Making an eBPF Virtual Machine Faster on Microcontrollers: Verified Optimization and Proof Simplification

Shenghao Yuan, Benjamin Lion, Frédéric Besson, Jean-Pierre Talpin

Making an eBPF Virtual Machine Faster on Microcontrollers: Verified Optimization and Proof Simplification

Shenghao Yuan[0000−0002−8467−5827], Benjamin Lion, Frédéric Besson[0000−0001−6815−0652], and Jean-Pierre Talpin[0000−0002−0556−4265]

Inria (firstname.lastname@inria.fr)

Abstract. As a revolutionary kernel extension technology, Berkeley Packet Filters (BPF) has been applied for various operating systems from different domains, from servers (Linux's extended BPF) to microcontrollers (RIOT-OS rBPF). Previous works have formally proved the memory isolation property for the non-optimized rBPF virtual machine in the Coq proof assistant. In this paper, we introduce a verified optimization for rBPF, and highlight a novel proof approach for optimization correctness: a simplification process is first used to transform monadic models with option state to simplified non-monadic models with inline arguments; then the optimization correctness theorem is split into i) proving simplification correct and ii) proving the optimization correctness on simplified models. Our proof approach enjoys a fruitful proof simplification. Preliminary experiments demonstrate satisfying performance.

Keywords: BPF · Optimization · Verification · Monad Simplification.

1 Introduction

One of the worst decision in software engineering is to compromise the correctness of the application in order to gain some runtime performance. As the saying goes, *premature optimization is the root of all evil*. The optimization of a program often comes with trade-off and good practices therefore advocate to first produce a modular and correct application, then looking for places to optimize.

This design strategy has been applied on the rBPF [19] virtual machine, whose correctness has been fully verified in the Coq proof assistant [3]. The correctness proof ensures that the interpreter is isolated: it will not allow rBPF instructions to modify memory regions in which it has insufficient permissions. The defensive function *check_mem* acts as a safeguard to enforce this requirement. The first proof of correctness in [18] defines a verified implementation of the *check_mem* function which checks every memory access at runtime against the list of all memory blocks, regions and permissions registered with the CompCert memory model of the virtual machine. As *check_mem* is often used by the interpreter (typically, every time a memory instruction is called), optimizing its implementation would obviously speed up execution. Since the rBPF instruction

set is used by operating systems for embedded devices and microcontrollers with low energy and resources, both correctness and performance are critical.

The interpreter has been formalized in Coq, candidates for optimization can be formally defined as a new implementation of the *check_mem* function and can be proved to preserve the semantics. The difficulty is to reason compositionally on the correctness of the interpreter with respect to the new *check_mem* function.

The formalization of the rBPF interpreter uses the native *check_mem* function, which is not optimized. For a given rBPF memory instruction, the *check_mem* function iterates over all memory regions, in sequence, to find whether the instruction points to a valid memory region. While such implementation is correct, the performance can be improved by changing the order in which *check_mem* iterates over the memory regions.

In this paper, we consider an alternative implementation of the *check_mem* function as a candidate for runtime optimization. When an rBPF memory instruction is executed for the first time, the *check_mem* function iterates sequentially over the memory regions until the accessed memory is reached, as the non-optimized *check_mem* does. The accessed memory region is stored in a cache, so that when the same instruction is called for a second time, the *check_mem* first looks at the cache before iterating over all the other memory regions.

We provide the following contributions. The first contribution is to prove that the new *check_mem* function is correct, *i.e.,* that the new interpreter performs the same behaviours as the non-optimized one. The main contribution is however the workflow employed to equally optimise and simplify these proofs, by the definition of a series of transformations on the monadic definition of the interpreter. The last contribution is to experimentally demonstrate that the new *check_mem* function is an optimization, by running benchmarks on the optimized and non-optimized interpreters.

The background material is presented in Section 2. The design choices for the new *check_mem* function are presented in Section 3, and the proof of correctness of the new interpreter is given in Section 4. Benchmarks are given in Section 5 and demonstrates that the new function is an optimization for the presented cases. Section 6 presents related works and Section 7 concludes.

2 Background

The CompCert Verified Compiler. CompCert is a C compiler, both programmed and proved correct in Coq, that generates efficient code for the ARM, PowerPC, RISC-V, and x86 processors. Two important CompCert concepts are used in this paper:

- A value $v \in val$ in CompCert can either be a 32-bit $Vint(i)$ or 64-bit $Vlong(i)$ integer, a pointer $Vptr(b, o)$ to a block b with offset o, a floating-point number or the undefined value $Vundef$.
- A CompCert memory m consists of a collection of separate blocks ($b \in block$) with a fixed size.

BPFs. Berkeley Packet Filters [11] (BPF) was designed originally for the purpose of network packet filtering. The Linux community adopted the concept of BPF and implemented eBPF (extended BPF) to run custom in-kernel VM code, for the untrusted kernel extension [6]. eBPF was then ported to micro-controllers, yielding RIOT-OS [2]'s specification: rBPF.

