# Formal Verification of an Implementation of the Roughtime Server

by

Christian Altamirano

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

September 2021



Author .....
Department of Electrical Engineering and Computer Science
August 6, 2021

Certified by .....
Adam Chlipala
Associate Professor
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chairman, Department Committee on Graduate Theses

**Formal Verification of an Implementation of the Roughtime Server**

by

Christian Altamirano

Submitted to the Department of Electrical Engineering and Computer Science
on August 6, 2021, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

## Abstract

Formal verification has been used in the past few decades to prove correctness of programs. This thesis provides a verification of a simpler implementation of Roughtime [1], a protocol that consists of securely querying the current time via a client-server interaction. The tool that was used is Bedrock2 [3], a work-in-progress Coq framework suitable for reasoning about low-level code, developed in the Programming Languages and Verification group at MIT CSAIL.

Thesis Supervisor: Adam Chlipala
Title: Associate Professor

# Acknowledgments

I'd like to thank Adam Chlipala for his mentorship throughout the time I've been working in his group, starting from letting me join his group as an undergrad up until now. Thank you for giving me guidance as well as reassurance even when I struggled to keep up with the work. I'd also like to thank Sam Gruetter for his help during the entire time I've been part of this group. I enjoyed our meetings. Overall they were a very productive time from which I've learned a lot and got better at writing Coq code. Lastly, I thank my family for always staying with me and supporting me.

# Contents

# Chapter 1

# Introduction

Formal verification is an area within computer science that focuses on proving correctness of programs and has shown promising results throughout the years. At a high level it consists of representing programs as mathematical objects, about which one can state and prove theorems. In certain cases this even seems necessary for high-importance systems, for instance a bug in an online payment system could lead to a vulnerability that could cost millions of dollars. It is known that extensive testing as well as code review can catch most bugs but these methods do not offer an actual guarantee that no bugs exist. Formal methods allow us to be certain that, given a specification, a program behaves as expected.

There exist several languages and interactive proof assistants such as Coq, Isabelle, etc. as well as tools and frameworks that facilitate verification. At a high level, verification consists of a specification, the source code, and a proof of correctness. The specification is generally given by the user, the source code is simply the written code, and the proof of correctness is a machine-checked proof that the code satisfies the desired specifications. This is what proof assistants aim to facilitate and automate if possible. There are some ideal scenarios in which the proof step is automated, called push-button automation, in which case the user would have to input their program together with a specification and get an output that tells them whether or not it is correct. One example is HyperKernel [10]. However, in general cases we are still far from achieving such a degree of automation, which means that in most cases a user

would have to use their mental power and write a proof using a proof assistant.

While this program verification process seems straightforward, it also comes with many challenges such as what is a good specification, what assumptions should we make, and what is our trusted code base. As will be discussed later, tools such as Bedrock2 include verified compilers from source language to machine code, so in this case we have to assume that hardware reads machine code correctly and behaves as expected. Another assumption we make is the correctness of the OCaml compiler and Coq.

In this thesis, we have worked on verifying a partial implementation of the Roughtime protocol using Bedrock2. At a high level, Roughtime consists of a server and a client, both of which operate under a defined protocol for request and response messages with the objective that the client can securely make a request to the server and the server can securely send the current timestamp to the client. We have made a few modifications to the Roughtime server implementation and verified that this refines the Roughtime message protocol and that it does not produce failures such as memory corruption. We decided not to verify the security and cryptographic related demands about the Roughtime message response because so far there is no standard method to even state such goals. We also made use of Coq's extensive type system for our specifications, which turned out to be useful.

# Chapter 2

# Background and Related Work

In this chapter we will discuss what programming languages are being used as well as tools that make use of them. We also give a brief introduction to relevant concepts, and finally we discuss what similar work has been done.

## 2.1 The Coq proof assistant

Coq is a formal proof assistant written in OCaml. It is purely functional and has dependent types as it uses the theory of the Calculus of Inductive Constructions. In Coq's type system, most types are defined as arrow types which indicate function types or inductive types which can be recursively defined. The arrow type $A \to B$ indicates the type of functions that map an element from type $A$ to an element of type $B$, and the inductive datatypes can be defined by the user. For example, the natural numbers can be defined in this way:

```
Inductive nat : Set :=  O : nat | S : nat -> nat
```

This means that a natural number is either zero or the successor of another natural number. In this example, the natural numbers are a recursive type because a natural number can be the successor of another natural number.

In Coq, a proof consists of building a proof term using the application of axioms or previously proven theorems. Part of the process is automated by Coq to a fair

extent just so that the user does not have to manually construct a proof term that could be much bigger than the statement of a theorem itself. To do this Coq offers a proof mode in which the user has a proof state that displays the hypotheses and the desired goal. Coq also provides a language called Ltac used to create tactics to achieve further automation.

