End-to-end Mechanised Proof of an eBPF Virtual Machine for Microcontrollers

Anonymized submission to CAV'22

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 proposes a methodology to directly derive the verified C implementation of an eBPF virtual machine from a Gallina specification within the Coq proof assistant. 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 its safety and security properties of its Gallina specification. Our refinement methodology ensures that the isolation property of the specification holds in the verified C implementation. Preliminary experiments demonstrate satisfying performances.

Keywords: proof methodology · virtual machines · fault isolation.

1 Introduction

Hardware-enforced memory isolation is often not available on micro-controller units (MCU), as memory protection units usually trade coarse-grain isolation for significant price and performance overheads, and architecture dependencies: Trustzone, Sanctum [4], Sancus [28], *etc.* To mitigate development variability and cost, common practice for MCU operating system design (RIOT [3], FreeRTOS, TinyOS, Fushia, and others [11]) advises 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 [22], 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 scripts like Wasm [10] or the Berkeley packet filters (BPF [23]), using an interpreter or a just-in-time compiler (JIT) on the target device.

Femto-containers To this end, RIOT adopts extended Berkeley packet filters (eBPF) to implement so-called femto-containers: tiny virtual machine instances interpreting eBPF scripts. Compared to more expressive languages (e.g., Wasm), experimentsshow that eBPF requires a smaller memory footprint [35] and, Femto-containers run user-provided eBPF scripts in isolated memory spaces and restrict or wrap accesses to system calls. The Linux kernel features an eBPF JIT compiler whose security depends on a sophisticated online verifier [27]. Its memory footprint is incompatible with an an MCU, and its use would requires delegation of trust to a 3rd party.While a VM incurs slower execution speed compared to JIT-compiled code, it can, however, run untrusted, erroneous, adversary code in an open, unattended and possibly hostile environment, and still isolate faults.

Approach & Goals This paper investigates an approach that trades high-performance on low-power devices for defensive programming and runtime 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 eBPF femto-containers, we investigate a methodology allowing the verified extraction of CompCert C code directly from a mechanically proved eBPF interpreter.

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 a virtual instruction set, i.e. a virtual machine or a container, implemented using a system programming language (C). To this aim, we introduce a direct, end-to-end, validation workflow. The semantic 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 and branching. Then, C code is 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 that of the corresponding source instructions, as defined by their monadic specification. Our method uses a monadic subset of Gallina, the functional language embedded in the theorem prover Coq, of sufficient expressivity to specify the eBPF's semantics, and the verified extraction of equivalent Clight code, while provably implementing all required defensive runtime checks.

Plan The rest of the paper is organised as follows. Section 2 explicits the contributions of our proof method. Section 3 provides some background on CompCert and the ∂x tool. Section 4 presents our methodology for formally refining a monadic Gallina program into a C program. Section 5 defines the proof model of our virtual machine: its semantics and consistency and isolation theorems. Section 6 refines the proof model of our femto-container into a synthesis model ready for code generation with CompCert. Section 7 proves the equivalence between the synthesis and implementation models. Section 8 case studies the performance of our generated VM implementation with respect to off-the-shelf RIOT femto-containers. Section 9 presents related works and Section 10 concludes. *The source code and proofs of our virtual machine, its generated code and benchmark data are available to the CAV'22 evaluation committee on an anonymized repository.*

2 Contributions

Our contribution is first methodological: we propose a workflow of proof-oriented programming, using the functional language Gallina embedded in the theorem prover Coq. Our goal is to benefit from both the proof efficiency of verified programming (automation apart) and the proof exhaustivity of theorem proving.

Requirements Implementing a fault-isolating virtual machine for MCUs faces two major requirements. A first requirement is to embed the virtual machine inside the RIOT micro-kernel, hence, to minimize its code size and execution environment. A second requirement is to minimise the verification gap between the proof model and the running code. An obvious approach would be to use the existing Coq extraction mechanism to compile the Gallina model into OCaml. To this end, Coq extraction to OCaml should then be trusted and the OCaml runtime would become part of the Trusted Computing Based, and would further need to be trimmed down to fit space requirements. Another approach would be to directly generate C code using, e.g., $F^\star$'s KreMLin compiler, which does however not yet provide a mechanised equivalence guarantee between source $Low^\star$ programs and extracted C code.

Proof Methodology Our ambition is to minimise the verification gap and provide an end-to-end security proof linking our Gallina model to the 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 AST. To reach this goal, our starting point is a model of the eBPF semantics. The model is written in Gallina, the functional language of the Coq proof assistant. For this model, we prove that all the memory accesses are valid and isolated to dedicated memory areas, thus ensuring isolation. From this model, we derive an executable version that we prove equivalent. It is written in a restricted subset of Gallina that fits the constraints of an experimental C code extractor ∂x, used to implement the Pip proto-kernel in Coq [12], that produces a Clight abstract syntax tree. We prove that each Clight function performs the same state transitions as its Coq definition.