Both eBPF and rBPF take eBPF binary programs as input because the rBPF ISA is a subset of the eBPF one. The main difference is:

- eBPF depends on a sophisticated online verifier (*e.g.,* [7]) to guarantee security, *e.g.,* a binary eBPF program can not cause any memory leaks.
- As an MCU architecture cannot host such a large verifier, rBPF has a tiny verifier and adopts dynamic defensive strategies in its interpreter to ensure security. For instance, the rBPF interpreter only allows executing an rBPF memory instruction after the defensive function *chek_mem* checks this memory operation is valid at run-time. This requires rBPF users to additionally declare all memory regions used by their eBPF programs.

CertrBPF [18,20] is a fully verified version of RIOT-OS rBPF that

- proves that rBPF isolates faults in the Coq proof assistant, and
- provides an end-to-end verification workflow to generate verified C programs from the CertrBPF specification in Gallina, the functional language embedded in the Coq proof assistant.

3 Design

This section introduces the *check_mem* optimization algorithm as well as the corresponding implementation.

3.1 High level intuition

Two scenarios for the optimized *check_mem* function are graphically depicted in Figure 1. We detail the steps in each scenario.

The input binary, on the left of Figure 1, contains two memory instructions, at location i and j. The list of memory regions are displayed next to the input binary array, and duplicated for readability (although, in practice, only one array of memory regions exists). The cache, in doted line, is initialized with 0.

When a memory instruction of the input binary is interpreted, the *check_mem* function is called. As the cache is initially empty, the optimized *check_mem* behaves, on the first call, as the non-optimized *check_mem* function and iterates sequentially over the memory regions to find if the instruction is valid. When the function finds the valid memory region (*e.g., mr_a* for the i-th instruction and *mr_b* for the j-th instruction), the cache is updated with the index of that memory region (*e.g., a* for the i-th instruction and b for the j-th instruction).

The second time that an instruction is interpreted, the optimized *check_mem* function behaves differently to the non-optimized *check_mem* function. The memory region that the cache points to is first checked. If that memory region is valid

for the instruction, the cache keeps the reference, and we call such case a *cache hitting*. If not, all other memory regions are checked, in sequence, until one a valid one is found. The cache is updated with the new valid one (*e.g.,* the j-th memory instruction, on the second time, belongs to the memory region mr_c).

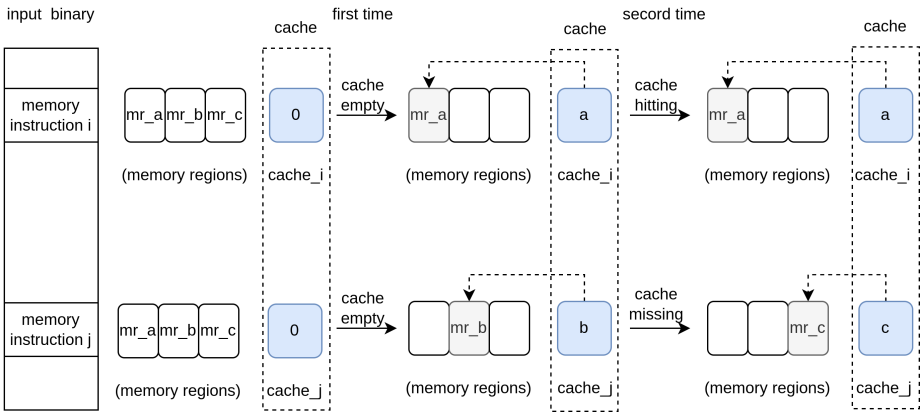

Fig. 1. check_mem optimization

3.2 Optimized check_mem function

We fix *perm*, *chk*, *mrs*, *cache*, *addr*, and *pc*, to respectively denote a CompCert permission, a CompCert memory chunk (size of memory block to access), a list of memory regions, a cache where elements of the list are indexes in the list of memory region, a memory address, and a program counter. We also write $l[n \mapsto v]$ for updating the n-th element of list l with value v.

The new optimized *check_mem* (Algorithm 1) is explained as follows: it first checks if the cache is empty (line 2), if the cache is not empty, the algorithm calls the function *check_one_mem_region* to check if the memory address *addr* is within the history memory region with the index *cache[pc]* (line 3). If the result is valid which represents cache hitting, then the algorithm directly returns *true* and the old cache (line 7). The other two cases cache empty (line 9) and cache missing (line 5) are similar: performing the iteration of all memory regions by the recursive function *check_all_mrs* (line 10).

The iteration is from right to left which is implemented according to the argument n starting from the number of memory regions. When n arrives 0, representing that there is no valid pointers for all memory regions, the *check_all_mrs* function returns $false$ and the old *cache* (line 12). *check_all_mrs* skips the region *cache_id* during the iteration (line 14). If *check_all_mrs* finds that the address *addr* is not within the current memory region indexed by n, it invokes the recur-

sion to check the $(n-1)^{th}$ memory region (line 18). Otherwise, it returns *true* and the updated *cache* with the new memory region (line 20).

The function *check_one_mem_region* is used to calculate if the input address *addr* satisfies that the memory interval $[addr,\ addr + chk)$ is in the range of the *id*-th memory region, it also checks alignment and permission of the interval.