## 2.2 Hoare logic

Several program verifiers use Hoare logic and/or its extension separation logic to reason about a program's behavior. Hoare logic consists of propositions of the form $\{P\}C\{Q\}$ where $P$ and $Q$ are assertions about the program state and $C$ is the program itself. We call such propositions Hoare triples. The Hoare triple $\{P\}C\{Q\}$ means that if a program state satisfies $P$, then after running $C$ it must satisfy $Q$. Hoare logic also uses axiomatic semantics to derive Hoare triples: for instance, one rule is the following:

$$\frac{\{P\}\, C_1\, \{Q\} \qquad \{Q\}\, C_1\, \{R\}}{\{P\}\, C_1; C_2\, \{R\}}$$

where $C_1; C_2$ represents the sequence of commands $C_1$ and $C_2$. Then, this rule allows us to prove a Hoare logic predicate for a sequence of commands. Another interesting rule is the while loop rule:

$$\frac{\{P \wedge B\}\, S\, \{P\}}{\{P\}\, \textbf{while}\, B\, \textbf{do}\, S\, \{\neg B \wedge P\}}$$

The predicate $P$ is called the loop invariant that cannot be inferred from the pre and post-conditions; it is known to be an undecidable problem to come up with an invariant, and therefore in many cases it has to be specified by the user.

If we use this while loop rule, Hoare logic does not actually guarantee total correctness of programs as technically it proves that if the program has finished running then it is correct. However, there is no termination guarantee. Fortunately this can

be solved with a small extension that is in fact used in Bedrock2.

## 2.3 Separation logic

If we only used Hoare logic for reasoning about programs we would have trouble being able to reason about disjoint memory locations: let's take a look at the following seemingly trivial example: Let's assume $A$ is an array and $i$ and $j$ are some positive integers.

$$\{A[i] = 4 \wedge A[j] = 4\}\, A[i] := 3\, \{A[i] = 3 \wedge A[j] = 4\}$$

Here the command stores 3 in the $i^{th}$ position of the array $A$, and therefore it seems obvious that after such a command is executed it happens that $A[i]$ is 3 and $A[j]$ stays as 4. However, this is true only if $i \neq j$. Otherwise such a proposition is actually false. Then, we can write something like the following:

$$\{i \neq j \wedge A[i] = 4 \wedge A[j] = 4\}\, A[i] := 3\, \{i \neq j \wedge A[i] = 3 \wedge A[j] = 4\}$$

which seems slightly verbose, and if we were to consider a big array we would have many conjuncts of the form $i \neq j$ around, and overall it would be cumbersome to reason about programs involving array manipulations. Here is where we make use of separation logic, an extension of Hoare logic that allows reasoning about different memory locations. In separation logic, assertions are over a chunk of memory or heap, and we have two binary operators on such assertions referred to as *star* and *magic wand*.

- The star operator $P * Q$ is an assertion on a heap $h$ that states that $h$ can be split in two disjoint parts $h_1, h_2$ such that $P$ holds in $h_1$ and $Q$ holds in $h_2$.

- The magic want operator $P \mathrel{-\!\!*} Q$ is an assertion on a heap $h$ that states that $h$ can be extended with another disjoint heap $h'$ such that $P$ holds on $h'$ and $Q$ holds on $h \cup h'$.

In the previous example, the triple will now look like the following:

$$\{A[i] = 4 * A[j] = 4\}\, A[i] := 3\, \{A[i] = 3 * A[j] = 4\}.$$

Note that the assertion $A[i] = 4 * A[j] = 4$ already implies that $A[i]$ and $A[j]$ represent disjoint locations in memory and therefore $i \neq j$. It turns out that for our purposes we would only need the star operator as we will only manipulate relatively simple array data structures.

## 2.4 Related work

There exist several other tools as well as projects that use formal verification. Some of those automate only a portion of the verification process, and others aim to fully automate verification at the expense of some trade-offs. For instance, a perfect fully automated verifier that can tell us everything about a program cannot exist, as several proof states involve existentials, quantifiers and recursive types which make the goals undecidable.

For example, Verifast [7] is a tool with a similar goal as Bedrock2: it contains a domain specific language (DSL) that allows the user to specify pre and post-conditions (these two are part of the specification) to a program, then uses symbolic execution to run forward through the program which then passes a formula to an SMT solver that proves or disproves the formula. In this DSL, the while loop structure requires an extra predicate for the invariant specified by the user. In this project they are able to automate the process via heuristics such as bounding the depth of recursive functions, and even then they have limitation such as not handling conditional branches because they exponentially increase the size of the SMT query.

Another interesting example is Ironfleet [6]. This tool aims at automatically verifying distributed systems, which is much more complex than verifying a single program. They make use use of Dafny, a tool that uses Hoare logic to reason about programs that works similarly as the example above. It has a similar DSL and also transforms

goals into SMT queries.

Other than just programs, researchers have also gone farther and verified entire operating systems. This is the case of Hyperkernel, CertiKOS [5] and seL4 [8]. These three implement operating system kernels and prove their functional correctness. In fact, the latter two even define and prove security properties such as non-interference: there must not be any data exposure from Alice to Bob. Another project that also uses this security property is SFSCQ [11], a file system with a proof of its correctness and security.

