* and *Reloading* stages guarantee the ARM calling convention. As the ARM binary can only ‘see’ ARM registers and memory blocks, the rBPF register map is stored in the special block `st_blk` and its start location ( $\text{Vptr}(st\_blk, o)$ ) is stored in  $r_1$  with the argument passed by the `jit_call` function. In the layout of `st_blk`, cells  $[4 * i, 4 * i + 4)$  have the value of  $R_i$  ( $0 \leq i \leq 10$ ), and  $[44, 48)$  have the rBPF PC’s value.

*Core Mapping.* The rBPF `Alu` instructions include common arithmetic operations where the destination operator is a general rBPF register ( $R_0 - R_9$ ) and the source could be a rBPF register ( $R_0 - R_{10}$ ) or a 32-bit immediate number.

$op ::=$	<code>ADD   SUB   MUL   DIV   OR   AND   XOR   MOV</code>	general
	<code>LSH   RSH   ARSH</code>	shift
$ins ::=$	<code>Alu op dst src   ...</code>	instruction

The core mapping of `JITALU` includes two general rules and one specific rule.

- *G1*: maps a register-based rBPF operation (source is a register) into its corresponding ARM instruction operating the related ARM registers, *e.g.*, `Alu ADD Rd Rs` is translated into `add rd rd rs`.
- *G2*: maps an immediate rBPF operation according to the provided range  $i$ .
  - *G2.1*: If the immediate constant  $i$  is in the range  $[0, 255]$ , each instruction is directly mapped to an 8-bit-immediate ARM instruction.
  - *G2.2*: If  $i$  is within the range  $[256, 65535]$ , it is first copied into ARM register  $r_{11}$  using `movw`, and then mapped to an ARM instruction with  $r_{11}$  as the second operand. `movw` writes an immediate value to the low 16 bits of the destination register.
  - *G2.3*: Otherwise,  $i$  is loaded into  $r_{11}$  using `movt` and `movw` before performing the operation. `movt` modifies the high 16 bits.
- *S1*: For the rBPF division and shift instructions, the immediate operation is mapped if the constant  $i$  is valid, *i.e.*,  $i \neq 0$  for division and  $(0 \leq i \leq 31)$  for shifts. For completion, `JITALU` returns failure if it encounters an invalid  $i$ . Note that validity is however a pre-condition guaranteed by the host virtual machine [37], which analyzes script prior to execution and, among other things, checks the validity of immediate instructions.

*Interaction.* In the `Core` stage, source instructions operate over rBPF registers, while the `jited` ARM code operates on ARM registers. Hence, a consistent

interaction between rBPF and ARM registers is mandatory.  $\text{JIT}_{\text{ALU}}$  generates extra binary code performing the interaction in the *Load* and *Store* stages, which relies on two special sets  $LD$  (rBPF registers that have been loaded into ARM registers) and  $ST$  (rBPF registers that should be updated in the *Store* stage). For each rBPF ALU instruction,  $\text{JIT}_{\text{ALU}}$  adopts two rules to produce memory instructions and update the register sets before it performs the core mapping.

- *I1*: if the rBPF destination register  $R_d$  isn't in the  $LD$ , i/ if  $r_d$  is an ARM callee-saved register, generate ' $\text{str } r_d [sp, \#(d * 4)]$ ' for spilling; ii/ generate ' $\text{ldr } r_d [r_{12}, \#(d * 4)]$ ' for the *Load* stage; iii/ add  $R_d$  into  $LD$  and  $ST$ .
- *I2*: if the rBPF source is a register  $R_s$  that isn't in the  $LD$ , generate the same code as *I1* but only add  $R_s$  into  $LD$ .

After all rBPF ALU instructions are *jit*ed,  $\text{JIT}_{\text{ALU}}$  updates the rBPF register map by generating ' $\text{str } r_i [r_{12}, \#(i * 4)]$ ' for all  $r_i \in ST$ . Then, to preserve the ARM calling convention,  $\text{JIT}_{\text{ALU}}$  resets all modified ARM callee-saved registers  $r_i$  from the stack frame by ' $\text{ldr } r_i [sp, \#(i * 4)]$ '.

