Verified functional programming of an IoT operating system's bootloader

Shenghao Yuan and Jean-Pierre Talpin
INRIA, IRISA, Rennes, France
firstname.lastname@inria.fr

***Abstract*—The fault of one device on a grid may incur severe economical or physical damages. Among the many critical components in such IoT devices, the operating system's bootloader comes first to initiate the trusted function of the device on the network. However, a bootloader uses hardware-dependent features that make its functional correctness proof difficult. This paper uses verified programming to automate the verification of both the C libraries and assembly boot-sequence of such a, real-world, bootloader in an operating system for ARM-based IoT devices: RIoT. We first define the ARM ISA specification, semantics and properties in F⋆ to model its critical assembly code boot sequence. We then use Low⋆, a DSL rendering a C-like memory model in F⋆, to implement the complete bootloader library and verify its functional correctness and memory safety. Other than fixing potential faults and vulnerabilities in the source C and ASM bootloader, our evaluation provides an optimized and formally documented code structure, a reasonable specification/implementation ratio, a high degree of proof automation and an equally efficient generated code.**

***Index Terms*—verified programming, IoT kernel, case study**

I. Introduction

Among the critical components in the operating system stack of an embedded device, the first whose reliability is put to trial is the bootloader to check, load and execute the image of the operating system or unikernel. Failure to boot renders an embedded device useless, leaving its possibly networked and mission-critical function unattended until maintenance. Well-known mechanisms, such as "Trusted Boot" [1] or "Verified Boot" [2], mainly focus on the validation of loaded images. However, a bootloader is itself a small but complex piece of software, tightly coupled to its hardware platform, making it quite difficult to be verified, especially when hand-written in an unsafe language like C and/or assembly code.

Program verification techniques have become popular to ensure the correctness of programs written in unsafe languages like C. Deductive programming tools would allow one to verify a bootloader at its source C code (maybe not its necessary assembly boot sequence). As such, VCC [3], Verifast [4] and refinedC [5] allow to verify C or Java programs annotated with pre- and post-conditions. These conditions however introduce an heterogeneous syntax that rapidly scales the amount of annotations in proportion of the source code: for instance, a 14 lines long C program may require about 20 lines of annotations [5, Sec. 2.2]. Another choice to formal verify a bootloader is to homogeneously express the implementation and verification conditions in a proof assistant. e.g. SABLE [6], in Isabelle/HOL [7], and the first-stage bootloader [8], in Coq [9]. Although their specification may automatically be generated, for instance by using VST [10], verification conditions still require to undergo a time-consuming process of manual proof. In fact, [9] only proves part of its Sanctum-like bootloader functionally correct.

In this paper, we adopt a verified programming methodology to implement and verify a real-world bootloader. Our approach gains both proof automation while maintaining homogeneous specifications and implementations in the F⋆ [11] programming environment. We implement a verified *riotboot* [12], the bootloader of a friendly Operating System for IoT devices, RIoT [13], together with the ARM Cortex-M architecture of its target platforms.

The source code of *riotboot* is a mixture of C and assembly code. The C code involves the C memory model and, inevitably, pointers. We use Low⋆ [14], a low-level subset of F⋆ that supports the C memory model and enjoys a translation to C that does not require a runtime library using the KreMLin compiler [15]. The hardware-dependent part of *riotboot*, written in assembly code, would not be modeled using existing F⋆ libraries. The most related project, VALE [16], [17], only supports the verification of x84/x64 architectures in F⋆. In this paper, we make the following contributions:

- *ARM-F⋆*: We define a complete F⋆ model of an instruction set available in most ARM platforms, including modes, conditionals and suffixes. Our model includes 1/ the formal syntax and operational semantics of the chosen ISA in F⋆, 2/ a formal specification of its critical properties (as explained in the ARM assembly user guide), and 3/ a series of lemmas in F⋆ to automate verification of programs.
- *Verified riotboot*: We use Low⋆ and our library ARM-F⋆ to model *riotboot* and verify its functional correctness and memory safety in the F⋆ environment. Our workflow contains: 1/ a model of *riotboot*'s C modules in Low⋆ and of its assembly code in ARM-F⋆, 2/ a functional correctness proof of *riotboot* in the F⋆ environment, and 3/ extracted C and assembly code from our F⋆ model.
- *Evaluation*: We show potential faults and vulnerabilities found with our F⋆/Low⋆ model, compare it with existing formally verified bootloaders by highlighting an optimized amount of required specification and, foremost, the high degree of proof automation gained.

As a result, we benefit from bare-metal executable code verified against all critical requirements at minimal specification cost and development time (i.e. one month). Our workflow provides a principled type-driven approach allowing IoT developers to specify and verify system- and application-specific properties in a way that maximizes proof automation while facilitating specification, in ways that could additionally be minimized by using static analysis, or be extended to deal with physical and hardware constraints [18].

The rest of the article is organized as follows. Section II gives a short introduction to verified programming in F⋆. Section III briefly introduces the modules of Riotboot. Section IV formalizes the ARM assembly documentation [19] in F⋆. Section V specifies *riotboot* in F⋆/Low⋆, verifies its functional correctness, and evaluates our model. Sections VI-VII conclude by discussing related and future works.

II. A brief overview of F⋆ and Low⋆

F⋆ is a general-purpose functional programming language that, in the spirit of Liquid Haskell or Agda, is meant at verifying programs. In this aim, F⋆ supports a dependent-type system allowing to express type refinements of both pure and imperative functions with logical properties pertaining to their value domain, pre- and post-conditions. For instance, the type of the `tot`(al) function `abs` accepts any integer and returns its absolute value: `v:int -> Tot v:int{v>=0}` is its type. The `st`(ateful) function `get`, reading the value `v` of a reference `r` in the memory heap `h` has type `r:ref a -> ST v:a`, pre-condition `requires fun h -> (contains h r)`, saying that `h` musts contain `r`, and post-condition `ensures fun h v _ -> v = (sel h r)`, saying that the returned value `v` is exactly that of the `sel`(ected) heap location. Low⋆ can be seen as a domain-specific language embedded in F⋆ whose purpose is to render the computational model of imperative system languages like C. As a result, Low⋆ enjoys the powerful specification and proof capabilities of F⋆ while being able to generate verified C code readily usable without resource-hungry runtime library.

Subroutines used during the image validation process are typical Low⋆ programs. For instance, considering function `rb_hdr_t2uint16_t` in details, it marshals header `struct`(ure) into an `uint16_t` buffer for input to the `fletcher32` image validation algorithm. It consists of a type and logical specification with a `val` declaration, and an implementation with a `let` declaration. It takes two arguments `s` and `d` whose types are specified between arrows: `rb_hdr_t` and `d:B.buffer UInt16.t`. The first one is just the datatype of a *riotboot* header data-structure. The second is a Low⋆ buffer (i.e. B.buffer) containing 16bits unsigned integers with type refinement behind brackets `{}` saying that the length of buffer `B.length d` should be larger than the value of the constant `UInt16.v offset_chksum`.

```
val rb_hdr_t2uint16_t: rb_hdr_t->d:B.buffer UInt16.t
    {B.length d> UInt16.v offset_chksum}->ST unit
 (requires (fun h0 -> B.live h0 d))
 (ensures (fun h0 v h1 -> (M.modifies
    (M.loc_buffer d) h0 h1) /\ B.live h1 d))
let rb_hdr_t2uint16_t s d =
 d.(0ul)<-uint32_to_uint16(s.magic_number);
 d.(1ul)<-uint32_to_uint16(s.magic_number >>^ 16ul);
 .../...;
 d.(5ul)<-uint32_to_uint16(s.start_addr >>^ 16ul);
 ()
```

The function body behind the `let` sequentially marshals the raw header data from `s` into the buffer `d` by performing a series of assignments and shifts. The function returns nothing, but has side-effects: it populates buffer `d`. Its assumptions and guarantees are specified by predicates in the monad `ST`. The pre-condition is defined by a function with the initial memory state `h0` as a parameter. By stating `B.live h0 d`, it requires `d` to be a live memory area in `h0`. The post-condition is stated as a function taking the result `v` and initial and final memory states `h0` and `h1` as parameters. It says that the function returns a modified and live memory buffer `d`. To obtain this guarantee, the effect of each statement in the sequence is collected from a sentence to the next one by using monadic binding. This propagated information is then checked against the declared post-condition.

III. Riotboot overview

The bootloader of RIoT: *riotboot*, expects flash memory to be supplied and formatted in slots to host operating system images. The core of *riotboot* consists of two modules: *choose_image* and *cpu_jump_to_image*.

```
1  void kernel_init(void){
2   uint32_t version = 0; int slot = -1;
3   for (unsigned i = 0;i < riotboot_slot_numof;i++){
4    const riotboot_hdr_t *riot_hdr=
       riotboot_slot_get_hdr(i);
5    if (riotboot_slot_validate(i)){continue;}
6    if (riot_hdr->start_addr !=
       riotboot_slot_get_image_startaddr(i)){continue;}
7    if (slot == -1 || riot_hdr->version > version)
8     {version = riot_hdr->version; slot = i;}
9   }
10  if (slot != -1) { riotboot_slot_jump(slot); }
11  while (1) {} }
```

Function `choose_image` consists of a for-loop that chooses a suitable image from a list of slots in flash memory. It first selects an image header in that list (lines 3-4) and validates its header (line 5) using the *fletcher32* checksum algorithm (below). If no valid image is present, *kernel_init* falls into an infinite loop (line 11) whose behavior may actually be reduced to *nop* by an optimizing compiler. If several valid images are present in the list, it chooses that with the latest version number (line 7).

Function `cpu_jump_to_image` is written in Cortex-M assembly code and performs a "long jump" to execute the imaged system. Line 2 sets the stack pointer (MSP) to the image address. Line 3 skips the image header. Lines 4-5 set the destination address and force the processor state to `Thumb` mode. Finally, line 6 branches execution at the destination. Such operations cannot be performed in a system language: a tempting `(*image_addr)()` in C would result in sharing the memory space of the bootloader with the image.

```
1 static inline void cpu_jump_to_image(uint32_t
      image_addr) {
2   __set_MSP(*(uint32_t*) image_addr);
3   image_addr += 4;
4   uint32_t destination = *(uint32_t*) image_addr;
5   destination |= 0x1;
6   __asm("BX %0" :: "r" (destination)); }
```

Our workflow starts with the definition of the ARM ISA in F⋆: its syntax, operational semantics and properties Then, the *choose_image* function is modelled in F⋆/Low⋆ and the *cpu_jump_to_image* function is expressed in ARM-F⋆. The model's functional correctness is automatically verified by F⋆

using the Z3 SMT-solver, and the verified model is used to extract executable C code by the Kremlin compiler and ASM assembly code using the ARM-F⋆ print module. The synthesis of extraction result finally produces a verified `riotboot`.

IV. Formalizing the ARM ISA in F⋆

This section selects a general ARM instruction set, defines its syntax and semantics and proves its properties derived from the ARM ASM user guide to provide useful lemmas.

A. Syntax

The syntax of the ARM assembly language is shown in Fig.1. It comprises of three kinds of instructions[1].