A certified BPF interpreter This paper introduces CertrBPF: the verified model and implementation of an rBPF interpreter in Coq. We formalize the syntax and semantics of all rBPF instructions, implement a formal model of the interpreter, complete the proof of critical properties of our model, 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 theorem prover, 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 TCB and memory footprint: CertrBPF is a memory-efficient, fully isolated, kernel-level virtual machine for eBPF, that isolates any runtime software fault using defensive code and does not necessitate offline verification.

Systems Integration & Micro-benchmarks We integrate CertrBPF as a drop-in replacement for the corresponding non-verified module in the RIOT operating system. We then comparatively evaluate the performance of CertrBPF integrated in RIOT, running on various 32-bit microcontroller architectures. Our benchmarks demonstrate that, in practice, CertrBPF's memory footprint is on par, and incurs only a small execution slow-down, well worth its security gains.

3 Background

This section describes essential features of BPF, 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 BPF [23] 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 [8]. eBPF was then ported to micro-controllers, yielding rBPF [34]. From an architectural point of view, eBPF is designed as a 64-bit register-based VM, using fixed-size 64-bit instructions and a reduced instruction set architecture (ISA). eBPF uses a fixed-sized stack (512 bytes) and defines no heap interaction, which limits VM memory overhead in random-access memory (RAM). The eBPF 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 [16] is a C compiler that is both programmed and proved correct using the Coq proof assistant. The compiler is structured into compiler passes using several intermediate languages. Each intermediate language is equipped with a formal semantics and each pass is proved to preserve the observational behaviour of programs.

The Clight Intermediate Language [18] 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 Gallina syntax from a C program.

CompCert Values and Memory Model [18,17] are shared across all the intermediate languages. 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 behaviour. Yet, as most of the C operators are strict in $Vundef$, and because branching over $Vundef$ or dereferencing $Vundef$ are undefined behaviours, 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 a 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 $\emptyset$) for invalid offsets o and invalid permissions.

The ∂x tool [12] extracts C code from a Gallina source program in the form of a CompCert C abstract syntax tree (AST). ∂x emerged from the toolchain used to design and verify the Pip proto-kernel. Its aim was to allow writing most of Pip's source code in Gallina in a style as close to C as possible. The goal of ∂x is to provide C programmers with 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, the language that ∂x can handle is only 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. But ∂x does not mandate a particular monad for code extraction.

∂x proceeds in two steps. First, given a list of Gallina functions, or whole modules, it generates a simple 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 [6], using the Coq-Elpi plugin [32]. That step can also process external functions (appearing as `extern` 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, one 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. A major design choice in the C-like subset of Gallina used by ∂x is memory management: the generated code should be executed without a garbage collector. This constrains the Coq types that can actually be used in ∂x: recursive inductive types such as lists cannot automatically be converted. However, this subset is particularly relevant for programs in which one wants to precisely control the management of memory and how data structures are represented in memory, such as an operating system or our rBPF virtual machine.

4 A generic method 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 a C implementation of a fault-isolating eBPF virtual machine. The novelty of our approach is to provide an end-to-end correctness proof, within the Coq proof assistant, that reduces the hurdle of reasoning directly over the C code. The verification of safety properties is performed on a Gallina specification. This specification is then refined into a C-ready Gallina implementation. At this stage, the proofs may be complex depending on the safety properties at stake and the distance between the specification and the C-ready implementation. From the C-ready implementation, we leverage ∂x to automatically generate a CompCert C AST. The verification of this step is also performed within the Coq proof assistant using translation validation. Because ∂x generates C code in a syntax-directed manner, the proof is systematic and minimizes the hurdle of reasoning over the Clight semantics of CompCert. 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.

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

In the remainder, we write $\emptyset$ for **None**, $\lfloor x \rfloor$ for **Some** x and **do** $x \leftarrow f; g$ for $bindM f (\lambda x.g)$.

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 Section 5), the absence of error ensures that the BPF 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: $wf : \mathcal{P}(state)$. We prove that well-formedness is an inductive property of the *step* function.

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

$$\forall st\ st'.\ st \in wf \wedge step\ st = \lfloor st' \rfloor \Rightarrow st' \in wf$$

We also prove 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 prove a simulation relation between the *step* and $step_{\partial x}$ functions. A direct consequence of the simulation theorem 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 of VST to get a Clight $step_C$ program (see Section 6). As this pass is not trusted, we prove 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 $step_{\partial x}$:*

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

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 behaviours. In particular, all memory accesses are valid. As the memory model does not allow to forge pointer, this is already a strong isolation property. *All the theorems stated in this section are expressed and proved using the Coq proof assistant.*

5 A proof-oriented virtual machine model

The proof model of our memory-isolated VM first requires preliminary definitions to denote the syntax and state of the interpreter, and auxiliary functions, to denote dynamic security checks. The rBPF instruction set, Fig. 1, features binary arithmetic and logic operations, negation, (un)conditional jumps relative to an offset, operations to load/store values from/to registers/memory, and termination. It operates on 64-bit *registers* $\{R0, \ldots, R10\}$, categorized as 32-bit immediate or register sources *src* and destination registers *dst* modulo 16-bit offsets *ofs*. The source operand of a `Load` instruction can only be a register *reg*. Compared to the RIOT rBPF implementation, our specification does not comprise the call instruction, which performs trusted API calls. As control flows outside of the VM, their security is delegated to the host OS.