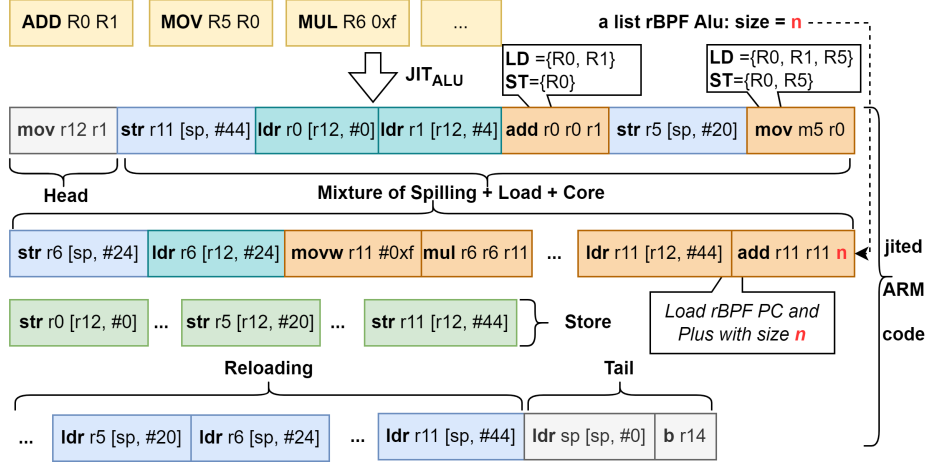
*Example.* Figure 5 illustrates the entire  $\text{JIT}_{\text{ALU}}$  process. Consider a source rBPF ALU snippet composed of  $n$  instructions: ' $[ADD R_0 R_1; MOV R_5 R_0; MUL R_6 0xf; \dots]$ '. The *Head* is always ' $\text{mov } r_{12} r_1$ '. Then, the *Spilling* stage saves  $r_{11}$  in the stack frame because it will be modified later. For the first rBPF instruction, the *jit*ed code copies  $R_0$  and  $R_1$  into  $r_0$  and  $r_1$ , and then performs the ARM addition. The initial  $LD$  and  $ST$  are  $\emptyset$ , the updated state is  $LD = \{R_0, R_1\}$  and  $ST = \{R_0\}$ . For the second rBPF instruction, the *jit*ed code requires a *Spilling* stage to save  $r_5$  first, then performs the move and updates  $LD$  and  $ST$  with  $R_5$ . After the  $n$ -th rBPF instruction, there are two instructions to update the rBPF's PC with the length of the input list. The *Store* stage updates all modified rBPF registers in  $ST$  and PC, and the *Reloading* stage resets all used call-save registers to their previous values stored in the stack frame during the *Spilling* stage. The last stage is *Tail*.

## 5.2 JIT Correctness

We employ the standard CompCert framework to prove the JIT compiler correct. The proof initially refines the source rBPF semantics into an intermediate model (after analysis), and subsequently refines into the target HAVM semantics.

*Machine State.* The rBPF state is a pair  $state ::= (R, M)$ , consisting of a CompCert memory model  $M$  and the register map  $R$ , which associates (32-bit) values with the rBPF registers ( $R_0 - R_{10}$  and  $PC$ ).

*Transition Semantics of rBPF.* The core of rBPF's semantics is a transition function  $T(ins, st) = [st']$  that determines the new state  $st'$  after executing instruction  $ins$  in the initial state  $st$ . In particular, the program counter PC is incremented. For simplification, we only present the transition rule of arithmetic

Fig. 5: JIT<sub>ALU</sub> example.

instructions in one execution step  $step_{rBPF}$ . The first two premises model the actions of reading and decoding the  $n$ -th instruction  $ins$ , which is pointed to by the program counter  $PC$ , from the list  $C$ . Then, the rule executes  $ins$  and returns a new state.

$$\frac{R[PC] = \mathbf{Vint}(n) \quad C[n] = [ins] \quad ins = \mathbf{Alu} \ op \ dst \ src \quad T(ins, (R, M)) = [(R', M')]}{C \vdash (R, M) \xrightarrow{\epsilon} (R', M')}$$

*Transition Semantics of the Analyzer.* The first module of *JITCompiler* is an analyzer that generates a list of analysis results  $BL$ , a pair of entry point and a list of (decoded) rBPF ALU instructions, from the input rBPF binary  $C$ . The refined semantics only replaces the previous arithmetic rule with the following one. When  $PC$  is an entry point and its related rBPF ALU list is in  $BL$ , a refined transition function  $T^L$  is used to sequentially execute all instructions in  $l$ .

$$\frac{R[PC] = \mathbf{Vint}(n) \quad (n, l) \in BL \quad T^L(l, (R, M)) = [(R', M')]}{C, BL \vdash (R, M) \xrightarrow{\epsilon} (R', M')}$$

We prove that, for one step ' $step_A$ ' of this refined machine,  $TS_A$  has a backward simulation relation with respect to several steps ' $step_{rBPF}^*$ ' of the source machine  $TS_{rBPF}$ .

