End-to-end Mechanized Proof of an eBPF Virtual Machine for Micro-controllers

Shenghao Yuan[1 [0000−0002−8467−5827]], Frédéric Besson[1 [0000−0001−6815−0652]], Jean-Pierre Talpin[1 [0000−0002−0556−4265]], Samuel Hym[2], Koen Zandberg[1], and Emmanuel Baccelli[1,3 [0000−0001−6239−9983]]

[1] Inria (firstname.lastname@inria.fr)
[2] Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France (samuel.hym@univ-lille.fr)
[3] Freie Universität Berlin, Germany

Abstract. RIOT is a micro-kernel dedicated to IoT applications that adopts eBPF (extended Berkeley Packet Filters) to implement so-called femto-containers. As micro-controllers rarely feature hardware memory protection, the isolation of eBPF virtual machines (VM) is critical to ensure system integrity against potentially malicious programs. This paper shows how to directly derive, within the Coq proof assistant, the verified C implementation of an eBPF virtual machine from a Gallina specification. Leveraging the formal semantics of the CompCert C compiler, we obtain an end-to-end theorem stating that the C code of our VM inherits the safety and security properties of the Gallina specification. Our refinement methodology ensures that the isolation property of the specification holds in the verified C implementation. Preliminary experiments demonstrate satisfying performance.

Keywords: mechanized proof · virtual machines · fault isolation.

1 Introduction

Hardware-enforced memory isolation (*e.g.,* Trustzone, Sanctum [6], Sancus [30]) is often not available on micro-controller units (MCU) which usually trade coarse-grain isolation for price and performance. To mitigate development variability and cost, common practices for MCU operating system design (RIOT [3], FreeRTOS, TinyOS, Fushia, and others [14]) advise to run all the device's code stack in a shared memory space, which can only be reasonably safe if that code can be trusted. While standard in safety-critical system design, such a trust requirement is oftentimes unsuitable for networked MCUs, where the extensibility of the OS kernel at runtime is an essential functionality. When system reconfiguration does not affect the entire network (via, *e.g.,* leader election), extensibility can easily be provided offline, by employing library OSs or unikernels [24], to reconfigure network endpoints independently (*e.g.,* cloud apps). Otherwise, the best solution is to load and execute system extensions (configurations, protocols,

firewalls, etc) as assembly-level Wasm [13] or Berkeley Packet Filters [25] scripts using an interpreter or a Just-In-Time (JIT) compiler on the target device.

Femto-containers. RIOT adopts the extended Berkeley Packet Filters (eBPF) and tailors it to resource-constrained MCUs by implementing so-called femto-containers: tiny virtual machine instances interpreting eBPF scripts. Compared to more expressive languages, like Wasm, experiments show that RIOT's eBPF implementation, rBPF, requires less memory [39]. The Linux kernel features an eBPF JIT compiler whose security depends on a sophisticated online verifier [29]. As an MCU architecture cannot host such a large verifier, executing JIT code would imply delegation of trust to a third-party, offline, verifier. The alternative is to rely on a defensive VM. Though a VM may be slower than a JIT, it can run untrusted, erroneous, adversary code in an open, and possibly hostile environment, and still isolate faults to protect its host's integrity.

Approach & Goals. This paper investigates an approach that trades high performance on low-power devices for defensive programming and low memory footprint. Our primary goal is to prevent faults that could compromise host devices and, by extension, force networked devices to reboot and resynchronize (*i.e.,* fault tolerance protocols). To maximize trust in the implementation of rBPF, our refinement methodology allows the verified extraction of C code directly from its mechanically proved definition in Gallina, the functional language embedded in the Coq proof assistant [4].

Method. To mechanically prove the correctness of an interpreter, a conventional approach consists in defining the reference semantics in a proof assistant and in showing that an executable optimized interpreter produces the same output. In this paper, our goal is to verify the interpreter of the virtual rBPF instruction set, implemented with the system programming language C. To this aim, we introduce a direct, end-to-end, validation workflow. The semantics of the source instruction set is directly defined by monadic functional terms in our proof assistant. We prove that this semantics enforces safety and security requirements regarding memory isolation and control-flow integrity. Then, C code is automatically derived from these monadic functional terms to implement the expected virtual machine. We prove that the extracted C code has the same stateful behavior as the monadic specification. Our method uses a monadic subset of Gallina of sufficient expressiveness to specify rBPF's semantics, supports the verified extraction of equivalent Clight [20] code, while provably implementing all required defensive runtime checks.

Plan. The rest of the paper is organized as follows. Sec. 2 states our contributions. Sec. 3 provides background on BPF and its variants, CompCert and the ∂x code extraction tool. Sec. 4 presents our workflow to formally refine monadic Gallina programs into C programs. Sec. 5 defines the proof model of our virtual machine: its semantics, consistency and isolation theorems. Sec. 6 refines the

proof model of our femto-container into a synthesis model ready for code generation with CompCert. Sec. 7 proves the refinement between the synthesis and implementation models. Sec. 8 introduces our verified verifier which establishes the invariants needed by the VM. Sec. 9 case studies the performance of our generated VM implementation with respect to off-the-shelf RIOT femto-containers. Sec. 10 presents related works and Sec. 11 concludes.

2 Contributions

Implementing a fault-isolating virtual machine for MCUs faces two major challenges. One is to embed the VM inside the MCU's micro-kernel and, hence, to minimize its code size and execution environment. A second challenge is to minimize the verification gap between its proof model and the running code. We address these challenges and present the first end-to-end verification and synthesis of a full-scale, real-world, virtual machine for the BPF instruction set family: CertrBPF, an interpreter tailored to the hardware and resources constraints of MCU architectures running the RIOT operating system. CertrBPF employs a workflow of proof-oriented programming using the functional language Gallina embedded in the proof assistant Coq. The verified refinement and extraction of an executable C program is performed directly from its proof model. We report the successful integration of CertrBPF into the open source IoT operating system RIOT and the evaluation of its performance against micro-benchmarks.

A certified rBPF interpreter. CertrBPF is a verified model and implementation of rBPF in Coq. We formalize the syntax and semantics of all rBPF instructions, implement a formal model of its interpreter (femto-container), complete the proof of critical properties of our model, and extract and verify CompCert C code from this formalization. This method allows us to obtain a fully verified virtual machine. Not only is the Gallina specification of the VM proved kernel- and memory-isolated using the proof assistant, but the direct interpretation of its intended semantics as CompCert C code is, itself, verified correct. This yields a fully verified binary program of maximum security and minimal memory footprint and reduced the Trusted Computing Base (TCB): CertrBPF, a memory-efficient kernel-level virtual machine that isolates runtime software faults using defensive code and does not necessitate offline verification.

End-to-end proof workflow. An obvious choice is to use the existing Coq extraction mechanism to compile the Gallina model into OCaml. The downside of this approach is that Coq extraction has to be trusted. Moreover the OCaml runtime needs to be trimmed down to fit space requirements of our target architecture and also becomes part of the TCB. Our ambition is instead to minimize the verification gap and provide an end-to-end security proof linking our Gallina model to, bare-metal, extracted C code. Our intended TCB is hence restricted to the Coq type-checker, the C semantics of the CompCert compiler and a pretty-printer for the generated C Abstract Syntax Tree (AST).

To reach this goal, our starting point is a model of the rBPF semantics written in Gallina.We use this proof model to certify that all the memory accesses are valid and isolated to dedicated memory areas, thus ensuring isolation. From this proof model, we then derive a synthesis model of which we extract an executable version in Clight, that we finally prove to perform the same state transitions.

Systems Integration & Micro-benchmarks. We integrate CertrBPF as a drop-in replacement of the current, non-verified, rBPF interpreter in the RIOT operating system. We then comparatively evaluate the performance of CertrBPF integrated in RIOT, running on various 32-bit micro-controller architectures. Our benchmarks demonstrate that, in practice, CertrBPF not just gains security, but reduces memory footprint as well as execution time.

3 Background

This section describes essential features of rBPF, of the CompCert compiler, and of the ∂x code generation tool, that are required by our refinement methodology.