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

Fig. 1: Core syntax of rBPF instruction set

Machine state A semantic state *st* is a tuple $\langle I, L, R, F, M, MR \rangle$ consisting of a sequence of instructions I, the current location L, registers R, an interpreter flag F, a memory M and memory regions MRs. The flag F indicates the state of the rBPF interpreter – normal *BPF_OK*, denoted F_n – terminated *BPF_SUCC*, denoted F_t – or error-tolerable *BPF_ILLEGAL_DIV/MEM*, ..., denoted F_e.

A collection MR of memory regions *mr* forms a CompCert memory model. A region $mr = \langle start, size, perm, ptr \rangle$ is defined by its physical *start* address, its *size*, its access *perm*ission and a pointer *ptr (= Vptr b 0)* to the corresponding block b where the CompCert memory model stores it. We write $I(L_{pc})$ for the instruction located at the program counter L_{pc}. $R[r]$ retrieves the value of the register r in the register map R. $R[r \leftarrow v]$ updates it with value v. Functions *Alu* and *Cmp* reuse the Compcert's operators over the *val* type. The *Alu* function returns *None* if an error occurs *e.g.* division by zero. Functions *load* and *store* are those of CompCert's memory model (see Section 3).

$$
\begin{array}{ll}
\mathbf{Alu} : op \to val \to val \to option\ val & \quad \mathbf{Cmp} : cmp \to val \to val \to bool \\
\mathbf{load} : chk \to mem \to block \to Z \to option\ val & \\
\mathbf{store} : chk \to mem \to block \to Z \to val \to option\ mem &
\end{array}
$$

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* requires the value v to be non-zero. The value v of an arithmetic or logical shift instructions should be within the specific range $n \in \{32, 64\}$ of the given instruction. CertrBPF has both 32-bit and 64-bit ALU instructions, but this paper only regards the latter for simplicity.

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

Function *check_mem* returns a valid pointer (*Vptr b ofs*) if there exists a unique memory region *mr* in *MRs* such that 1) the permission *mr.perm* is at least *Readable* for `Load` and *Writable* for `Store`:, i.e. $mr.perm \geq p$; and 2) the offset *ofs* is aligned, i.e. $ofs\%Z(chk) = 0$ and in bounds, i.e. $ofs \leq max\_unsigned - Z(chk)$, and the interval $[ofs, hi\_ofs)$ is in the range of mr.Otherwise, *check_mem* returns the null pointer $V nullptr$. The function *Z(chk)* maps *byte*, *halfword*, *word* and *double* to 1, 2, 4, and 8.

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

Semantics Functions *run* and *step* 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 *run* processes a sequence of instructions I in nominal state F_n, checking the *pc* bounds. The second stage *step* processes individual instructions $I(L_{pc})$. MR 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 the *pc* gets out of bounds, the flag turns to F_e. Result `None` marks transitions to crash states that the theorem prover should weave as unreachable (ghosts) using carefully crafted definitions of the `check_alu` and `check_mem` functions.

```
run = λs. if 0 ≤ s.L < length(I) then match step s with
            | Some ((), t) => if t.F≠ Fn then Some((),t) else run
            | None => None
          else Some((),s{F=Fe})

step = λs. match s.I(s.L) with
| Exit -> Some ((), s{F = Ft})
| Ja ofs -> Some ((), s{L = s.L+ofs+1})
| Jump cmp dst ofs -> if Cmp(cmp,s.R[dst],s.R[src])
                      then Some ((), s{L = s.L+ofs})
                      else Some ((), s{L = s.L+1})
| Alu Neg dst -> let Some v=Val(Neg,s.R[dst])in Some ((),s{R[dst]=v})
| Alu op dst src -> if ¬check_alu(op,s.R[src]) then s{F = Fe}
                    else match Val(op,s.R[dst],s.R[src]) with
                    | Some v -> Some ((), s{R[dst] = v})
                    | None -> None
| Load ck dst reg ofs -> match check_mem(Rd,ck,s.R[reg]+ofs,s.MR) with
      | Vptr b ofs -> match load(ck,M,b,ofs) with
        | Some v -> Some ((), s{R[dst] = v})
        | None -> None
      | _ -> Some ((), s{F = Fe})
| Store ck dst src ofs-> match check_mem(Wr,ck,s.R[src]+ofs,s.MR) with
      | Vptr b ofs -> match store(ck,M,b,ofs) with
        | Some N -> Some ((), s{M = N})
        | None -> None
      | _ -> Some ((), s{F = Fe})
| _ -> Some ((), s{F = Fe})
```