Finally, there is a very similar tool as Bedrock called VST (Verified Software Toolchain) [2] which is built on top of CompCert [9], an existing verified compiler for a large subset of C. Like Bedrock2, VST also makes use of separation logic and provides lemmas and tactics for users to prove correctness of their implementations. One relatively small difference between these two is that VST does not prove termination of programs but Bedrock2 does, by making a small extension to the program logic rules.

# Chapter 3

# The Bedrock2 project

Bedrock2 is a tool built entirely on Coq that serves as a framework for low-level code formal verification. It includes a C-like DSL that we will refer to as Bedrock C and a verified compiler composed of several layers that translate code from Bedrock C all the way down to RISC-V machine instructions. That compiler is written entirely in Coq and makes very good use of Coq's datatypes to formally define such languages. Bedrock2 uses separation logic to reason about program behaviors and also includes several lemmas and tactics to facilitate such proof. In the following sections we will discuss in more detail how this tool works.

## 3.1 Bedrock2 semantics

By making use of Coq's inductive datatypes, programs in Bedrock are defined as commands which can have several forms; for instance, they can be an assignment of the form $x := 1$ or simply sequences of commands. Many compilers make use of object oriented programming and define classes for the abstract syntax tree of a given programming language; in this case we make use of Coq's inductive datatypes for this purpose.

```
Inductive cmd :=
| skip
```

```
| set (lhs : string) (rhs : expr)
| unset (lhs : string)
| store (_ : access_size) (address : expr) (value : expr)
| stackalloc (lhs : string) (nbytes : Z) (body : cmd)
| cond (condition : expr) (nonzero_branch zero_branch : cmd)
| seq (s1 s2: cmd)
| while (test : expr) (body : cmd)
| call (binds : list string) (function : string) (args: list expr)
| interact (binds : list string) (action : string) (args: list expr).
```

Similarly, expressions are terms such as $x + 1$ that can also be defined with datatypes as follows:

```
Inductive expr  : Type :=
| literal (v: Z)
| var (x: string)
| load (_ : access_size) (addr:expr)
| inlinetable (_ : access_size) (table: list Byte.byte) (index: expr)
| op (op: bopname) (e1 e2: expr).
```

Here `bopname` stands for binary operation name such as adding or subtracting, which is widely used in programming languages and specifically in C. This is is defined as follows:

```
Inductive bopname := add | sub | mul | mulhuu | divu | remu | and | or | xor
    | sru | slu | srs | lts | ltu | eq.
```

Also, the type `access_size` is a simple variant type

```
Variant access_size := one | two | four | word.
```

This is simply the size of a machine word, which can be either one byte, two bytes, four bytes or a word in the target architecture (if 64-bit then it has 8 bytes). This

allows for flexibility for defining word byte sizes which is especially useful for writing low-level code.

Given these definitions, any program in Bedrock C (and a substantial subset of C) can be written as a command. As for actually writing code, in normal cases a parser would be needed but in this case Coq already has a built-in notation tool that allows one to define custom notations and parse them as we process lines of Coq code. This might not be in general enough to parse larger programming languages but is enough for our purposes.

## 3.2 Bedrock2 separation logic

We have explained in the previous chapter how separation logic intuitively works, and here we will explain how this is actually defined and implemented. The most basic memory predicates are `scalar8`, `scalar16`, `scalar32` and `scalar`, used for words of size 1, 2, 4, $w$ bytes respectively where $w$ is the width of a pointer in bytes in the target architecture.

Each of these definitions is a memory predicate on two words: a pointer *addr* and a value $x$, simply stating that there is a value $x$ at address *addr* of word size depending on which scalar we use.

A slightly more complicated separation logic predicate is defined using the `array` fixpoint definition. At a high level, it is a predicate on a start address $s$ and a list of words $l$ and states that starting at address $s$ we can find the words $l$ contiguously. Of course, we also need to specify a word width for how many bytes separate the words as well as a memory predicate, which is generally one of the scalar predicates defined in the previous paragraph.

## 3.3 The Bedrock2 compiler

As mentioned before, Bedrock2 also includes a verified compiler from source code down to RISC-V instructions. This compiler is written in Coq and makes use of the

datatypes in the previous section as well as different datatypes for RISC-V instructions. By verified compiler here we mean that for any program in the source code (Bedrock C) the generated RISC-V code should behave as expected based on the source code. This means that if we were to use it, our trusted code base would be only the Coq programming language and the RISC-V semantics.

In this thesis we don't make use of this compiler because we would have to either write the entire Roughtime server implementation in Bedrock C or write a partial implementation but then somehow add the generated RISC-V instructions on top of the rest of the implementation, which seems quite tedious. Therefore, our focus stays at the top-level Bedrock C source code.

## 3.4 Bedrock2 program logic

For a given program we want to make assertions and be able to prove properties based on inference rules such as the one explained in 2.2. Bedrock2 also includes a logic to reason about these programs in this way.

This includes lemmas about programs as well as tactics to automate proving. For example, one lemma could have the form of

```
Lemma load_word_of_sep addr value R m (Hsep : ((scalar addr value) * R) m)
    : Memory.load Syntax.access_size.word m addr = Some value.
```

This lemma states that if we have a value at some address then calling load on that address gives us that value.

