# Towards a Verified First-Stage Bootloader in Coq

by

Zygimantas Straznickas

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2020

Author . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Department of Electrical Engineering and Computer Science
May 12, 2020

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Adam Chlipala
Associate Professor of Computer Science
Thesis Supervisor

Accepted by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

# Towards a Verified First-Stage Bootloader in Coq

by

Zygimantas Straznickas

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 2020, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

## Abstract

A cryptographic secure boot and attestation system usually depends on a measurement root — a first-stage bootloader written to ROM that loads the boot payload into the machine's memory, cryptographically signs it and ensures that the boot payload begins execution from a clean-slate environment. We implement a first-stage bootloader heavily inspired by the Sanctum project [5], describe its specification in Coq relative to low-level RISC-V semantics, state the correctness of the implementation as a theorem in Coq and prove a few major lemmas necessary for establishing correctness.

Thesis Supervisor: Adam Chlipala
Title: Associate Professor of Computer Science

# Contents

# List of Figures

# Chapter 1

# Introduction

In [5], Lebedev et al. proposed a secure boot and attestation system for the Sanctum processor, used to prove cryptographically to outside observers that a computer's state was correctly established according to the user's specification. The security of this system relies on the first-stage bootloader — a program that runs as the very first piece of code when the machine is turned on. Its purpose is to copy over the actual executable — e.g. the operating system — from untrusted disk storage to trusted memory and cryptographically sign the executable, combining it with the computer's device-specific key.

Normally, this first-stage bootloader would be stored in the device's ROM memory, written there during the manufacturing process and immutable after that. If a security-compromising bug was discovered, there would be no way to patch it without entirely replacing the hardware. This makes the correctness of the bootloader even more important.

Formal-verification techniques have been getting more popular and practical in ensuring program correctness. Sanctum is especially amenable to such techniques because it is designed for hardware that uses the RISC-V instruction set [29]. As an open standard, RISC-V's instruction semantics are publicly available and can be used as a base against which to verify program behavior. This thesis is a case study in formally verifying a low-level program all the way down to RISC-V instruction semantics, including the modelling of custom hardware DMA-style features available to the CPU.

The goal of this project is to design, implement and verify a first-stage bootloader inspired by the Sanctum bootloader. The property being verified is functional correctness. We show that under appropriate assumptions the bootloader, starting from an arbitrary state, will always terminate and, after completing execution, will have the second-stage boot image ready in memory, signed and certified, with all other memory and state being zeroed out. We do not verify the cryptographic-security aspects of the bootloader — this is considered out of scope. We implement the bootloader for the RISC-V architecture with a Sanctum-inspired direct memory access mechanism to load a user-configurable OS boot image to memory. In order to keep the performance of cryptographic operations practical, and to investigate possible interoperation of verified and nonverified code, we also use cryptographic routines implemented in C and verify a "bridge" from the verified part of the code to these functions, limiting our need to trust nonverified C compilers. More specifically, we assume the C compiler will compile the cryptographic functions correctly according to the cdecl calling convention, and verify the correctness of the function call preamble and postamble in the bootloader.

We implement the bootloader in the Coq proof assistant (using the Bedrock2 framework), state the correctness theorem about the desired properties relative to the riscv-coq RISC-V specification, and prove most lemmas necessary for the correctness theorem.

# Chapter 2

# Background

## 2.1 RISC-V

RISC-V is an instruction-set architecture (ISA) managed and developed by the RISC-V Foundation. It is an open and patent-free ISA with a public specification and a permissive licence. It is intended to be a practical ISA, suitable for real-world implementations and able to match the performance of current mainstream ISAs.

RISC-V defines both 32-bit and 64-bit variants. Its specification is divided into the base instruction set and multiple extensions. As of the time of writing, the following extensions are fully defined:

- Multiply-Divide
- Atomic
- Floating-point arithmetic for single, double and quad precision
- Compressed Instructions

and there are plans to define other extensions, including for SIMD instructions, bit manipulation, etc.

RISC-V is supported by many mainstream software-development tools including GCC, GNU binutils and QEMU.

## 2.2 Coq

Coq [26] is a proof assistant — a software environment to write and manage formal mathematical proofs. Its theoretical foundations are based on the Calculus of Inductive Constructions [19], a higher-order intuitionistic type theory with inductive types. Some additional axioms like the law of excluded middle have been proven consistent with Coq foundations and can be imported by users selectively.

Gallina, the language used to define terms and functions in Coq, is a functional language that supports defining algebraic data types (in the form of inductive types), pattern matching and a typeclass mechanism. Coq also includes Ltac and Ltac2, scripting languages used to construct proof terms dynamically.

The two main uses of Coq are formalizing pure mathematics and constructing formally verified software. Pure mathematics developed in Coq include the formalization of the Odd-Order theorem [11] and work on homotopy type theory [2] [28] and category theory [12]. Verified software written in Coq includes the CompCert C compiler [17] and fiat-crypto [9], a project for generating elliptic-curve-arithmetic code.

## 2.3 riscv-coq

The riscv-coq project [8] is an implementation of the RISC-V ISA specification in Coq. It is based on the RISC-V Haskell specification [3] and is partly programmatically generated from it. It currently supports the RV32I and RV64I architectures and A and M extensions.

The riscv-coq project defines an abstract interface of a RISC-V machine and describes RISC-V instructions by how they change the abstract machine. It also describes the encoding and decoding of instructions and proves that these operations are mutually inverse.

The definition of the abstract machine interface can be seen in Listing 1. It is monadic in nature and can be parametrized by an arbitrary monad to support nondeterministic implementations.

riscv-coq also defines several concrete platforms that implement the abstract interface. The basic implementation just defines the operations as expected, representing registers and memory as maps. Other implementations add additional functionality to the hardware, like memory-mapped input/output (MMIO), or extend the state representation of the machine to enable proving more properties about the programs running on it, e.g. tracking of executable addresses in memory or counting execution metrics (jumps, reads, writes, etc.)