Exit terminates the program with flag F_t and unconditional jump Ja increments the pc by $ofs + 1$. A conditional $Jump$ does so when function Cmp says that "$src\,cmp\,dst$" is true or skips. 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 and increments the pc. For simplicity, we omit the case of immediate srcs, which would read src instead of $R[src]$ in the rules below. If the result is `None`, so becomes the monadic state (undefined behavior). Our definition of $check\_alu$, and well-formedness conditions (see Section 5.1) ensures that this will never happen and that, in case of error, the execution terminates with flag F_e. The Neg instruction is a simpler Alu instruction with only one operand (Neg). Similarly, the semantics of memory instructions discerns legal accesses (*Load-Store*) from invalid ones using the $check\_mem$ function, whose careful definition should avoid undefined behaviors. We write $s.F$ for the value of field F in record s and $s\{F = v\}$ updates it to v.

5.1 Proof of software-fault isolation

Our proof model M_p formalizes the semantics of eBPF. It is implemented in Coq using Gallina. Assessing its correctness consists of proving two essential properties: 1) the well-formedness of the virtual machine's state, that is, its registers and memory invariants, and 2) software-fault isolation, that is, the isolation of all transitions to a crash state $\emptyset$ using runtime safety checks (e.g. $check\_alu$, $check\_mem$), ergo the impossibility of a transition to an undefined behavior. The register invariant $Registers\_Inv$ states that all registers contain 64-bits integer values. This rules out 32 bits integers, $Vundef$ but also pointers and floating-point numbers, for which the **Alu** function would be undefined.

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

The memory consistency invariant is, expectably, a bit more administrative. It states that CompCert memory regions 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 should point to disjoint physical address spaces in m.

Definition 2 (memory_inv). $\forall mr \in MRs,\ m. \exists b, start, size.\ s.t.$

$$\begin{aligned}&Mem.valid\_block\ m\ b\ \wedge\ is\_byte\_block\ b\ m \wedge mr.start = Vint\ start\ \wedge\\&mr.size = Vint\ size \wedge \forall mr' \in MRs, mr' \neq mr \rightarrow mr'.ptr \neq mr.ptr \wedge\\&Mem.range\_perm\ m\ b\ 0\ (Int.unsigned\ size)\ Cur\ mr.perm \wedge mr.perm \geq Readable\end{aligned}$$

These two invariants implement well-formedness as proposed in Sec. 4. Therefore, the following Coq Theorem `sem_preserve_inv` instantiates Theorem 1 and states that well-formedness is preserved by the $step$ function. Similarly, Theorem `inv_ensure_no_undef` instantiates 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 property. 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 will never crash it.

```
Theorem sem_preserve_inv:
  ∀ (st1 st2: state) (Hinv : registers_inv st1 /\ memory_inv st1)
    (Hsem : step st1 = Some (tt, st2)),
      registers_inv st2 /\ memory_inv st2.

Theorem inv_ensure_no_undef:
  ∀ (st1: state) (Hflag : st1.flag = BPF_OK)
    (Hinv : registers_inv st1 /\ memory_inv st1), step st1 ≠ None.
```

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. To this aim, the synthesis model M_s 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. In order to extract a verified implementation of rBPF of competing efficiency with RIOT's, the synthesis model refines our proof model in several respects.

1/ Renaming. Each Coq inductive type may correspond to several C types (e.g. $Vint/Vlong$ to *signed* or *unsigned*, 32-bits or 64-bits) and require careful encoding. 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.

2/ Erasure. There are *ghost functions* present in M_p to identify undefined behaviors (branches to state `None`). These branches can be erased in M_s since our safety theorem proves that no undefined behaviors is possible.

3/ Masking. M_p implements an eager decoding of all 64-bit binary instructions and all specific fields. However, most instructions do not use all of these fields. Instead, M_s adopts a fine-grained decoding, i.e. masking operations, yielding a verifiable optimisation reducing computation and reusing code. For example, given a binary instruction '0f 02 00 00 00 00 00', M_p first computes *opcode = 0x0f, dst = R0, src = R2, imm =...*, then compares the opcode with all 78 possible ones to find out that the instruction is *add64 dst src*. Instead, M_s first masks '$opcode\&0x07$' to identify *ALU64 dst*, then masks '$opcode\&0x08$' to match *ALU64 dst src*, compares '$opcode\&0xf0$' with the 13 possible alu opcodes, and only computes *dst* and *src*.