Bedrock2 also includes tactics for the use of these lemmas, as it would be very time-consuming for the user to memorize all the lemmas and figure out when to apply them. Bedrock2 has a tactic called `straightline` that processes code line by line and leaves subgoals if needed or fails if it's not possible to proceed. For example, if somehow we are stepping through a line in which we are loading an address, the lemma `load_word_of_sep` will be applied. Ideally one would be able to just use `straightline` and be able to step through any line of code that is not a while

loop. As for while loops, the user has to specify an invariant and a well-founded measure (well-founded means that it must have a minimum element and guarantees the loop termination); in most examples of Bedrock2, we use decreasing for loops as opposed to increasing for loops as this facilitates the measure definitions and the proofs of subgoals. Finally, many of the tactics also involve linear arithmetic solvers for equations involving the usual integers, naturals and also machine words.

## 3.5 Bedrock2 examples

Several examples have been written and verified by using the Bedrock2 machinery explained above. One example is a swap function that simply swaps values at two addresses and another one is a binary search algorithm. The most complex example is the lightbulb demo [4] : it consists of an Ethernet-connected IoT lightbulb controller that includes a proper specification, source code for its functionality, and a full proof of its correctness. This one makes use of the compiler as it generates RISC-V code that is run on a FPGA.

# Chapter 4

# The Roughtime implementation

In this chapter we describe the source code used in the Roughtime server implementation and the modifications we have done to it to make verification feasible.

## 4.1 Roughtime

As mentioned in the introduction, Roughtime is a protocol that consists of a client and server interaction to ensure that both requests and responses are made securely. The client's request does not include any other information than the client's nonce, and the server's response sends the current timestamp back to the client together with security bytes. The timestamp consists of a time interval with a midpoint and a radius. A sample of its output is as follows:

```
Sending request to 173.194.206.158, port 2002.
Received reply in 25602µs.
Current time is 1627981226278941µs from the epoch, ±1000000µs
System clock differs from that estimate by 10569µs.
```

## 4.2 The Roughtime protocol

The Roughtime protocol describes how messages should be sent back and forth between the server and the client. Here we are only interested in messages from the

server to the client. The contents of a message are defined by a list of tags and data corresponding to each tag where each piece of data is either a byte string or another nested message. For example, the protocol for the message sent from the server to the client uses the following tree structure.

- SREP
  - ROOT
  - MIDP
  - RADI
- SIG\x00
- INDX
- PATH
- CERT
  - SIG\x00
  - DELE
    - MINT
    - MAXT
    - PUBK

This means that the message has five tags and for instance the data corresponding to the SREP tag is a nested message. Each tag here has a special meaning, and most of them are used for security purposes; for example, the SIG\x00 tags correspond to signature bytes. The only places where the timestamp data lives are in MIDP and RADI which stand for the midpoint and radius of the current timestamp.

The Roughtime protocol also requires the data to be read as 4-byte words, and therefore the number of bytes in total must be a multiple of four. The values must also be encoded in little-endian form.

## 4.3 Implementation

A good portion of the implementation of the Roughtime protocol for both the server and client sides is devoted to cryptographic security checks, as its main goal is to be secure. However, our focus will diverge from that; meaning that we will implement the part where data is written into a buffer containing the server response and will keep the existing security checks.

# Chapter 5

# A first attempt towards verification

In this chapter we will show our first attempt at verifying Roughtime, our initial implementation of the Roughtime server, its verification and the challenges that arose.

## 5.1 Implementation

The first implementation of this program was done by purely hard-coding our main function which is called `CreateTimestampMessage`. This function takes an argument, which is the address pointer to which bytes will be written. Then, it creates a hard-coded Roughtime response message by adding each 4-byte word corresponding to the headers and tags. As for the data written in the buffer, we simply store garbage words using a loop that stores the hexadecimal `0x42` for each 4-byte word. It looks something like the following:

```
Definition createTimestampMessage :=
  let buffer := "buffer" in
  let i := "i" in
  ("createTimestampMessage", ([buffer], []:list String.string, bedrock_func_body:(
    store4(buffer, coq:(0x"5"));
    store4(buffer + coq:(4), coq:(0x"40"));
    store4(buffer + coq:(8), coq:(0x"40"));
```

```
store4(buffer + coq:(12), coq:(0x"a4"));
store4(buffer + coq:(16), coq:(0x"13c"));

store4(buffer + coq:(20), coq:(Z_of_string "SIG"));
store4(buffer + coq:(24), coq:(Z_of_string "PATH"));
store4(buffer + coq:(28), coq:(Z_of_string "SREP"));
store4(buffer + coq:(32), coq:(Z_of_string "CERT"));
store4(buffer + coq:(36), coq:(Z_of_string "INDX"));

i = (coq:(64)); while (i) { i = (i - coq:(4));
  store4(buffer + coq:(100) - i, coq:(0x"42"))
};
```

Each line of code basically stores a 4-byte word at a given address, and by doing this we fill our buffer with data. We only included part of the code because the rest is repetitive, as we decided to hard-code everything for this experiment. Note that this code when printed generates C code that functionally does something similar as the original Roughtime source code, although due to the random data dumped it will not behave the same as the original source code, and for instance will not pass the security checks; if we get rid of these security checks as well as some assertion checks in the Roughtime client code, then we would be able to replace part of the original Roughtime server implementation with our generated C code and be able to obtain a working client-server interaction.