BPF, eBPF and rBPF. Originally, the purpose of Berkeley Packet Filters [25] (BPF) was network packet filtering. The Linux community extended it to provide ways to run custom in-kernel VM code, hooked into various subsystems, for varieties of purposes beyond packet filtering [10]. eBPF was then ported to micro-controllers, yielding RIOT's specification: rBPF [38]. Just as eBPF, rBPF is designed as a 64-bit register-based VM, using fixed-size 64-bit instructions and a reduced instruction set architecture. rBPF uses a fixed-size stack (512 bytes) and defines no heap interaction, which limits the VM memory overhead in RAM. The rBPF specification, however, does not define special registers or interrupts for flow control, nor support virtual memory: the host device's memory is accessed directly and only guarded using permissions.

The CompCert Verified Compiler. CompCert [18] is a C compiler that is both programmed and proved correct using the Coq proof assistant. The compiler is structured into passes using several intermediate languages. Each intermediate language is equipped with a formal semantics and each pass is proved to preserve the observational behavior of programs.

The Clight Intermediate Language. Clight [20] is a pivotal language which condenses the essential features of C using a minimal syntax. The Verified Software Toolchain (VST) [2] verifies C programs at the Clight level that are obtained by the CLIGHTGEN tool. Though we do not reuse the proof infrastructure of VST, we are reusing CLIGHTGEN in order to get a Clight syntax from a C program.

CompCert Values and Memory Model [20,19]. The memory model and the representation of values are shared across all the intermediate languages of CompCert. The set of values *val* is defined as follows:

$$val \ni v ::= Vint(i) \mid Vlong(i) \mid Vptr(b, o) \mid Vundef \mid \ldots$$

A value $v \in val$ can be a 32-bit integer $Vint(i)$; a 64-bit integer $Vlong(i)$, a pointer $Vptr(b, o)$ consisting of a block identifier b and an offset o, or the undefined value $Vundef$. The undefined value $Vundef$ represents an unspecified value and is not, strictly speaking, an undefined behavior. Yet, as most of the C operators are strict in $Vundef$, and because branching over $Vundef$ or dereferencing $Vundef$ are undefined behaviors, our proofs will ensure the absence of $Vundef$. CompCert values also include floating-point numbers; they play no role in the current development. CompCert's memory consists of a collection of separate arrays. Each array has a fixed size determined at allocation time and is identified by an uninterpreted block $b \in block$. The memory provides an API for loading values from memory and storing values in memory. Operations are parameterised by a memory chunk k which specifies how many bytes should be written or read and how to interpret bytes as a value $v \in val$.

For instance, the memory chunk *Mint32* specifies a 32-bit value and *Mint64* a 64-bit value. The function *load* k m b o takes a memory chunk k, a memory m, a block b and an offset o. Upon success, it returns a value v obtained from the memory by reading bytes from the block b starting at index o. Similarly, the function *store* k m b o v takes a memory chunk k, a memory m, a block b, an offset o and a value v. Upon success, it returns an updated memory m' which is identical to m except that the block b contains the value v encoded into bytes according to the chunk k starting at offset o. The isolation properties offered by CompCert memory regions are worth mentioning: *load* and *store* operations fail (return $None$) for invalid offsets o and invalid permissions.

The ∂x tool. ∂x emerged from the toolchain used to design and verify the Pip proto-kernel [15]. Its aim was to allow writing most of Pip's source code in Gallina in a style as close to C as possible. ∂x extracts C code from a Gallina source program in the form of a CompCert C AST. The goal of ∂x is to provide C programmers with readily reviewable code and thus avoid misunderstanding between those working on C/assembly modules (that access hardware) and those working on Coq modules (the code and proofs). To achieve this, ∂x handles a C-like subset of Gallina. The functions that are to be converted to C rely on a monad to represent the side effects of the computation, such as modifications to the CPU state. Yet ∂x does not mandate a particular monad for code extraction.

∂x's workflow. ∂x proceeds in two steps. First, given a list of Gallina functions, or whole modules, it generates an intermediate representation (IR) for the subset of Gallina it can handle. The second step is to translate this IR into a CompCert C AST. Since Coq has no built-in reflection mechanism, the first step is written in Elpi [8], using the Coq-Elpi plugin [37]. That step can also process external functions (appearing as `extern` in the extracted C code) to support separate compilation with CompCert. In order to obtain an actual C file, ∂x also provides a small OCaml function that binds the extracted C AST to CompCert's C pretty-printer. Even though the ∂x language is a small subset of Gallina, it inherits much expressivity from the use of Coq types to manipulate values. For example, we can use bounded integers (*i.e.,* the dependent pair of an integer with the

proof that it is within some given range), that can be faithfully and efficiently represented as a single `int` in C. To this end, ∂x expects a configuration mapping Coq types to C.

∂x memory management. A major design choice in the C-like subset of Gallina used by ∂x is memory management: its generated code executes without garbage collection. This affects the Coq types that can actually be used in ∂x: recursive inductive types, such as lists, cannot automatically be converted. However, this Gallina subset is particularly relevant to programs in which one wants to precisely control memory management and decide how to represent data structures in memory. This is typically the case of an operating system or, in our case, the rBPF virtual machine.

4 A Workflow for End-to-End Verification in Coq

This section gives an overview of our methodology to derive a verified C implementation from a Gallina specification. In the following sections, the methodology will be instantiated to derive the C implementation of a fault-isolating rBPF virtual machine and its verifier. Our approach provides an end-to-end correctness proof, within the Coq proof assistant, that reduces the hurdle of reasoning directly over the C code.

As shown in Fig. 1, the original rBPF C implementation is first formalized by a proof model in Gallina, and the verification of expected properties (*e.g.,* safety) is performed within the Coq proof assistant. This specification is then refined into an optimized (and equivalent) synthesis model ready for C-code extraction.

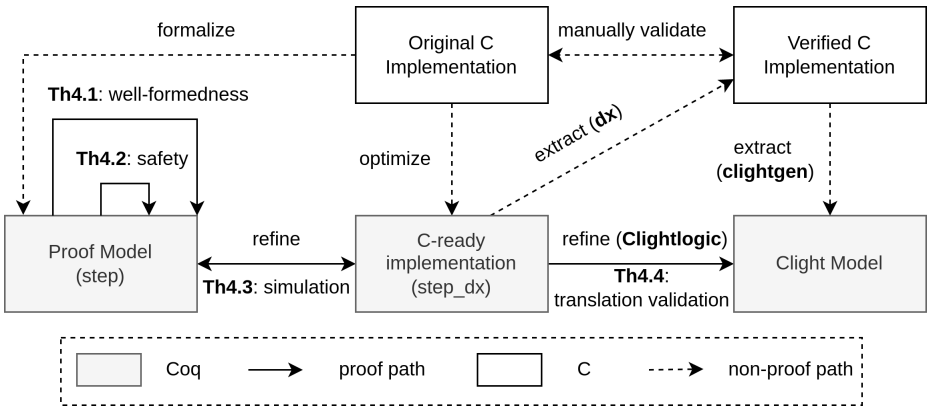

Fig. 1: End-to-end verification and synthesis workflow

The refinement & optimization principle employed by our method consists of deriving a C-ready implementation, in Gallina, that is as close as possible

to the expected target C code. This principle allows to i) prove optimizations correct, ii) improve the performance of the extracted code and, iii) facilitate review and validation of extracted code with the system designers. From the C-ready Gallina implementation, we leverage ∂x to automatically generate C code and verify it: i) the generated C code is first parsed as a CompCert Clight model by the CLIGHTGEN tool of VST and ii) it is proved to refine the source Gallina model in Coq using translation validation. Because ∂x generates C code in a syntax-directed manner, a minimal *Clightlogic* is designed to facilitate the refinement proof. The rest of the section explains these different steps in details.