**Listing 1** The interface of an abstract RISC-V machine in riscv-coq. Source: riscv-coq.

```
Class RiscvProgram{M}{t}`{Monad M}`{MachineWidth t} := mkRiscvProgram {
  getRegister: Register -> M t;
  setRegister: Register -> t -> M unit;

  loadByte   : SourceType -> t -> M w8;
  loadHalf   : SourceType -> t -> M w16;
  loadWord   : SourceType -> t -> M w32;
  loadDouble : SourceType -> t -> M w64;

  storeByte   : SourceType -> t -> w8 -> M unit;
  storeHalf   : SourceType -> t -> w16 -> M unit;
  storeWord   : SourceType -> t -> w32 -> M unit;
  storeDouble : SourceType -> t -> w64 -> M unit;

  makeReservation  : t -> M unit;
  clearReservation : t -> M unit;
  checkReservation : t -> M bool;

  getPC: M t;
  setPC: t -> M unit;

  step: M unit; (* updates PC *)

  raiseExceptionWithInfo{A: Type}(isInterrupt: t)
                        (exceptionCode: t)(info: t): M A;
}.
```

## 2.4 Bedrock2

Bedrock2 [7] is a low-level, imperative, C-like programming language implemented in Coq and compiling into RISC-V assembly. It is designed for formal verification, making it easy to prove theorems about Bedrock2 programs. The language supports the usual imperative language features like mutable variables, branches, loops and functions. At the moment of writing, it only supports 32- and 64-bit words as datatypes. Arrays and structures are supported as Coq-level aliases with helpers to generate field-access code metaprogramming-style, but are not supported as native data types and cannot automatically be allocated on the stack. In order to make reasoning about Bedrock2 programs easier, nonstatic features like function pointers, recursion and nonterminating programs are intentionally not supported.

The Bedrock2 project implements the Bedrock2 language as a deep embedding in Coq. It describes the semantics of Bedrock2 and proves a correctness theorem for the compiler. Using this theorem lets users transport proofs about Bedrock2 programs to proofs about the compiled RISC-V assembly in RISC-V semantics.

The Bedrock2 semantics are also extensible — in additional to usual statement types like if statements and function calls, it also supports an "external call" statement type. The semantics of these external calls are provided by the user, together with a function to compile them to RISC-V and a proof of compilation correctness. One important use case of external calls is exposing hardware capabilities of custom RISC-V machines to the Bedrock2 layer. Bedrock2 provides an example external-call implementation of memory-mapped input/output (MMIO).

## 2.5 Predicate-Transformer Semantics

Introduced by Edsger Dijkstra, Predicate Transformer Semantics [6] is a way to define the semantics of a programming language by specifying a *predicate transformer* to each statement in the language, that is, specifying how each statement transforms which predicates apply to the program's state before and after executing that statement.

Weakest-precondition transformers are a popular concrete definition of a predicate transformer. Given a desired postcondition $Q(\text{state})$, they specify, for each statement, what precondition

$$P = wp(\text{statement}, \text{state}, Q)$$

should hold before the statement is executed.

For example, we can define the semantics of a variable-assignment statement $k := V$ by saying

$$wp(k := V, \text{state}, Q) = Q(\text{state}[k \mapsto V])$$

That is, to prove that $Q$ holds after the assignment, we need to prove that $Q$ would hold before the assignment if we replaced $k$'s valuation with $V$. Such definitions of $wp$ can be given for all statements in the language to allow for backwards reasoning about the program's behavior.

## 2.6 Separation Logic

Hoare logic [14] is a formal system for reasoning about program behavior. Its assertions are expressed in the form of Hoare triples — $\{P\}C\{Q\}$ — meaning "if the predicate $P$ holds before executing a statement $C$, the predicate $Q$ will hold after its execution."

Separation logic [22] is an extension of Hoare logic to support reasoning about non-intersecting regions of memory. It adds a concept of a heap $h$, which is represented by a set of tuples of $k \mapsto v$, describing which memory addresses map to which values. Hoare logic triples can then reference this heap when defining preconditions and postconditions. Separation logic also adds a connective $\star$, where for two propositions $P$ and $Q$, $P * Q$ means that there exists a way to divide the heap $h$ into two nonintersecting $h_1$ and $h_2$ such that $P$ is valid on $h_1$ and $Q$ is valid on $h_2$.

In Bedrock2, separation logic is implemented as a way to construct predicates on memory. Specifically, memory is represented by a data type `mem :> map.map word byte` and Bedrock2 provides ways to construct statements of type `mem -> Prop`. These include "primitive" statements like

**Listing 2** Examples of primitive separation-logic statements in Bedrock2. Source: Bedrock2.

```
Definition emp : mem -> Prop := fun m => m = empty.
Definition ptsto k v : mem -> Prop :=
  fun m => m = put map.empty k v.
```

as well as the connective

**Listing 3** Example of the star connective in Bedrock2. Source: Bedrock2.

```
Definition sep (p q : mem -> Prop) m :=
    exists mp mq, split m mp mq /\ p mp /\ q mq.
```

representing the star connective. This connective also has an infix notation of ∗ in Bedrock2. A sample separation-logic statement about the machine's memory in Bedrock2 would look like:

```
Check (exists R,
    ((ptsto pointer1 dataWord) *
    (ptsto_array pointer2 dataBlock) *
    R) initialMemory).
```

The above statement says that the initial memory has a `dataWord` pointed to by `pointer1`, a `dataBlock` pointed to by `pointer2`, and is otherwise fully described by some separation-logic statement `R`.

## 2.7 Related Work

### 2.7.1 Sanctum

Sanctum [5] is a platform to support strong isolation of independent software modules running on a computer. It consists of hardware modifications to the underlying

CPU platform and several software components: the measurement root, the security monitor and the signing enclave.

In particular, the measurement root is stored in the machine's ROM and is the first fragment of code entered after boot. It acts as a first-stage bootloader and is responsible for the following tasks:

- loading the security monitor into memory;
- generating the necessary cryptographic keys for the Sanctum system;
- cryptographically signing the loaded security monitor;
- zeroing out the memory used by intermediate computations;
- jumping into the security monitor and resuming boot.

The guarantee provided by the measurement root is that after it finishes executing, the second-stage boot image is loaded and signed, and the machine is in a "clean" state otherwise.

The first-stage bootloader implemented in this thesis is directly inspired by the Sanctum measurement root and follows its architecture (while making a few simplifications and assumptions about the hardware).

Parts of the Sanctum project have been formally verified: [25] defines TAP (Trusted Abstract Platform), an idealized model of a software isolation system, and proves that a simplified specification of the Sanctum platform is a refinement of TAP. However, the proof is limited to the specification of Sanctum, not its implementation.