- Twelve arithmetic instructions from the 'Add with Carry' **adc** to the 'Store' instruction **str**.
- The logical instructions **mov** and four bitwise operations: conjunction **and**, disjunction **orr** and **orn**, exclusion **eor**.
- The shift instructions: Arithmetic Shift Right **asr**, Logical Shift Left **lsl**, Logical Shift Right **lsr** and Rotate Right **ror**.

$$
\begin{array}{rlll}
ci & ::= \{cond\}\ i \mid \{s\}\ i \mid \{s\}\ \{cond\}\ i & & \\
i & ::= \mathbf{adc}\ r_d\ r_n\ op_2 & \mid \mathbf{add}\ r_d\ r_n\ op_2 & \mid \mathbf{bx}\ r_d \\
 & \mid \mathbf{cmn}\ r_n\ op_2 & \mid \mathbf{cmp}\ r_n\ op_2 & \mid \mathbf{ldr}\ r_d\ r_n\ o \\
 & \mid \mathbf{mul}\ r_d\ r_n\ r_m & \mid \mathbf{neg}\ r_d\ r_n & \mid \mathbf{nop} \\
 & \mid \mathbf{sub}\ r_d\ r_n\ op_2 & \mid \mathbf{str}\ r_d\ r_n\ o & \\
 & \mid \mathbf{and}\ r_d\ r_n\ op_2 & \mid \mathbf{eor}\ r_d\ r_n\ op_2 & \mid \mathbf{mov}\ r_d\ op_2 \\
 & \mid \mathbf{orn}\ r_d\ r_n\ op_2 & \mid \mathbf{orr}\ r_d\ r_n\ op_2 & \mid \mathbf{asr}\ r_d\ r_n\ r_s \\
 & \mid \mathbf{lsl}\ r_d\ r_n\ r_s & \mid \mathbf{lsr}\ r_d\ r_n\ r_s & \mid \mathbf{ror}\ r_d\ r_n\ r_s \\
cond & ::= EQ \mid NE \mid CS \mid CC \mid MI \mid PL \mid VS \mid VC \mid LT & & \\
 & \mid LE \mid GT \mid GE \mid AL & & \\
r & ::= r_0 \mid r_1 \mid \ldots \mid r_{12} \mid sp \mid lr \mid pc & & \\
op_2 & ::= c \mid r \mid r\ sop & & \\
sop & ::= ASRshift\ sh_2 \mid LSLshift\ sh_1 & & \\
 & \mid LSRshift\ sh_2 \mid RORshift\ sh_1 & & \\
sh_n & \in [1, 30+n],\ \ o, c\ \in\ Int32,\ \ s\ \in\ String & &
\end{array}
$$

Fig. 1. Core syntax of the ARM assembly language

Conditional instructions execute when a condition flag is set by a prior instruction. A compound conditional instruction can be built by composing a simple one with a condition or a suffix or both condition and suffix.

Conditional code `cond` defines the condition that must be met for an instruction to execute. It can be equal (`EQ`), unequal (`NE`), negative (`MI`), positive or zero (`PL`), etc.

Optional suffix, if specified, sets the condition flag after the instruction is executed. Otherwise, the instruction has no effect on the condition flags.

General purpose registers are r_0-r_{12} and three special registers:the stack pointer register (`sp`), the link register(`lr`) and the program counter register (`pc`). The Application Program Status Register (APSR) holds the program status flags. The suffix of a register appearing in an instruction has a specified meaning. For instance, r_d stands for the destination register and r_n represents the register holding the first operand.

[1]The classification follows the same flags update principle: *str* and *adc* use the same function to update flags, while *mov* and bitwise instructions adopt another. Please refer to the ARM ASM user guide for details.

Operands and shifts are found as second operand op_2 of many ARM arithmetic and logical instructions. They can be a constant c, a register r or a register with a shift value.

B. Machine state

In the spirit of the VALE project [16], [17], we represent the machine state as a record (`arm_state`) consisting of:

- memory, as a map from physical addresses to bytes,
- registers, as functions mapping register names to values,
- status flags, as the negative (`N`), zero (`Z`), carry (`C`), and overflow (`V`) condition flags of the APSR register.
- ISA modes ARM, Thumb16 and Thumb32,
- and a Boolean field `ok` representing the processor state.

A valid state (`ok = true`) indicates that the machine has safely executed until the current state. For instance, a valid state ensures that no segmentation fault occurred. An invalid memory access or update would make the state invalid (`ok = false`) and stop the execution of the machine.

```
type addr = int32
type mem_entry = | Mem32 : v:int32 -> mem_entry
type memory = FStar.Map.t addr mem_entry
type arm_state = {
    regs      : reg -> int32;
    flags     : flag; (*i.e. the APSR register*)
    mem       : memory;
    isa_mode  : mode;
    ok        : bool; }
```

C. Operational Semantics

We now define the operational semantics of key instructions from Fig. 1 in F⋆ by employing the methodology of VALE. The complete definition of the operational semantics of ARM instructions can be found in a GitLab repository [20]. Firstly some auxiliary functions are defined to check the validity of ARM instructions (as per the reference manual [19]), as shown in the Figure 2. Then the rules of the operational semantics are defined, as shown in Figure 3.

a) Valid functions: Most ARM instructions have constraints regarding the usage of registers and operands, e.g., the destination register of most instructions can not be the program counter. This paper defines validity functions to constrain the parameters of each instruction. In F⋆ and VALE, they can be modeled as predicates and enforced as pre-conditions to using the instructions. Fig. 2 defines the valid usage of the destination register (r_d) for a given instruction i and the current mode `m`: `valid(i,` r_d`, m)`. For instance, the addition with carry instruction in mode $Thumb_i$ ($i \in \{16, 32\}$) cannot use *pc* or *sp* as destination register. It can only use *pc* in $Thumb_{32}$ mode with a constant operand $op_2 \in [0, 4095]$.

The *exception_pc* function says that r_d can be *pc* only for the Thumb32 ADD instruction and with a constant op_2 in range 0-4095. The definition of *exception_pc* and all other validity predicates can be found in Appendix A. They are:

- `valid(i,`r_n`,m)` defines a similar constraint for r_n: using *pc* or *sp* as 1^{st} operand in most instructions is invalid.
- `valid(i,`op_2`,m)` defines the validity condition for operand op_2, e.g., the register may not be *pc* or *sp*, the shift register may not be *pc*.
- `valid(o,m)` defines the range of the offset appearing in the instructions depending on the memory mode.
- `valid(i,m)` says there is no ARM or 16-bit Thumb `ORN`.

$$valid(i, r_d, m) \overset{\text{def}}{=} \begin{cases} r_d \neq pc & \text{if } i = \textbf{adc} \text{ and } m = ARM \\ r_d \neq pc \;\&\&\; rd \neq sp & \text{if } i = \textbf{adc} \text{ and } m = Thumb_i \\ r_d \neq pc \;\&\&\; rd \neq sp & \text{if } i = \textbf{add} \text{ and } m = ARM \mid Thumb16 \\ exception\_pc(i, r_d) & \text{if } i = \textbf{add} \text{ and } m = Thumb32 \\ .../... \end{cases}$$

Fig. 2. The valid function of the destination register

b) Semantics: We first introduce the key operational semantics rules of the simple ARM instructions used for the bootloader, i.e., **add**, **bx**, **mov**, **orr**. Then, we introduce a special rule: the *memory_unsafe* rule. Fig 3 exemplifies rules for selected instructions. The complete set of rules can be found in Appendix A (some are too large).

$$\frac{st.ok \wedge valid(rd) \wedge valid(rn) \wedge valid(op_2)}{(ADD\ rd\ rn\ op_2, st) \rightarrow st[rd/[[rn]] + [[op_2]], pc/_{[[pc]]+1}]}(add)$$

$$\frac{st.ok \wedge [[rd]].bit(0) = 0 \wedge valid(rd)}{(BX\ rd, st) \rightarrow st[st.isa\_mode/Thumb_{16}, pc/[[rd]]]}(bx1)$$

$$\frac{st.ok \wedge [[rd]].bit(0) = 1 \wedge valid(rd)}{(BX\ rd, st) \rightarrow st[st.isa\_mode/_{ARM}, pc/[[rd]]]}(bx2)$$

$$\frac{st.ok \wedge valid(rd) \wedge valid(op_2)}{(MOV\ rd\ op_2, st) \rightarrow st[rd/[[op2]], pc/[[pc]] + 1]}(mov)$$

$$\frac{st.ok \wedge valid(rd) \wedge valid(rn) \wedge valid(op_2)}{(ORR\ rd\ rn\ op_2, st) \rightarrow st[rd/[[rn]] \mid [[op_2]], pc/[[pc]] + 1]}(orr)$$

$$\vdots$$

Fig. 3. Semantics of the simple ARM instruction set

All rules satisfy two preconditions: the memory flag *ok* is true and all operands are valid. Then,

(add) adds the values in r_n and op_2, stores the result in r_d and updates the *pc* register.

(bx*) causes a branch to the address stored in r_d and switches the instruction set:

(bx1) If bit(0) is 0, then the processor changes to ARM state,

(bx2) if bit(0) is 1, the processor remains in Thumb state.

(mov) copies the value of op_2 into r_d and updates the *pc* register.

(orr) performs a bit-wise OR operation on r_n and op_2, stores the result in r_d and updates the *pc*.

If an operand *op* of an instruction *i* is invalid, the memory flag is cleared (*ok=false*). If the memory flag is false, the processor aborts.

$$\frac{\neg st.ok \vee \neg valid(op)}{(ins, st) \rightarrow \textbf{abort}} \quad (memory\_unsafe)$$

1) Conditional Instructions: have the form '$\{cond\}$ i' (Sec. IV-A). Conditional execution of ARM instructions can reduce the number of branch instructions to improve code density and save computation overhead. All simple ARM instructions can be executed conditionally by relying on the condition code *c* of the instruction and the value of the condition flags in the APSR, i.e. the memory state *st*. To introduce the semantics of conditional instructions, we first define function *cond(c,st)*, Fig. 4.

$$cond(c, st) \overset{\text{def}}{=} \begin{cases} (st.flags).z & \text{if } c = EQ \ (*Equal*) \\ not\ ((st.flags).z) & \text{if } c = NE \ (*Not\ equal*) \\ (st.flags).c & \text{if } c = CS \ (*Carry\ set*) \\ not\ ((st.flags).c) & \text{if } c = CC \ (*Carry\ clear*) \\ ... \\ true & \text{if } c = AL \ (*Default*) \end{cases}$$

Fig. 4. The *cond* function for conditional instructions

cond(c,st) defines the condition that must be met for an instruction to execute. For instance, code `EQ` expects condition equality, which corresponds to the flag `Z` set to true. Code `NE` expects condition inequality and flag `Z` to false. Hence, a conditional instruction ins demands two rules:

- The *cond_true* rule: if the conditional instruction satisfies both the precondition of the simple instruction rule and $cond(c, st)$ is true, then the conditional instruction performs the expected operation, referred to as ins_{pre} for premises and ins_{post} for conclusion from, e.g., Fig.3.

$$\frac{st.ok \wedge cond(c, st) \wedge (ins_{pre})}{(ins, st) \rightarrow st[ins_{post}]}(cond\_true)$$

- The *cond_false* rule: if the conditional instruction meets the precondition of the simple instruction rule but $cond(c, st)$ is false, then the instruction only updates the value of *pc*.

$$\frac{st.ok \wedge \neg cond(c, st) \wedge (ins_{pre})}{(ins, st) \rightarrow st[pc/_{[[pc]]+1}]}(cond\_false)$$

2) Instructions with Condition Suffix: This section mainly discusses the semantics of instructions with condition suffix, i.e. of the form '$\{s\}$ i'.