Proof-oriented specification. Our specification takes the form of an executable abstract machine in monadic form. It uses the standard option-state monad M.

$$
\begin{aligned}
&M\ a\ state := state \to \mathbf{option}(a \times state)\\
&returnM : a \to M\ a\ state := \lambda a.\lambda st.\mathbf{Some}(a, st)\\
&bindM : M\ a\ state \to (a \to M\ b\ state) \to M\ b\ state :=\\
&\qquad \lambda A.f.\lambda s.\mathbf{match}\ A\ s\ \mathbf{with}\ |\ \mathbf{None} \Rightarrow \mathbf{None}\ |\ \mathbf{Some}(x, s') \Rightarrow (f\ x)\ s'
\end{aligned}
$$

In the remainder, we write $\emptyset$ for **None** and $\lfloor x \rfloor$ for **Some** x.

The monad threads the state along computations to model its in-place update. The safety property of the machine is implemented as an inline monitor: any violation leads to an unrecoverable error, *i.e.,* the unique error represented by $\emptyset$. One step of the machine has the following signature:

$$step : M\ r\ state$$

where r is the type of the result. The *step* function implements a defensive semantics, checking the absence of error, dynamically. For our rBPF interpreter (see Sec. 5), the absence of error ensures that the rBPF code only performs valid instructions. In particular, all memory accesses are restricted to a sandbox specified as a list of memory regions. Function *step* is part of the TCB and, therefore, a mis-specification could result, after refinement, in an invalid computation. The purpose of the error state is to specify state transitions that would escape the scope of the safety property and, therefore, shall never be reachable from a well-formed state $st \in wf \subseteq \mathcal{P}(state)$. We require well-formedness to be an inductive property of the *step* function.

Theorem 1 (Well-formedness). *The step function preserves well-formedness.*

$$\forall st, st', r.\ st \in wf \wedge step\ st = \lfloor (r, st') \rfloor \Rightarrow st' \in wf$$

We also require that well-formedness is a sufficient condition to prevent the absence of error and, therefore, the safety of computations.

Theorem 2 (Safety). *The step function is safe,* i.e., *a well-formed state never leads to an error.*

$$\forall st.\ st \in wf \Rightarrow step\ st \neq \emptyset$$

C-ready implementation. Our methodology consists in refining the *step* function into an interpreter $step_{\partial x}$ complying with the requirements of ∂x. As ∂x performs syntax-directed code generation, the efficiency of the extracted code crucially depends on $step_{\partial x}$. In order to preserve the absence of errors, we need a simulation relation between the *step* and $step_{\partial x}$ functions. A direct consequence of the simulation thoerem is that $step_{\partial x}$ never raises an error.

Theorem 3 (Simulation). *Given simulation relations $Rs \subseteq state \times state'$ and $Rr \subseteq r \times r'$, the function $step_{\partial x}$ simulates the function step.*

$$\forall s_1, s_1', s_2, r.(s_1, s_2) \in Rs \wedge step\ s_1 = \lfloor r, s_1' \rfloor \Rightarrow \exists s_2', r'. \bigwedge \begin{cases} step_{\partial x}\ s_2 = \lfloor r', s_2' \rfloor \\ (s_1', s_2') \in Rs \\ (r, r') \in Rr \end{cases}$$

Translation Validation of C code. The next stage consists in refining the $step_{\partial x}$ function into a Clight program by relying on ∂x to get a C program and on the CLIGHTGEN tool to get a Clight $step_C$ program (see Sec. 6). As this pass is not trusted, we require the following translation validation theorem.

Theorem 4 (Translation Validation). *Given a simulation relation $Rs \subseteq state' \times val \times mem$ and a relation $Rr \subseteq res \times val$, the Clight code $step_C$ refines the function $step_{\partial x}$:*

$$\begin{aligned} &\forall r, s, s', v, k, m.(s, v, m) \in Rs \Rightarrow step_{\partial x}\ s = \lfloor (r, s') \rfloor \Rightarrow \\ &\quad \exists m', r'. Callstate(step_C, [v], k, m) \rightarrow^{*t} ReturnState(r', call\_cont(k), m') \wedge \\ &\qquad (s', v, m') \in Rs \wedge (r, r') \in Rr \end{aligned}$$

Theorem 4 states that, if $step_{\partial x}\ s$ runs without error and returns a result (r, s'), then, the Clight function $step_C$ successfully runs with argument v and, after a finite number of execution steps, returns a result r' and a memory m' that preserve the refinement relations. In our encoding, the unique argument v is a pointer to the memory allocated region refining the interpreter state and k represents the continuation of the computation. A corollary of Theorem 4 is that the Clight code $step_C$ is free of undefined behaviors. In particular, all memory accesses are valid. As the memory model does not allow to forge pointers, this yields a strong isolation property. In the remainder of this paper, for our rBPF virtual machine, we prove all the aforementioned properties within the Coq proof assistant.

5 A Proof-Oriented Virtual Machine Model

For our proof model, we define an explicit syntax for rBPF. We also define the state of the interpreter and semantic functions, in particular those implementing dynamic security checks. The rBPF instruction set, Figure 2, features binary arithmetic and logic operations, negation, (un)conditional jumps relative to an offset, operations to load/store values from/to registers/memory, function calls, and termination. There are eleven 64-bit registers $\{R0, \ldots, R10\}$; an immediate is 32-bit wide and an offset is 16-bit wide.

$$
\begin{array}{lll}
(\mathit{Operands}) & \mathit{dst}, \mathit{reg} \in \mathit{registers}\ ,\ \mathit{src} \in \mathit{registers} \cup \mathit{immediate} \\
& \mathit{imm} \in \mathit{immediate}\ ,\ \mathit{ofs} \in \mathit{offset} \\
(\mathit{Chunk}) & \mathit{chk} ::= \mathit{byte} \mid \mathit{halfword} \mid \mathit{word} \mid \mathit{doublewords} \\
(\mathit{Operators}) & \mathit{op} ::= \mathit{add} \mid \mathit{sub} \mid \mathit{mul} \mid \mathit{div} \mid \mathit{and} \mid \mathit{or} \mid \\
& \quad \mathit{lsh} \mid \mathit{rsh} \mid \mathit{mod} \mid \mathit{xor} \mid \mathit{mov} \mid \mathit{arsh} \\
& \mathit{cmp} ::= \mathit{eq} \mid \mathit{neq} \mid \mathit{lt} \mid \mathit{gt} \mid \mathit{le} \mid \mathit{ge} \mid \mathit{set} \mid \mathit{slt} \mid \mathit{sgt} \mid \mathit{sle} \mid \mathit{sge} \\
(\mathit{Instruction}) & \mathit{ins} ::= \texttt{Exit} \mid \texttt{Call}\ \mathit{imm} \mid \texttt{Neg}\ \mathit{dst} \mid \texttt{Ja}\ \mathit{ofs} \mid \texttt{Jump}\ \mathit{cmp}\ \mathit{dst}\ \mathit{src}\ \mathit{ofs} \\
& \quad \mid \texttt{Alu}\ \mathit{op}\ \mathit{dst}\ \mathit{src} \mid \texttt{Load}\ \mathit{chk}\ \mathit{dst}\ \mathit{reg}\ \mathit{ofs} \mid \texttt{Store}\ \mathit{chk}\ \mathit{dst}\ \mathit{src}\ \mathit{ofs}
\end{array}
$$

Fig. 2: Core syntax of rBPF instruction set

Machine state. A semantic state st is a tuple $\langle I, L, R, F, M, MRs \rangle$ consisting of a sequence of instructions I, the current location L, registers R, an interpreter flag F, a memory M and a specification of available memory regions MRs. The flag F characterizes the state of the rBPF interpreter. It may be i) a normal state, written F_n; ii) a final state, written F_t; iii) or an error state, written F_e. An error state $f \in F_e$ means that the defensive checks of the interpreter have detected that an invalid behavior is about to occur.