### 2.7.2 Bootloaders

Coreboot [27] is an open-source implementation of a general-purpose bootloader. It supports several CPU architectures including x86-64, ARMv8 and RISC-V. While most of Coreboot is implemented in C/assembly and is unverified, parts of the project use the SPARK programming language and have been formally checked to have no runtime errors. Coreboot is capable of performing measurements of the payload but needs to use a platform-provided root of trust.

SABLE [4] is an implementation of an open-source, formally verified bootloader. It is verified in the style of seL4 [15] — by proving that its abstract specification's behavior matches the behavior of the corresponding C implementation. However, the compilation from C to machine code is unverified, and the compiler has to be trusted. Similar to Coreboot, SABLE can support measured booting but depends on an external root of trust for that.

### 2.7.3 Trusted Platform Modules

Several authors have published work on formal verification of the Trusted Platform Module specification as well as its implementations, including the first-stage bootloader measurement part. [13] defines an abstract model of the TPM 1.2 specification and verifies several of its security properties. [18] implements a verified reference implementation of a part of the TPM specification. It verifies the code up to C semantics, requiring a trusted compiler.

In comparison, the scope of this thesis is much narrower: it implements only the first-stage bootloader (the software part of the root of trust) and does not handle later-stage setup. However, the formal verification is end-to-end: the correctness theorem specifies how the implementation transforms the state of an abstract RISC-V machine according to RISC-V assembly semantics, bypassing the need to trust a C compiler for everything but the performance-sensitive external C functions.

# Chapter 3

# Implementing the First-Stage Bootloader

## 3.1 Structure

As the design of the bootloader is based on the Sanctum [5] system's first-stage bootloader, we also draw heavy inspiration from its original implementation in C [16].

The bootloader is composed of three abstract parts:

- a preamble, used to initialize the memory, set up the stack and load the payload binary into the machine's memory (using the DMA-based loader mechanism described below).
- higher-level structured code to compute the cryptographic attestation of the payload binary.
- a postamble, used to clean up the memory and reset the device's state.

Ideally, we would want to implement as much of the project as possible in Bedrock2 to maximize the use of helpful tools the language provides. This includes both code-level features like function calls or local variables and proof-level features like the lemma library and proof automation. Unfortunately, because the preamble and

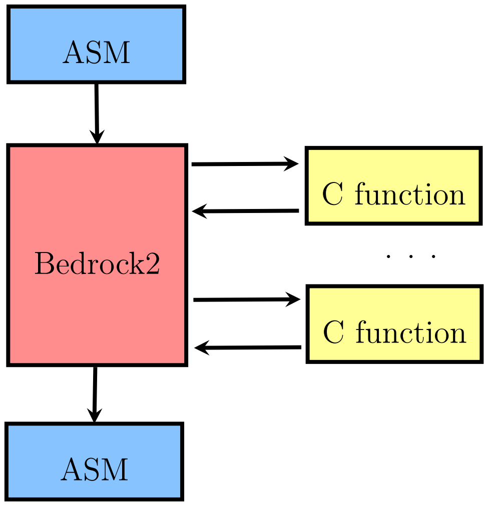

Figure 3-1: Structure of the first-stage bootloader

postamble need access to all of the machine's memory to zero it out, their behavior cannot be reasoned about as Bedrock2 programs. For that reason, we implement them directly in RISC-V assembly. Only the middle part, which is mostly composed of a sequence of straightline calls to Bedrock2 functions and external C functions, is implemented in Bedrock2.

The bootloader uses cryptographic primitives to certify and sign the boot image during the boot process. Most of the running time of the bootloader is spent in these cryptographic operations, so they have to be fast enough to keep the bootloader practical. Unfortunately, the Bedrock2 compiler (as of the time of writing) does not optimize the code very well. Because of this, and as a case study of integrating untrusted pregenerated code into a verified program, we use the cryptographic routines implemented in C and compiled with the GNU Compiler Collection (GCC) and jump into them from verified Bedrock2 code. We describe these C functions by their "external specifications" and, throughout the development, assume that they correctly describe the behavior of the resulting assembly.

Throughout this project, because the verified part of our program interfaces with some unverified parts, we need the following assumptions to ensure our proofs are meaningful:

- The machine that is running the code is a single-core 64-bit RISC-V abstract machine with a DMA boot-image-loading mechanism. The machine must have a known fixed memory layout.

- GCC compiles code correctly, and the compiled function uses the cdecl calling convention.
- The external specifications of C functions and the DMA boot-image loader are written correctly.

## 3.2 Specification

The correctness proof takes the form of a Coq theorem describing that the program's RISC-V assembly terminates and, after termination, the final state of the machine satisfies the specification. More specifically, the precondition we need is quite weak: it suffices that

- The bootloader code is in memory.
- The program counter is pointing to the first instruction of the bootloader.
- The memory contains an arbitrarily initialized region for the bootloader's working memory.
- The memory addresses of the external C functions (in the "real world") are undefined in the abstract machine's memory.

(Note that the preconditions do not require zeroed-out memory or registers because it's shown that the program will not attempt to read them before their values are initialized.)

In this project, we focus on proving functional correctness for the bootloader. We define a functional program in Coq that describes the nonzero part of the machine's memory post-execution. Because the project uses code from external C functions for which functional correctness has not necessarily been proven, the functional specification of the bootloader is parametrized by functional specifications of the C functions, for example,

```
Variable hash_functional_spec : list byte -> list byte.
Variable hash_output_size:
    forall i, length (hash_functional_spec i) = 64.
```

Given these parameters, we then define a function that describes the bootloader's output when interpreted as a functional program (of arity zero — conceptually, a bootloader doesn't take any arguments):

```
Definition bootloader_output : list byte := ... .
```

This lets us finally state the postcondition of the theorem:

- The bootloader terminates.
- After termination, there is a block of memory in the machine whose value is equal to `bootloader_output`.
- The program counter is pointing to the first instruction of that block.
- All other memory in RAM that the program ever wrote to is zeroed-out.
- All registers are zeroed-out.

In this thesis, we state the specification as a theorem in Coq and prove several major lemmas needed for it, including the correctness of the Bedrock2 code relative to Bedrock2 semantics and the correctness of assembly routines relative to riscv-coq semantics. We do not prove the full-correctness theorem unconditionally — that requires some more proof-engineering work.

## 3.3 External Calls

It is useful for machine models to have a way of modelling "external" actions that are not reflected in default hardware-architecture semantics. Such external actions can be used to represent memory-mapped input/output (MMIO), more direct hardware mechanisms like direct memory access (DMA), or interactions with code written outside the formalization. Specifically, in this project the two kinds of external calls used are

- Boot image loader — an in-hardware mechanism to put the secondary boot image into the device memory during the boot process using DMA.
- External C calls — calls into machine code compiled from C for performance reasons.