**Lemma 2** ( $TS_A$  simulates  $TS_{rBPF}$  in one step).  $\forall C BL t st st'$ ,

$$\text{Analyzer } C = [BL] \wedge (\text{step}_A C BL) st t st' \rightarrow (\text{step}_{rBPF}^* C) st t st'$$

*Transition Semantics of HAVM.* The *Combiner* module calls  $\text{JIT}_{\text{ALU}}$  to generate all binary code lists from the analyzing results and combines all `jited` code into one list. The target semantics only changes the arithmetic rule compared to the source semantics. Where  $PC$  is an entry point and its related `jited` list located in  $bl$  starting from offset  $ofs$ , the transition function  $T^{ARM}$  calls the symbolic ARM interpreter `bin_exec` to execute the `jited` code.

$$\frac{R[PC] = \text{Vint}(n) \ ((n, ofs), bl) \in TP \ T^{ARM}(ofs, bl, (R, M)) = [(R', M')]}{C, TP \vdash (R, M) \xrightarrow{\epsilon} (R', M')}$$

**Lemma 3** proves that one step ‘ $\text{step}_A$ ’ of  $TS_A$  has a forward simulation relation with one step ‘ $\text{step}_{HAVM}$ ’ of the target machine  $TS_{HAVM}$ . Since the semantics of  $TS_{HAVM}$  encompasses rBPF and ARM, this proof features some interesting inter-operations: i/ Both machines start from the same rBPF state; ii/ When  $TS_{HAVM}$  executes its ALU rule using  $T^{ARM}$ , we prove a simulation between the rBPF state of  $TS_A$  and the ARM state of  $TS_{HAVM}$ ; iii/ After completing the ALU rule, we prove that the `jited` code respects the ARM calling convention, and both machines achieve the same final rBPF state.

**Lemma 3** ( $TS_A$  simulates  $TS_{HAVM}$  in one step).  $\forall C BL TP t st st'$ ,

$$\text{Combiner } BL = [TP] \wedge (\text{step}_A C BL) st t st' \rightarrow (\text{step}_{HAVM} C TP) st t st'$$

From **Lemma 2** and **Lemma 3**, we can prove that *JITCompiler* is correctness because the forward simulation in **Lemma 3** can be reconstructed into a backward proof, and composed to a complete simulation proof from target to source.

### 5.3 JIT Vertical Refinement

The goal of this section is to design a verified and optimized *JITCompiler C* implementation. The refinement process is step-wise.

*Removing Intermediate Representation.* *JITCompiler* adopts a modular design for proof simplification. Expectedly, *JITCompiler* is memory-consuming and of low efficiency, as it takes additional memory to save analysis results (**Figure 3**, middle). *JITCompiler<sub>opt</sub>* instead operates as “find a rBPF `Alu`, *jit* immediately, and check the next one”, with minimal resources and better performances.

*Refining Data Structure.*  $JITCompiler_{opt}$  refines data structures for optimization and synthesis requirements. For example,  $LD$  and  $ST$  are implemented as sorted **ListSets**, which cannot be directly mapped to a C type. We refine ListSet as a Coq Record type  $regSet$  that states which rBPF registers are modified (e.g., flagged *true*). Then ‘ $LD : regSet$ ’ can be extracted as ‘ $_{Bool} LD[11]$ ’ in C.

$$\mathbf{Record} \text{ regSet} := \{ f\_R0 : \mathbf{bool}; \dots; f\_R10 : \mathbf{bool} \}$$

*$\partial x$  Refinement.*  $JITCompiler_{\partial x}$  adopts an option-state monad to model effectful behaviours. For example, reading rBPF input binary  $p$  and writing the **jited** code into the pre-allocated list  $tp\_bin$  with a proper offset in  $jit\_state$ .