A memory region $mr = \langle start, size, p, ptr \rangle \in MRs$ associates a permission $p \in \{Readable, Writable\}$ to the address range $[start, start + size)$. We make the link between concrete physical addresses and the CompCert memory model using the pointer *ptr (= Vptr b 0)* where the block b is the abstract representation of the address *start*. We write $I(L)$ for the instruction located at the program counter L. $R[r]$ retrieves the value of the register r in the register map R. Functions *alu* and *cmp* reuse the CompCert's operators over the *val* type. The *alu* function returns $\emptyset$ if an error occurs, *e.g.,* division by zero. Functions *load* and *store* are those of CompCert's memory model (see Sec. 3).

$$
\begin{array}{ll}
\textbf{alu} : \mathit{op} \to \mathit{val} \to \mathit{val} \to \mathit{option\ val} & \textbf{cmp} : \mathit{cmp} \to \mathit{val} \to \mathit{val} \to \mathit{bool} \\
\textbf{load} : \mathit{chk} \to \mathit{mem} \to \mathit{block} \to Z \to \mathit{option\ val} & \\
\textbf{store} : \mathit{chk} \to \mathit{mem} \to \mathit{block} \to Z \to \mathit{val} \to \mathit{option\ mem} &
\end{array}
$$

Dynamic checks. Function *check_alu* dynamically checks the validity of an arithmetic to avoid *div-by-zero* and *undefined-shift* errors. For division instructions, *check_alu* mandates the second argument to be non-zero. For arithmetic and logical shift instructions, the second argument has to be below $n \in \{32, 64\}$ depending on whether the ALU instruction operates on 32 or 64 bit operands. For simplicity, the paper only considers 64-bit ALU instructions but CertrBPF also has the 32-bit variants.

$$
\mathit{check\_alu}(\mathit{op}, v) \stackrel{\text{def}}{=} \begin{cases} v \neq 0 & \mathit{if}\ \mathit{op} \in \{\mathit{div}, \mathit{mod}\} \\ 0 \leq v < n & \mathit{if}\ \mathit{op} \in \{\mathit{lsh}, \mathit{rsh}, \mathit{arsh}\} \\ \mathit{true} & \mathit{otherwise} \end{cases}
$$

Function *check_mem* returns a valid pointer ($Vptr\ b\ ofs$) if there exists a unique memory region mr in MRs such that i) the permission $mr.perm$ is at least *Readable* for `Load` and *Writable* for `Store`, *i.e., mr.perm* $\geq p$; ii) the offset *ofs* is aligned, *i.e.,* $ofs \% Z(chk) = 0$; iii) in bounds, *i.e., ofs* $\leq max\_unsigned - Z(chk)$,

iv) and the interval $[ofs, hi\_ofs)$ is in the range of mr. Otherwise, *check_mem* returns the null pointer *Vnullptr*. The function *Z(chk)* maps memory chunks *byte*, *halfword*, *word* and *double* to 1, 2, 4, and 8, respectively.

$$
\begin{array}{l}
check\_mem(p, chk, addr, MRs) \stackrel{\text{def}}{=} \textbf{if } \exists!\ mr \in MRs,\ b. \\
\quad \textbf{let } ofs = addr - mr.start \textbf{ and } hi\_ofs = ofs + Z(chk) \textbf{ in} \\
\quad\quad (mr.ptr == Vptr\ b\ 0)\ \wedge\ (mr.perm \geq p)\ \wedge\ (ofs \% Z(chk) == 0) \wedge \\
\quad\quad\quad (ofs \leq max\_signed - Z(chk))\ \wedge\ (0 \leq ofs \wedge hi\_ofs < mr.size)) \\
\quad\quad \textbf{then } Vptr\ b\ ofs \textbf{ else } Vnullptr
\end{array}
$$

Semantics. Functions *interp* and *sem* formalize the implementation of our proof model M_p in the Coq proof assistant by defining a monadic interpreter of rBPF. The top-level recursion *interp* processes a (monotonically decreasing) $fuel$ argument and a state s. The function *sem* processes individual instructions $I(L_{pc})$. MRs and I are read-only. During normal execution, the flag remains F_n. If the flag turns to F_t or F_e while processing an instruction, execution stops. For instance, if $fuel$ reaches zero, the flag turns to F_e. We write $s.F$ for the value of field F in record s and $s\{F = v\}$ updates it to v.

```
interp = λfuel s. if fuel == 0 then ⌊((), s{F=F_e})⌋ else
    match sem s with
    | ⌊((), t)⌋ => if t.F≠ F_n then ⌊((), t)⌋
                   else interp (fuel-1) t{L = t.L+1}
    | ∅ => ∅

sem = λs. match s.I(s.L) with
| Exit => ⌊((), s{F = F_t})⌋
| Call imm => let f_ptr = bpf_get_call imm in
                if f_ptr == Vnullptr then ⌊((), s{F = F_e})⌋
                else ⌊((), s{R0 = exec_function f_ptr})⌋
| Ja ofs => ⌊((), s{L = s.L+ofs})⌋
| Jump c dst ofs => if cmp(c, s.R[dst], s.R[src])
                      then ⌊((), s{L = s.L+ofs})⌋  else ⌊((), s)⌋
| Neg dst => ⌊((), s{R[dst]= ¬ s.R[dst]})⌋
| Alu op dst src => if check_alu(op, s.R[src]) then
                    match alu(op, s.R[dst], s.R[src]) with
                    | ⌊v⌋ => ⌊((), s{R[dst] = v})⌋ | ∅ => ∅
                    else ⌊((), s{F = F_e})⌋
| Load chk dst reg ofs =>
  match check_mem(Readable, chk, s.R[reg]+ofs, s.MRs) with
  | Vptr b ofs => match load(chk, s.M, b, ofs) with
                  | ⌊v⌋ => ⌊((), s{R[dst] = v})⌋ | ∅ => ∅
  | _ => ⌊((), s{F = F_e})⌋
| Store chk dst src ofs =>
  match check_mem(Writable, chk, s.R[dst]+ofs, s.MRs) with
  | Vptr b ofs => match store(chk, s.M, b, ofs, S.R[src]) with
                  | ⌊N⌋ => ⌊((), s{M = N})⌋  | ∅ => ∅
  | _ => ⌊((), s{F = F_e})⌋
| _ => ⌊((), s{F = F_e})⌋
```

Result $\emptyset$ marks transitions to crash states that are proved unreachable given our carefully crafted definitions of the `check_alu` and `check_mem` functions. Note that the interpreter *interp* does not check the range of branching offsets (*i.e.,* `0 <=s.L< length(s.I)`) and register-out-of-bounds. This properties are statically verified, once and for all, by the verifier of Sec. 8.

Exit terminates the program with flag F_t. The *Call* instruction selects (using *bpf_get_call*) the trusted system API service designated by an immediate number *imm*. It then calls the chosen service if available (*i.e.,* not a null pointer). Unconditional jump *Ja* increments the *pc* by *ofs* and a conditional *Jump* does so when $cmp(c, src, dest)$ holds. For an arithmetic operation `Alu` *op dst src*, *check_alu* first checks the validity of *op* with source *src*, evaluates *op* against destination *dst* using *alu*, stores the result v in register *dst*. For simplicity, we omit the case of immediate *src*s. If the result is $\emptyset$, so becomes the monadic state (undefined behavior). Our definition of *check_alu*, and well-formedness conditions (see Sec. 5.1) ensures that this will never happen and that, in case of error, the execution terminates with flag F_e. Similarly, the semantics of memory instructions (*Load-Store*) validates memory accesses using the *check_mem* function. Its definition ensures the absence of undefined behaviors.

5.1 Proof of Software-Fault Isolation

Our proof model M_p formalizes the semantics of rBPF. It is implemented in Coq using Gallina. Assessing its correctness consists of proving two essential properties: i) the well-formedness of the virtual machine's state, that is, its registers, memory and verifier invariants, and ii) software-fault isolation, that is, the isolation of all transitions to a crash state $\emptyset$ using runtime safety checks (*e.g., check_mem*), ergo the impossibility of a transition to an undefined behavior.

The register invariant states that all registers contain 64-bit integer values. This rules out 32-bit integers, *Vundef* but also pointers and floating-point numbers, for which the **alu** function may be undefined.