It is important to note that the definitions of these external calls are used mainly in the proof side of the project, to specify the behavior of components external to the produced RISC-V program. Their main purpose is to be able to express the statement "if the RISC-V code executes a sequence of instructions that is defined to trigger the external call, the machine's state will change in this exact way."

The main criteria considered when designing the external-call interface for this project were:

- **External-call semantics must be coherent with respect to compiled code.** The goal of this project is to produce a valid RISC-V binary blob with a correctness proof. This means that any external call in the semantics has to be triggered by ordinary-looking RISC-V instructions (those in the binary blob), and we cannot invent arbitrary new instructions to represent external calls.
- **Preserving Bedrock2 proofs.** The Bedrock2 programming language and its proof of compilation correctness are built on top of the machine RISC-V semantics. Since we would like to reuse Bedrock2 and all its proof machinery, it is important to change the RISC-V semantics in a very controlled way, so that it's easy to modify the Bedrock2 proofs.
- **It should be easy to understand the effect of the external call.** The external calls are effectively additional logical assumptions in our correctness proof, so the proof's validity in the real world depends on these assumptions

being right (exactly describing the effects of the mechanism being modelled and logically uncontradictory).

### 3.3.1 External Calls as Undefined Memory Handlers

We chose to represent external calls as handlers of reads and writes to undefined memory. This was inspired by preexisting implementation of an MMIO abstraction in the RISC-V semantics. However, the MMIO abstraction was limited to specifying how values are written to or read from a specific undefined memory address — the definitions of MMIO actions have no access to directly referencing the RAM of the abstract RISC-V machine, except for the exact address that's being read from/written to.

This is not sufficient for our use case, as the external calls we use all modify the memory of the machine. So, our interface for external calls is generalized to allow this access. We choose to represent external call specifications using predicate-transformer semantics: if we want the postcondition $R$ to hold after the call is executed, the semantics gives us a precondition $P(state, R)$ that needs to be satisfied before the call. The interface is shown in Listing 4. It is a parameter to the description of an abstract RISC-V machine — that is, each machine is parametrized by an external specification. (In this project, we use a concrete external specification that describes the functions we want to use.)

### 3.3.2 Boot-image Loader

The Sanctum hardware platform contains a DMA-based mechanism to load a user-modifiable second-stage boot image to the machine's memory. The image loader is controlled MMIO-style, by reading and writing values to predefined memory addresses. Specifically, to activate the loader, the machine issues an MMIO write of the payload's destination address. After the write is received, the loader continuously, nonatomically fills the destination with the payload's data. In the mean time, the machine polls for the loader's completion by repeatedly issuing MMIO reads to a

**Listing 4** The (simplified) external specification interface for an abstract RISC-V machine. Adapted from the Bedrock2 MMIO interface.

```
(* In the following code,
   word - a type of n-bit words
   mem - a dictionary from word to byte, representing the machine memory
   regs - a dictionary from Z to word, representing the registers
   list LogItem - the machine's external call trace *)

Class ExtSpec
    {W: Words}
    {mem : map.map word byte}
    {regs: map.map Register word}
    := {
  (* weakest precondition semantics for ext_load: *)
  ext_load:
    (* given the load address, the current memory, registers and trace *)
    word -> mem -> regs -> list LogItem ->
    (* and some desired proposition about the memory, registers and the
       load's return value after executing the external call, *)
    (mem -> regs -> HList.tuple byte n -> Prop) ->
    (* return a pre-condition that we need to satisfy. *)
    Prop;
  ext_store: (* equivalent *)
}.
```

specific address. Once an MMIO read returns a value greater than 0, this signals that the loader is done and the bootloader can proceed.

Or, in C-like pseudocode:

**Algorithm 1** Using the DMA loader

```
$(*loaderExtCallStart) \leftarrow destinationAddress$
done $\leftarrow 0$
while done $= 0$ do
    done $\leftarrow (*loaderExtCallDone)$
end while
```

Modelling this mechanism poses a challenge because it's nondeterministic: writing to the machine's memory is happening in the background, and the loader doesn't specify the exact order and timing of memory modification. To model this, we use the notion of undefined memory in the riscv-coq semantics — the machine model can have certain ranges of the memory undefined (which is distinct from them being set to 0). Any non-MMIO read from such undefined memory addresses fails, and the execution can't proceed — so the proof of the bootloader's functional correctness also proves that no such reads happened.

We use undefined memory to signify that the loader is working on that memory. First, we write the desired destination address to *loaderExtCallStart* and, as part of the MMIO write's postcondition, the block of predefined size at the address becomes undefined. Then, as we poll *loaderExtCallDone* and eventually read 1, the postcondition makes that memory block defined again, and states that it contains the payload data.

We use the execution trace to model the nondeterministic execution time of the loading mechanism. Each read of the *loaderExtCallDone* MMIO address adds an entry to the execution trace, with a label that lets us distinguish loader-generated events. The behavior of MMIO-reading *loaderExtCallDone* depends on the number of these log entries in the trace — and therefore on how many times it has been read before. For some parameter $k$, the first $k - 1$ reads of *loaderExtCallDone* will do nothing to the memory and return 0. The $k$th read will finalize the loading — change the destination memory from undefined to containing the boot binary — and return 1. All subsequent reads will just return 1.

Effectively, this specification means that for some fixed $k$, the boot image is loaded after $k$ polls. This, combined with the metric-tracking mechanism for the reference RISC-V machine, lets us prove the exact number of instructions it takes to load the boot image as a function of $k$, and as a consequence proves termination for any fixed $k$.

### 3.3.3 C calls

The bootloader uses the following cryptographic functions implemented in C:

```c
void hash(const void * in_data,
  size_t in_data_size,
  hash_t * out_hash);

void init_hash(hash_context_t * hash_context);

void extend_hash(hash_context_t * hash_context,
  const void * in_data,
  size_t in_data_size);

void finalize_hash(hash_context_t * hash_context,
  hash_t * out_hash);

void create_secret_signing_key(const key_seed_t * in_seed,
  secret_key_t * out_secret_key);

void compute_public_signing_key(const secret_key_t * in_secret_key,
  const public_key_t * out_public_key);

void sign(void * in_data,
  size_t in_data_size,
  public_key_t * in_public_key,
  secret_key_t * in_secret_key,
  signature_t * out_signature);
```

More specifically, these functions are thin wrappers around preexisting C implementations of SHA3 [23] and Ed25519 elliptic curve cryptography [20].