$$\mathbf{Record} \text{ jit\_state} := \{ \dots; p : \text{list int64}; \dots; tp\_kv : \dots; tp\_bin : \text{list int} \}$$

We use  $\partial x$  to extract an executable C code  $JITCompiler_C$  from  $JITCompiler_{\partial x}$  using a global state where Coq lists are mapped to C pointers.

$$\mathbf{struct} \text{ jit\_state} \{ \dots; \text{uint64}_t * p; \dots; \dots \text{ tp\_kv}; \text{uint32}_t * \text{tp\_bin} \}$$

The end-to-end proof of the JIT compiler refinement proceeds in two steps: i/ from  $JITCompiler$  to  $JITCompiler_{\partial x}$ , we prove that the refinement is correct (see [Lemma 4](#)) and, ii/ from  $JITCompiler_{\partial x}$  to  $JITCompiler_C$ , we reuse the  $\partial x$  end-to-end verification workflow.

**Lemma 4 ( $\partial x$ -Refinement Correctness).** *Suppose that  $Compiler_{\partial x}$  is the refinement of  $Compiler$ .  $Compiler$  and  $Compiler_{\partial x}$  must generate the same result  $tp$  when they accept the same input program  $p$ .*

$$\begin{aligned} \forall p \text{ tp } st_1, \text{ Compiler } p = [tp] \wedge p \in st_1 \wedge \text{Allocate}(tp) \in st_1 \rightarrow \\ \exists st_2, \text{ Compiler}_{\partial x} st_1 = [(unit, st_2)] \wedge p \in st_2 \wedge tp \in st_2 \end{aligned}$$

where  $x \in st$  if  $x$  is a field of state  $st$  and  $\text{Allocate}(x)$  creates an empty list of the same size as  $x$ .

## 6 HAVM: a hybrid Interpreter for rBPF

This section introduces the first (and fully-verified) hybrid rBPF interpreter HAVM, which interprets the composition of a rBPF binary script with **jited** ARM binary code.

*HAVM Design.* HAVM is formalized as a monadic function in Gallina. First, we highlight several fields in the monadic state of HAVM: 1/  $R.pc$  is the PC of rBPF register map  $R$ ; 2/  $tp\_kv$  is the offset-pairs list; 3/  $M$  is the CompCert memory including a special memory block  $jit\_blk$  storing the **jited** code list.

Then, we extend the standard rBPF interpreter of [\[39\]](#) to implement HAVM. Its step function  $hybrid\_step$  interprets different rBPF instructions. For rBPF **Alu** instructions, it directly calls `jit_call`, which is a monadic instantiation



of our transition function  $T^{ARM}$ . For rBPF memory instructions, *hybrid\_step* inherits the defensive semantics from the vanilla rBPF VM: the *check\_mem* function guarantees the (verified) safety of all memory operations.

```

hybrid_step(hst) =
  match hst.p[hst.R.pc] with | ../.
  | Alu op dst src  $\Rightarrow$  jit_call(hst.tp_kv[hst.R.pc], hst)
  | Mem dst src ...  $\Rightarrow$  if check_mem(...) then safe_mem_op else ...

```

*Refinement Proof.* The vertical refinement proof focuses on the proof from  $TS_{HAVM}$  to the monadic model  $HAVM$  (see Lemma 5) where the simulation relation  $st \sim hst$  is defined as  $st.R = hst.R \wedge st.M = hst.M$ .

**Lemma 5 (Interpreter Refinement).**  $\forall st_1 st_2 t hst_1, st_1 \sim hst_1 \wedge$

$$step_{havm} st_1 t st_2 \rightarrow \exists hst_2, hybrid\_step hst_1 = [(unit, hst_2)] \wedge st_2 \sim hst_2$$

*C Implementation.* We use  $\partial x$  to extract a verified C implementation  $HAVM_C$ . The C version `jit_callC` is implemented by the verified ‘*bin\_exec*’ built-in function. This allows us to prove that the refinement from  $HAVM_{\partial x}$  to  $HAVM_C$ : `jit_call` is equivalent to `jit_callC` due to the verified CompCert built-in mechanism. The Non-Alu cases reuse most of the refinement proofs of [37].

## 7 Evaluation: Case Study of RIOT’s Femto-Containers

We integrated our JIT compiler and the HAVM into the RIOT-OS to provide the same functionalities as the previous vanilla-rBPF module.