Definition 1 (register_inv). $\forall r \in registers. \exists l. R[r] = Vlong\ l$

As expected, the memory consistency invariant is a bit more elaborate. It states that each CompCert memory region *mr* register 8-bit integer blocks b of memory m, designated by a pointer *mr.ptr* to the 32-bit physical *mr.start* address of b, the 32-bit *mr.size* of b and at least *Readable* permissions *mr.perm* across $[0, size)$. Finally, every two regions point to disjoint physical address spaces in m (as per CompCert's memory regions for $mr'.ptr \neq mr.ptr$).

Definition 2 (memory_inv). $\forall mr \in MRs,\ m.\ \exists b, start, size.\ s.t.$
$mr.ptr = Vptr\ b\ 0\ \wedge\ Mem.valid\_block\ m\ b\ \wedge\ is\_byte\_block\ b\ m\ \wedge$
$mr.start = Vint\ start\ \wedge\ mr.size = Vint\ size\ \wedge\ mr.perm \geq Readable\ \wedge$
$Mem.range\_perm\ m\ b\ 0\ (Int.unsigned\ size)\ Cur\ mr.perm\ \wedge$
$(\forall mr' \in MRs, mr' \neq mr \rightarrow mr'.ptr \neq mr.ptr)$

Linux eBPF has a verifier to statically analyze eBPF programs and only accept those which are free of undefined behaviors. Our CertrBPF's verifier,

introduced in Sec. 8, ensures the weaker invariant given by Definition 3. The invariant stipulates the minimal pre-condition so that the interpreter can safely run a sequence of instructions I. More precisely, the invariant states that each instruction $I[i]$ references registers within the range $[0, 10]$ and that the target of every jump instruction is within the program range *i.e.,* $0 \leq i + ofs + 1 \leq length(I) - 1$.

Definition 3 (verifier_inv). $\forall i,\ I,\ ofs.\ 0 \leq i \leq length(I) - 1 \rightarrow$
$0 \leq get\_dst(I[i]) \leq 10\ \wedge 0 \leq get\_src(I[i]) \leq 10\ \wedge$
$((I[i] = Ja\ ofs \vee I[i] = Jump\ \_\ \_\ \_\ ofs) \rightarrow 0 \leq i + ofs + 1 \leq length(I) - 1)$

These three invariants implement well-formedness as proposed in Sec. 4. Therefore, the following Coq Theorem *sem_preserve_inv* proves Theorem 1 and states that well-formedness is preserved by the *interp* function. Similarly, Theorem *inv_ensure_no_undef* proves Theorem 2. This proves that the dynamic checks of the model M_p are sufficient to ensure the absence of error. In particular, all memory accesses are valid and performed within the dedicated memory regions. As a result, our model ensures software fault isolation. The corollary of Theorems *sem_preserve_inv* and *inv_ensure_no_undef* is that our virtual machine, obtained by refinement of the proof model, will always isolate code from other memory regions of the operating system and never crash it.

```
Theorem sem_preserve_inv:  ∀ (st st': state) (fuel: nat)
    (Hinv: register_inv st ∧ memory_inv st ∧ verifier_inv st)
    (Hsem: interp fuel st = ⌊(tt, st')⌋),
        register_inv st' ∧ memory_inv st' ∧ verifier_inv st'.
Theorem interp_no_undef:  ∀ (st: state) (fuel: nat)
    (Hinv: register_inv st ∧ memory_inv st ∧ verifier_inv st),
        interp fuel st ≠ ∅ .
```

6 A Synthesis-Oriented eBPF Interpreter

The coding style of the proof model M_p is quite different from the original RIOT implementation in C and lacks optimizations used in the latter to improve runtime performance. The synthesis model M_s firstly refines M_p into an optimized, safe and behaviorally equivalent monadic model which is then automatically transformed into an effectful implementation model M_c using ∂x.

Synthesis model M_s. M_s refines our proof model by following the principle "make M_s as close as possible to the expected target C code". M_s also refines Coq types because each Coq inductive type may correspond to several C types (*e.g.,* $Vint/Vlong$ to *signed* or *unsigned*, 32-bit or 64-bit). The case of $Vptr$ is particularly delicate, as the target type contextually relies on bit-size and signedness. To sort this out, we rename Coq types to match the correct C type. For example, $val64\_t, valu32\_t, vals32\_t$ are Val types mapped to `unsigned long long`, `unsigned int` and `int`, respectively.

Equivalence. Both M_p and M_s use the same monadic state *st* as in Sec. 5. Hence, the simulation relation $R \subseteq st \times st$, required by Theorem 3, is equality. As a result, we prove the stronger result that both $interp : nat \to M\ unit$, the M_p interpreter, and $interp\_dx : nat \to M\ unit$, the M_s interpreter, denote the exact same function.

```
Theorem equivalence_relation: ∀ (st: state) (fuel: nat),
    interp fuel st = interp_dx fuel st.
```

∂x configuration & Implementation model M_c. To extract the implementation model, we supply *∂x* with our monad *M* and a mapping relation from Gallina to C, Table 1.

Table 1: Mapping relation from Gallina to C

	Gallina	C
Types	*reg/sint32_t/valptr8_t ...*	*unsigned int/int/unsigned char* ...*
Constructions	*true/Int.repr(-2)/F_n ...*	*1/-2/0...*
Constants	*Val.addl/subl/mull/Z.eqb ...*	*+/-/*/==/ ...*
Functions	*eval_pc: M sint32_t ...*	*int eval_pc(struct state *) ...*
Code struct	*if-then-else, match-pattern ...*	*if-else, switch-case ...*

Inductive types map to C types, *e.g., reg* to *unsigned int* (note that a many-to-one relation from Gallina to C is legal). Gallina constructs & constant functions map to C operators & constants, *e.g.,* '*Val.addl*' to '+', '*Int.repr*(−2)' and '*true*' to '−2' and '1', etc. Gallina functions map to C functions. For any function operating the monadic state, the target C function has an additional argument *st* of type *struct state*∗ which corresponds to the implicit state of the monad. Gallina's *match-pattern* translates to C's *switch-case*, etc.

Code extraction with ∂x. The extracted C implementation preserves the structure of the original Gallina code, and the extracted C functions directly operate on actual memory locations as CompCert memory operations map to C expressions with a dereference. Consider the example of the `step_mem_st_reg` function.

```
Definition step_mem_st_reg (src: val64_t) (addr: valu32_t) (op: int8_t):
    M unit :=
  do opcode_st <- get_opcode_mem_st_reg op;
  match opcode_st with
  | op_BPF_STXW =>
    do addr_ptr <- check_mem Writable Mint32 addr;
      if eq_ptr_null addr_ptr then
        upd_flag BPF_ILLEGAL_MEM
      else (** i.e. Mem.storev Mint32 addr_ptr src *)
        do _ <- store_mem_reg Mint32 addr_ptr src; returnM tt
  ...
```

CompCert's Byte *int8_t* is mapped to *unsigned char*. Constructs *op_BPF_STXW*, *BPF_ ILLEGAL_MEM* and *Writable* are respectively mapped to '99', '-2' and

'2U'. The constant function `eq_ptr_null` is translated into an operation to check whether a pointer is null. The '`match opcode_st with`' is extracted to '`switch (opcode_st) case`'. Functions `step_mem_st_reg`, `check_mem` and `store_mem_reg` in C have an additional monadic argument *st*.

```
void step_mem_st_reg(struct bpf_state* st, unsigned long long
    src, unsigned int addr, unsigned char op){
  unsigned char opcode_st;
  unsigned char *addr_ptr;
  opcode_st = get_opcode_mem_st_reg(op);
  switch (opcode_st) {
    case 99:
      addr_ptr = check_mem(st, 2U, 4U, addr);
      if (addr_ptr == 0) {
        upd_flag(st, -2); return;
      } else { // i.e. *(unsigned int *) addr_ptr = src
        store_mem_reg(st, 4U, addr_ptr, src); return;
      }
    ...
```

7 Simulation Proof of the C rBPF Virtual Machine

In this section, we explain how to establish Theorem 4 for the Clight code of our virtual machine, derived from ∂x, and compiled into a Clight AST in Coq using the CLIGHTGEN tool.