Flag update: When suffix *s* is specified, conditional flags are updated after performing the default action of instruction i. The `N` and `Z` flags update according to the result of i, while the other two relate to specific instructions. Three update functions are defined to classify the scenarios:

- The `upd_arithmetic` function updates the four condition flags according to the result of an instruction such as **adc**, **add**, **cmn**, **cmp**, **mul**, or **sub**.
- The `upd_logical` function is used to update N, Z and C flags after performing the **mov** instruction or bitwise instructions such as **and**, **eor** and **orr**.
- The `upd_shift` function updates the three flags when shift operations (i.e. **asr**, **lsl**, **lsr** and **ror**) are performed.

Fig. 5 shows the semantics of the **add** instruction with condition suffix. The complete rules are found in Appendix 9. Compared to rules in Fig. 3, instructions with condition suffixes mainly add flags to the updated memory state. For instance, the *adds* rule calls the *upd_arithmetic* to update the N, Z, C and V flags according to the result.

In addition, if a conditional instruction has both condition and suffix, its semantic rule is composed with the aforementioned ones. For instance, the **add** instruction has two rules,

$$\frac{st.ok \wedge valid(rd) \wedge valid(rn) \wedge valid(op_2)}{(ADDS\ rd\ rn\ op_2, st) \rightarrow st[rd/[[rn]] + [[op_2]], flags/upd\_arithmetic([[rn]] +_i [[op_2]]), pc/[[pc]] + 1]} \quad (adds)$$

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(rn) \wedge valid(op_2)}{(ADDSC\ c\ \ rd\ rn\ op_2, st) \rightarrow st[rd/[[rn]] + [[op_2]], flags/upd\_arithmetic([[rn]] +_i [[op_2]]), pc/_{[[pc]]+1}]} \quad (addsc1)$$

$$\frac{st.ok \wedge \neg cond(c, st) \wedge valid(rd) \wedge valid(rn) \wedge valid(op_2)}{(ADDSC\ c\ \ rd\ rn\ op_2, st) \rightarrow st[pc/_{[[pc]]+1}]} \quad (addsc2)$$

Fig. 5. Semantics of instructions with condition suffix

Fig.5. *addsc1* is derived from the *adds* and *cond_true* rules. *addsc2* is an instance of the *cond_false* rule.

D. Properties of the ARM ISA

Based on the semantics of ARM instructions defined in *ARM-F⋆*, we specify the correctness requirements of isolation, branch, and no-effect listed in the ASM manual [19] and formalize them as `Lemmas`. Doing so allows us to prove the correctness of any ARM assembly code sequence modeled in F⋆. These lemmas are summarized in Tab I.

TABLE I
ASM PROPERTIES FROM THE ARM COMPILER USER GUIDE

Name	Specification
Isolation	A processor in one instruction set state cannot execute instructions from another instruction set.
Branch	Transformations between different modes only depend on the jump instruction [i.e. `BX` in the paper].
No-effect	If the condition test of a conditional instruction fails, the instruction has no effect.

Let i be the instruction that is to be executed next, st_0 be the current memory state, st_1 (i.e. $Eval(i, st_0)$) be the next memory state after executing i, and S_x be an instruction set in x mode. Theorem 1 says that the ARM operational semantics should ensure that the processor never receives any instruction of the wrong instruction set in the current state.

Theorem 1 (Isolation)
If $st_0.isa\_mode = x$, $st_0.ok$ and $st_1.ok$ then $i \in S_x$

In our ARM model, most instructions can execute in ARM, Thumb16 or Thumb32 mode. The exception is the instruction **orn**, only available in the Thumb32 mode, as defined by the $valid(i, m)$ function. So the isolation property (1) is equivalent to saying the validity holds if $st_0.ok$ and $st_1.ok$ are true. This is formalized by the following F⋆ lemma.

```
val isolation: i:ins -> st0:arm_state -> Lemma
  (requires (let st1 = eval_cond_ins i st0 in
                st0.ok=true /\ st1.ok = true))
  (ensures  (match i with
                | ORN ... -> st0.isa_mode == Thumb32
                | _ -> true))
```

Second, we need to prove that the processor only relies on the branch instruction to perform mode transformation. Theorem 2 says that, if executing an instruction results in a memory state of a different mode as the current one, then the instruction can only be the jump instruction **bx**.

Theorem 2 (Branch)
If $st_0.isa\_mode \neq st_1.isa\_mode$ then $i = bx$.

The formalization and proof of this property in F⋆ proceed by case analysis base on the definition of *mode*. The F⋆ lemma below depicts the case of $st_0.isa\_mode = ARM$.

```
val branch__arm: i:ins ->st0:arm_state -> Lemma
  (requires (let st1 = eval_cond_ins i st0 in
    st0.ok == true /\ valid_ins i st0 == true /\
    st0.isa == ARM /\ st1.isa_mode =!= ARM))
  (ensures (match i with
    | BX _ | BXc _ _ -> true | _ -> false))
```

Lastly, Theorem 3 says that, if the condition test of a conditional instruction fails, the instruction 1/ does not execute, 2/ does not write a value in the destination register, 3/ does not change any flag, and 4/ does not raise an exception.

Theorem 3 (No-effect) *Let* c, rd *be the condition code and the destination register (if present) of the instruction* i. *If* $Cond(c, st) = false$, *then* $st_1.pc = st_0.pc + 1$, $st_1.rd = st_0.rd$, $st_1.flags = st_0.flags$ *and* $st_1.ok = st_0.ok$.

We decompose the proof into two lemmas. First, nop_equiv_nopc says that **nop** and **nopc** have the same effect on a memory state. Second, $cond\_fails\_equiv\_nop$ shows equivalence of a failed conditional instruction with **nop**.

```
val nop_equiv_nopc: st0:arm_state -> Lemma
  (requires (st0.ok = true))
  (ensures   (∀ c. eval_cond_ins NOP st0 ==
                eval_cond_ins (NOPc c) st0))
val cond_fails_equiv_nop: i:ins -> st0:arm_state ->
    Lemma (requires (st0.ok = true /\ (valid_ins i
    st0 = true) /\ (getCondition i st0 = false)))
  (ensures (eval_cond_ins i st0 == eval_cond_ins NOP
     st0))
```

nop doesn't have a destination register, therefore we apply the $cond\_fails\_equiv\_nop$ lemma to easily prove 2/. Then, the other part of the property can be proved by applying the aforementioned lemmas and three lemmas below:

- $nopc\_skip$: **nopc** skips and updates the pc.
- $nopc\_flags$: The updated memory state has the same value of flags as the previous state.
- $nopc\_memory\_safe$: The updated memory state is safe.

Along the way, we prove some auxiliary lemmas which will be useful for the verification of our case study. First, we can formalize the memory safety of an executed list of instructions L. Let st_0 be the initial memory state and st_1 (i.e.$EvalL(L, st_0)$) the final memory one, the $list\_ok\ L\ st_0$ function says that no instruction in L generates an exception if its current state is memory-safe.

Lemma 1 (List Memory Safety)
If $st_0.ok$ and $(list\_ok\ L\ st_0)$ then $st_1.ok$.

Assuming a memory-safe initial state, its F⋆ encoding is a lemma stipulating that no instruction in the list produces an unsafe state, and that the final state is memory-safe, by induction on the list L.

```
val list_memory_safety: l:list ins -> st0:arm_state
    -> Lemma
 (requires (st0.ok=true /\ (list_ok l st0)))
 (ensures  (let st1 = eval_list_ins l st0 in st1.ok
    = true))
let rec list_memory_safety l st0 =
  match l with
  | [] -> ()
  | hd :: tl -> let st1 = eval_ins hd st0 in
    list_memory_safety tl st1
```

Additionally useful lemmas apply to specific instructions, such as *nop_equiv_nopc* and the 'load after store' lemma.

Lemma 2 (Load after Store)
Let $st_1 = Eval(\mathbf{str}, r_d, r_n, o, st_0)$ and $st_2 = Eval(\mathbf{ldr}, r_d, r_n, o, st_1)$ then $st_0.r_d = st_1.r_d = st_2.r_d$.

The lemma stipulates that the destination register always remains unchanged when the processor first executes the store instruction *str* with some registers and operands, and only executes the load instruction *ldr* with the same parameters. In F⋆, $load\_after\_store\_aux$ operates on two instructions, the intermediate and final memory states st' and st_1.

```
val load_after_store: rd: reg -> rn:reg -> o:
    operand -> st0:arm_state -> Lemma
  (requires (let (str, ldr, st', st1) =
                load_after_store_aux rd rn o st0 in
                 valid_ins str st0 == true /\
                 valid_ins ldr st' == true /\
                 st0.ok == true))
  (ensures  (let (str, ldr, st', st1) =
                load_after_store_aux rd rn o st0 in
                 eval_reg rd st0 == eval_reg rd st' /\
                 eval_reg rd st' == eval_reg rd st1))
```

V. Evaluation Case Study

Our goal is to specify the *riotboot* protocol and verify its correctness in F⋆. We first give the details of its implementation and verification. Next, we evaluate our formal model from the following perspectives: Bug-fixing/optimization, verification cost, and comparison with existing verified bootloaders.

A. Implementation & Verification

Fig. 6 gives the structure of the *riotboot* specification and details the modular decomposition of its verification. The assumptions (R_i) and guarantees (E_i) of each of these modules w.r.t. functional correctness and memory safety. *riotboot* assumes that the image list `images` is available (R) and guarantees the expected properties (E). The *riotboot* in F⋆ has the same structure as its C version: *choose_image* and *cpu_jump_to_image*.

choose_image has two sub modules that are modeled in Low⋆. *validate_header* matches the image header's checksum to that calculated using the Fletcher32 checksum. *choose_version* is used to select the fletcher32-valid image of latest version and execute it using *cpu_jump_to_image*.

In ARM-F⋆, *cpu_jump_to_image* corresponds to the ARM assembly code below. The input $image\_address$ is stored in $R0$. $i0$ copies the input to $R1$, $i1$ is to set MSP, $i2$ is to skip sp register (by 1 `int32` word instead of 4bits in ASM). $i3$ is to set thumb bit, i.e. bit[0] of $R0$ is 1. $i4$ causes a branch to the address contained in $R0$ and changes the instruction set to Thumb mode.