Algorithm 1: The check_mem optimization algorithm

```
   Data: (perm : permission), (chk : memory_chunk), (mrs :
         list memory_region), (cache : list nat), (addr : val), (pc : int)
   Result: (is_valid : bool), (new_cache : list nat)
 1 check_mem :
 2 if cache[pc] ! = null then
 3 |   is_valid ← check_one_mem_region(mrs[cache[pc]], perm, chk, addr) ;
 4 |   if ! is_valid then
 5 |   |   return check_all_mrs(size(mrs), id, perm, chk, mrs, cache, addr, pc);
   |   |    /* cache missing */
 6 |   else
 7 |   |   return (true, cache) ;                          /* cache hitting */
 8 else
 9 |   return check_all_mrs(size(mrs), 0, perm, chk, mrs, cache, addr, pc) ;
   |    /* cache empty */
   Data: (n : nat), (cache_id : nat), (perm : permission), (chk :
         memory_chunk), (mrs : list memory_region), (cache : list nat), (addr :
         val), (pc : int)
   Result: (is_valid : bool), (new_cache : list nat)
10 check_all_mrs :
11 if n = 0 then
12 |   return (false, cache);
13 else if n = cache_id then
14 |   return check_all_mrs(n − 1, cache_id, perm, chk, mrs, cache, addr, pc);
15 else
16 |   is_valid ← check_one_mem_region(mrs[n], perm, chk, addr) ;
17 |   if ! is_valid then
18 |   |   return check_all_mrs(n − 1, cache_id, perm, chk, mrs, cache, addr, pc);
19 |   else
20 |   |   return (true, cache[pc ↦ n]);                   /* cache updating */
```

3.3 Implementation

The Coq development reuses the CertrBPF Gallina specification which makes use of an option-state monad M to capture undefined behaviors (an option type has two cases: $\emptyset$ denoting failure and $\lfloor x \rfloor$ denoting success with result x) and memory-related side effects. We recall the definition of an option-state monad,

for any value type A and state type *state*:

$$M\ state\ A := state \to \mathbf{option}(A \times state)$$

The optimized rBPF interpreter takes a new state that lifts the existing CertrBPF state with an additional field *cache*. We stipulate:

- *cache* represents a list of caches with the same length of the input rBPF binary list, *i.e.,* each memory instruction in the binary list has the same location as the corresponding cache in the *cache* list.
- the value of each cell of *cache* should be within range $[0, mrs\_num]$ where mrs_num is the number of memory regions. Initially, each *cache* cell is empty (0 by default), when it is updated with a *cache_id*, this cache points to the corresponding memory region with index $cache\_id - 1$.

Our new interpreter implementation introduces a flag '$opt\_flag : bool$': when users set the flag with $true$, the optimization is enabled. If $opt\_flag = false$, our specification is equivalent to the formally verified CertrBPF interpreter where *fuel* is for the termination of the interpreter (4096 by default) and *ctx_ptr* points to the start address of a special memory region of rBPF, named context, that is used to store all input values for rBPF programs. The bpf interpreter returns a value from an option-state monad with *val* the type of the bpf interpreter status:

$$bpf\_interp\ (opt\_flag :\ bool)\ (fuel :\ nat)\ (ctx\_ptr :\ val) :\ M\ state\ val.$$

Benefiting from the CertrBPF workflow[18], an executable C program could be automatically extracted from our Gallina model.

4 Proof

The monadic definition of the rBPF interpreter is such that every sub function of the interpreter is a monadic function defined over the same state. The benefit of having a monadic model is that the binding operator of the monad composes the value and threads the state over each function. For instance, given $f_1 : A \to M\ state\ B$ and $f_2 : B \to M\ state\ C$, then the composition is simply the binding of f_1 and f_2, written as $do\ x \leftarrow f_1; f_2$. The drawback, however, is that every function is defined over the same option-state monad with a global state. As a consequence, if the global state changes (*e.g.,* due to an optimization), all invariants have to be proved again. More details are given in subsection 4.1.

Alternatively, each function of the model can be defined over the projection of the state that the function modifies. For instance, if the function f_1 only modifies the substate s of the state *state*, the projection f_1' has the new signature $f_1' : A \times s \to (B \times s)$. The benefit of such approach is that if the global state *state* is modified (*e.g.,* due to an optimization), but the modification is not in s, then the invariant of f_1' still hold, without additional proof. The drawback, however, is that composition of f_1' and f_2' (for some projection of f_2) is no longer as simple

as the monadic binding as the signatures may not coincide. We call *simplification* the transformation that takes a function defined over an option-state monad and returns the projected function as detailed above.

Our strategy is then the following. We keep the monadic model for design, as monadic composition makes specification easier. We use the *simplification* transformation, as detailed in subsection 4.2, on the rBPF interpreter with optimization, and prove the correctness of the *check_mem* optimization. Last, we prove that the simplification is correct with the equivalence proof in subsection 4.3.

4.1 Challenge

Following the workflow of CertrBPF, our new proof model adopts the monadic form with an option *state* monad M. The standard refinement proof adopted by CompCert and CertrBPF is to prove the theorem *optimization_correctness* that the two models (non-optimized and optimized) preserve a proper forward simulation relation.