Simulation Relation. A crucial ingredient of Theorem 4 is the simulation relation between the Gallina state monad and the Clight state which is essentially made of a CompCert memory. The Gallina state comprises a CompCert memory that models the various memory regions available to the rBPF program. This memory may also contain other blocks that are not modified by the virtual machine but represent other kernel data-structures. The simulation relation stipulates that such blocks also exist in the Clight memory and have the same content. The Clight memory contains additional blocks (*i.e., state_block*, *ins_block* and *mrs_block*) to model the other fields of the Gallina state. The layout and content of those blocks are depicted in Fig. 3.

Solid arrows in Fig. 3 are simulation relations between *state_block* and st_{rbpf}. Solid lines are the equalities between the rBPF memory m and blocks in rBPF-Clight memory. Dashed lines indicate relations of pointers to blocks in CompCert memory. The encoding exploits the fact that each field of the Gallina state has a known length. Thus, every field can be encoded as a continuous sub-block. As a result, the program counter is obtained from the first 4 bytes: loading a memory chunk of type *Mint32* at offset 0 retrieves the *pc* field of the Gallina state. The next 4 bytes encode the enumerated type flag. Here, each constructor of type flag is assigned an integer. The next 11×64 bits are used to encode the register

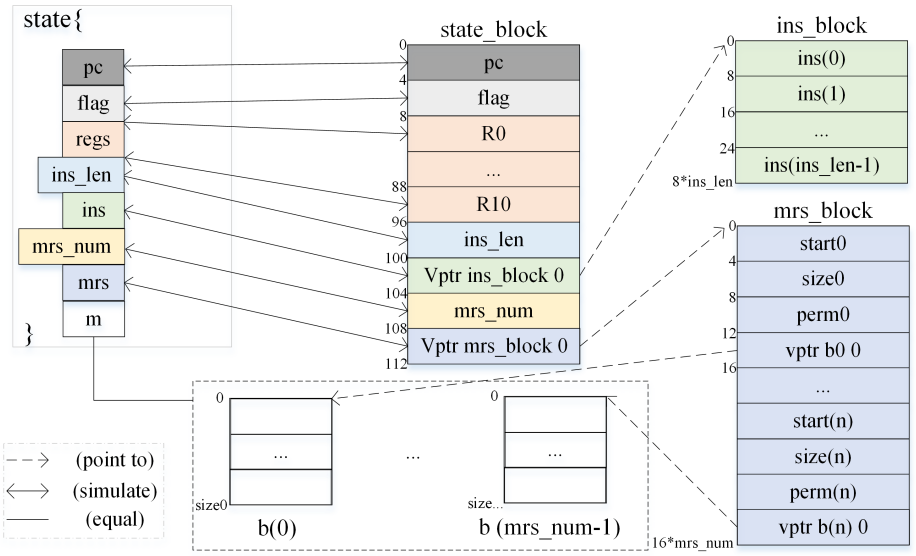

Fig. 3: Simulation relation R between st_{rbpf}, left, and rBPFClight, right.

bank of the Gallina state.

$$Rs(state, state\_block, m) \stackrel{\text{def}}{=} \begin{cases} st_{rbpf}.pc & = load\ Mint32\ m_{clight}\ state\_block\ 0 \\ st_{rbpf}.flag & = load\ Mint32\ m_{clight}\ state\_block\ 4 \\ st_{rbpf}.R0 & = load\ Mint64\ m_{clight}\ state\_block\ 8 \\ \dots & \end{cases}$$

The next elements of the Clight block represent the lists of instructions and of memory regions. In a functional language, lists are potentially of unbounded length and have a polymorphic type. Here, our lists always have fixed lengths and elements of fixed size. As a result, a list is directly encoded by a field specifying its length followed by a pointer to its memory block. The elements of the list are stored continuously in the pointed block.

Systematic Proof of Simulation. Since the ∂x tool is syntax-directed, there is a systematic correspondence between the source Gallina and the target C code. We exploit this property to design a minimal Clight logic geared toward our simulation proof. Our *Clightlogic* generalizes the translation validation theorem (Theorem 4) to accommodate Gallina functions and C functions with multiple arguments. In that case, we have a precondition which states that the Gallina and C arguments are linked pairwise by a refinement relation. Most of the arguments are numeric values and, in this case, the refinement relation states that the Gallina and C values are the same. The *Clightlogic* also provides a syntax-directed proof principle for each pair of Gallina/C syntactic construct. For instance, the

bindM operator translates to a sequence in the C code. Also, the result of a Gallina function call is bound to a local variable in C. Moreover, the local variable v below stands for the monadic state in C and points to the state memory block.

$$\partial x(bindM\ f\ (\lambda x.g)) = (vx = f_C(v); g_C(v, vx))$$

To exploit this pattern, our invariants take the form of an association list mapping each local variable to a set of C values that is obtained by partially evaluating a refinement relation with the Gallina value computed by the function (Fig. 3). To evaluate f, one needs to have a refinement relation Rs between the Gallina state st and the C value of v in memory m. Now, suppose that $fst = \lfloor r, st' \rfloor$. Since f_C is a correct refinement of f, relations $Rs(st', v, m')$ and $Rr(r, x)$ hold for the value x of the local variable vx in the current environment. We conclude by mapping $vx \mapsto Rr\ r$ and use this invariant to refine g by g_C.

The translation validation theorem proves a forward simulation relation from Coq to Clight. A backward simulation relation can be constructed as Gallina programs are functions and Clight is *determinate*.

8 CertrBPF Verifier

Linux eBPF's compiler and runtime system do not enforce type or memory safety. Instead, safety is verified prior to execution using a static analyzer that checks programs validity. As both the size and complexity cannot fit the requirements of an MCU architecture, CertrBPF instead provides a simple (linear time) but formally verified verifier, CertrBPF-verifier, which ensures the invariant $verifier\_inv$ (Def. 3). Accordingly, it scans an input rBPF program (*i.e.,* a list of 64-bit bytecode instructions) and rejects it when: i) a source or destination register is greater than 10. ii) the offset of a jump instruction is out of the instruction sequence bounds. iii) or the last instruction is not the *Exit* instruction (opcode `0x95`).

Static verification of these properties allows the interpreter to skip unnecessary dynamic checks. Our verifier adopts the same end-to-end verification method as the interpreter, Sec. 4. The virtual machine state in CertrBPF-verifier is a strict subset of the interpreter's state: $st_v = \langle I, M \rangle$ consists of a sequence of instructions I and a memory M.

```
Theorem verifier_well_formedness_and_safety :
  ∀ (st: verifier_state) (b: bool),
    verifier st = ⌊(b , st)⌋.
Theorem verifier_imply_inv :
  ∀ (st: verifier_state) (st': state)
    (Hinclude: st ⊂ st') (Hpre : verifier st = ⌊(true, st)⌋),
      verifier_inv st'.
```

Theorem $verifier\_well\_formedness\_and\_safety$ proves both Theorem 1 and Theorem 2. The verifier has the following properties: i) no assumption (every state is well-formed); ii) never crashes (safety); iii) never modifies the VM state.

In addition, the Coq theorem $verifier\_imply\_inv$ states that if the $verifier$ returns $true$, $verifier\_inv$ holds. Considering that the verifier's proof and synthesis models are exactly the same, the simulation relation $R_v \subseteq st_v \times st_v$ required by Theorem 3 is equality. CertrBPF-verifier reuses the *Clightlogic* to prove the simulation proof of its C implementation.

9 Evaluation: Case Study of RIOT's Femto-Containers

We integrate CertrBPF as a drop-in replacement for the existing non-verified module optimized for size (vanilla-rBPF) in the IoT operating system RIOT to provide the expected femto-container functionalities [39].