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 the **Simulation** theorem, simply is equality. rBPF's **Simulation** theorem is defined with *step* : *M unit*, the M_p interpreter, and *step_dx* : *M unit*, the M_s interpreter, in Coq.

```
Theorem equivalence_relation: ∀ st, step st = step_dx 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, Tab. 1. 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 holds an additional argument *st* of type *struct state∗* which corresponds to the implicit state of the monad. Gallina *match-pattern* translates to C *switch-case*, etc.

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)/BPF_OK/Mint32 ...*	*1/-2/0/4...*
Constants	*Val.addl/Val.subl/Val.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 ...*

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 (src64: val64_t) (addr: valu32_t) (pc:
↪  sint32_t) (dst: reg) (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 comp_eq_ptr8_null addr_ptr then upd_flag BPF_ILLEGAL_MEM
      else (** i.e. Mem.storev Mint32 addr_ptr src64 *)
        do _ <- store_mem_reg Mint32 addr_ptr src64; returnM tt ...
```

CompCert's Int and Byte *sint32_t* and *int8_t*, the register type *reg*, are mapped to *int*, *unsigned char* and *unsigned int*. Constructs *op_BPF_STXW*, *BPF_ILLEGAL_MEM* and *Writable* are respectively mapped to '99', '-2' and '2U'. The constant function `comp_eq_ptr8_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 src64,
↪  unsigned int addr, int pc, unsigned int dst, 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 = src64
        store_mem_reg(st, 4U, addr_ptr, src64); 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 Coq AST using the CLIGHTGEN tool of the VST toolchain.

Simulation Relation. A crucial ingredient of this theorem 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 is depicted in Figure 2.

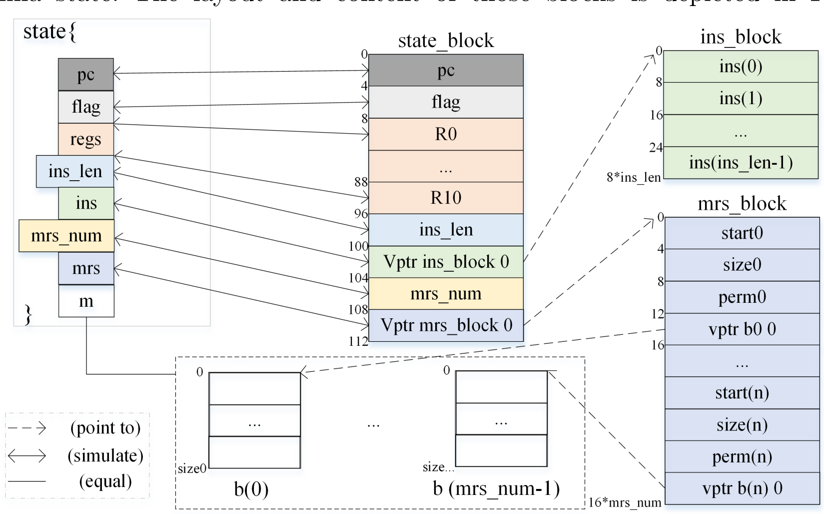

Fig. 2. Simulation relation R between st_{rbpf}, left, and rBPFClight, right.

Solid arrows in Fig. 2 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 4 first 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 the flag type is assigned an integer. The next 10×64 bits are used to encode the register bank of the Gallina state.

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

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 source Gallina and 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 of Section 4 (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, e.g., 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. 2). 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 $f st = \lfloor r, st' \rfloor$. Since f_C is a correct refinement of f, we have $Rs(st', v, m')$ and $Rr(r, x)$ 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 for the refinement of g by g_C.

Status of the Proof At the time of writing, we completed the proofs of isolation for model M_p, of simulation with model M_s, of translation for most ALU and JUMP instructions. The proof of memory operations still requires work as the validation of memory addresses is recursive and therefore requires strengthening the refinement relation. Yet, we are confident that the complete proof of the 78 instructions of rBPF will be completed in a timely manner.

8 Evaluation: case study of RIOT's femto-containers

In this section we evaluate CertrBPF in a concrete use case. We integrate CertrBPF as a drop-in replacement for the corresponding non-verified module optimized for size (vanilla-rBPF) in the IoT operating system RIOT to provide the expected femto-container functionalities [35].

Experimental Setup. We carry out our measurements on a selected set of popular, commercial, off-the-shelf low-power IoT hardware, representative of modern 32-bit microcontroller architectures: Arm Cortex-M, Espressif ESP32, and RISC-V. Correspondingly, the boards we used are the Nordic nRF52840 development kit, the Espressif WROOM-32 board, and the Sipeed Longan Nano GD32VF103CBT6 development board. All code is compiled with GCC using size optimization enabled and the `-foptimize-sibling-calls` GCC option to reduce all tail-recursive calls and bound the stack size. This is critical to our isolation theorem as it relies on the implicit CompCert assumption of the absence of stack overflow.