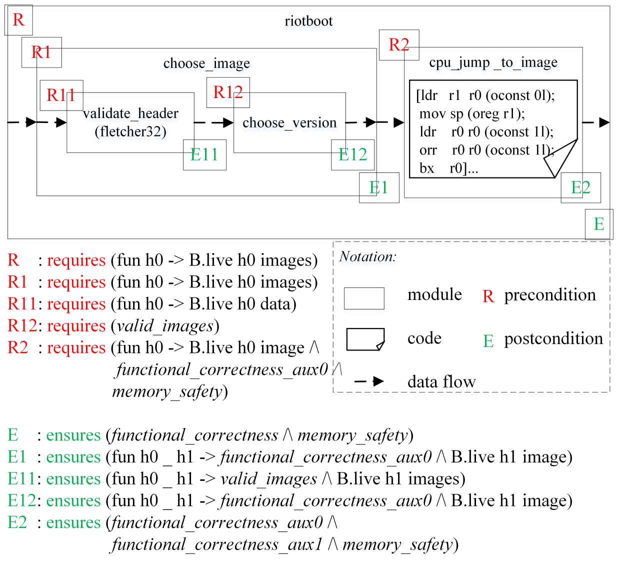
riotboot
R
R1
choose_image
R11
validate_header
(fletcher32)
E11
R12
choose_version
E12
E1
R2
cpu_jump _to_image
[ldr r1 r0 (oconst 0l);
mov sp (oreg r1);
ldr r0 r0 (oconst 1l);
orr r0 r0 (oconst 1l);
bx r0]...
E2
E
R : requires (fun h0 -> B.live h0 images)
R1 : requires (fun h0 -> B.live h0 images)
R11: requires (fun h0 -> B.live h0 data)
R12: requires (valid_images)
R2 : requires (fun h0 -> B.live h0 image /\ functional_correctness_aux0 /\ memory_safety)
Notation:
module
R precondition
code
E postcondition
data flow
E : ensures (functional_correctness /\ memory_safety)
E1 : ensures (fun h0 _ h1 -> functional_correctness_aux0 /\ B.live h1 image)
E11: ensures (fun h0 _ h1 -> valid_images /\ B.live h1 images)
E12: ensures (fun h0 _ h1 -> functional_correctness_aux0 /\ B.live h1 image)
E2 : ensures (functional_correctness_aux0 /\ functional_correctness_aux1 /\ memory_safety)

Fig. 6. The simplified structure of verified *riotboot*

```
let i0: ins = LDR R1 R0 (OConst 0l)
let i1: ins = MOV SP (OReg R1)
let i2: ins = LDR R0 R0 (OConst 1l)
let i3: ins = ORR R0 R0 (OConst 1l)
let i4: ins = BX  R0
let cpu_jump_to_image_ins = [i0; i1; i2; i3; i4]
```

Theorem 4 (Functional Correctness) If *riotboot* finds a suitable image i, then 1/ i should be fletcher32-valid and be latest comparing with all valid images (*functional_correctness_aux0*) and 2/ the registers satisfy $sp = i.start\_addr$, $pc = i.start\_addr \,|\, 0x1$ and the processor mode is *Thumb* (*functional_correctness_aux1*).

Functional correctness is defined by two auxiliary lemmas according to the code structure of *riotboot*: *choose_image* requires the *images* is available (R_1) and ensures the first lemma (E_1), while *cpu_jump_to_image* assumes the liveness of the selected image (R_2) and guarantees the second lemma (E_2). *choose_image* calls the *fletcher32* function to validate an image header, which requires the *fletcher32*'s input *data* is live (R_{11}). Liveness is defined by the post-condition of *rb_hdr_t2uint16_t* (see Sec II). *fletcher32* ensures all returned images are valid (i.e. $valid\_images$ of E_{11}). *choose_version* assumes all input images valid (R_{12}) and guarantees that the selected image has the latest version (E_{12}).

Theorem 5 (Memory Safety) *riotboot* requires an initially safe memory state and yields a safe final memory state.

Since Low⋆'s hyper-stack memory model guarantees memory safety of *choose_image*, we only need to prove that *cpu_jump_to_image* is memory-safe. We use the *list_memory_safety* lemma to help F⋆'s SMT-solver to inductively prove this property.

B. Discussion

Building the *riotboot* case study in Low⋆ based on our ARM-F⋆ opens to interesting discussions regarding the memory models of Low⋆ and the ARM model, the validity of the booted image, and the extracted C and assembly code.

Memory Model: The *choose_image* module is encoded in Low⋆. It is based on its hyper-stack memory model while the *cpu_jump_to_image* function uses the ARM ISA memory model: a map from physical addresses to bytes. Hence a potential problem to compose the specification and factor the verification of two modules with different memory models. Fortunately, the technique of [16] can be reproduced to reconcile them by constraining the interface between the two modules. Finally, verified C code is generated from the Low⋆ implementation of *riotboot*, with its extracted ARM code inlined, constituting a fully verified bootloader implementation.

Validity of the booted image: *riotboot* uses the fletcher32 algorithm to validate the checksum of the selected image. In the case study, a refinement type is introduced to prove the termination of fletcher32. To guarantee functional correctness of the algorithm, a solution is to add a predicate to the postcondition of the Low⋆ code relying on HACSPEC [21] to verify the functional correctness of cryptographic algorithms encoded in a Rust-like specification language from which F⋆ can be generated and used as the basis for proofs. In the present case study, we relied on the verified implementation of the fletcher32 algorithm provided by HACSPEC to trust image validation. Its generated F⋆ code appears in App. C.

Extracted Code: Our hybrid program *riotboot* uses both the Low⋆ language and our ARM assembly model. Code extraction relies on Low⋆'s KreMLin compiler to generate C code and on VALE to generate assembly code. They are composed as a standalone program. Below is the assembly code generated from the *cpu_jump_to_image* module:

```
__asm__ __volatile__ (/* disable optimizations */
  "ldr  r1, [%0] \n\t"      /* r1 = *image_addr */
  "msr  msp, r1 \n\t"       /* MSP = r1 */
  "ldr  %0, [%0, #4] \n\t"  /* r0=*(image_addr+4) */
  "orr  %0, %0, #1 \n\t"    /* sets thumb bit */
  "bx   %0 \n\t"            /* branch to image */
  :                         /* No outputs */
  : "r" (image_addr)        /* input image_addr */
  : "r0"  );                /* r0 may be modified */
```

C. Evaluation

Our F⋆/Low⋆ bootloader implementation relates to the *RIOT* [22] and *RIOT in Rust* [23] projects as part of Inria's *Future-Proof IoT* Challenge [24].

Monadic type checking improvements: We rapidly spotted an infinite loop in the original C&Rust versions of *riotboot* preventing code generation from Low⋆, as it would be given the Div(ergent) monad. Instead, we introduced an if statement (for the case no valid image is found).

```
let kernel_init() = ...
 if B.is_null slot then P.(printf "No valid image.\n
    " done) else (*call the ASM boot sequence*)
```

Strong typing in F⋆ also allowed us to spot and correct comparisons of header sequence numbers (i.e. versions) with header start addresses in the Rust version of *riotboot* [25].

```
pub fn choose_image ...=
  let mut image: Option<u32> = None; ...
  // fix: Option<u32> -> Option<&Header>
  if header.sequence_number <= image ...
  // fix: image -> image.sequence_number
  image = Some(header.start_addr) ...
  // fix: header.start_addr -> header
```

Refinement types also allowed optimizations in *riotboot* while maintaining a verified equivalence with its unoptimized translation. For example, the if-statement on line 6 of *kernel_init* (Sec III) has an unnecessary condition that can be omitted: the left part of the condition is equal to the statement `riotboot_slot_get_hdr(i)->start_addr` according to the definition of *riot_hdr* (line 4). But the right part is also equal to that statement according to the definition of `riotboot_slot_get_image_startaddr`.

```
riotboot_slot_get_image_startaddr(unsigned slot){
  return riotboot_slot_get_hdr(slot)->start_addr; }
```

Verification Cost: But first and foremost, our case study demonstrates that the verified programming workflow presented in the paper has a major impact on validation costs as most verification conditions generated by its type checker can automatically be discharged by F⋆'s companion SMT solver Z3. Verification conditions in *riotboot* are easy to define and express in F⋆/Low⋆. The number of refinement types, pre- and post- conditions we specified are listed in Table II:

TABLE II
DATA STATISTICS OF VERIFICATION CONDITIONS

Module	Refinement Type	Pre-/Post-condition
Choose Image	14	11 / 18
Cpu Jump to Image	0	11 / 26

Refinement types in *riotboot* are used to set the length or scope of some parameters and can be directly derived from the source code. e.g. an input variable *slot*, representing an index in an image table, should be less than the table's length.

```
riotboot_slot_get_hdr(unsigned slot){
  assert(slot < riotboot_slot_numof);  ...
```

Our F⋆/Low⋆ implementation expresses the requirement $index \in [0, length - 1]$ by a refinement type:

```
val choose_image_aux : ... ->
  index:nat{0<=index /\ index <= hdrs_len-1} -> ...
```

Most pre/post-conditions in *Choose Image* concern the liveness of buffer pointers holding image headers. F⋆/Low⋆ requires a pointer to reference a live memory buffer before operation. The preconditions of *Cpu Jump to Image* express this memory safety condition and the post-conditions enforce them for all intermediate states generated by the instructions.

Comparison: Table III compares our F⋆ model with related verified bootloaders.

TABLE III
COMPARISON OF VERIFIED BOOTLOADERS

Name	SourceCode	Model	Proofs	Language
SABLE	600+	250+	400+	Isabelle/HOL
First-stage	200+	n.a.	n.a.	Coq
riotboot	150+	180+	12	F⋆/Low⋆

n.a. no artifact or data available.

To our knowledge, SABLE is the first formally verified bootloader. It uses the methodology of seL4 [26] and adopts the Isabelle/HOL proof assistant. Its source code is over 600 lines and its formal specification 250 lines. The verification effort of SABLE represents more than 400 lines of proof.

The first-stage bootloader, another verified bootloader, formally verifies Sanctum's secure boot [27] (more than 200 lines of C) down to its RISC-V instruction semantics in Coq. Currently, this project is carrying on the whole correctness proof and, at the time of writing, no data or artifact are available for comparison.

Compared with related works, our verified implementation of *riotboot* with about 150 lines of C code in F⋆/Low⋆. The formal specification has a similar code size, and verification benefits a high degree of proof automation using the Z3 SMT solver. To prove the *riotboot* functional correctness and memory safety, only 7 auxiliary lemmas needed to be defined and 12 lines of manual proof declared.

VI. Related Works

A. Bootloaders

Secure boot and trusted boot are two well-known features of bootloaders not to conflate with the verified programming of a bootloader. Secure boot is a valuable feature to help maintain the integrity of a platform at runtime, for example *Android's Verified Boot*. Trusted boot, defined by Trusted Computing Group (TCG), is a process to let a running application check if the system has booted into a trusted environment, e.g., *ARM's Trusted Boot*. While designed with the highest engineering skills, neither secure or trusted boot have provers' verified implementations. In this paper, our goal is to additionally propose a method to guarantee the functional correctness of a bootloader at minor additional engineering costs. Although some tools, like BootStomp [28], allow to identify bootloader vulnerabilities, our method allows to formally verify the absence thereof (up to the considered memory model).