To fully implement these C functions as external calls for Bedrock2, we proceed in three stages:

- First, we construct a system to compile the C code into RISC-V and programmatically generate its function metadata in the form of Coq definitions.

- Then, we define our custom, extended RISC-V abstract machine that is capable of executing the abstractly defined effects of the external calls when we jump to their addresses.

- Finally, we connect the RISC-V semantics to Bedrock2, by defining a way to compile Bedrock2 “interact” statements to RISC-V instructions and proving that this compilation is correct w.r.t. our extended RISC-V semantics.

**Compiling and Exposing C Functions**

The first stage of using C function as external calls is getting the actual function assembly code and making it available in Coq. For this, the functions are compiled using GCC [24], each of them fully inlined. This produces an ELF file with a symbol for each function.

Using GNU binutils we then inspect the ELF file and extract the information about each desired function: its name and address in the file. In addition, we read the extended GCC compilation log to see how much stack memory each function requires. (While in general GCC might fail to calculate a function's stack size, the cryptographic functions we use are written using a small static subset of C, so GCC is able to determine the bound on their stack size.) This information is then used to generate a Coq definition describing these functions and containing enough information to compile calls into these functions. An example of such a definition is shown in Listing 5.

Finally, the ELF file is stripped of all the metadata stored before and after the code sections to produce a minimal binary blob containing the functions.

Clearly, this procedure of making compiled C function code accessible to Coq is a part of the trusted base for this project — its correctness depends on the correctness of GCC and the build pipeline. One future direction of this work could be replacing GCC with CompCert [17] to eliminate this need for trust.

**Listing 5** Example generated information about C functions in Coq

```
Definition external_functions := {|
    functions := [
        ...
        {|
            id := "hash";
            addr := (0)%Z; (* address relative to base_offset *)
            stack_use := 64;
        |};
        ...
    ];
    base_offset := 0x"1234";
    total_size := (1234)%Z;
|}.
```

### Writing C Function External Specifications

Compiling the C functions into RISC-V assembly gives us the program-level information we need. But to incorporate C functions into our proofs, we also need a description of their behavior. We do this by using our extension-specification mechanism, by giving a specification for each such function.

Semantically, we want jumping into the function's address to behave like "calling a function": what happens at the program level when you jump into a valid GCC-compiled function is that some side effects happen, caller-saved registers are clobbered, and eventually the execution jumps back to the value of the *ra* register.

The tool we have to achieve this is the extensibility of the implementation of our RISC-V abstract machine. We want to make sure that while in the "real world" the jump to an external function would execute some external assembly, in the "proof world" it should trigger the external call, and its specification should take effect. There are many ways of doing that, with the trade-off being flexibility of defining external calls versus the difficulty of proving that they are coherent with respect to the rest of the system. In this project, we use the action of *undefined memory loads* to intercept external calls.

In the real world, an external call begins by jumping into a function somewhere in the memory, and we assume we will eventually reach the instruction to jump to the return address. In the proof world, the memory of external-call instructions is represented as undefined — so a jump to it is followed by a fetch of the first instruction. That fetch tries to read undefined data and triggers the RISC-V external-specification handler. The specification handler's postcondition takes effect, saying that a particular memory modification happens and the instruction fetched is a Jump instruction to the value held in the $ra$ register. In other words, we skip all but the last instruction of the external call (instead "magically" applying the postcondition) and make it so the first instruction fetch returns the Jump instruction that returns from the external call back into the outer function flow.

While some parts of the external specification are unique to the function being specified — the `hash` function will behave differently from `sign` — a lot of the behavior is shared. The functions themselves have some similarities:

- The function return types are *void*.
- The function arguments are either integers or pointers to byte arrays whose sizes are either statically known or passed as other arguments.

In addition, we need to include the cdecl calling convention in the specification. For that, we split the specification into two parts: the common part that talks about the calling convention and the return type, and function-specific parts that actually describe what each function does to the device's memory. As an example, the specification of an external call to the hash function can be seen in Listing 6.

### Connecting Bedrock2 with External-Call Semantics

The next step is to make it so that invocations of Bedrock2's `interact` statement would actually compile to the instructions necessary to trigger the RISC-V external specification described above. For that, we need to define a way to compile these `interact` statements into RISC-V and prove the correctness of this step.

The compilation step is straightforward. We only need to implement the *cdecl* calling convention: copy over caller-saved registers to a preassigned memory location, set up the stack pointer and the return address, and jump to the address of the desired function.

The semantics-transfer-correctness theorem is more complicated. We need to prove that our description of what an external call does in Bedrock2 semantics corresponds to what the compiled code does in our custom abstract RISC-V machine semantics. The actual effects of individual cryptographic functions are the easy part — all of our functions only modify the noninstruction memory, and the effect of that is the same in Bedrock2 and RISC-V semantics. The mechanically difficult part of the proof is showing the correctness of our implementation of the *cdecl* calling convention. This includes showing that the register values are preserved through the function call and that the call arguments are copied into the right registers.

In this thesis, we stated the semantics-transfer-correctness theorem in Coq and were able to prove several lemmas to use in its proof, but in the end had to assume it as an axiom instead of fully proving it for time reasons.

## 3.4 Assembly

Raw RISC-V assembly is used to write the pre- and postambles of the bootloader. To avoid needing to parse a separate assembly language, we write down the instructions as variants of the Instruction algebraic data type in Coq, defined by the riscv-coq project. This also lets us use Coq functions for lightweight metaprogramming capabilities.

The two main assembly routines used in the code are the memory-clearing routine, which zeroes out a particular block of memory, and the routine to interact with the DMA boot image loader and load the image in the desired location. The routine code can be found in Listing 7.

While the DMA loader looks very much like normal assembly code, the memory-zeroing routine is more interesting. It is parametrized by the loop-unrolling level — how many Sw instructions should be executed in a single loop iteration. Different numbers achieve different balances between performance and code size. The main benefit of using a parametrized description is that this lets us prove the correctness of this routine for any unrolling level. The user can then choose different levels for different places in the codebase and get correctness proofs for free.