## 5.2 Specification

Our first specification is a much weaker one than desired; it merely states that the source code can be run without any issue such as memory corruption. In a later chapter we will discuss the actual specification, as it's not straightforward to even define it.

In Bedrock2 the weak specification looks like the following:

```
Notation array32 := array scalar32 (word.of_Z 4).

Definition message_val : list semantics.word.

Instance spec_of_createTimestampMessage : spec_of "createTimestampMessage" := fun fun
    forall p_addr buf R m t,
      (array32 p_addr buf * R) m /\
      List.length buf = 356%nat /\
      WeakestPrecondition.call functions "createTimestampMessage" t m [p_addr]
      (fun t' m' rets => t = t' /\ rets = nil /\ (array32 p_addr message_val * R) m')
```

In the precondition we have a separation logic predicate that states that there is a pointer `p_addr` to a list of words `buf` and that such list has length 356, which is the same as having 356 words (each 4 bytes) of allocated memory at `p_addr`. In the postcondition, we only state that at the end we have an array pointer to a list with all the values that we hard-code beforehand. Finally, note that we actually need the memory layout predicate in our precondition to prove that at least the code runs.

## 5.3 Verification

In Bedrock2 notation one would prove the specification as follows:

```
Lemma createTimestampResponse_ok :
    program_logic_goal_for_function! createTimestampResponse.
  Proof.
      repeat straightline.
  Admitted.
```

In an ideal world we would just be able to use the `straightline` tactic to step through the entire program as it's supposed to be trivial. However this is not the case as we will see.

The first line of code of the program has the form `store4(buffer, coq:(0x"5"));` and in order to go through such a line the Bedrock2 tactics require a memory predicate that contains a separation logic conjunct of the form `scalar32 buf 5`, which makes sense since in order to store data into a location we need four bytes of memory allocated at such location. However, all we have is an `array` predicate that stores a list of values with length 356. An easy way to solve this issue is to simply destroy the list. Coq's `destruct` tactic to perform case analysis on the list discards the empty list case and ends up with the first element of the list and the remaining list of 355 elements. This seems to work as unfolding the array gives us something of the form `scalar32 buff h * array32 (buff+4) t` and now we can step through this line of code freely by destroying the list each time.

Now there is clearly some kind of problem: we would have to split a list every time we have a store command that operates on an array pointer (this would actually apply to the load command as well). In fact, this is what we did on our first experiment to see how far we could go. Since we have hard-coded our program we only need to destruct our list 356 times at most. This turns out to be very expensive both memory- and time-wise, especially because it becomes challenging for the linear solvers to deal with and/or simplify large arithmetic terms.

It turns out that after destructing the list 10 times, the linear arithmetic solver queries take more than a minute due to the memory predicates containing overall quadratic-sized arithmetic terms. Attempting to measure the execution time and memory usage was pointless as because of the time limitation it was not even possible to complete the proof.

One way to solve this issue would be to at each step split and then merge the buffer list to keep the proof state as simple as possible and not have to spend large amounts of memory and time. This has in fact been done for another proof in Bedrock2. However, this would again be farther from being automated as it required around 10

lines of Coq code to do so.

This suggests that we need a lemma to apply to each of these lines of code to reason about memory predicates involving array pointers, which we will see in the next chapter.

# Chapter 6

# Array pointer logic

In this chapter we discuss the implementation of lemmas that reason about array pointers and decisions made on what these lemmas are as well as tactics that use these; as we've discussed before these were very much needed to not have to manually split arrays before store or load commands.

## 6.1 The array store lemmas

We define lemmas for storing to array pointers at given addresses.

For the store lemmas we will need a way to update a Coq list as after a store command our list of values will be updated. Interestingly enough, such definition is not part of the Coq standard library so we make use of our own definition `upd` which stands for updating a list at a given index if possible, and in fact we define the more general `upds` which stands for updating a list with several values starting from a given index as long as indices allow us. The definitions are as follows:

```
Definition upds {E: Type}(l: list E)(i: nat)(xs: list E): list E :=
  (firstn i l) ++ (firstn (length l - i) xs) ++ (skipn (length xs + i) l).
Definition upd {E: Type}(l: list E)(i: nat)(x: E): list E := upds l i [x].
```

With that in mind, the store lemmas look like this

```
Lemma array_store_of_sep (addr addr' : word) n (oldvalues : list word)
```

```
    (value : word) size sz R m (post:_->Prop)
    (_ : addr' = word.add addr (word.of_Z ((word.unsigned size) * (Z.of_nat n))))
    (Hsep : sep (array (truncated_word sz) size addr oldvalues) R m)
    (_ : (n < length oldvalues)%nat)
    (Hpost : forall m, sep (array (truncated_word sz) size addr
        (upd oldvalues n value)) R m -> post m)
    : exists m1, Memory.store sz m addr' value = Some m1 /\ post m1.
```