Coreboot [29] is an open-source firmware platform delivering a lightning fast and secure boot. Some libraries of Coreboot, e.g. libgfxinit, written in the SPARK language, can automatically be proved to have no runtime errors, but most of Coreboot, written in C and assembly, is unverified.

Instead, SABLE is a formally verified bootloader developed using Isabelle/HOL. SABLE's method proves that the formalized behavior of bootloader's implementation, in C, satisfies its abstract specification requirements. Compared to our approach, the proof scale is quite important (more than 400 lines of proof) and compilation from C to machine code still remains is unverified (which it could using, e.g., CompCert).

[8] presents the verification of a first-stage bootloader in Coq. The paper considers Sanctum system's bootloader deployed on the RISC-V architectures. One advantage of the approach is to reuse existing Coq projects, for instance the riscv-coq project [30] which implemented the RISC-V ISA specification in Coq. However, this method requires a fully manual proof in Coq, and has, at the time of writing, not delivered a complete and available correctness theorem.

Our method improves related works by employing verified programming to enforce functional correctness properties at compile-time in a way that maximizes proof automation, as presented in Sec V-C.

B. Verified assembly languages

Pioneering works such as CompCert [31], seL4 and Sail [32] have formalized many architecture specifications, such as the x86/x64, ARM and RISC-V ISAs, allowing embedded systems designers to verify the expected properties of low-level programs using the artifacts of these projects, and complete detailed manual proofs using Isabelle/HOL or Coq.

To the best of our knowledge, the closest and only related work to ARM-F⋆, presented in this paper is the verified assembly language environment VALE. VALE is a tool to formally verify high-performance applications written in assembly language by relying on existing verification frameworks, such as Dafny [33] and F⋆. Currently, it includes a limited ARM ISA for the Dafny verification framework and doesn't support the verification of ARM assembly code in F⋆. Our ARM-F⋆ model covers a complete ARM ISA as found in practical applications like *riotboot*, which comprises registers (e.g. *sp*), advanced instructions like **orr** and **bx**, and mode transformations between *ARM* and *Thumb* ISAs. The ARM-F⋆ model also formalizes the correctness requirements listed in the ASM manual and provides both a methodology and useful lemmas for reuse in practical applications.

VII. Conclusion and Future Works

In this paper, we have formalized the ARM instruction set in F⋆, and developed a verified implementation of the RIoT bootloader. Our formalization of the ARM ISA supports a general instruction set available in most ARM platforms, different ISA modes, and conditional instructions with a condition suffix mechanism. We also specify the correctness requirements from the ARM ASM manual and prove them as `Lemmas` in F⋆. Next, we model the RIoT bootloader in Low⋆, and verify functional correctness properties and memory safety of its main components. Our evaluation shows that, not only strong typing in the verified *riotboot* fixes potential vulnerabilities, provides an optimized code structure, but most importantly gains from a high degree of proof automation.

Our next project is to verify RIoT's rBPF subsystem [34] using the same methodology as for *riotboot*. We expect that an F⋆-verified rBPF will provide a more industrial-size experience to highlight the effectiveness of our workflow. Our final goal is to build useful libraries for the F⋆/Low⋆ community to verify low-level embedded programs, and also provide a set of verified subsystems for the RIoT community.

Acknowledgment

The authors are grateful to members of the HACSPEC project for providing the validated and automatically generated F⋆ implementation of the fletcher32 checksum algorithm used in this case study (appendix C)

References

[1] ARM, "Arm trusted firmware," 2021. [Online]. Available: https://github.com/ARM-software/arm-trusted-firmware

[2] GOOGLE, "Verifying boot," 2021. [Online]. Available: https://source.android.com/security/verifiedboot/verified-boot.html

[3] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, "Vcc: A practical system for verifying concurrent c," in *International Conference on Theorem Proving in Higher Order Logics*. Springer, 2009, pp. 23–42.

[4] B. Jacobs and F. Piessens, "The verifast program verifier," Citeseer, Tech. Rep., 2008.

[5] M. Sammler, R. Lepigre, R. Krebbers, K. Memarian, D. Dreyer, and D. Garg, "Refinedc: Automating the foundational verification of c code with refined ownership types," 2021.

[6] S. D. Constable, R. Sutton, A. Sahebolamri, and S. Chapin, "Formal verification of a modern boot loader," Electrical Engineering and Computer Science, Tech. Rep., 2018.

[7] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: a proof assistant for higher-order logic*. Springer Science & Business Media, 2002, vol. 2283.

[8] Z. Straznickas, "Towards a verified first-stage bootloader in coq," Ph.D. dissertation, Massachusetts Institute of Technology, 2020.

[9] Y. Bertot and P. Castéran, *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[10] A. W. Appel, "Verified software toolchain," in *European Symposium on Programming*. Springer, 2011, pp. 1–17.

[11] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang, "Secure distributed programming with value-dependent types," *ACM SIGPLAN Notices*, vol. 46, no. 9, pp. 266–278, 2011.

[12] RIoT, "riotboot," 2021. [Online]. Available: https://github.com/RIOT-OS/RIOT/tree/master/bootloaders/riotboot

[13] E. Baccelli, C. Gündoğan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "Riot: An open source operating system for low-end embedded devices in the iot," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4428–4440, 2018.

[14] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hritcu, K. Bhargavan, C. Fournet, and N. Swamy, "Verified low-level programming embedded in F*," *PACMPL*, vol. 1, no. ICFP, pp. 17:1–17:29, Sep. 2017. [Online]. Available: http://arxiv.org/abs/1703.00053

[15] F. Team, "Kremlin," 2021. [Online]. Available: https://github.com/FStarLang/kremlin

[16] A. Fromherz, N. Giannarakis, C. Hawblitzel, B. Parno, A. Rastogi, and N. Swamy, "A verified, efficient embedding of a verifiable assembly language," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–30, 2019.

[17] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson, "Vale: Verifying high-performance cryptographic assembly code," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 917–934. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/bond

[18] J.-P. Talpin, J.-J. Marty, S. Narayan, D. Stefan, and R. Gupta, "Towards verified programming of embedded devices," in *DATE 2019 - 22nd IEEE/ACM Design, Automation and Test in Europe*. Florence, Italy: IEEE, Mar. 2019, pp. 1445–1450. [Online]. Available: https://hal.inria.fr/hal-02193635

[19] A. Limited, "Arm compiler armasm user guide v5.06," 2016. [Online]. Available: https://developer.arm.com/documentation/dui0473/m

[20] S. YUAN and J.-P. Talpin, "verified riotboot in fstar," 2021. [Online]. Available: https://gitlab.inria.fr/syuan/memocode-riotboot

[21] hacspec, "A specification language for crypto primitives in rust," 2021. [Online]. Available: https://github.com/hacspec/hacspec

[22] RIoT, "Riot-os," 2021. [Online]. Available: https://github.com/RIOT-OS/RIOT

[23] ——, "Riot-rs," 2021. [Online]. Available: https://github.com/future-proof-iot/RIOT-rs

[24] Inria, "Riot-fp," 2021. [Online]. Available: https://future-proof-iot.github.io/RIOT-fp/about

[25] RIoT, "riotboot-rs," 2021. [Online]. Available: https://github.com/kaspar030/riotboot-rs

[26] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "Sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 207–220. [Online]. Available: https://doi.org/10.1145/1629575.1629596

[27] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 857–874. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan

[28] N. Redini, A. Machiry, D. Das, Y. Fratantonio, A. Bianchi, E. Gustafson, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Bootstomp: on the security of bootloaders in mobile devices," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 781–798.

[29] T. C. D. Team, "Coreboot," 2021. [Online]. Available: https://www.coreboot.org/

[30] M. P. Group, "riscv-coq:risc-v specification in coq," 2021. [Online]. Available: https://github.com/mit-plv/riscv-coq

[31] X. Leroy, "The CompCert C verified compiler: Documentation and user's manual," Inria, Intern report, Nov. 2020. [Online]. Available: https://hal.inria.fr/hal-01091802

[32] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell, "Isa semantics for armv8-a, risc-v, and cheri-mips," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3290384

[33] K. R. M. Leino, "Dafny: An automatic program verifier for functional correctness," in *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 2010, pp. 348–370.

[34] K. Zandberg and E. Baccelli, "Minimal Virtual Machines on IoT Microcontrollers: The Case of Berkeley Packet Filters with rBPF," in *PEMWN 2020 - 9th IFIP/IEEE International Conference on Performance Evaluation and Modeling in Wired and Wireless Networks*, Berlin / Virtual, Germany, Dec. 2020, to be published in the proceedings of IFIP/IEEE PEMWN 2020. [Online]. Available: https://hal.inria.fr/hal-03019639

APPENDIX

This appendix lists parts of our implementation of `riotboot` referenced in this article. As already mentioned, the complete implementation can be downloaded from a GitLab repository for evaluation purposes: https://gitlab.inria.fr/syuan/memocode-riotboot.

Section IV-C defines valid functions to describe the constraints of most ARM instructions: The *exception_pc* function says that r_d can be *pc* only for a Thumb32 instruction and with a constant *c* in range 0-4095; The $valid(i, r_n, m)$ function says users are suggested to use *pc* or *sp* as the first operand in most ARM instructions;

The $valid(i, op_2, m)$ function says if the second operand is a register, it should not be *pc* or *sp* (i.e. *reg_not_in_operand* (reg, op_2)), and if it is a register with a shift, the shift register should not be *pc* (i.e. $no\_reg\_shift(op_2)$); The $valid(o, m)$ function says the offset in ARM mode should be in range [-4095,4095], in Thumb32 mode is [-255,4095] and in Thumb16 should be [0,124]; The $valid(i, m)$ says `ORN` is only available in the Thumb32 instruction set.

A. Semantics of the ARM instruction set

This section details the complete operational semantics of the ARM instruction set as outline in Figures 9-11 in the style of a state transition system subject to the validity preconditions.