Proof-wise, both the DMA boot-image loader and the memory zeroing routine's correctness lemmas are proven using induction. For the zeroing routine, we simply induct on how much memory is left to erase. The image loader is more complicated: its external specification is parametrized by a number $k$ — how many polls it will take for the loading to finish. Holding this number constant, we can induct on $\big(k-$(current loop iteration)$\big)$. This amounts to proving

- **Base case:** if $\big(k-$(current loop iteration)$\big) = 0$, we have done all the polling we need — this iteration will finish the loading and prove the postcondition.

- **Inductive step:** if $\big(k-$(current loop iteration)$\big) > 0$, we have more polling to do — we need to poll once and then apply the inductive hypothesis.

## 3.5 Bedrock2 Part

The Bedrock2 part of the bootloader code mainly consists of a straightline sequence of function calls. Two of them — memory copying and zeroing routines — are themselves functions written in Bedrock2, while the others are external calls of cryptographic C functions, expressed as `interact` statements.

The Bedrock2 code contains naive implementations of routines to zero out/copy blocks of memory (corresponding to `memset` and `memcpy` functions in C). While the project already contains a RISC-V assembly routine to zero out memory, it would require additional software-engineering work to make it usable from Bedrock2, so rewriting and reproving a separate Bedrock2 version was considered easier. Listing 8 contains the code and specification of the memory-zeroing-out routine, and the memory-copying routine is defined analogously. Both routines are simple loops so their correctness can be proved by induction on how many iterations are left. Using lemmas and tactics from the Bedrock2 framework, proving that these functions adhere to specifications is much less laborious than proving properties of the assembly routines.

The whole Bedrock2 part of the code is represented as a function (the "main" function), its effect captured by a specification. Specifically, since the function takes no arguments and has no returns, its specification describes the changes in the machine's memory and IO trace. The proof of the specification simply combines the specifications of memory-modification routines and external calls applied sequentially, keeping track of the states of the memory and registers along the way. This is again made easy by the Bedrock2 framework and its lemmas and tactics, and by the fact that the effects of the external C functions are deterministic (that is, a specification maps the machine state pre call to a single unique machine state post call).

The "main" function is then compiled into RISC-V, which gives us both the RISC-V instruction list and the proof of correctness, transported from Bedrock2 to the RISC-V semantics. Both of them are then combined with the preamble and postamble to produce the whole RISC-V assembly block and its correctness proof.

### 3.5.1 Arrays and Structures

Most functions used in the Bedrock2 part operate on byte arrays. Ideally, we would want to declare some of these byte arrays as local variables allocated on the stack and let the Bedrock2 compiler manage them for us. Unfortunately, at the time of writing Bedrock2 did not support arrays and structures as native datatypes. To get around this, we define a structure that contains all local byte arrays and allocate a memory block for it by hand (i.e. put it in a predefined place in the device's memory). In practice this means including a statement about this block's existence in the "main" function's precondition. We then use the fields of this "stack" structure in place of local variables.

**Listing 6** Example RISC-V external-specification definition.

```
Definition common_part_of_ext_spec
    (n numargs: nat) specific_ext_spec initMem initReg post :=
    (* if the following preconditions hold, *)
    exists raddr args,
      (* ra contains the value of some address *)
      map.get initReg ra = Some raddr /\
      (* a predicate that says the registers contain the call's input
         arguments *)
      arg_relationship numargs initReg args /\
      (* after the call, the following postconditions hold: *)
      (forall finalReg,
          (* the registers that aren't caller_saved_registers are
             preserved *)
          (forall r, (~(In r caller_saved_registers) \/ r = ra) ->
             map.get initReg r = map.get finalReg r) ->
          (* the function-specific postcondition holds, and the read
             returns a Jalr instruction. *)
          let read_result :=
            (LittleEndian.split n (encode (Jalr zero ra 0%Z))) in
          specific_ext_spec initMem args
            (fun (finalMem: mem) => post finalMem finalReg read_result)).

Definition hash_spec := fun (initMem: mem) (args: list word) post =>
  forall finalMem,
  exists inPtr inSize outPtr inData outDataBefore R,
  (* given that the initial memory contains two memory regions *)
  ((ptsto_array inPtr inData) *
   (ptsto_array outPtr outDataBefore) *
    R) initMem /\
  (* and our arguments passed are reasonable, *)
  args = [inPtr; inSize; outPtr] /\
  length inData = (Z.to_nat (word.unsigned inSize)) /\
  ((* after the execution, the input block didn't change *)
    ((ptsto_array inPtr inData) *
    (* while the output block now contains the result of the functional
       spec of the hash function *)
     (ptsto_array outPtr (hash_functional_spec inData)) *
     R) finalMem ->
    post finalMem).
```

**Listing 7** The assembly routines to use the DMA loader and zero out a block of memory .

```
Definition DMA_loader :=
      (compile_lit_64bit t0 BOOT_IMAGE_DEST)
      ++ (compile_lit_64bit t1 BOOT_IMAGE_LOADER_BASE)
      ++ [[
           Sw t0 t1 0;
           Fence 0 0
         ]]
      ++ (compile_lit_64bit t1 BOOT_IMAGE_LOADER_POLL))
      ++ [[
           Lw t0 t1 0;
           Bnez t0 (-4);
           Fence 0 0;
           Fence_i;
           Jr ra
         ]].

    (* generates a list of assembly instructions that zero out n
       consecutive memory addresses starting with register t *)
    Fixpoint n_32s_zeroed (n: nat) (t: Register) : list Instruction :=
      match n with
      | O => nil
      | S p =>  [[ Sw a1 zero (Z.of_nat n * -4)  ]] ++ (n_32s_zeroed p t)
      end.

    (* an assembly loop to zero out memory between a0 and a1, parametrized
       by how much the loop should be unrolled *)
    Definition clear_memory (n : nat) : list Instruction :=
        := n_32s_zeroed n a1
           ++ [[
                 Addi a1 a1 (-4 * n);
                 Blt a0 a1 (-4 * (n+1));
                 Jalr zero ra 0 ]].
```