*Implementation.* The whole project, available on [36], consists of more than 70k lines of Coq code: The CompCert variant is completed by 6k lines and the rBPF-related transitive systems are approx. 1k lines long. The specification of the JIT compiler 1k lines large and our main proof effort, the JIT correctness theorem, demanded 45k proof code. The vertical refinement to monadic form contains the JIT part (about 4k lines) and the HAVM part (about 3k lines). From the monadic models to the final C implementation, about 10k lines proof code, we rely on the existing end-to-end verification workflow of [37].

*Experiment.* Our experiments are performed on a *nrf52840dk* development board which uses an Arm Cortex-M4 micro-controller, a popular 32-bit architecture (arm-v7m). The experimental benchmark code is compiled using the Arm GNU toolchain version 12.2. The compilation is using level 2 optimization enabled and the GCC option `-foptimize-sibling-calls` to optimize all tail-recursive calls and in turn, bound the stack usage. We also enable `-falign-functions=16` to reduce the performance variation caused by the instruction cache on the device. Lastly, we compare the HAVM implementation against both CertrBPF and Vanilla-rBPF using real-world benchmarks shown in Table 1.

Interpreter	incr	square	bitswap	fib	sock_buf	memcpy	fletcher32	bsort
vanilla-rBPF	8.44 $\mu$ s	8.50 $\mu$ s	42.25 $\mu$ s	94.38 $\mu$ s	325 $\mu$ s	887 $\mu$ s	2283 $\mu$ s	11 722 $\mu$ s
CertrBPF	5.13 $\mu$ s	5.50 $\mu$ s	34.75 $\mu$ s	92.38 $\mu$ s	300 $\mu$ s	822 $\mu$ s	1980 $\mu$ s	10 697 $\mu$ s
HAVM	<b>4.38 <math>\mu</math>s</b>	<b>4.37 <math>\mu</math>s</b>	<b>15.63 <math>\mu</math>s</b>	<b>51.75 <math>\mu</math>s</b>	<b>185 <math>\mu</math>s</b>	<b>588 <math>\mu</math>s</b>	<b>1196 <math>\mu</math>s</b>	<b>7120 <math>\mu</math>s</b>

Table 1: Execution time of real-world benchmarks

The first four benchmarks test purely computational tasks, mainly consisting of rBPF ALU operations. Then, two special benchmarks comprise more memory operations but fewer ALU operations (worst cases for HAVM): the classical BPF socket buffer read/write and memory copy functions. Finally, we benchmark the performance of actual IoT data processing algorithms such as the Fletcher32 hash function or a bubble sort. We observed that, for all real-world benchmarks, HAVM improves performance because of the numerical acceleration JIT-feature.

## 8 Lessons Learned

In this section, we clarify the prospects and limitations of the methodology proposed in the paper and its application to the rBPF JIT compiler.

*Our Goal and Limitations.* The methodology aims at proving the high-level specification of the JIT compiler correct and at extracting a verified C implementation directly from this specification, using refinement. As mentioned in [Section 7](#), our methodology requires a lot of manual proof effort.

Regarding its application to rBPF JIT compilation, our JIT compiler is, more precisely, a hardware accelerator for numerical operations: it only translates a subset of rBPF ISA, the ALU instructions, into ARM binary. We made this choice as the other memory operations of rBPF must be given a defensive semantics to meet the requirement of (memory) fault isolation. This defensive code is large and, if jitted, would significantly increase the binary code size for a limited performance gain.

*Adaptability: move to other Targets.* Should we consider instantiating our methodology and proofs to another target architecture, say RISC-V, then most of the current JIT design and proof techniques could be reused. The only modification would regard platform-dependent elements: the semantics model would need to be based on CompCert/RISC-V, and the JIT’s spilling and reloading stages would have to be modified to match RISC-V’s calling conventions.

*Adaptability: turn to Linux eBPF.* However, our ARM-ALU JIT compiler may not directly be extended to a full-fledged Linux eBPF JIT compiler, as eBPF doesn’t have a defensive semantics for memory operations. Instead, Linux’ eBPF uses a sophisticated verifier to validate memory operations, which would not fit

the memory resources onboard IoT devices, as it is larger than 20k lines of C code, and be an end-to-end verification project in its own rights.