The simulation relation $match\_states \subseteq state \times state$ is straightforward: the equality between all other fields in two states, except for the *cache*.

$$match\_states(st_1,\ st_2) \stackrel{def}{=} \bigwedge \begin{cases} st_1.pc = st_2.pc \\ st_1.flag = st_2.flag \\ \dots \end{cases}$$

```
Theorem optimization_correctness: ...
  (Hsim: match_states st1 st2)
  (Hinterp: bpf_interp false fuel input_v st1 = ⌊(res, st1')⌋),
  ∃ st2',
    bpf_interp true fuel input_v st2 = ⌊(res, st2')⌋ /\
    match_states st1' st2'.
```

Theorem *optimization_correctness* only considers the case when the monadic interpreter returns successfully because the existing isolation proof [18] guarantees the monadic CertrBPF interpreter (*i.e.*, $opt\_flag = false$) never crashes.

We illustrate the challenge of directly proving the correctness of the optimization within our monadic model and present our solution in the next section.

We first introduce the function tree of our Gallina model because the proof process of the theorem follows this tree structure. We highlight four nodes of the function tree, as depicted in Figure 2:

- `bpf_interp`: The top function of our new CertrBPF model;
- `step`: It interprets single rBPF instruction with an initial state, and results a new state;
- `upd_reg`: The leaf node that updates the register map field of the global monad state;

– `check_mem`: The key function where the $opt\_flag = true$ branch enables the *check_mem* optimization. We underline that the optimized case and the non-optimized one return different states as the former can modify the cache field of the global state.

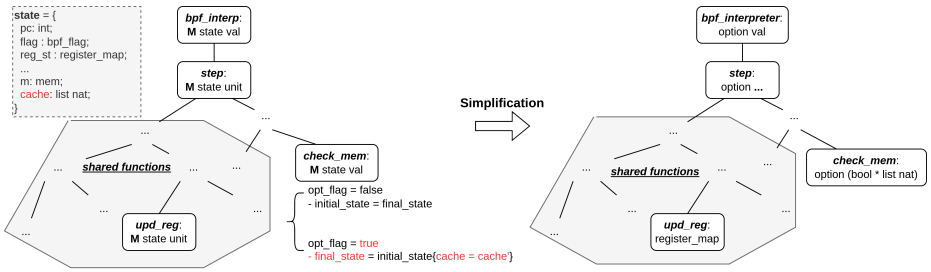

Fig. 2. Function tree of the optimized interpreter. Left is the monadic model with the global state. Right is our simplified model. The grey region includes the shared functions whose simulation proofs can be eliminated on the right side.

To complete this standard state-based refinement proof, it then requires a forward simulation proof of each node in Figure 2. The main reason is that all monadic functions in the tree share the same global state and `check_mem` has different effects on this state, depending on the optimization flag opt_flag. The existing experience from CertrBPF has shown evidence that this way is quite complex and spends a lot of time proving many trivial but very detailed lemmas.

There are several potential ways that may tackle the challenge.

– *dependent type*: CertrBPF adopts 'implementation-proof-separation' way that this proof model doesn't contain any proof invariant. We could try to re-implement CertrBPF by embedding some invariant in a dependent-type form, and it may simplify the final simulation proof.
– *monadic laws*: Our proof model is monadic, therefore we could consider proof based on monadic laws. Similar to monad transformer in Haskell, we define monad projection and injection, along with the related laws, in order to declare the side-effect on the global state of each sub functions *e.g.,* `upd_reg` and `check_mem`.

Our solution is much more direct: we forget the monadic structure, and replace the global state with proper inline arguments, by a so-called *simplification* process. It results in a fruitful proof simplification of the final theorem: we omit the simulation proof of most shared functions since they have the exactly same behavior in both optimized and non-optimized models.

4.2 Simplification

The *simplification* is to

– replace the global state with the *essential* local parameters derived from fields of the state;
– replace the monad along with monad operations to the normal Gallina functions, and
– simplify the simulation relation (`match_states`) to the equivalence between outputs of models, in our cases, they are the non-optimized model and the *check_mem* optimized model.

As depicted in Figure 3, the *simplification* process consists of four steps:

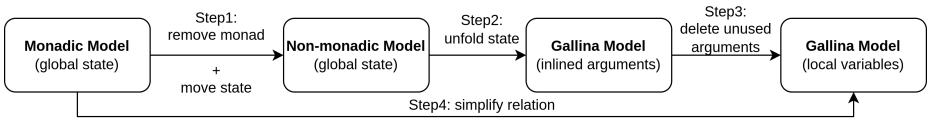

Fig. 3. The simplification process.

$$
\begin{aligned}
& f\ (\_ : a) : \ M\ b \\
(step1) \quad & \Rightarrow f_1\ (\_ : a)\ (\_ : state) : option\ (b \times state) \\
(step2) \quad & \Rightarrow f_2\ (\_ : a)\ (\_ : t_1)\ (\_ : t_2)\ \ldots\ (\_ : t_n) : option\ (b \times t_1 \times t_2 \times \ldots \times t_n) \\
(step3) \quad & \Rightarrow f_3\ (\_ : a)\ (\_ : t_j)\ (\_ : t_j)\ \ldots : \ option\ (b \times t_i \times t_j \times \ldots) \\
(step4) \quad & (f, f^{opt}) \in R \Rightarrow (f_3, f_3^{opt}) \in R_{simpl}
\end{aligned}
$$

Step1: remove monad along with standard (and non-standard) monad operations. We unfold our option state monad M and its operations, then we move the initial state to an argument for readability: this syntax-level transformation doesn't modify the semantics of Gallina programs.