This lemma is general enough for words of different sizes. It states that if we have an array pointer `addr` to a list of old values `oldvalues`, then going through a store command at address `addr'` ends up with a similar separation logic predicate where we update `oldvalues` with a new value `value` at some position `n` assuming that `n` is within bounds and that `addr'` is the address `addr` at offset `n`. This lemma turns out to be straightforward to apply, but it contains too many quantified variables to be computed, and for that we have another lemma.

```
Lemma array_store_of_sep' (addr addr' : word) (oldvalues : list word)
    (value : word) size sz R m (post:_->Prop)
    (Hsep : sep (array (truncated_word sz) size addr oldvalues) R m)
    (_ : 0 < word.unsigned size)
    (_ : let offset := word.unsigned (word.sub addr' addr) in
         (Z.modulo offset (word.unsigned size) = 0) /\
         (let n := Z.to_nat (offset / word.unsigned size) in
            (n < List.length oldvalues)%nat /\
    (forall m, sep (array (truncated_word sz) size addr
        (upd oldvalues n value)) R m -> post m)))
    : exists m1, Memory.store sz m addr' value = Some m1 /\ post m1.
```

In this lemma we compute the offset on the fly and need to use these let bindings for that purpose. We also make use of the previous proven lemma inside its proof, as these two lemmas basically state the same property.

## 6.2 Load lemmas

For the load lemmas, we only need to access an element at a given index of a given list which is defined in the Coq standard library as `nth`. With this in mind we also define our load lemmas which look like the following

```
Lemma array_load_of_sep' (addr addr': word) (values : list word) size sz R m
    (Hsep : sep (array (truncated_word sz) size addr values) R m)
    (_ : 0 < word.unsigned size)
    : let offset := word.unsigned (word.sub addr' addr) in
      (Z.modulo offset (word.unsigned size) = 0) ->
      (let n := Z.to_nat (offset / word.unsigned size) in (n < List.length values)%na
      Memory.load sz m addr' =
      Some (truncate_word sz (nth n values (word.of_Z 0)))).
```

## 6.3 Array tactics

We also define our custom tactic for automating the use of these array store and load lemmas. We only have locally defined a tactic called `array_straightline` that serves our purposes and allows us to step through load and store instructions. The way this tactic works is as follows: Let us consider the example:

$$\{\texttt{array}\, A\, [1;2;3] * \texttt{array}\, B\, [1] * R\}\ \texttt{store}(A+4, 3)\ \{\texttt{array}\, A\, [1;3;3] * \texttt{array}\, B\, [1] * R\}$$

In this case, the store instruction has the same meaning as $A[1] = 3$ assuming four-byte words. Then, the tactic would look at the precondition memory predicate and look for one that contains an array with address an expression that symbolically matches the address we want to store to, which is $A + 4$ (this can be easily handled using Ltac's `context` keyword). Then the tactic infers that the array that will be used for the lemma will be $A$ and not $B$. Based on that we now know both the store pointer as well as the array base pointer and can apply the array store lemma from

the previous section and verify this code.

In this example this tactic worked because it was the case that the array address expression existed inside the store address expression. One could say that in general array addresses must be of the form *basePointer* + *offset* or similar and therefore must contain the base pointer as a sub-expression. However, this is all up to the user adhering to conventions; for example, if a user splits an array then the base pointer gets changed, and our tactic would most likely fail. Therefore, if a user keeps array predicates intact, this tactic will work fine.

# Chapter 7

# An improved and verified implementation of Roughtime

In this final chapter we describe our last attempt at verifying Roughtime and how we were able to verify a good portion of it.

## 7.1 Implementation

The implementation used in the first experiment was merely hard-coded instructions that stored data into some array pointer. In this implementation we still hard-code some of the data but also copy data from one pointer to another. In fact, in the actual Roughtime implementation there are three places where data is filled and later copied into the array pointer for the response message bytes. We define three functions that construct a response message by hard-coding the bytes needed for the header and tags and copying the data from another pointer. This data can be either some signature bytes or the timestamp. One such function looks like the following:

```
Definition createSignedResponse :=
  let buffer := "buffer" in
  let time := "time" in
  let radius := "radius" in
```

```
  let root := "root" in
  ("createSignedResponse", ([buffer; radius; time; root], []: list String.string, bed
    store4(buffer + coq:(0), coq:(Ox"3"));
    store4(buffer + coq:(4), coq:(Ox"4"));
    store4(buffer + coq:(8), coq:(Ox"c"));

    store4(buffer + coq:(12), coq:(Z_of_string "RADI"));
    store4(buffer + coq:(16), coq:(Z_of_string "MIDP"));
    store4(buffer + coq:(20), coq:(Z_of_string "ROOT"));

    store4(buffer + coq:(24), radius);
    memcpy(buffer + coq:(28), time, coq:(2));
    memcpy(buffer + coq:(36), root, coq:(16))))).
```