Finally, this paper does not discuss JIT optimizations, unlike modern JIT compilers, which have sophisticated optimization strategies. Proving their correctness in Coq would also be a non-trivial verification task.

However, we believe that these two last limitations could be relaxed once we complete a fully verified “JIT-all” compiler for rBPF. Essentially, the last mile of our journey toward a complete JIT compiler would be one capable of calling a verified ARM binary implementation of the (verified) defensive “*check\_mem*” function and to embed it in the JITed code block. This would essentially amount to verifying an adhoc linker between the JITed code and the embedded *check\_mem*’s binary code.

## 9 Related Works

*Verified Compilers, OS kernels, and VMs.* There is a rich literature on verified software design. Verified compilers include CompCert [17] (from C to assembly) and CakeML [26] (from ML to binary), etc. Verified OS kernels comprise for instance SeL4 [15] (L4 microkernel) and CertiKOS [11] (multi-cores). Verified virtual machines have been developed for richer scripting languages, such as Java VM [21], JavaScript VM [7], and Ethereum [40].

Our work build upon CertrBPF [37], the first verified eBPF VM for RIOT providing a service of so-called femto-containers [39]. CertrBPF provides an end-to-end verification workflow from monadic Gallina models to executable C implementation. However, it has no JIT compiler. The main novelty presented in this paper is the first and fully verified JIT compiler for RIOT rBPF, reusing and enriching the CompCert and CertrBPF projects.

*Verified JITs.* Barriere et al. [3,4] extend the CompCert backend to support general-purpose JIT compilation. They adopt an additional memory model for defining the behaviours of `jited` code and require unverified C glue code to obtain a runnable JIT compiler. Wang et al. [35] extend CompCert to extract a verified JIT compiler *Jitk* from classic BPF (not eBPF) to assembly code in OCaml. All aforementioned extensions rely on unverified TCB consisting of the OCaml runtime, an assembler, and a linker, which are not suitable for a security-critical and resources-limited OS kernel like RIOT. Myreen [25] proves a JIT compiler from a simple stack-based bytecode language to x86 in the HOL4 proof assistant. Van Geffen et al. [10] present an optimized JIT compiler for Linux eBPF, embedded with automated static analysis. Nelson et al. [28,34] develop the domain-specific language *Jitterbug* to write JITs and prove them correct.

All the above approaches only verify the JIT correctness in a high-level abstract model, but do not produce a verified C implementation which is vital for, e.g., field deployment on networks of micro-controllers (IoT) or embedded devices. This paper fills this verification gap: the JIT correctness proof is conducted over an abstract specification in Coq and then propagated down to a concrete C implementation of the JIT compiler.

*End-to-end Verification.* There are various solutions for extracting executable C code from high-level programs, but most of them are not compatible with our goal: *i.e.*, a verified JIT C implementation running the real-time OS kernel deployed on IoT devices. Some of them are unverified, *e.g.*, KaRaMeL [30] (from F\* to C) and Codegen [33] (from Gallina to C). Some require a garbage collector, *e.g.*, CertiCoq [1] and Euf [24] (from Gallina to C) or CakeML (from Standard ML to binary). The Cogent framework [31] (from Cogent to Isabelle/HOL and C) is verified but depends on calls to foreign C functions to perform loops, and Ruplicola [29] (from Gallina to bedrock2, a C-like language) has only been tested for small algorithms. The end-to-end refinement method proposed in the paper instead reuses the existing verification workflow and proof efforts of CertiBPF and CompCert to provide the first, fully verified and resource-efficient, hybrid virtual machine, HAVM.

## 10 Conclusion

As use-cases for eBPF virtual machines multiply, their applicability encompasses not only PCs and servers but also low-power devices based on microcontrollers. In this context, we presented an end-to-end design, proof, and synthesis methodology to bring the first BPF Just-in-Time compiler tailored to the hardware and resources constraints of popular low-power microcontroller architectures, proven correct end-to-end using the proof assistant Coq. We combined our proven JIT implementation with the BPF interpreter provided in the RIOT operating system to create a hybrid virtual machine, HAVM: a defensive, kernel-privileged service capable of accelerating numerical tasks at runtime using partial JIT compilation. Benchmarking HAVM in practice on Cortex-M microcontrollers show that HAVM achieves significant execution speed improvements compared to prior works.