Results. We first evaluate the memory footprint of the CertrBPF interpreter, compared to vanilla-rBPF in our use case. In terms of Flash, our measurements show that CertrBPF actually reduces the footprint by 37 % on RISC-V and by 25 % on ESP32, while incurring 6 % increase 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 92 B to 144 B on all platforms.

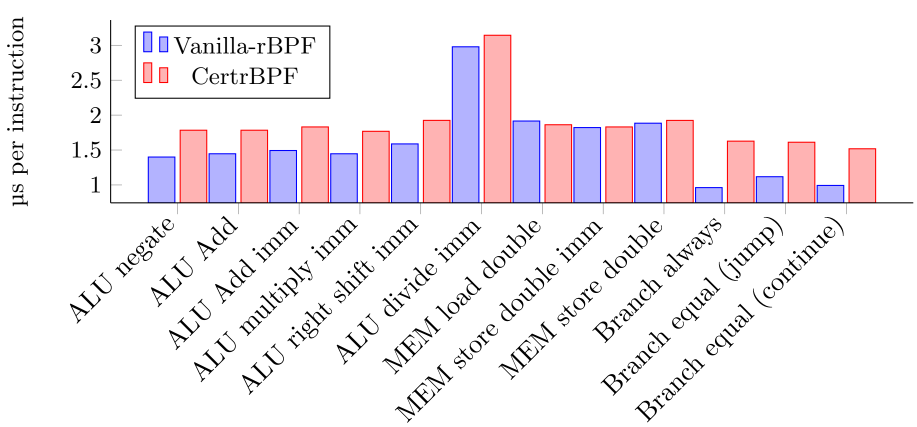

Fig. 3: Time per instructions on the Cortex-M4 platform

Next, we micro-benchmark the performance of fundamental 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, Figure 3. 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 4 shows the performance we measured depending on the size of the window. We use this as blueprint for computation load scaling. To visualize the trade-off between runtime speed and flash size, we also ran sliding window benchmarks with forced function inlining on the certrBPF code. We observed that the resulting VM would then be 3208 B in size while averaging 1024 µs per execution versus 1341 µs, hinting at fine-tuning opportunities to increase speed by up to 30 %.

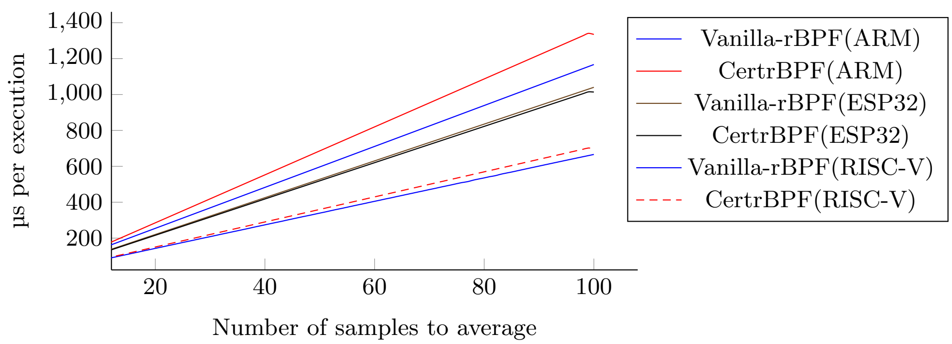

Fig. 4: Sliding window average on Cortex-M, ESP32, and RISC-V.

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, Figure 3, that the execution slow-down is most acute for Branch instructions, on Cortex-M. However, on all platforms (RISC-V, ESP32 and Cortex-M) our micro-benchmarks show that most instructions suffer minimal slow-down with CertrBPF compared to vanilla-rBPF. This behavior is also visible in our sensor data processing benchmark, Fig. 4, where the performances of CertrBPF and vanilla-rBPF are almost identical on ESP32 and RISC-V. All in all, the security gains brought by CertrBPF can be considered worth the performance loss we measured.

9 Related Works

Methodologies for systems and compilers verification The verification of compilers [16], static analysers [13] and operating systems [14,9] have been the subjects of colossal development and verification 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 key contribution in the development of the verified SeL4 micro-kernel, for instance, is the CAmkES [15] framework, which allows to modularly co-design the specification (proof model) and the implementation (code model) of system components. In CertiKOS [9], this approach generalizes to the 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 [20], a layered Linux KVM hypervisor architecture for multiprocessor hardware. The CompCert project [16] 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 [13], employs static analysis of CompCert C code using a verified core abstract interpreter with composable abstract domains. Our problem statement is methologically simpler: to build a safe and small VM that interprets eBPF virtual instructions on networked microcontrollers. We choose the radical approach of proof-oriented programming (à la Low*, Vale) to prove an eBPF interpreter embedded in Coq correct and to directly extract verified code from its definition.