1) Semantics of the simple ARM instruction set:

Mode: The processor must be in the correct *instruction set state* for the ARM instructions it is executing. ARM instructions are 32 bits wide. Thumb instructions are 16 or 32-bits wide. This paper models three kinds of modes in F⋆:

```
type mode = ARM | Thumb32 | Thumb16
```

Condition Flags: The APSR register is a record (`flag`) holding the negative (`N`), Zero (`Z`), Carry (`C`), and Overflow (`V`) condition flags. The processor uses them to determine whether or not to execute conditional instructions.

```
type flag = {    (*true => 1; false => 0*)
    n : bool;    (*Negative*)
    z : bool;    (*Zero*)
    c : bool;    (*Carry*)
    v : bool;    (*Overflow*)  }
```

$$exception\_pc(i, r_d) \overset{\text{def}}{=} \begin{cases} 0 \leq n \leq 4095 & if\ r_d = pc\ and\ i.op_2 = c \\ false & if\ r_d = pc \\ true & otherwise \end{cases}$$

$$valid(i, r_n, m) \overset{\text{def}}{=}$$
$$\begin{cases} r_n \neq pc\ \&\&\ r_n \neq sp & if\ i = \textbf{adc}\ and\ m = ARM \\ r_n \neq pc\ \&\&\ r_n \neq sp & if\ i = \textbf{add}\ and\ m = ARM \\ \ldots \end{cases}$$

$$valid(i, op_2, m) \overset{\text{def}}{=}$$
$$\begin{cases} \quad reg\_not\_in\_operand(pc, op_2) & \\ \&\&\ reg\_not\_in\_operand(sp, op_2) & if\ i = \textbf{adc}|\textbf{add}\ \ldots \\ \quad reg\_not\_in\_operand(pc, op_2) & \\ \&\&\ reg\_not\_in\_operand(sp, op_2) & \\ \&\&\ no\_reg\_shift(op_2)\ \ if\ i = \textbf{and} & and\ m = ARM \\ \quad reg\_not\_in\_operand(pc, op_2) & \\ \&\&\ reg\_not\_in\_operand(sp, op_2) & \\ \&\&\ 1 \leq i.sh \leq 32\ \ if\ i = \textbf{asr} & and\ m = Thumb_i \\ \quad \ldots \end{cases}$$

$$reg\_not\_in\_operand(reg, op_2) \overset{\text{def}}{=} \begin{cases} true & if\ op_2 = c \\ reg \neq r & if\ op_2 = r || r\ sop \end{cases}$$

$$no\_reg\_shift(op_2) \overset{\text{def}}{=} \begin{cases} r \neq pc & if\ op_2 = r\ sop \\ true & otherwise \end{cases}$$

$$valid(o, m) \overset{\text{def}}{=} \begin{cases} -4095 \leq o \leq 4095 & if\ m = ARM \\ -255 \leq o \leq 4095 & if\ m = Thumb32 \\ 0 \leq o \leq 124 & if\ m = Thumb16 \end{cases}$$

$$valid(i, m) \overset{\text{def}}{=} \begin{cases} m \neq ARM\ \&\&\ m \neq Thumb_{16} & if\ i = \textbf{orn} \\ true & otherwise \end{cases}$$

Fig. 7. The valid functions and related functions

a) Auxiliary Definitions: The memory model in ARM assembly is exposed by four operations declared as total functions in F⋆.

- `eval_mem`: reads from memory at given address.
- `upd_mem`: writes into memory at given address.
- `eval_reg`: reads from a given register (`[[reg]]`).
- `upd_reg`: writes into given register (`reg/val`).

'[[_]]' is also overloaded to get the value of condition flags, for instance `[[flags.c]]` returns the Carry value. In F⋆, these operations are encoded as follows:

```
unfold let eval_mem (addr: int32) (s:arm_state): Tot int32 =
  load_mem addr s.mem
let upd_mem (a:int32) (v:int32) (s:arm_state):Tot arm_state=
  {s with mem = store_mem a v s.mem}
unfold let eval_reg (r:reg) (s:arm_state) : Tot int32 =
  s.regs r
let upd_reg (r:reg) (v:int32) (s:arm_state): Tot arm_state =
  {s with regs = regs_make (fun (r':reg) ->
    if r = r' then v else s.regs r') }
```

Some symbols used in the Fig. 9 and Fig. 10 are explained below:

- $+$, $-$, $\times$, $\neg$ designate operations in range $[-2^{31}, 2^{31}-1]$.
- &, |, $\otimes$, and $\sim$ are bitwise AND, OR, exclusive OR and NOT operations respectively.

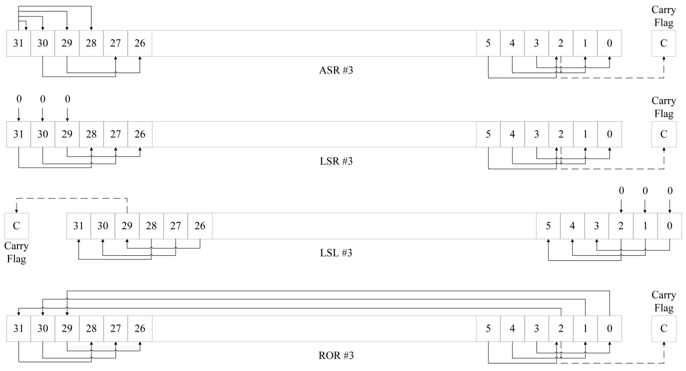

Fig. 8. Four shift operation: examples.

- $\gg_a$ is the arithmetic right shift operation.
- $\ll$ is the logical left shift operation.
- $\gg_l$ is the logical right shift operation.
- $\gg_r$ is the rotate right shift operation.

The unit of a memory cell is a 32-bit integer, so the *pc* register usually increases by 1 (i.e. 4 bytes).

In order to better explain the shift operations, Fig. 8 shows four examples, where

- *ASR #n* moves the left-hand 32-n bits of a register to the right by n places, into the right-hand 32-n bits of the result. It copies the original bit(31) of the register into the left-hand n bits of the result.
- *LSR #n* moves the left-hand 32-n bits of a register to the right by n places, into the right-hand 32-n bits of the result. It sets the left-hand n bits of the result to 0.
- *LSL #n* moves the right-hand 32-n bits of a register to the left by n places, into the left-hand 32-n bits of the result. It sets the right-hand n bits of the result to 0.
- *ROR #n* moves the left-hand 32-n bits of a register to the right by n places, into the right-hand 32-n bits of the result. It also moves the right-hand n bits of the register into the left-hand n bits of the result.

Note that the shift operations don't modify the Carry flag if the instruction lacks the condition flag suffix.

2) Instructions with Condition Suffix: Most instructions can update the condition flags when the suffix *s* is specified. But there are two special cases: the instructions **cmp** and **cmn** always update this flag, while the instructions **bx, ldr, neg, nop** and **str** never do (they don't support that suffix). This section mainly discusses the semantics of instructions with condition suffix, i.e. of the form '$\{s\}\ i$'.

Three update functions are defined to classify the scenarios:

- The `upd_arithmetic` function updates the four condition flags according to the result of an instruction.
  - `C` = 1 if an addition instruction (**adc/add/cmn**) produces a carry, or a subtraction instruction (**cmp/sub**) produce a borrow, otherwise `C` = 0.
  - `V` = 1 if the result of a signed add, subtract, or compare is greater than or equal to 2^{31}, or less than -2^{31}
- The `upd_logical` function is used to update N, Z and C flags after performing the **mov** instruction or bitwise instructions.

$$\frac{st.ok \wedge valid(rd) \wedge valid(rn) \wedge valid(op_2)}{(ADC\ rd\ rn\ op_2, st) \rightarrow st[rd/_{[[rn]]+[[op_2]]+[[flags.c]]}, pc/_{[[pc]]+1}]} \qquad (adc)$$

$$\frac{st.ok \wedge valid(rd) \wedge valid(rn) \wedge valid(op_2)}{(ADD\ rd\ rn\ op_2, st) \rightarrow st[rd/_{[[rn]]+[[op_2]]}, pc/_{[[pc]]+1}]} \qquad (add)$$

$$\frac{st.ok \wedge valid(rd) \wedge valid(rn) \wedge valid(op_2)}{(AND\ rd\ rn\ op_2, st) \rightarrow st[rd/_{[[rn]]\&[[op_2]]}, pc/_{[[pc]]+1}]} \qquad (and)$$

$$\frac{st.ok \wedge valid(rd) \wedge valid(rn) \wedge valid(rs)}{(ASR\ rd\ rn\ rs, st) \rightarrow st[rd/_{rn \gg_a rs}, pc/_{[[pc]]+1}]} \qquad (asr)$$

$$\frac{st.ok \wedge [[rd]].bit(0) = 0 \wedge valid(rd)}{(BX\ rd, st) \rightarrow st[st.isa\_mode/_{Thumb_{16}}, pc/_{[[rd]]}]} \qquad (bx1)$$

$$\frac{st.ok \wedge [[rd]].bit(0) = 1 \wedge valid(rd)}{(BX\ rd, st) \rightarrow st[st.isa\_mode/_{ARM}, pc/_{[[rd]]}]} \qquad (bx2)$$

$$\frac{st.ok \wedge valid(rn) \wedge valid(op_2)}{(CMN\ rn\ op_2, st) \rightarrow st[flags/_{upd\_arith([[rn]]+[[op_2]])}, pc/_{[[pc]]+1}]} \qquad (cmn)$$

$$\frac{st.ok \wedge valid(rn) \wedge valid(op_2)}{(CMP\ rn\ op_2, st) \rightarrow st[flags/_{upd\_arith([[rn]]-[[op_2]])}, pc/_{[[pc]]+1}]} \qquad (cmp)$$

$$\frac{st.ok \wedge valid(rd) \wedge valid(rn) \wedge valid(op_2)}{(EOR\ rd\ rn\ op_2, st) \rightarrow st[rd/_{[[rn]] \otimes [[op_2]]}, pc/_{[[pc]]+1}]} \qquad (eor)$$

$$\frac{st.ok \wedge valid(rd) \wedge valid(rn) \wedge valid(o)}{(LDR\ rd\ rn\ o, st) \rightarrow st[rd/_{\{\{[[rn]]+[[o]]\}\}}, pc/_{[[pc]]+1}]} \qquad (ldr)$$

$$\frac{st.ok \wedge valid(rd) \wedge valid(rn) \wedge valid(rs)}{(LSL\ rd\ rn\ rs, st) \rightarrow st[rd/_{[[rn]] \ll [[rs]]}, pc/_{[[pc]]+1}]} \qquad (lsl)$$

$$\frac{st.ok \wedge valid(rd) \wedge valid(rn) \wedge valid(rs)}{(LSR\ rd\ rn\ rs, st) \rightarrow st[rd/_{[[rn]] \gg_l [[rs]]}, pc/_{[[pc]]+1}]} \qquad (lsr)$$

$$\frac{st.ok \wedge valid(rd) \wedge valid(op_2)}{(MOV\ rd\ op_2, st) \rightarrow st[rd/_{[[op2]]}, pc/_{[[pc]]+1}]} \qquad (mov)$$

$$\frac{st.ok \wedge valid(rd) \wedge valid(rn) \wedge valid(rm)}{(MUL\ rd\ rn\ rm, st) \rightarrow st[rd/_{[[rd]] \times [[rm]]}, pc/_{[[pc]]+1}]} \qquad (mul)$$

$$\frac{st.ok \wedge valid(rd) \wedge valid(rm)}{(NEG\ rd\ rm, st) \rightarrow st[rd/_{\neg[[rm]]}, pc/_{[[pc]]+1}]} \qquad (neg)$$

$$\frac{st.ok}{(NOP, st) \rightarrow st[pc/_{[[pc]]+1}]} \qquad (nop)$$

$$\frac{st.ok \wedge valid(rd) \wedge valid(rn) \wedge valid(op_2) \wedge valid(st.isa\_mode)}{(ORN\ rd\ rn\ op_2, st) \rightarrow st[rd/_{[[rn]] \mid (\sim[[op_2]])}, pc/_{[[pc]]+1}]} \qquad (orn)$$

$$\frac{st.ok \wedge valid(rd) \wedge valid(rn) \wedge valid(op_2)}{(ORR\ rd\ rn\ op_2, st) \rightarrow st[rd/_{[[rn]] \mid [[op_2]]}, pc/_{[[pc]]+1}]} \qquad (orr)$$

$$\frac{st.ok \wedge valid(rd) \wedge valid(rn) \wedge valid(rs)}{(ROR\ rd\ rn\ op_2, st) \rightarrow st[rd/_{[[rn]] \gg_r [[op_2]]}, pc/_{[[pc]]+1}]} \qquad (ror)$$

$$\frac{st.ok \wedge valid(rd) \wedge valid(rn) \wedge valid(o)}{(STR\ rd\ rn\ o, st) \rightarrow st[\{[[rn]] + [[o]]\}/_{[[rd]]}, pc/_{[[pc]]+1}]} \qquad (str)$$

$$\frac{st.ok \wedge valid(rd) \wedge valid(rn) \wedge valid(op_2)}{(SUB\ rd\ rn\ op_2, st) \rightarrow st[rd/_{[[rn]]-[[op_2]]}, pc/_{[[pc]]+1}]} \qquad (sub)$$