We are carrying on designing a fully verified JIT-all compiler for RIOT that translates all rBPF instructions into binary. One of the most challenging aspect of this project is to link and embed tailor-optimized *check\_mem* algorithms into *jit*ed code (using loop unrolling, partial evaluation).

## References

1. Anand, A., Appel, A.W., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Bélanger, O.S., Sozeau, M., Weaver, M.Z.: Certicoq : A verified compiler for Coq. In: CoqPL (2017)
2. Baccelli, E., Gündoğan, C., Hahm, O., Kietzmann, P., Lenders, M.S., Petersen, H., Schleiser, K., Schmidt, T.C., Wählisch, M.: RIOT: An open source operating system for low-end embedded devices in the IoT. *IoT-J* **5**(6), 4428–4440 (2018)
3. Barrière, A., Blazy, S., Flückiger, O., Pichardie, D., Vitek, J.: Formally verified speculation and deoptimization in a JIT compiler. *Proceedings of the ACM on Programming Languages* **5**(POPL), 26 (Jan 2021). <https://doi.org/10.1145/3434327>

4. Barrière, A., Blazy, S., Pichardie, D.: Formally verified native code generation in an effectful jit: Turning the compcert backend into a formally verified jit compiler. *Proc. ACM Program. Lang.* **7**(POPL) (jan 2023), <https://doi.org/10.1145/3571202>
5. Bertot, Y., Castéran, P.: *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer (2013)
6. Bratman, H.: A alternate form of the “uncol diagram”. *Commun. ACM* **4**(3), 142 (mar 1961), <https://doi.org/10.1145/366199.366249>
7. Desharnais, M., Brunthaler, S.: Towards Efficient and Verified Virtual Machines for Dynamic Languages. In: *CPP*. p. 61–75. ACM (2021)
8. Dunchev, C., Guidi, F., Coen, C.S., Tassi, E.: ELPI: fast, embeddable,  $\lambda$ prolog interpreter. In: *LPAR. Lecture Notes in Computer Science*, vol. 9450, pp. 460–468. Springer (2015)
9. Fleming, M.: A Thorough Introduction to eBPF. *Linux Weekly News* (2017)
10. Geffen, J.V., Nelson, L., Dillig, I., Wang, X., Torlak, E.: Synthesizing JIT compilers for in-kernel dsls. In: *CAV. LNCS*, vol. 12225, pp. 564–586. Springer (2020)
11. 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: *OSDI*. pp. 653–669. USENIX (2016)
12. Hutchings, B.: Executing arbitrary code within the kernel (2021), <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-29154>
13. Jomaa, N., Torrini, P., Nowak, D., Grimaud, G., Hym, S.: Proof-oriented design of a separation kernel with minimal trusted computing base. In: *AVOCS*. vol. 76. Electronic Communications of the EASST (2018)
14. Keshri, R.: confidentiality problem (2021), <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-20320>
15. Klein, G., Norrish, M., Sewell, T., Tuch, H., Winwood, S., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R.: seL4: formal verification of an OS kernel. In: *SOSP*. p. 207. ACM Press (2009)
16. Krysiuk, P.: Linux kernel vulnerability in netapp products (2021), <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-38300>
17. Leroy, X.: Formal verification of a realistic compiler. *Communications of the ACM* **52**(7), 107–115 (2009)
18. Leroy, X., Appel, A.W., Blazy, S., Stewart, G.: The CompCert Memory Model, Version 2. Research Report RR-7987, INRIA (2012)
19. Leroy, X., Blazy, S.: Formal verification of a C-like memory model and its uses for verifying program transformations. *JAR* **41**(1), 1–31 (2008)
20. Letouzey, P.: A new extraction for Coq. In: Geuvers, H., Wiedijk, F. (eds.) *Types for Proofs and Programs, Second International Workshop, TYPES 2002*, Berg en Dal, The Netherlands, April 24–28, 2002, Selected Papers. *Lecture Notes in Computer Science*, vol. 2646, pp. 200–219. Springer (2002)
21. Lochbihler, A.: *A Machine-Checked, Type-Safe Model of Java Concurrency : Language, Virtual Machine, Memory Model, and Verified Compiler*. Ph.D. thesis, Karlsruhe Institute of Technology (2012)
22. McCanne, S., Jacobson, V.: The BSD Packet Filter: A New Architecture for User-level Packet Capture. In: *Usenix Winter Conference*. vol. 46, pp. 259–270. USENIX (1993)
23. Mogul, J., Rashid, R., Accetta, M.: The packer filter: An efficient mechanism for user-level network code. In: *SOSP*. p. 39–51. ACM (1987)
24. Mullen, E., Pernsteiner, S., Wilcox, J.R., Tatlock, Z., Grossman, D.:  $\mathcal{C}$ uF: minimizing the Coq extraction TCB. In: *CPP*. pp. 172–185. ACM (2018)