Background on BPF and its verified implementations [24] introduced 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. [26] verifies a JIT compiler from BPF to Intel architectures using the HOL4 theorem prover and an x86 compiler as TCB. [30] uses CompCert to extract an Ocaml translator from BPF to assembly code, verified using the theorem prover Coq, using the OCaml runtime, an assembler, and a linker as TCB. [33] is 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 microcontrollers (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 Indeed, as Google's Project Zero recently pointed out a series on JIT-compiler exploits, interest for isolating byte-code interpreters inside virtual machines has regained. [21] 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 [29,16]. [5] proposes the formal verification of an optimized and secure Javascript interpreter in HOL. Its proof methodology is based on concepts of bisimulation. The interpreter targets optimal security and run-time performance. To target MCU devices, our eBPF 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. [36] presents a different and ambitious workflow using the deductive programming environment

Why3 [7] 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 Etherium's protocols.

Background on converting Gallina programs into executables Just as the proof-oriented approach advocated by dependently-typed functional languages like F* and Liquid Haskell, 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 [19] 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 [25] is another tool to compile Gallina to C. It considers a carefuly 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 [31] 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. We chose ∂x [12] for its simplicity and because it doesn't 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.

10 Conclusion and Future Works

We propose a refinement methodology from Gallina programs to C code. All the refinement steps are mechanically verified using the Coq proof assistant to minimize the TCB. Our case study is the rBPF VM which we prove to isolate software faults and not to err at runtime. Performances are at par with the vanilla rBPF implementation. Yet, the optimised rBPF implementation uses Gcc's label-as-values extensions. As it cannot be expressed by the CompCert semantics, this optimisation is out of reach of CertrBPF. As future work, we aim at instantiating our proof model to Wasm, or to that of a (fault-isolating) JIT compiler, one challenge being that Linux' 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 that would substantially improve the efficiency of our VM.

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 workshop, Paris, 2017 (2017)
2. Appel, A.W., Dockins, R., Hobor, A., Beringer, L., Dodds, J., Stewart, G., Blazy, S., Leroy, X.: Program logics for certified compilers. Cambridge University Press (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. IEEE Internet of Things Journal **5**(6), 4428–4440 (2018)
4. Costan, V., Lebedev, I., Devadas, S.: Sanctum: Minimal hardware extensions for strong software isolation. In: 25th USENIX Security Symposium (USENIX Security 16). pp. 857–874. USENIX Association, Austin, TX (Aug 2016), https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan
5. Desharnais, M., Brunthaler, S.: Towards efficient and verified virtual machines for dynamic languages. In: Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs. p. 61–75. CPP 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3437992.3439923, https://doi.org/10.1145/3437992.3439923
6. Dunchev, C., Guidi, F., Coen, C.S., Tassi, E.: ELPI: Fast, embeddable, λProlog interpreter. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9450, pp. 460–468. Springer (2015). https://doi.org/10.1007/978-3-662-48899-7_32, https://doi.org/10.1007/978-3-662-48899-7_32
7. Filliâtre, J.C., Paskevich, A.: Why3—where programs meet provers. In: European symposium on programming, pp. 125–128. Springer Berlin Heidelberg (2013)
8. Fleming, M.: A Thorough Introduction to eBPF. Linux Weekly News (2017)
9. Gu, L., Vaynberg, A., Ford, B., Shao, Z., Costanzo, D.: CertiKOS: a certified kernel for secure cloud computing. In: Proceedings of the Second Asia-Pacific Workshop on Systems - APSys '11. p. 1. ACM Press, Shanghai, China (2011). https://doi.org/10.1145/2103799.2103803, http://dl.acm.org/citation.cfm?doid=2103799.2103803
10. 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: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 185–200. PLDI 2017, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3062341.3062363, https://doi.org/10.1145/3062341.3062363
11. Hahm, O., et al.: Operating Systems for Low-end Devices in the Internet of Things: a Survey. IEEE Internet of Things Journal **3**(5), 720–734 (2015)
12. Jomaa, N., Torrini, P., Nowak, D., Grimaud, G., Hym, S.: Proof-oriented design of a separation kernel with minimal trusted computing base. In: 18th International Workshop on Automated Verification of Critical Systems (AVOCS 2018). Oxford, United Kingdom (Jul 2018). https://doi.org/10.14279/tuj.eceasst.76.1080, https://hal.archives-ouvertes.fr/hal-01816830