Fig. 9. Semantics of the simple ARM instruction set

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(rn) \wedge valid(op_2)}{(ADCC\ c\ rd\ rn\ op_2, st) \rightarrow st[rd/_{[[rn]]+[[op_2]]+[[flags.c]]}, pc/_{[[pc]]+1}]} \quad (adcc)$$

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(rn) \wedge valid(op_2)}{(ADDC\ c\ rd\ rn\ op_2, st) \rightarrow st[rd/_{[[rn]]+[[op_2]]}, pc/_{[[pc]]+1}]} \quad (addc)$$

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(rn) \wedge valid(op_2)}{(ANDC\ c\ rd\ rn\ op_2, st) \rightarrow st[rd/_{[[rn]]\&[[op_2]]}, pc/_{[[pc]]+1}]} \quad (andc)$$

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(rn) \wedge valid(rs)}{(ASRC\ c\ rd\ rn\ rs, st) \rightarrow st[rd/_{rn\ \gg_a\ rs}, pc/_{[[pc]]+1}]} \quad (asrc)$$

$$\frac{st.ok \wedge cond(c, st) \wedge [[rd]].bit(0) = 0 \wedge valid(rd)}{(BXC\ c\ rd, st) \rightarrow st[st.isa\_mode/_{Thumb_{16}}, pc/_{[[rd]]}]} \quad (bxc1)$$

$$\frac{st.ok \wedge cond(c, st) \wedge [[rd]].bit(0) = 1 \wedge valid(rd)}{(BXC\ c\ rd, st) \rightarrow st[st.isa\_mode/_{ARM}, pc/_{[[rd]]}]} \quad (bxc2)$$

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(rn) \wedge valid(op_2)}{(EORC\ c\ rd\ rn\ op_2, st) \rightarrow st[rd/_{[[rn]]\ \otimes\ [[op_2]]}, pc/_{[[pc]]+1}]} \quad (eorc)$$

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(rn) \wedge valid(o)}{(LDRC\ c\ rd\ rn\ o, st) \rightarrow st[rd/_{\{\{[[rn]]+[[o]]\}\}}, pc/_{[[pc]]+1}]} \quad (ldrc)$$

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(rn) \wedge valid(rs)}{(LSLC\ c\ rd\ rn\ rs, st) \rightarrow st[rd/_{[[rn]]\ll[[rs]]}, pc/_{[[pc]]+1}]} \quad (lslc)$$

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(rn) \wedge valid(rs)}{(LSRC\ c\ rd\ rn\ rs, st) \rightarrow st[rd/_{[[rn]]\gg_l[[rs]]}, pc/_{[[pc]]+1}]} \quad (lsrc)$$

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(op_2)}{(MOVC\ c\ rd\ op_2, st) \rightarrow st[rd/_{[[op2]]}, pc/_{[[pc]]+1}]} \quad (movc)$$

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(rn) \wedge valid(rm)}{(MULC\ c\ rd\ rn\ rm, st) \rightarrow st[rd/_{[[rd]]\times[[rm]]}, pc/_{[[pc]]+1}]} \quad (mulc)$$

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(rm)}{(NEGC\ c\ rd\ rm, st) \rightarrow st[rd/_{\neg[[rm]]}, pc/_{[[pc]]+1}]} \quad (negc)$$

$$\frac{st.ok \wedge cond(c, st)}{(NOPC\ c, st) \rightarrow st[pc/_{[[pc]]+1}]} \quad (nopc)$$

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(rn) \wedge valid(op_2) \wedge valid(st.isa\_mode)}{(ORNC\ c\ rd\ rn\ op_2, st) \rightarrow st[rd/_{[[rn]]\ |\ (\sim[[op_2]])}, pc/_{[[pc]]+1}]} \quad (ornc)$$

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(rn) \wedge valid(op_2)}{(ORRC\ c\ rd\ rn\ op_2, st) \rightarrow st[rd/_{[[rn]]\ |\ [[op_2]]}, pc/_{[[pc]]+1}]} \quad (orrc)$$

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(rn) \wedge valid(rs)}{(RORC\ c\ rd\ rn\ op_2, st) \rightarrow st[rd/_{[[rn]]\ \gg_r\ [[op_2]]}, pc/_{[[pc]]+1}]} \quad (rorc)$$

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(rn) \wedge valid(o)}{(STRC\ c\ rd\ rn\ o, st) \rightarrow st[\{[[rn]] + [[o]]\}/_{[[rd]]}, pc/_{[[pc]]+1}]} \quad (strc)$$

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(rn) \wedge valid(op_2)}{(SUBC\ c\ rd\ rn\ op_2, st) \rightarrow st[rd/_{[[rn]]-[[op_2]]}, pc/_{[[pc]]+1}]} \quad (subc)$$

Fig. 10. Semantics of conditional ARM instruction sets

  – C: updates the flag during calculation of 2^{nd} operand.
  – V: does not affect the flag.
- The upd_shift function updates the three flags.
  – C: The flag is updated to the last bit shifted out.
  – V: does not affect the flag.

Fig. 11 shows the semantics rules of some instructions with condition suffix.

B. *Functional correctness of the assembly boot code*

This section is the proof of functional correctness of the core assembly boot sequence code of the bootloader.

```
val functional_connectness_aux2_0: st:arm_state ->
    Lemma
  (requires (st.ok=true))
  (ensures  (let st0 = eval_cond_ins i0 st in
             let r0' = eval_reg R0 st in
             let r0  = eval_reg R0 st0 in
             let r1_0 = eval_reg R1 st0 in
               r1_0 == (eval_mem r0' st) /\
               r0 == r0'
             ))
let functional_connectness_aux2_0 st = ()

val functional_connectness_aux2_1: st:arm_state ->
    Lemma
  (requires (st.ok=true))
  (ensures  (let st1 = eval_cond_ins i1 st in
             let r0' = eval_reg R0 st in
             let r0  = eval_reg R0 st1 in
             let r1' = eval_reg R1 st in
             let r1 = eval_reg R1 st1 in
             let sp = eval_reg SP st1 in
               sp == r1 /\
               r1 == r1' /\
               r0 == r0'
             ))
let functional_connectness_aux2_1 st = ()

val functional_connectness_aux2_2: st:arm_state ->
    Lemma
  (requires (st.ok=true))
  (ensures  (let st2 = eval_cond_ins i2 st in
             let r0' = eval_reg R0 st in
             let r0 = eval_reg R0 st2 in
             let r1' = eval_reg R1 st in
             let r1 = eval_reg R1 st2 in
             let addr = Int32.int_to_t (add_mod (
    Int32.v r0') (Int32.v 1l)) in
             let sp' = eval_reg SP st in
             let sp = eval_reg SP st2 in
              r0 == eval_mem addr st /\
              r1 == r1' /\
              sp' == sp
             ))
let functional_connectness_aux2_2 st = ()

val functional_connectness_aux2_3: st:arm_state ->
    Lemma
  (requires (st.ok=true))
  (ensures  (let st3 = eval_cond_ins i3 st in
             let r0' = eval_reg R0 st in
             let r0 = eval_reg R0 st3 in
             let r1' = eval_reg R1 st in
             let r1 = eval_reg R1 st3 in
             let sp' = eval_reg SP st in
             let sp = eval_reg SP st3 in
               bit_n (Int32.v r0) 31 == true /\
               r0  == Int32.int_to_t (logor (Int32.v
     r0') (Int32.v 1l)) /\
               r1 == r1' /\
               sp' == sp
             ))

#push-options "--ifuel 50 --fuel 50 --z3rlimit 320"
let functional_connectness_aux2_3 st = ()
#pop-options

val functional_connectness_aux2_4: st:arm_state ->
    Lemma
  (requires (st.ok=true /\
             (let r0 = eval_reg R0 st in
              bit_n (Int32.v r0) 31 == true)
              ))
  (ensures  (let st4 = eval_cond_ins i4 st in
             let pc  = eval_reg PC st4 in
             let r0'  = eval_reg R0 st in
             let r0  = eval_reg R0 st4 in
             let r1' = eval_reg R1 st in
             let r1 = eval_reg R1 st4 in
             let sp' = eval_reg SP st in
             let sp = eval_reg SP st4 in
              st4.isa_mode == Thumb16 /\
              sp == sp' /\
              r0 == r0' /\
              r1 == r1' /\
              pc == r0
             ))
let functional_connectness_aux2_4 st = ()

val functional_connectness_aux1: st:arm_state ->
    Lemma
  (requires (st.ok = true))
  (ensures  (let st' = eval_list_ins cplist st in
             let st0 = eval_cond_ins i0 st in
             let st1 = eval_cond_ins i1 st0 in
             let st2 = eval_cond_ins i2 st1 in
             let st3 = eval_cond_ins i3 st2 in
             let st4 = eval_cond_ins i4 st3 in
               st' == st4
              ))
let functional_connectness_aux1 st = ()

val functional_connectness_aux2: st:arm_state ->
    Lemma
  (requires (st.ok = true))
  (ensures  (let st0 = eval_cond_ins i0 st in
             let st1 = eval_cond_ins i1 st0 in
             let st2 = eval_cond_ins i2 st1 in
             let st3 = eval_cond_ins i3 st2 in
             let st4 = eval_cond_ins i4 st3 in
             let r0' = eval_reg R0 st in
             let addr = Int32.int_to_t (add_mod (
    Int32.v r0') (Int32.v 1l)) in
             let sp = eval_reg SP st4 in
             let r1 = eval_mem r0' st in
             let pc  = eval_reg PC st4 in
             let r0  = eval_reg R0 st4 in
              st4.isa_mode == Thumb16 /\
              sp == r1 /\
              r0 == Int32.int_to_t (logor (Int32.v (
    eval_mem addr st)) (Int32.v 1l)) /\
              pc == r0
              ))

#push-options "--ifuel 50 --fuel 50 --z3rlimit 320"
let functional_connectness_aux2 st =
  let st0 = eval_cond_ins i0 st in
  let st1 = eval_cond_ins i1 st0 in
  let st2 = eval_cond_ins i2 st1 in
  let st3 = eval_cond_ins i3 st2 in
    functional_connectness_aux2_0 st;
    functional_connectness_aux2_1 st0;
    functional_connectness_aux2_2 st1;
```