**Listing 8** The implementation and specification of `zero_out_memory`. Memory-copying function defined similarly.

```
Definition zero_out_memory := ((
      (* function name *)
      "zeroOut",
      (* argument list *)
      (addr :: numBytes :: nil),
      (* list of returned variables *)
      nil,
      (* function body *)
      bedrock_func_body:(
        ctr = (constr:(0));
        while (ctr < numBytes) {
            store4(addr + ctr, constr:(0));
            ctr = (ctr + constr:(4))
        }
      ))).

Instance spec_of_zero_out_memory: spec_of "zeroOut" :=
  fun functions =>
    forall initTrace initMem blockAddr blockSize initData R,
      (* given that the initial memory contains an array initData at
      blockAddr *)
      ((ptsto_array blockAddr initData)  * R) initMem) ->
      (* and they're all appropriately sized, *)
      Z.to_nat (word.unsigned blockSize) = List.length initData ->
      (word.unsigned blockSize) + (word.unsigned blockAddr) < 2^width ->
      (* if a function is called with arguments [blockAddr; blockSize], *)
      WeakestPrecondition.call functions "zeroOut" initTrace initMem
        [blockAddr; blockSize]
        (* the following postcondition holds: *)
        (fun finalTrace finalMem rets =>
            (* the IO trace won't change *)
            initTrace = finalTrace /\
            (* nothing will be returned *)
            rets = [] /\
            (* and the final memory will contain an array of zeroes
               instead of initData *)
            ((ptsto_array blockAddr
                         (List.repeat Byte.x00 (word_to_nat blockSize))))
             * R) finalMem).
```

# Chapter 4

# Discussion

## 4.1 Ergonomics of Coq

This project used Coq as the programming language and software-engineering environment. While for the most part the developer experience was very pleasant (and tremendously improved by Proof General [1] / Company-Coq [21]), there were a few aspects that we found to slow down Coq development. In this section we want to share our experience with Coq ergonomics-wise and discuss the difficulties we experienced.

### 4.1.1 Implicit Arguments

Coq users often want to make their definitions and proofs as abstract as possible. To help that, Coq supports *implicit arguments*: arguments that do not have to be provided explicitly if their values can be inferred from context. They can be very useful in situations where an argument's type depends on a previous argument. For example, in definitions like

```
Fixpoint reverse {A} (l: list A) : list A := ...
```

it is much easier to write `reverse [0; 1; 2]` instead of `reverse nat [0; 1; 2]`. However, if used too much, implicit arguments can hurt code quality instead of helping it.

### Hidden by Default

By default, implicit arguments are not printed in the proof view or when inspecting terms. This can make debugging issues confusing in situations where terms are equal in their explicit parameters, but not in implicit parameters.

A prototypical situation that happened multiple times throughout the project is when in the proof view we need to prove `P a b c`, and in our hypothesis list we see `H1: P a b c`. Hoping to automate the proof, we write an Ltac script:

```
match goal with
| H: ?A |- ?A => apply H
end.
```

Unfortunately, the match statement fails. Eventually, after some confusion, the author would remember to set the printing mode to print the implicit variables, and would realize that the terms are indeed different: `@P x a b c` and `@P y a b c` with $x \neq y$.

### Implicits Without Full Simplification

In situations like above, the author would sometimes encounter an even stranger situation: while the match statement would fail, directly writing `apply H1` would work. This was again caused by the (structural) mismatch of implicit parameters. However, in this case they were definitionally equal — which is why the `apply` tactic would work.

One way to fix this problem is to apply `simpl` to both the goal and the hypothesis before applying the match statement. However, due to the size and construction of proof terms when using Bedrock2, simplifying everything would take an impractically long time.

In this project, we solved this problem by writing a tactic that would selectively simplify all implicit parameters in commonly used definitions and calling the tactic before using match statements. The downside of this approach is that every such definition has to be included in the tactic explicitly. An alternative approach would be to use the `simpl never` argument annotation to make sure Coq never simplifies terms that have the potential to be slow. However, the slowness is not a binary property, and it sometimes makes sense to decide whether to simplify these terms on a case-by-case basis.

### 4.1.2 Performance

While Coq's performance has improved significantly throughout the years, there are still cases where slow performance becomes a blocker to verification work.

- **Slow tactics.** Working with Bedrock2 often involves manipulating big terms where even simple normalization might take a lot of time. When working with such terms, it is important either to simplify relevant subterms selectively or to aggressively use `simpl never` annotations. Naturally, most other tactics will also be slow for such terms.

- **Slow QED.** In some cases (especially after constructing proofs about Bedrock2 program specifications), going through the proof is fast enough but the QED step takes a prohibitively long time (at least several hours, with no termination observed). For this project, we used a workaround from the Bedrock2 project: in cases where QED is slow and it is not clear how to make it faster, we replace it with

  ```
  Lemma ...
  Proof.
    ...
    all: fail.
  Admitted.
  ```

  which has the effect of failing if there are any unsolved goals left. Since we end the proof with `Admitted`, we lose some correctness guarantees — it is possible to construct proofs that pass the proof mode but get rightly flagged as incorrect during the QED checking. Fortunately, it is (empirically) difficult to build such proofs by accident so the workaround provides enough confidence in the proof's correctness.

## 4.2 Future Work

### 4.2.1 Trusting GCC

By far the biggest weakness of this verification approach is its use of cryptographic C functions, and as a consequence GCC. While the C functions are written in a static subset of C and don't use any "dangerous" features, ideally we would want an end-to-end proof of correctness instead of having to include GCC in our trusted code base. There are several ways to address this:

- **Rewriting the cryptographic functions in Bedrock2**. At the time of writing the Bedrock2 compiler can't optimize code very well, so this would likely make the bootloader unusably slow. In addition, since the bootloader uses public-key cryptography, implementing the functions would require a lot of time and effort.

- **Integrating with fiat-crypto**. Fiat-crypto [9] is a project for generating correct-by-construction elliptic-curve-arithmetic code. While its original version only supported outputting C code, there is ongoing work to make fiat-crypto output Bedrock2 code and connect them on a proof level. Implementing asymmetric cryptography operations using fiat-crypto would save a lot of time and would probably improve the performance by a little bit (due to better arithmetic-operation constructions), but we would still lose out on register allocation and peephole optimizations of GCC.

- **Integrating with CompCert**. Another option is to reuse the cryptographic C code but compile it with CompCert. This would get us provably correct compilation and good performance (potentially within a factor of 2 from GCC's output). However, it would be necessary to build a bridge between Bedrock2 semantics and CompCert semantics, which might be nontrivial.