13. Jourdan, J.H., Laporte, V., Blazy, S., Leroy, X., Pichardie, D.: A formally-verified C static analyzer. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 247–259. ACM, Mumbai India (Jan 2015). https://doi.org/10.1145/2676726.2676966, https://dl.acm.org/doi/10.1145/2676726.2676966
14. 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: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP '09. p. 207. ACM Press, Big Sky, Montana, USA (2009). https://doi.org/10.1145/1629575.1629596, http://portal.acm.org/citation.cfm?doid=1629575.1629596
15. Kuz, I., Liu, Y., Gorton, I., Heiser, G.: CAmkES: A component model for secure microkernel-based embedded systems. Journal of Systems and Software **80**(5), 687–699 (2007). https://doi.org/https://doi.org/10.1016/j.jss.2006.08.039, https://www.sciencedirect.com/science/article/pii/S016412120600224X
16. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**(7), 107–115 (Jul 2009). https://doi.org/10.1145/1538788.1538814, https://dl.acm.org/doi/10.1145/1538788.1538814
17. Leroy, X., Appel, A.W., Blazy, S., Stewart, G.: The CompCert Memory Model, Version 2. Research Report RR-7987, INRIA (Jun 2012), https://hal.inria.fr/hal-00703441
18. Leroy, X., Blazy, S.: Formal verification of a C-like memory model and its uses for verifying program transformations. Journal of Automated Reasoning **41**(1), 1–31 (2008), http://xavierleroy.org/publi/memory-model-journal.pdf
19. Letouzey, P.: A new extraction for coq. In: International Workshop on Types for Proofs and Programs. pp. 200–219. Springer (2002)
20. Li, S.W., Li, X., Gu, R., Nieh, J., Hui, J.Z.: Formally verified memory protection for a commodity multiprocessor hypervisor. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 3953–3970. USENIX Association (Aug 2021), https://www.usenix.org/conference/usenixsecurity21/presentation/li-shih-wei
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). https://doi.org/10.5445/KSP/1000028867
22. 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: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems. p. 461–472. ASPLOS '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2451116.2451167, https://doi.org/10.1145/2451116.2451167
23. McCanne, S., Jacobson, V.: The bsd packet filter: A new architecture for user-level packet capture. In: USENIX winter. vol. 46 (1993)
24. Mogul, J., Rashid, R., Accetta, M.: The packer filter: an efficient mechanism for user-level network code. ACM SIGOPS Operating Systems Review **21**(5), 39–51 (1987)
25. Mullen, E., Pernsteiner, S., Wilcox, J.R., Tatlock, Z., Grossman, D.: Œuf: minimizing the coq extraction TCB. In: Andronick, J., Felty, A.P. (eds.) Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018. pp. 172–185. ACM (2018). https://doi.org/10.1145/3167089

26. Myreen, M.O.: Verified just-in-time compiler on x86. In: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 107–118 (2010)
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: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). pp. 41–61. USENIX Association (Nov 2020), https://www.usenix.org/conference/osdi20/presentation/nelson
28. 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: 22nd USENIX Security Symposium (USENIX Security 13). pp. 479–498. USENIX Association, Washington, D.C. (Aug 2013), https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/noorman
29. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Goos, G., Hartmanis, J., van Leeuwen, J., Steffen, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, vol. 1384, pp. 151–166. Springer Berlin Heidelberg, Berlin, Heidelberg (1998). https://doi.org/10.1007/BFb0054170, http://link.springer.com/10.1007/BFb0054170, series Title: Lecture Notes in Computer Science
30. Porncharoenwase, S., Bornholt, J., Torlak, E.: Fixing code that explodes under symbolic evaluation. In: International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 44–67. Springer (2020)
31. Tanaka, A.: Coq to C translation with partial evaluation. In: Lindley, S., Mogensen, T.Æ. (eds.) Proceedings of the 2021 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM@POPL 2021, Virtual Event, Denmark, January 18-19, 2021. pp. 14–31. ACM (2021). https://doi.org/10.1145/3441296.3441394
32. Tassi, E.: Coq-Elpi, Coq plugin embedding Elpi (2021), https://github.com/LPCIC/coq-elpi
33. 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)
34. Zandberg, K., Baccelli, E.: Minimal virtual machines on IoT microcontrollers: The case of Berkeley Packet Filters with rBPF. In: 2020 9th IFIP International Conference on Performance Evaluation and Modeling in Wireless Networks (PEMWN). pp. 1–6. IEEE (2020)
35. Zandberg, K., Baccelli, E.: Femto-Containers: DevOps on Microcontrollers with Lightweight Virtualization & Isolation for IoT Software Modules (Nov 2021), https://hal.inria.fr/hal-03263164, preprint
36. Zhang, X., Li, Y., Sun, M.: Towards a formally verified EVM in production environment. In: Bliudze, S., Bocchi, L. (eds.) Coordination Models and Languages. pp. 341–349. Springer International Publishing, Cham (2020)

Appendix: source code, proofs and benchmarks

The source code and proofs of our virtual machine, its generated code and benchmark data are available to the CAV'22 evaluation committee on the anonymized repository: https://anonymous.4open.science/r/AnonymousCAV22-59DC.