Step2: replace the global state with inline arguments. Assume the global *state* contains several fields with the signature $t_1 \times t_2 \times \ldots \times t_n$, we construct a new function by:

1. unfold the initial state as a list of arguments with types $t_1,\ t_2,\ \ldots\ , t_n$, and
2. replace the final state with the projection of all its components.

Step3: bottom-up delete unused arguments and outputs. According to the function tree of our monadic model, we remove all unused arguments and reduce fields of outputs starting from leaf nodes. Only modified components allow to output and their types are reserved in the type signature.

For instance, the initial monadic functions (see comments) and the final simplified versions of `upd_reg`, `step`, and `bpf_interpreter` are shown as follows:

– `upd_reg` only modifies the register map *regmap* field in the global state, therefore we only reserve this type in the input and output.

- `step` takes most fields of the monadic state, except for the bpf flag, because we only execute *step* when the interpreter status is normal. `step` returns a new pc (affected by *Branch* instructions), a new register map (modified by most instructions, *e.g.,* alu), a new CompCert memory (because of *Store* instructions), a bpf flag (produced by *e.g.,* div-by-zero, etc), and a new cache (updated by the *check_mem* optimization). Some fields are not returned, such as the binary instruction list l, the input bianry size *len* and the memory region list *mrs*, which naturally represents they are unchanged by `step`.
- `bpf_interpreter` takes all components of the global state as parameters and returns the final result if successful.

```
(**r Definition upd_reg (r: reg) (v: val) : M state unit := ... *)
Definition upd_reg (r: reg) (v: val) (rs: regmap): regmap := ...

(**r Definition step (opt_flag: bool): M state unit := ... *)
Definition step (opt_flag: bool) (pc: int) (cache: list nat)
  (l: list int64) (len: nat) (rs: regmap) (mrs_num: nat)
  (mrs: MyMemRegionsType) (m: mem):
    option (int * regmap * mem * bpf_flag * list nat) := ...
  match ins with
  | Alu  ... => match step_alu ... with
                | ∅ => ∅
                | ⌊(rs', f)⌋  => ⌊(pc, rs', m, f, cache)⌋
                end
  ...
(**r Definition bpf_interpreter (fuel: nat) (ctx_ptr: val)
       (opt_flag: bool): M state val := ... *)
Definition bpf_interpreter (opt_flag: bool) (fuel: nat) (ctx_ptr: val)
  (pc: int) (cache: list nat) (l: list int64) (len: nat)
  (rs: regmap) (mrs_num: nat) (mrs: MyMemRegionsType) (m: mem):
    option (val * int * regmap * mem * bpf_flag * list nat) := ...
```

Step4: simplify the simulation relation. Since the simplified model doesn't have any states, the simulation relation R can be equivalently replaced by a much simpler and more intuitive input-output relation R_{simpl}: the simulation relation of a pair of initial states of function f and f^{opt} is transformed into an input relation that the simplified f_3 and f_3^{opt} have the same input value of simplified arguments; the simulation of finial state is also translated into the relation that f_3 and f_3^{opt} have the same value for simplified output fields.

For example, the lemma *step_preserves_simulation_relation* declares an input-output relation derived from *match_states*.

- *input*: The initial simulation relation is replaced by the constraint that all inline input arguments (pc, register map, CompCert memory, and bpf flag) must be identical for the optimized function ($opt\_flag = true$) and the non-optimized one.
- *output*: The final simulation relation is expressed as two parts:

- *Explication*: all fields that are used in *match_states* also exist in the output should be identical, *e.g.,* the new pc value pc'.
- *Implication*: all fields that do not appear in the output are unmodified, *e.g.,* the read-only memory region list mrs.

```
Lemma step_preserves_simulation_relation: ...
  (Hstep: step false pc cache l len rs mrs_num mrs m = ⌊(pc', rs', m',
  ↪  f', cache')⌋),
  ∃ cache1,
    step true pc cache l len rs mrs_num mrs m = ⌊(pc', rs', m', f',
    ↪  cache1)⌋.
```

Last, the construction of the final state is replaced by finding a new *cache*.

4.3 Equivalence Proof

This section mainly discusses two theorems:

– *The simplification is correct* (Theorem 1): prove the equivalence between the initial monadic models and the simplified models;
– *The check_mem optimization is correct* (Theorem 2): prove that the non-optimized model and the optimized model are equivalent.