Implementation. The proof model of the interpreter (Sec. 5) consists of 2.4k lines of Coq code and the corresponding isolation proof (Sec. 5.1) is more than 4.8k lines long. The synthesis model, Sec. 6, is approx. 3.2k lines long and the equivalence theorem is completed by 0.6k proof code. The final step (Sec. 7) includes 10.8k translation validation proofs between the Gallina specification and the extracted Clight model. As for the CertrBPF verifier (Sec. 8), the proof and synthesis models sport 1.4k lines of Coq code. The corresponding proofs are more than 0.5k long and the last simulation proof is about 8.3k long. In addition, the *Clightlogic* implementation has 4.4k lines of Coq code.

Experimental evaluation setup. Our experimental objects are the original non-verified rBPF interpreter (*i.e.,* vanilla-rBPF) and the automatically extracted and verified CertrBPF interpreter (without RIOT's API). We carry out our measurements on a selected set of popular, commercial, off-the-shelf low-power IoT hardware, representative of modern 32-bit micro-controller architectures and boards: i) Nordic nRF52840 (Arm Cortex-M); ii) Espressif WROOM-32 (Espressif ESP32); iii) Sipeed Longan Nano GD32VF103CBT6 (RISC-V). All code is compiled with GCC using size optimization enabled and the `-foptimize-sibling-calls` GCC option to remove all tail-recursive calls and thus bound the stack size. This is critical to our isolation theorem as it relies on the implicit CompCert assumption that the stack cannot overflow. To avoid a possible mismatch between the CompCert semantics and the GCC semantics, we also pass the following options: i) `-fwrapv`, `-fwrapv-pointer` mean that both signed and pointer arithmetic wrap around according to the two's-complement encoding; ii) `-fno-strict-aliasing` means that there is no aliasing assumption.

Results. We first evaluate the memory footprint of the CertrBPF interpreter, compared to vanilla-rBPF. We measure i) *Flash size*: all read-only data, including the actual code; ii) *Stack*: the approximate ram used for stack space; iii) *Context*: the static RAM. In terms of Flash, our measurements show that CertrBPF actually reduces the footprint by 47% on RISC-V and by 35% on ESP32, and a 10% decrease on Cortex-M. In terms of stack requirements, CertrBPF reduces the footprint by 33% on Cortex-M, by 22% on RISC-V, and by 4% on ESP32. The context memory, however, increases from 92B to 144B on all platforms.

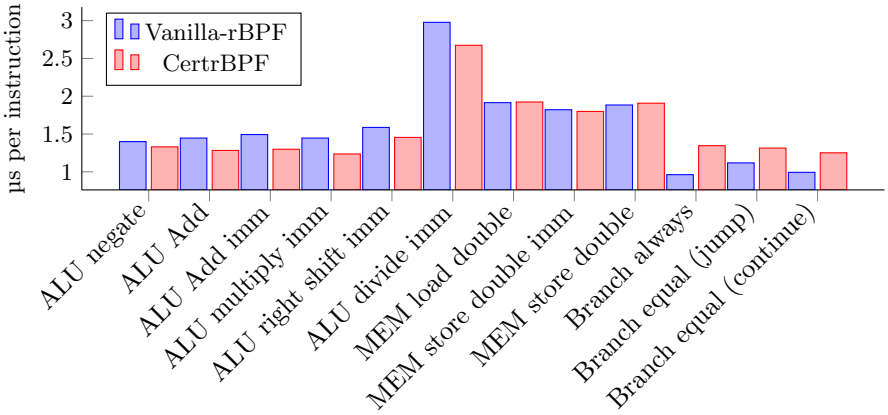

Fig. 4: Time per instructions on the Cortex-M4 platform

Next, we micro-benchmark the performance of core operations: single instructions from the arithmetic logic unit (ALU), for memory access (MEM) and branch instructions, with a mix of register and immediate value for the operands, Fig. 4. These results are averages over 1000 single identical instruction calls with a single return statement to make the application exit.

Finally, we benchmark the performance of actual IoT data processing, hosted in a femto-container with RIOT running on our selected hardware. In this use case, a sliding window average is performed within the femto-container, on available sensor data points. Figure 5 shows the performance we measured depending on the size of the window. We use this as blueprint for computation load scaling.

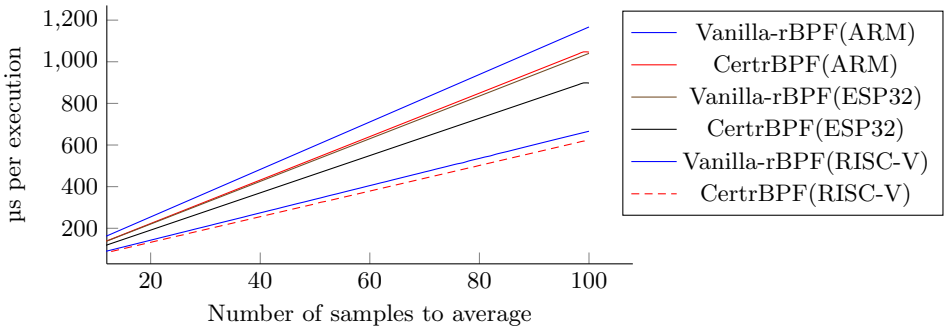

Fig. 5: Sliding window average on Cortex-M, ESP32, and RISC-V.

Key take-away. We observe that CertrBPF generally decreases the memory footprint. One reason is that calls to the RIOT API are currently not supported by CertrBPF. We observe, Fig. 4, that the execution slow-down is acute for Branch instructions, on Cortex-M. However, on all other platforms (RISC-V,

ESP32 and Cortex-M), our micro-benchmarks show that most instructions enjoy speed-up with CertrBPF compared to vanilla-rBPF. This behavior is also visible in our sensor data processing benchmark, Fig. 5, where CertrBPF performs better than vanilla-rBPF on three platforms. All in all, CertrBPF gains both security and reduces memory footprint as well as execution time.

10 Related Works

Methodologies for systems and compilers verification. The verification of compilers [18], static analyzers [16], and operating systems [17,12] have been the subjects of vast development and verification efforts due to the sheer code size of the artifacts at stake. These full-scale case studies gave rise to new strategies and methodologies to address the challenge of verifying large software. One such approach is Cogent [35] which aims at developing verified applications on top of the SeL4 [17] micro-kernel. Cogent [35] consists of a functional language with linear types to specify source programs and produces C code with Isabelle/HOL proof information. It provides a framework to prove that the extracted C code refines a high-level Isabelle/HOL functional correctness specification in the Isabelle/HOL proof assistant. Our method differs from co-specification in Cogent in that it is direct: it directly translates Coq specifications into C code and performs the end-to-end verification in Coq. CertiKOS [12] uses a multi-layered, refinement-based, and modular definition of a micro-kernel from its low-level memory model to its user-level interface and services. It is adopted in SeKVM [22], a layered Linux KVM hypervisor architecture for multiprocessor hardware. The CompCert project [18] adopted this "divide-and-conquer" strategy to decompose the verification of a full-scale ANSI C compiler into that of its successive transformations from source program to machine code, compositionally verifying each of the translation steps bisimilar. Its related static analyser, Verasco [16], employs static analysis of CompCert C code using a verified core abstract interpreter with composable abstract domains. Our problem statement is methodologically simpler: to build a safe and small VM that interprets rBPF virtual instructions on networked micro-controllers. We choose the radical approach of proof-oriented programming (à la Low* [34], Vale [5]) to prove an rBPF interpreter embedded in Coq correct and to directly extract verified code from its definition.

Background on BPF and its verified implementations. Mogul et al. [26] introduce a stack-based virtual machine to interpret packet filters into the BSD kernel that BPF extended to 32-bit instructions. BPF gained adoption in the Linux community and became eBPF (extended BPF), a virtual 64-bit RISC-like architectures. To our knowledge, verification of BPF runtime systems has mainly focused on JIT translation for operation on micro-kernels. Myreen [28] verifies a JIT compiler targeting x86 for a stack language using the HOL4 proof assistant. The generated code only preserves the semantics of the source code but does not ensure any isolation property. Porncharoenwase et al. [33] use CompCert to extract an OCaml translator from BPF to assembly code, verified using the proof assistant Coq, using the OCaml runtime, an assembler, and a linker as TCB. Van

Geffen et al. [11] present an optimized JIT compiler for Linux BPF with automated static analysis onboard, assuming offline verification using the Linux BPF verifier as TCB. For field deployment on networks of micro-controllers (IoT), all the above approaches would require a trusted, offline BPF verifier and, additionally, a secure upload protocol to sign verified scripts and perform authenticated uploads on target devices, which motivates our approach to use a fault-proof virtual machine instead.