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(rn) \wedge valid(op_2)}{(ADCSC\ c\ rd\ rn\ op_2, st) \rightarrow st[rd/_{[[rn]]+[[op_2]]+[[flags.c]]}, pc/_{[[pc]]+1}, flags/upd\_arith([[rn]] +_i [[op_2]] +_i [[flags.c]])]} \quad (adcsc)$$

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(rn) \wedge valid(op_2)}{(ADDSC\ c\ rd\ rn\ op_2, st) \rightarrow st[rd/_{[[rn]]+[[op_2]]}, pc/_{[[pc]]+1}, flags/upd\_arith([[rn]] +_i [[op_2]])]} \quad (addsc)$$

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(rn) \wedge valid(op_2)}{(ANDSC\ c\ rd\ rn\ op_2, st) \rightarrow st[rd/_{[[rn]]\&[[op_2]]}, pc/_{[[pc]]+1}, upd\_logical([[rn]] +_i [[op_2]])]} \quad (andsc)$$

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(rn) \wedge valid(rs)}{(ASRSC\ c\ rd\ rn\ rs, st) \rightarrow st[rd/_{rn \gg_a rs}, pc/_{[[pc]]+1}, upd\_logical([[rn]], [[rs]])]} \quad (asrsc)$$

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(rn) \wedge valid(op_2)}{(EORSC\ c\ rd\ rn\ op_2, st) \rightarrow st[rd/_{[[rn]] \otimes [[op_2]]}, pc/_{[[pc]]+1}, upd\_logical([[rn]] +_i [[op_2]])]} \quad (eorsc)$$

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(rn) \wedge valid(rs)}{(LSLSC\ c\ rd\ rn\ rs, st) \rightarrow st[rd/_{[[rn]] \ll [[rs]]}, pc/_{[[pc]]+1}, upd\_logical([[rn]], [[rs]])]} \quad (lslsc)$$

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(rn) \wedge valid(rs)}{(LSRSC\ c\ rd\ rd\ rs, st) \rightarrow st[rd/_{[[rn]] \gg_l [[rs]]}, pc/_{[[pc]]+1}, upd\_logical([[rn]], [[rs]])]} \quad (lsrsc)$$

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(op_2)}{(MOVSC\ c\ rd\ op_2, st) \rightarrow st[rd/_{[[op2]]}, pc/_{[[pc]]+1}, flags/upd\_arith([[op_2]])]} \quad (movsc)$$

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(rn) \wedge valid(rm)}{(MULSC\ c\ rd\ rn\ rm, st) \rightarrow st[rd/_{[[rd]] \times [[rm]]}, pc/_{[[pc]]+1}, flags/upd\_arith([[rn]] +_i [[rm]])]} \quad (mulsc)$$

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(rn) \wedge valid(op_2) \wedge valid(st.isa\_mode)}{(ORNSC\ c\ rd\ rn\ op_2, st) \rightarrow st[rd/_{[[rn]]\ |\ (\sim[[op_2]])}, pc/_{[[pc]]+1}, upd\_logical([[rn]] +_i [[op_2]])]} \quad (ornsc)$$

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(rn) \wedge valid(op_2)}{(ORRSC\ c\ rd\ rn\ op_2, st) \rightarrow st[rd/_{[[rn]]\ |\ [[op_2]]}, pc/_{[[pc]]+1}, upd\_logical([[rn]] +_i [[op_2]])]} \quad (orrsc)$$

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(rn) \wedge valid(rs)}{(RORSC\ c\ rd\ rn\ rs, st) \rightarrow st[rd/_{[[rn]] \gg_r [[rs]]}, pc/_{[[pc]]+1}, upd_logical([[rn]], [[rs]])]} \quad (rorsc)$$

$$\frac{st.ok \wedge cond(c, st) \wedge valid(rd) \wedge valid(rn) \wedge valid(op_2)}{(SUBSC\ c\ rd\ rn\ op_2, st) \rightarrow st[rd/_{[[rn]]-[[op_2]]}, pc/_{[[pc]]+1}, flags/upd\_arith([[rn]] +_i [[op_2]])]} \quad (subsc)$$

Fig. 11. Semantics of conditional ARM instructions with condition suffix

```
    functional_connectness_aux2_3 st2;
    functional_connectness_aux2_4 st3
#pop-options

val functional_connectness: st:arm_state -> Lemma
  (requires (st.ok=true))
  (ensures   (let st1 = eval_list_ins cplist st in
              let r0' = eval_reg R0 st in
              let sp = eval_reg SP st1 in
              let r0 = eval_reg R0 st1 in
              let addr = Int32.int_to_t (add_mod (Int32.v r0') (Int32.v 11)) in
              let pc = eval_reg PC st1 in
              let r1 = eval_mem r0' st in
                r0 == Int32.int_to_t (logor (Int32.v (eval_mem addr st)) (Int32.v 11)) /\
                sp == r1 /\
                pc == r0
                ))

#push-options "--ifuel 50 --fuel 50 --z3rlimit 320"
let functional_connectness st =
   functional_connectness_aux1 st; functional_connectness_aux2 st
#pop-options
```

C. Validated choose_image function

Finally, this section lists the verified fletcher32 and choose_image functions of the bootloader.

```
type fletcher = (pub_uint32 & pub_uint32)
type header = (pub_uint32 & pub_uint32 & pub_uint32 & pub_uint32)

let riotboot_magic : pub_uint32 =
  pub_u32 0x544f4952
let new_fletcher () : fletcher =
  (pub_u32 0x0, pub_u32 0x0)
let max_chunk_size () : uint_size =
  usize 360

let reduce_u32 (x_0 : pub_uint32) : pub_uint32 =
  ((x_0) &. (pub_u32 0xffff)) +. ((x_0) `shift_right` (pub_u32 0x10))

let combine (lower_1 : pub_uint32) (upper_2 : pub_uint32) : pub_uint32 =
  (lower_1) |. ((upper_2) `shift_left` (pub_u32 0x10))

```

```
let update_fletcher (f_3 : fletcher) (data_4 : seq
    pub_uint16) : fletcher =
  let max_chunk_size_5 = max_chunk_size () in
  let (a_6, b_7) = f_3 in
  let (a_6, b_7) =
    foldi (usize 0) (seq_num_chunks (data_4) (
    max_chunk_size_5)) (fun i_8 (
        a_6,
        b_7
      ) ->
      let (chunk_len_9, chunk_10) =
        seq_get_chunk (data_4) (i_8) (
    max_chunk_size_5)
      in
      let intermediate_a_11 = a_6 in
      let intermediate_b_12 = b_7 in
      let (intermediate_a_11, intermediate_b_12) =
        foldi (usize 0) (chunk_len_9) (fun j_13 (
            intermediate_a_11,
            intermediate_b_12
          ) ->
          let intermediate_a_11 =
            (intermediate_a_11) +. (
              cast U32 PUB (array_index
                (**) #pub_uint16 #chunk_len_9
                (chunk_10) (j_13)))
          in
          let intermediate_b_12 = (intermediate_b_12
    ) +. (intermediate_a_11) in
          (intermediate_a_11, intermediate_b_12))
        (intermediate_a_11, intermediate_b_12)
      in
      let a_6 = reduce_u32 (intermediate_a_11) in
      let b_7 = reduce_u32 (intermediate_b_12) in
      (a_6, b_7))
    (a_6, b_7)
  in
  let a_6 = reduce_u32 (a_6) in
  let b_7 = reduce_u32 (b_7) in
  (a_6, b_7)

let value (x_14 : fletcher) : pub_uint32 =
  let (a_15, b_16) = x_14 in
  combine (a_15) (b_16)

let header_as_u16_slice (h_17 : header) : seq
    pub_uint16 =
  let (magic_18, seq_number_19, start_addr_20, _) =
    h_17 in
  let magic_21 = u32_to_be_bytes (magic_18) in
  let seq_number_22 = u32_to_be_bytes (seq_number_19
    ) in
  let start_addr_23 = u32_to_be_bytes (start_addr_20
    ) in
  let u8_seq_24 = seq_new_ (pub_u8 0x0) (usize 12)
    in
  let u8_seq_25 =
    seq_update_slice (u8_seq_24) (usize 0) (magic_21
    ) (usize 0) (usize 4)
  in
  let u8_seq_26 =
    seq_update_slice (u8_seq_25) (usize 4) (
    seq_number_22) (usize 0) (usize 4)
  in
  let u8_seq_27 =
    seq_update_slice (u8_seq_26) (usize 8) (
    start_addr_23) (usize 0) (usize 4)
  in
  let u16_seq_28 = seq_new_ (pub_u16 0x0) (usize 6)
    in
  let (u16_seq_28) =
    foldi (usize 0) (usize 6) (fun i_29 (u16_seq_28)
     ->
      let u16_word_30 =
        array_from_seq (2) (
          seq_slice (u8_seq_27) ((i_29) * (usize 2))
     (usize 2))
      in
      let u16_value_31 = u16_from_be_bytes (
    u16_word_30) in
      let u16_seq_28 = array_upd u16_seq_28 (i_29) (
    u16_value_31) in
      (u16_seq_28))
    (u16_seq_28)
  in
  u16_seq_28

let is_valid_header (h_32 : header) : bool =
  let (magic_number_33, seq_number_34, start_addr_35
    , checksum_36) = h_32 in
  let slice_37 =
    header_as_u16_slice (
      (magic_number_33, seq_number_34, start_addr_35
    , checksum_36))
  in
  let result_38 = false in
  let (result_38) =
    if (magic_number_33) = (riotboot_magic) then
    begin
      let fletcher_39 = new_fletcher () in
      let fletcher_40 = update_fletcher (fletcher_39
    ) (slice_37) in
      let sum_41 = value (fletcher_40) in
      let result_38 = (sum_41) = (checksum_36) in
      (result_38)
    end else begin (result_38)
    end
  in
  result_38

let choose_image (images_42 : seq header) : (bool &
    pub_uint32) =
  let image_43 = pub_u32 0x0 in
  let image_found_44 = false in
  let (image_43, image_found_44) =
    foldi (usize 0) (seq_len (images_42)) (fun i_45
    (image_43, image_found_44
      ) ->
      let header_46 = array_index
        (**) #header #(seq_len images_42)
        (images_42) (i_45)
      in
      let (magic_number_47, seq_number_48,
    start_addr_49, checksum_50) =
        header_46
      in
      let (image_43, image_found_44) =
        if is_valid_header (
          (magic_number_47, seq_number_48,
    start_addr_49, checksum_50
          )) then begin
          let change_image_51 =
            not ((image_found_44) && ((seq_number_48
    ) <=. (image_43)))
          in
          let (image_43, image_found_44) =
            if change_image_51 then begin
              let image_43 = start_addr_49 in
              let image_found_44 = true in
              (image_43, image_found_44)
            end else begin (image_43, image_found_44
    )
            end
          in
          (image_43, image_found_44)
        end else begin (image_43, image_found_44)
        end
      in
      (image_43, image_found_44))
    (image_43, image_found_44)
  in
  (image_found_44, image_43)
```