Theorem 1 (Simplification Correctness). *Assume the monadic interpreter* ***bpf_interp*** *takes initial state* $st1$ *and successfully outputs result* res *and final state* $st2$*, the simplified version* ***bpf_interpreter*** *passes all fields of* $st1$ *as arguments, it returns the same result.*

```
Theorem simplification_correctness: ...
  (Hinterp: bpf_interp opt_flag fuel ctx_ptr st1 = ⌊(res, st2)⌋),
    bpf_interpreter opt_flag fuel ctx_ptr (pc_loc st1) (cache st1)
    (ins st1) (ins_len st1) (regs_st st1) (mrs_num st1)
    (bpf_mrs st1) (bpf_m st1) =
      ⌊(res, pc_loc st2, regs_st st2, bpf_m st2, flag st2, cache st2)⌋.
```

Proof. The key part is that the monadic *check_mem* function may modify the global state, therefore a related lemma is proved to show this function has no effect on all other fields that are used by all consequent monadic functions.

Theorem 2 (Optimization Correctness). *Assume the simplified rBPF interpreter accepts the same arguments, the one disabling the* ***check_mem*** *optimization returns the same result as another enabling the optimization.*

```
Theorem optimization_correctness_simpl: ...
  (Hmem_disjoint: memory_regions_disjoint mrs_num mrs m0)
  (Hcache_inv: cache_inv cache l mrs_num)
```

```
(Hinterp: bpf_interpreter false ... = ⌊(res, pc, rs, m, f, cache)⌋),
∃ cache1,
  bpf_interpreter true ... = ⌊(res, pc, rs, m, f, cache1)⌋.
```

This theorem requires two assumptions:

- the user declared memory regions *mrs* are disjoint: there is no overlap between any two memory regions. This one is directly derived from the memory invariant of the original isolation proof.
- Due to the new field *cache*, an additional invariant is used to formalize the stipulation mentioned in subsection 3.3 for proving Theorem 2: *cache_inv* specifies the length of the cache list and the valid range of each element in the list.

Proof. We first case analysis on rBPF instructions, the proof of the non-memory instructions is trivial because both non-optimized and optimized models execute identical behaviors. The memory instruction cases are non-trivial because the *check_mem* function of two models has different behaviors: the optimized model may update *cache*. Therefore we prove an important lemma *check_mem_preserves_simulation_relation* that indicates two models return the same result pointer *ptr* but different *cache*s.

```
Lemma check_mem_preserves_simulation_relation: ...
  (Hcheck: check_mem false mrs_num ... = Some (ptr, cache')),
  ∃ cache1,
    check_mem true mrs_num ...  = Some (ptr, cache1).
```

Next, the *check_mem* lemma proof consists of three cases of the cache in the optimized model:

- *cache not exists* ($cache\_id = 0$): This is the simplest case which represents the cache is empty and the optimized model performs the normal memory checking as same as the non-optimized version, the only difference is that once it finds a valid pointer, it updates the cache with the corresponding memory region index before output. In this case, the proof is trivial.
- *cache exists* ($cache\_id \neq 0$): the optimized model observes the cache is not empty, there are two cases,
  - *cache missing*: The input address is not valid in the corresponding memory region of *cache_id*, therefore the optimized model performs the normal memory checking as same as the non-optimized version but skips the memory region *cache_id* (Algorithm 1: line 16-17). This case proves by induction on the number of memory regions *mrs_num* and returns a new cache.
  - *cache hitting*: The proof is also by induction on *mrs_num*, and this case doesn't modify the cache because the input address is valid in the corresponding memory region of *cache_id*. The proof requires that the non-optimized version is also (and only) valid in this memory region where we use the assumption that memory regions are disjoint.

5 Evaluation

Our experimental objects are the original non-verified rBPF interpreter (*i.e.,* vanilla-rBPF), CertrBPF, and the optimized CertrBPF. Our measurements focus on the memory requirements of the virtual machines and the instruction execution throughput.

5.1 Experimental Evaluation Setup

Our experiments are performed on a *nrf52840dk* support board which uses an Arm Cortex-M4 microcontroller, a popular 32-bit architecture (arm-v7m). The code is compiled using the Arm GNU toolchain version 12.2. The compilation is using level 2 optimization enabled and the following GCC options:

- `-foptimize-sibling-calls`: optimize all tail-recursive calls and in turn, bound the stack usage;
- `-fwrapv`, `-fwrapv-pointer`: both signed and pointer arithmetic wrap around according to the two's-complement encoding;
- `-fno-strict-aliasing`: there is no aliasing assumption.

The last three options are passed for the purpose of avoiding a possible mismatch between the CompCert semantics and the GCC semantics.

5.2 Memory Footprint

We first evaluate the memory footprint of the CertrBPF interpreter and the CertrBPF-opt implementation, 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.

Size	Vanilla-rBPF	CertrBPF	CertrBPF (opt)
Flash	2018 B	**1502** B	2114 B
Stack	356 B	**68** B	96 B

Table 1. Memory footprint of rBPF engines

We compare the required memory by the different implementations in Table 1. In terms of Flash, the *check_mem* optimized CertrBPF increases the footprint because of the additional cache field, compared to CertrBPF, but this optimized version is comparable with Vanilla-rBPF. In terms of Stack, both CertrBPF and optimized CertrBPF have less stack usage, compared to Vanilla-rBPF. One reason is that Vanilla-rBPF has an extra call module that occupies approximately 108 B.