25. Myreen, M.O.: Verified just-in-time compiler on x86. In: POPL. pp. 107–118. ACM (2010)
26. Myreen, M.O., Owens, S.: Proof-producing synthesis of ml from higher-order logic. In: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming. p. 115–126. ICFP '12, Association for Computing Machinery, New York, NY, USA (2012), <https://doi.org/10.1145/2364527.2364545>
27. Nelson, L., Geffen, J.V., Torlak, E., Wang, X.: Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the linux kernel. In: OSDI. pp. 41–61. USENIX (2020)
28. Nelson, L., Geffen, J.V., Torlak, E., Wang, X.: Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the linux kernel. In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). pp. 41–61. USENIX Association, USA (Nov 2020)
29. Pit-Claudel, C., Philipoom, J., Jamner, D., Erbsen, A., Chlipala, A.: Relational compilation for performance-critical applications. In: PLDI. ACM (2022)
30. Protzenko, J., Zinzindohoué, J.K., Rastogi, A., Ramanandaro, T., Wang, P., Zanella-Béguelin, S., Delignat-Lavaud, A., Hritcu, C., Bhargavan, K., Fournet, C., Swamy, N.: Verified low-level programming embedded in F\*. PACMPL **1**(ICFP), 17:1–17:29 (Sep 2017). <https://doi.org/10.1145/3110261>
31. Rizkallah, C., Lim, J., Nagashima, Y., Sewell, T., Chen, Z., O'Connor, L., Murray, T.C., Keller, G., Klein, G.: A Framework for the Automatic Formal Verification of Refinement from Cogent to C. In: ITP. LNCS, vol. 9807, pp. 323–340. Springer (2016)
32. Sozeau, M., Boulier, S., Forster, Y., Tabareau, N., Winterhalter, T.: Coq coq correct! verification of type checking and erasure for coq, in coq. Proc. ACM Program. Lang. **4**(POPL), 8:1–8:28 (2020), <https://doi.org/10.1145/3371076>
33. Tanaka, A.: Coq to C translation with partial evaluation. In: PEPM@POPL. pp. 14–31. ACM (2021)
34. Van Geffen, J., Nelson, L., Dillig, I., Wang, X., Torlak, E.: Synthesizing jit compilers for in-kernel dsls. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification. pp. 564–586. Springer International Publishing, Cham (2020)
35. Wang, X., Lazar, D., Zeldovich, N., Chlipala, A., Tatlock, Z.: Jitk: A trustworthy In-Kernel interpreter infrastructure. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). pp. 33–47. USENIX Association, Broomfield, CO (Oct 2014)
36. YUAN, S.: Certrbpf-jit: A verified jit for riot-os rbpf (2024), <https://gitlab.inria.fr/x-SYuan/certrbpf-jit/-/tree/CAV24-AE>
37. Yuan, S., Besson, F., Talpin, J.P., Hym, S., Zandberg, K., Baccelli, E.: End-to-end mechanized proof of an ebpf virtual machine for micro-controllers. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification. pp. 293–316. Springer International Publishing, Cham (2022)
38. Zandberg, K., Baccelli, E.: Minimal virtual machines on IoT microcontrollers: The case of Berkeley Packet Filters with rBPF. In: PEMWN. pp. 1–6. IEEE (2020)
39. Zandberg, K., Baccelli, E., Yuan, S., Besson, F., Talpin, J.P.: Femto-containers: Lightweight virtualization and fault isolation for small software functions on low-power iot microcontrollers. In: Proceedings of the 23rd ACM/IFIP International Middleware Conference. p. 161–173. Middleware '22, Association for Computing Machinery, New York, NY, USA (2022), <https://doi.org/10.1145/3528535.3565242>
40. Zhang, X., Li, Y., Sun, M.: Towards a formally verified EVM in production environment. In: COORDINATION. LNCS, vol. 12134, pp. 341–349. Springer (2020)