Background on verified virtual machines. Lochbihler [23] presents the verified implementation of a virtual machine modeling the semantics, memory model and byte-code semantics of Java, all by using the proof methodology of translation validation [32,18]. Desharnais and Brunthaler [7] propose the formal verification of an optimized and secure Javascript interpreter in Isabelle/HOL. Its proof methodology is based on concepts of bisimulation. The interpreter targets optimal security and run-time performance. To target MCU devices, our rBPF VM instead seeks optimal run-time memory footprint, to support the expected capability of dynamically running several isolated services on a small device with shared memory. Zhang et al. [40] present a different and ambitious workflow using the deductive programming environment Why3 [9] to specify a virtual machine of Etherium byte-code (EVM) and verify functional correctness of smart contracts against it. The EVM is extracted to OCaml binary code, yielding a TCB consisting of the OCaml runtime and the implementation of Eth's protocols.

Background on converting Gallina programs into executables. Just as the proof-oriented approach advocated by dependently-typed functional languages like F* mentioned in Sec. 2, there are various alternatives to ∂x for extracting executables from Gallina programs. To begin with, Coq comes with a builtin extraction mechanism [21] that generates OCaml, Haskell or Scheme. This path has a rather large TCB (Coq extraction and a compiler). CertiCoq [1] is an ongoing project aiming at generating CompCert C code from Gallina using a specific IR and several passes. Once this effort is completed, it will allow one to rely on a small TCB. Œuf [27] is another tool to compile Gallina to C. It considers a carefully chosen subset of Gallina to tackle the tricky issue of verifying the reflection of Gallina into an AST. Both CertiCoq and Œuf, however, require a garbage collector and define how Coq inductives are represented at runtime. Codegen [36] converts Gallina to C with the goal of maximizing performance by, *e.g.,* allowing the user to control how Coq values are represented at runtime. Rupicola [31] considers an original and promising approach which regards a compiler as a partial decision procedure: it consists of a proof search procedure, which may fail, or else exhibit a target program in bedrock2 (a C-like low-level language AST embedded in Coq) with a proof of equivalence. It has, at present, only been tested for small algorithms. We chose to use ∂x for its simplicity and because it does not increase our TCB. It shares with Codegen the capability to configure the representation of values. Unlike Codegen, it produces C code that is structurally identical to source code. This direct and traceable translation simplifies

the verification of generated code w.r.t. source programs, and facilitates source program optimisations.

11 Conclusion and Future Works

This paper uses a refinement methodology to directly derive a verified C implementation of rBPF, the implementation of BPF hosted by the RIOT operating system, from a Gallina specification in Coq. All the refinement steps are mechanically verified using the Coq proof assistant to minimize the TCB. We prove our rBPF virtual machine to isolate software faults and not to produce runtime errors. Performances are at par with the vanilla rBPF implementation in RIOT.

Our future works aim at instantiating our proof workflow to a (fault-isolating) JIT compiler, one challenge being that Linux's approach of using a verifier will not be feasible on resource-constrained devices, and another being that certain operations might only be expressible in assembly code. This calls for further studies on ways to substantially improve the efficiency of our VM.

Acknowledgments The authors wish to thank the anonymous reviewers for their feedback and suggestions. This work is partly funded by Inria Challenge RIOT-fp, the ANR/BMBF project TinyPART, and the H2020 project Sparta.

Artifacts The source code and proofs of our virtual machine, its generated code and benchmark data are available on https://gitlab.inria.fr/syuan/rbpf-dx/-/tree/CAV22-AE.

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. Appel, A.W., Dockins, R., Hobor, A., Beringer, L., Dodds, J., Stewart, G., Blazy, S., Leroy, X.: Program logics for certified compilers. CUP (2014)
3. 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)
4. Bertot, Y., Castéran, P.: Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions. Springer (2013)
5. Bond, B., Hawblitzel, C., Kapritsos, M., Leino, K.R.M., Lorch, J.R., Parno, B., Rane, A., Setty, S., Thompson, L.: Vale: Verifying high-performance cryptographic assembly code. In: USENIX Security. pp. 917–934 (2017)
6. Costan, V., Lebedev, I., Devadas, S.: Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In: USENIX Security. pp. 857–874. USENIX (2016)
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, λProlog interpreter. In: LPAR. LNCS, vol. 9450, pp. 460–468. Springer (2015)

9. Filliâtre, J.C., Paskevich, A.: Why3—where programs meet provers. In: European symposium on programming, pp. 125–128. Springer (2013)
10. Fleming, M.: A Thorough Introduction to eBPF. Linux Weekly News (2017)
11. 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)
12. 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)
13. Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J.: Bringing the web up to speed with webassembly. In: PLDI. p. 185–200. ACM (2017)
14. Hahm, O., Baccelli, E., Petersen, H., Tsiftes, N.: Operating Systems for Low-End Devices in the Internet of Things: a Survey. IoT-J **3**(5), 720–734 (2016)
15. 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)
16. Jourdan, J.H., Laporte, V., Blazy, S., Leroy, X., Pichardie, D.: A Formally-Verified C Static Analyzer. In: POPL. pp. 247–259. ACM (2015)
17. 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)
18. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**(7), 107–115 (2009)
19. Leroy, X., Appel, A.W., Blazy, S., Stewart, G.: The CompCert Memory Model, Version 2. Research Report RR-7987, INRIA (2012)
20. 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)
21. Letouzey, P.: A new extraction for Coq. In: TYPES. pp. 200–219. Springer (2002)
22. Li, S.W., Li, X., Gu, R., Nieh, J., Hui, J.Z.: Formally verified memory protection for a commodity multiprocessor hypervisor. In: USENIX Security. pp. 3953–3970. USENIX (2021)
23. 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)
24. Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., Crowcroft, J.: Unikernels: Library operating systems for the cloud. In: ASPLOS. p. 461–472. ACM (2013)
25. 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)
26. Mogul, J., Rashid, R., Accetta, M.: The packer filter: An efficient mechanism for user-level network code. In: SOSP. p. 39–51. ACM (1987)
27. Mullen, E., Pernsteiner, S., Wilcox, J.R., Tatlock, Z., Grossman, D.: Œuf: minimizing the Coq extraction TCB. In: CPP. pp. 172–185. ACM (2018)
28. Myreen, M.O.: Verified just-in-time compiler on x86. In: POPL. pp. 107–118. ACM (2010)
29. 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)

30. Noorman, J., Agten, P., Daniels, W., Strackx, R., Herrewege, A.V., Huygens, C., Preneel, B., Verbauwhede, I., Piessens, F.: Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In: USENIX Security. pp. 479–498. USENIX (2013)
31. Pit-Claudel, C., Philipoom, J., Jamner, D., Erbsen, A., Chlipala, A.: Relational compilation for performance-critical applications. In: PLDI. ACM (2022)
32. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Steffen, B. (ed.) TACAS. LNCS, vol. 1384, pp. 151–166. Springer (1998)
33. Porncharoenwase, S., Bornholt, J., Torlak, E.: Fixing code that explodes under symbolic evaluation. In: VMCAI. LNCS, vol. 11990, pp. 44–67. Springer (2020)
34. Protzenko, J., Zinzindohoué, J.K., Rastogi, A., Ramananandro, 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
35. 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)
36. Tanaka, A.: Coq to C translation with partial evaluation. In: PEPM@POPL. pp. 14–31. ACM (2021)
37. Tassi, E.: Coq-Elpi, Coq plugin embedding Elpi (2021), https://github.com/LPCIC/coq-elpi
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.: Femto-Containers: DevOps on Microcontrollers with Lightweight Virtualization & Isolation for IoT Software Modules (2021), preprint
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)