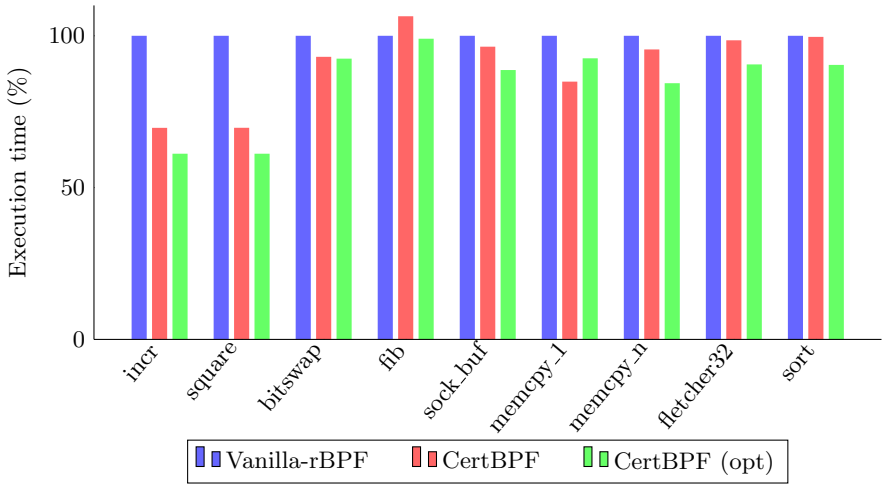

Fig. 4. Performance of runtime (optimzied) CertBPF generated code relative to Vanilla-rBPF generated code on a Arm Cortex-M4 processor. Shorter is better. The baseline, in blue, is Vanilla-rBPF without optimizations. The optimization is in green.

5.3 Experiments

We evaluate the performance of actual benchmarks using CertrBPF (enable and disable the *check_mem* optimization):

- The first four benchmarks test pure computation tasks mainly consisting of rBPF `alu` operations and one extra *exit* instruction (for the purpose of validating the rBPF verifier). These results are averaged over 1000 runs to guarantee accuracy.
- Then, we select three special cases with more memory operations but fewer `alu32` operations: the classical BPF socket buffer read/write, memory copies only 1 element (average over 1000 times), and memory copies many elements.
- Finally, we benchmark the performance of actual IoT data processing using the Fletcher32 algorithm or a sort algorithm.

Summary. As shown in Figure 4, the optimized CertrBPF speedups most of the benchmarks, compared to the original CertrBPF.

For the first four benchmarks, the possible reason for the performance improvement of the optimized CertrBPF may be that GCC with optimizations changes the layout of the final binary forms generated from the source C rBPF interpreters because of the additional cache-related functions. Therefore, the evaluation result produces slight difference, *e.g.,* incr in CertrBPF is 5.750 µs, and incr in CertrBPF (opt) is 5.376 µs.

We observe that the optimized CertrBPF is slower than the original CertrBPF in the case of *memcpy_1*, but still faster than Vanilla-rBPF. This slow-down is caused by copying only one element: it takes the additional expense to update

cache but then exits before benefiting from any acceleration of cache. The benchmark *memcpy_n* (n = 60) shows the optimized version enjoys speedup due to the extra cache. This behavior is also visible in our data processing benchmarks.

Discussion. We could image the worst case of the optimized CertrBPF: one rBPF program frequently switches one pointer to different memory regions, and it results in a lot of cache updating but never cache hitting. For this worst case, users should disable the *check_mem* optimization, another alternative is to design an advanced rBPF verifier that adopts static analysis techniques to determine whether the optimization should be enable or not.

6 Related Work

Monadification. Many existing research works on lifting non-monadic functions into a monadic form, *i.e.,* monadification, for quite various purposes: Martin et al. [5] describe an algorithm to automatically transform non-monadic programs into monadic form for structuring and modularizing functional programs. Simon et al. [17] present a framework in Isabelle/HOL for automatic memoization of recursive functions which uses monadification to produce immediate representation of recursive functions with the state monad. Akira et al. [14] consider the formally-verified transformation from Coq to low-level C code, they propose a monadification algorithm that inserts proper monads into programs for the preservation of critical properties, *e.g.,* absence of overflows.

The simplification presented in this paper is the opposite operation of monadification by forgetting the monadic structure and localizing the global state as proper arguments. This choice makes us benefit from a fruitful proof simplification: most shared functions between the optimized model and the non-optimized one are free of proof.

To the best of our knowledge, the closest related work is AutoCorres[8,9]. AutoCorres is a formally verified tool that abstracts monadic C representations (deep embedding in Isabelle/HOL) into shallow embedding forms in Isabelle/HOL. One of its steps, named 'local variable lifting', translates a monadic C representation into a simplified monadic representation where local variables are lifted out of the program's global state. AutoCorres and our simplification differ in the following ways:

- AutoCorres considers a verified transformation between two monadic representations, while our method targets the non-monadic form; and
- AutoCorres provides an automatic proof for this lifting step, whereas we handle our simplification process manually.