### 4.2.2 Not fully accounting for execution time

We prove (subject to assumptions) that the bootloader implemented for this project terminates, but one might also be interested in bounding its running time more strongly. In fact, the Bedrock2 framework already supports such reasoning: the riscv-coq semantics include `MetricRiscvMachine`, an implementation of an abstract RISC-V machine that keeps count of various metrics (like the number of instructions executed, reads and writes), and we can transport Bedrock2 proofs to proofs about this metric machine. Unfortunately, this approach doesn't capture the running time of cryptographic C functions — since we describe their effects using external calls, each invocation of such a function takes one step of the abstract RISC-V machine. Some potential ways to address it are:

- **Implementing cryptographic routines in Bedrock2 as described above**. This would give us metric tracking for free.
- **Using static analysis to bound execution time of C code**. Some static-analysis tools (e.g. [10]) support deriving worst-case running-time bounds for C programs written in "safe" subsets of C. Such tools could be used to analyze the C cryptographic functions and record the results as part of the generated metadata file for Coq, which we could combine with the rest of the proof. This would not give us end-to-end security — the static-analysis tool would have to be trusted — but it might be a worthwhile trade-off for practical applications.

### 4.2.3 Finishing the correctness proof

In this thesis we have stated the correctness theorem in Coq and proved most of the lemmas necessary for it, but we have not succeeded in unconditionally proving the full correctness theorem. While the author believes that the parts left to prove are correct and only pose mechanical/proof engineering difficulty, it is obviously necessary to finish the correctness theorem before this implementation can be fully trusted.

# Bibliography

[1] David Aspinall. Proof General: A Generic Tool for Proof Development. In Susanne Graf and Michael Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 38–43, Berlin, Heidelberg, 2000. Springer.

[2] Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Mike Shulman, Matthieu Sozeau, and Bas Spitters. The HoTT Library: A formalization of homotopy type theory in Coq. *arXiv:1610.04591 [cs, math]*, December 2016.

[3] Ian Clester, Samuel Gruetter, Andy Wright, and Adam Chlipala. RISC-V Specification in Haskell. Accessible at https://github.com/mit-plv/riscv-semantics. Git commit 8d00 fa6f 81d6 c1e9 b5d3 b579 7144 0158 7f7d c119.

[4] Scott Constable, Rob Sutton, Arash Sahebolamri, and Steve Chapin. Formal Verification of a Modern Boot Loader. *Electrical Engineering and Computer Science - Technical Reports*, August 2018.

[5] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. Technical Report 564, 2015.

[6] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[7] Andres Erbsen and Samuel Gruetter. Bedrock2: language and compiler for verified low-level programming. Accessible at https://github.com/mit-plv/bedrock2. Git commit e892 d189 1761 49c1 ef8a bafe a370 03b9 671d b977.

[8] Andres Erbsen and Samuel Gruetter. RISC-V Specification in Coq. Accessible at https://github.com/mit-plv/riscv-coq. Git commit 94c7 265e c21f b047 6ee3 ee78 cae9 3de4 1dde d99e.

[9] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1202–1219, San Francisco, CA, USA, May 2019. IEEE.

[10] C. Ferdinand. Worst case execution time prediction by static program analysis. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pages 125–127, Santa Fe, NM, USA, 2004. IEEE.

[11] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, and Sidi Ould Biha. A machine-checked proof of the odd order theorem. In *International Conference on Interactive Theorem Proving*, pages 163–179. Springer, 2013.

[12] Jason Gross, Adam Chlipala, and David I. Spivak. Experience Implementing a Performant Category-Theory Library in Coq. *arXiv:1401.7694 [cs, math]*, April 2014.

[13] Brigid Halling. *Towards a Formal Verification of the Trusted Platform Module.* PhD thesis, University of Kansas, 2013.

[14] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[15] Gerwin Klein, Michael Norrish, Thomas Sewell, Harvey Tuch, Simon Winwood, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, and Rafal Kolanski. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP '09*, page 207, Big Sky, Montana, USA, 2009. ACM Press.

[16] Ilia Lebedev and Jules Drean. Secure Bootloader for the Sanctum Project, May 2020. Accessible at https://github.com/mit-enclaves/secure_bootloader. Git commit 5527 f1ed 36bc 5bfe efaf 8e19 556d 4633 b76a b17c.

[17] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert - A Formally Verified Optimizing Compiler. In *RTS 2016: Embedded Real Time Software and Systems,8th European Congress, SEE*, Toulouse, France, January 2016. hal-01238879.

[18] Aybek Mukhamedov, Andrew D. Gordon, and Mark Ryan. Towards a Verified Reference Implementation of a Trusted Platform Module. In Bruce Christianson, James A. Malcolm, Vashek Matyáš, and Michael Roe, editors, *Security Protocols XVII*, Lecture Notes in Computer Science, pages 69–81, Berlin, Heidelberg, 2013. Springer.

[19] Christine Paulin-Mohring. *Introduction to the Calculus of Inductive Constructions*, volume 55. College Publications, January 2015.

[20] Orson Peters. Ed25519 C Implementation, May 2020. Accessible at https://github.com/orlp/ed25519. Git commit 439a c7a6 2b4d b959 2257 4fd1 83cf 3215 0058 0c82.

[21] Clément Pit-Claudel and Pierre Courtieu. Company-Coq: Taking Proof General one step closer to a real IDE, January 2016.

[22] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Copenhagen, Denmark, 2002. IEEE Comput. Soc.

[23] Markku-Juhani O. Saarinen. TinySHA3, May 2020. Accessible at https://github.com/mjosaarinen/tiny_sha3. Git commit dcbb 3192 047c 2a72 1f5f 851d b591 871d 4280 36a9.

[24] Richard M. Stallman and GCC DeveloperCommunity. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3.* CreateSpace, Scotts Valley, CA, 2009.

[25] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit A. Seshia. A Formal Foundation for Secure Remote Execution of Enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2435–2450, Dallas Texas USA, October 2017. ACM.

[26] The Coq Development Team. The Coq Proof Assistant, version 8.11.0, January 2020. Accessible at https://coq.inria.fr. Version 8.11.

[27] The Coreboot Development Team. coreboot. Accessible at https://www.coreboot.org. Accessed 05/10/2020.

[28] Benedikt and Grayson Daniel and others Voevodsky, Vladimir and Ahrens. UniMath/UniMath, May 2020. Accessible at https://github.com/UniMath/UniMath. Git commit 2aec 9849 a459 3df6 fb40 e598 e044 0072 1b3b fa62.

[29] Andrew Waterman, Yunsup Lee, David A. Patterson, Krste Asanovic, Volume I. User-level Isa, Andrew Waterman, Yunsup Lee, and David Patterson. *The RISC-V Instruction Set Manual.* 2014.