And its pretty printed C code output looks like the following.

```
static void createSignedResponse(uintptr_t buffer, uintptr_t radius,
    uintptr_t time, uintptr_t root) {
  _br2_store((buffer)+((uintptr_t)0ULL), (uintptr_t)3ULL, 4);
  _br2_store((buffer)+((uintptr_t)4ULL), (uintptr_t)4ULL, 4);
  _br2_store((buffer)+((uintptr_t)8ULL), (uintptr_t)12ULL, 4);
  _br2_store((buffer)+((uintptr_t)12ULL), (uintptr_t)1229209938ULL, 4);
  _br2_store((buffer)+((uintptr_t)16ULL), (uintptr_t)1346652493ULL, 4);
  _br2_store((buffer)+((uintptr_t)20ULL), (uintptr_t)1414483794ULL, 4);
  _br2_store((buffer)+((uintptr_t)24ULL), radius, 4);
  memcpy((buffer)+((uintptr_t)28ULL), time, (uintptr_t)2ULL);
  memcpy((buffer)+((uintptr_t)36ULL), root, (uintptr_t)16ULL);
  return;
}
```

This function takes as arguments the array pointer, the timestamp, the radius,

and the root of the Merkle tree (security-related data) and constructs a small message following Roughtime's protocol. Note that the radius is a four-byte word, and therefore it fits within one word and we simply pass it into the store function. On the other hand, the timestamp takes 8 bytes and the root takes 64 bytes and therefore, we have to pass on pointers to such data as arguments and then use the `memcpy` function to copy data.

By replacing the three places where the Roughtime server fills out data by our own implementation, we obtain a working Roughtime server implementation; we manually verify this by compiling all the code together and checking that our code does functionally the same as the original Roughtime server code.

## 7.2 The memcpy function

It is known that most C compilers have a function called `memcpy` as part of their standard library. For this thesis we implement our own `memcpy` function based on the primitives we have. Its implementation looks like this.

```
Definition memcpy : func :=
  let dst := "dst" in
  let src := "src" in
  let num := "num" in
  let i := "i" in
  ("memcpy", ([dst; src; num], nil: list string, bedrock_func_body:(
    i = (coq:(4) * num); while (i) {
      store4(dst + coq:(4) * num - i, load4(src + coq:(4) * num - i));
      i = (i - coq:(4))
    })).
```

And its pretty printed C code output looks like the following:

```
void memcpy(uintptr_t dst, uintptr_t src, uintptr_t num) {
  uintptr_t i;
```

```
  i = ((uintptr_t)4ULL)*(num);
  while (i) {
    _br2_store(((dst)+(((uintptr_t)4ULL)*(num)))-(i),
        _br2_load(((src)+(((uintptr_t)4ULL)*(num)))-(i), 4), 4);
    i = (i)-((uintptr_t)4ULL);
  }
  return;
}
```

Note that this implementation was made only for copying four-byte words. Also, we define a specification for it that looks like the following

```
Global Instance spec_of_memcpy : spec_of "memcpy" :=
    fun functions => forall src_ptr dst_ptr src dst buf1 buf2 num R m t,
        let off1 := \_ (src ^- src_ptr) in
        let off2 := \_ (dst ^- dst_ptr) in
        let n1 := Z.to_nat (off1 / 4) in
        let n2 := Z.to_nat (off2 / 4) in
        (off1 mod 4 = 0 /\ off2 mod 4 = 0 /\
         off1 + 4 * (\_ num) < 2^width /\
         off2 + 4 * (\_ num) < 2^width /\
         Z.of_nat n1 + \_ num <= Z.of_nat (List.length buf1) /\
         Z.of_nat n2 + \_ num <= Z.of_nat (List.length buf2)) /\
        (array32 src_ptr buf1 * array32 dst_ptr buf2 * R) m ->
        WeakestPrecondition.call functions "memcpy" t m [dst; src; num]
        (fun t' m' rets => t = t' /\ rets = nil /\
        (array32 src_ptr buf1 * array32 dst_ptr
            (upds buf2 n2 (firstn (Z.to_nat (\_ num)) (skipn n1 buf1))) * R) m').
```

This specification quantifies over a source array base pointer and also a destination array base pointer; this is needed as per our convention from Chapter 6 where we

want to always keep array predicates on array base pointers. Similarly as in the array lemmas in Chapter 6, we need several let bindings to perform some pre-computation for the offsets. As for our precondition we require that offsets are divisible by 4 and within range; and for our memory predicate we need some array at the source and some array at the destination. As for our post-condition, we will obtain a different array at the destination where we update the old list at several places with the copied data.

With this in mind, we verify our memcpy implementation, and the process is quite straightforward. The only interesting part is coming up with a good loop invariant that merely states that after each iteration the source was partially copied onto the destination.

## 7.3 Specification

For this specification we will actually show that the message constructed has the form described in Section 2 as per the Roughtime protocol. Such a definition of the protocol was only given in words, and now we will actually use Coq to formally define what the protocol requires, and for that we define our own datatype.