Verified Optimization. There has been a good deal of work on proving the correctness of optimizing transformations for various functional languages, such as CompCert[10], CakeML[12], and CertiCoq[1]. In the context of BPF, Linux eBPF adopts JIT techniques as well as many modern JIT-related optimizations

to accelerate the execution time of eBPF programs. However, existing formal verification research[13] [15] [16] on eBPF JITs primarily focuses on the correctness of the JITs compilation instead of verified optimizations.

When turning to RIOT-OS rBPF, both the unverified Vanilla-rBPF and verified CertrBPF consider fewer (verified) optimizations. To address it, this paper presents a verified *check_mem* optimization for rBPF.

7 Conclusion and Future Work

This paper presents a verified optimization algorithm for the formally verified CertrBPF virtual machine, and introduces a simplification process from monadic functions to non-monadic form for the purpose of simplify the proof that the optimization is correct. The algorithm implementation and proofs of our optimized virtual machine are formalized in Coq and are available on the repository[4].

Next step, we aim at: i) designing an algorithm, similar to AutoCorres, for the automatic simplification of monadic programs; ii) exploring a monadic framework in Coq to prove the correctness of optimized programs in a monad form: we plan a monadic state transformer to only modify state without changing monad, by reusing the existing monad transformer technique.

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. Paris, France (2017)
2. Baccelli, E., Gündoğan, C., Hahm, O., Kietzmann, P., Lenders, M.S., Petersen, H., Schleiser, K., Schmidt, T.C., Wählisch, M.: RIOT: An open source operating system for low-end embedded devices in the IoT. IoT-J **5**(6), 4428–4440 (2018)
3. Bertot, Y., Castéran, P.: Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions. Springer, Berlin, Heidelberg (2013)
4. CertrBPFOpt: A verified optimization of certrbpf (2023), https://gitlab.inria.fr/syuan/certrbpfopt
5. Erwig, M., Ren, D.: Monadification of functional programs. Science of Computer Programming **52**(1), 101–129 (2004). https://doi.org/https://doi.org/10.1016/j.scico.2004.03.004, special Issue on Program Transformation
6. Fleming, M.: A Thorough Introduction to eBPF (2017)
7. Gershuni, E., Amit, N., Gurfinkel, A., Narodytska, N., Navas, J.A., Rinetzky, N., Ryzhyk, L., Sagiv, M.: Simple and precise static analysis of untrusted linux kernel extensions. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 1069–1084. PLDI 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3314221.3314590, https://doi.org/10.1145/3314221.3314590
8. Greenaway, D.: Automated proof-producing abstraction of C code. Ph.D. thesis, UNSW Sydney (2014)

9. Greenaway, D., Lim, J., Andronick, J., Klein, G.: Don't sweat the small stuff: Formal verification of c code without the pain. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 429–439. PLDI '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2594291.2594296, https://doi.org/10.1145/2594291.2594296
10. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**(7), 107–115 (2009)
11. 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, San Diego, California, USA (1993)
12. Myreen, M.O., Owens, S.: Proof-producing synthesis of ml from higher-order logic. In: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming. p. 115–126. ICFP '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2364527.2364545, https://doi.org/10.1145/2364527.2364545
13. Nelson, L., Geffen, J.V., Torlak, E., Wang, X.: Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the linux kernel. In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). pp. 41–61. USENIX Association, USA (Nov 2020), https://www.usenix.org/conference/osdi20/presentation/nelson
14. Tanaka, A., Affeldt, R., Garrigue, J.: Safe low-level code generation in coq using monomorphization and monadification. J. Inf. Process. **26**, 54–72 (2018), https://api.semanticscholar.org/CorpusID:4571133
15. 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)
16. Wang, X., Lazar, D., Zeldovich, N., Chlipala, A., Tatlock, Z.: Jitk: A trustworthy In-Kernel interpreter infrastructure. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). pp. 33–47. USENIX Association, Broomfield, CO (Oct 2014), https://www.usenix.org/conference/osdi14/technical-sessions/presentation/wang_xi
17. Wimmer, S., Hu, S., Nipkow, T.: Verified memoization and dynamic programming. In: International Conference on Interactive Theorem Proving (2018), https://api.semanticscholar.org/CorpusID:14004609
18. Yuan, S., Besson, F., Talpin, J.P., Hym, S., Zandberg, K., Baccelli, E.: End-to-end mechanized proof of an ebpf virtual machine for micro-controllers. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification. pp. 293–316. Springer International Publishing, Cham (2022)
19. Zandberg, K., Baccelli, E.: Minimal virtual machines on IoT microcontrollers: The case of Berkeley Packet Filters with rBPF. In: PEMWN. pp. 1–6. IEEE, Berlin / Virtual, Germany (2020)
20. Zandberg, K., Baccelli, E., Yuan, S., Besson, F., Talpin, J.P.: Femto-containers: Lightweight virtualization and fault isolation for small software functions on low-power iot microcontrollers. In: Proceedings of the 23rd ACM/IFIP International Middleware Conference. p. 161–173. Middleware '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3528535.3565242, https://doi.org/10.1145/3528535.3565242