### 7.3.1 The entry datatype

Each message in the Roughtime protocol is comprised of a list of tags and entries for each tag; each entry is itself either a nested message or simply raw data, which makes it a perfect candidate for Coq's inductive datatypes which look like the following.

```
Inductive entry :=
| rec : list (prod string entry) -> entry
| val : list (Semantics.word) -> entry.
```

This means that an entry is either a list of pairs of tags and entries or a list of words. Together with that we also define a few functions for entries that will be

needed such as `size`, which computes the total size in bytes of an entry. With that, we also define a function that serializes an entry and converts it to a list of words:

```
Fixpoint flatten (e : entry) : list (Semantics.word).
```

It turns out we also will need a lemma that states that the size of a flattened entry is the same as the one that we obtain by just computing it:

```
Lemma flatten_length (e : entry) : Z.of_nat (List.length (flatten e)) = size e.
```

### 7.3.2 A stronger specification

Given our entry datatype, we can now define a good specification for our message response. We also define a way to construct it by creating a function that takes the time data, root data and radius and constructs a list of correct bytes corresponding to the response, which we call `srep_entry`. Then our specification looks like the following.

```
Instance spec_of_createSignedResponse : spec_of "createSignedResponse" :=
     fun functions => forall buf_addr buf_data radius time_addr time_data
     root_addr root_data R m t,
         (array32 buf_addr buf_data * array32 root_addr root_data *
            array32 time_addr time_data * R) m ->
         List.length buf_data = Z.to_nat 25 ->
         List.length time_data = Z.to_nat 2 ->
         List.length root_data = Z.to_nat 16 ->
         WeakestPrecondition.call functions "createSignedResponse" t m
         [buf_addr; radius; time_addr; root_addr]
           (fun t' m' rets => t = t' /\ rets = nil /\
              (array32 buf_addr (flatten (srep_entry radius (val time_data)
                (val root_data))) * array32 root_addr root_data *
                    array32 time_addr time_data * R) m').
```

Here in the pre-condition we state all the array pointers needed as well as lengths needed for bounds, and in the post-condition we state that our buffer pointer array points to a list of bytes made by constructing a correct signed response.

## 7.4 Verification

As for the verification, we use our tactics for reasoning about array pointers and are able to step through the code very easily; the proof for the signed response bytes looks like the following

```
Lemma createSignedResponse_ok :
    program_logic_goal_for_function! createSignedResponse.
  Proof.
    repeat straightline.
    subst_words.
    repeat array_straightline.
    memcpy_call.
    memcpy_call.
    split; try split; auto.
    large_sep_assumption.
  Qed.
```

The tactic `memcpy_call` automates the memcpy function calls, and the last tactic `large_sep_assumption` is very specialized for our case to prove that one separation logic predicate implies another. The entire proof took just a few seconds to process, which is much better than the incomplete proof from Chapter 5 that already took minutes. The fact that we proved array lemmas and load lemmas saved us a lot of time when stepping through store and load commands. Overall the execution of the Coq file for the Roughtime proofs took around 34.25 seconds and used around 655 megabytes of memory on a 2015 Macbook Pro. This is relatively reasonable compared to the other examples that use Bedrock2.

## 7.5 Conclusion

Overall we have verified three functions that fill hard-coded data and also copy data into a buffer. The original Roughtime server implementation is much more general but it would require a lot of complexity the verification process. For instance, it makes use of C structs which are still not usable in Bedrock2 as of now. A possible further work could be to write an implementation closer to the original one once C structs are fully developed in Bedrock 2.

# Bibliography


[1] Roughtime, `https://roughtime.googlesource.com/roughtime/`.

[2] Andrew W. Appel. Verified Software Toolchain. Berlin, Heidelberg, 2012. Springer-Verlag.

[3] Bedrock2, `https://github.com/mit-plv/bedrock2`.

[4] Andres Erbsen, Samuel Gruetter, Joonwon Choi, Clark Wood, and Adam Chlipala. Integration Verification across Software and Hardware for a Simple Embedded System. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 604–619, New York, NY, USA, 2021. Association for Computing Machinery.

[5] Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao, and David Costanzo. CertiKOS: A Certified Kernel for Secure Cloud Computing. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, New York, NY, USA, 2011. Association for Computing Machinery.

[6] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jay Lorch, Bryan Parno, Justine Stephenson, Srinath Setty, and Brian Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. ACM - Association for Computing Machinery, October 2015. https://www.microsoft.com/en-us/research/publication/ironfleet-proving-practical-distributed-systems-correct/.

[7] Bart Jacobs and Frank Piessens. The VeriFast Program Verifier. Technical Report CW-520, Katholieke Universiteit Leuven, 2008. `https://people.cs.kuleuven.be/~bart.jacobs/verifast/verifast.pdf`.

[8] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. SeL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[9] Xavier Leroy. Formal Verification of a Realistic Compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[10] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 252–269, New York, NY, USA, 2017. Association for Computing Machinery.

[11] Atalay İleri, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Proving Confidentiality in a File System Using DISKSEC. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 323–338, USA, 2018. USENIX Association.