Formally Verifying the Correctness and Safety of OpenTitan Boot Code using CBMC

Jacob Gosch Søndergaard,
Kristoffer Skagbæk Jensen

Spring 2021

AALBORG UNIVERSITY

STUDENT REPORT

Department of Computer Science
Selma Lagerlöfs Vej 300
9220 Aalborg East, DK
Telefon +45 9940 9940
Telefax +45 9940 9798
https://www.cs.aau.dk/

Title:
Formally Verifying the Correctness and Safety of OpenTitan Boot Code using CBMC

Theme:
Formal Verification

Project Period:
Software 10th Semester
2021/02/01 - 2021/06/18

Project Group:
SV107f21

Participants:
Jacob Gosch Søndergaard
Kristoffer Skagbæk Jensen

Supervisors:
Danny Bøgsted Poulsen
Kim Guldstrand Larsen
René Rydhof Hansen

Number of pages: 117

Date of Completion:
June 18th, 2021

Abstract:

The correctness and safety of heavily relied upon software is crucial. OpenTitan is an open-source silicon root-of-trust project. Based on an extensive analysis of the OpenTitan project, we, together with SV106f21, developed C code that corresponds to the initial boot stage of OpenTitan. CBMC is a bounded model checker for ANSI-C verification. We verify numerous safety properties for the developed boot code using CBMC. In addition, we propose an overview of the CBMC architecture and theory as well as a structured approach to verify C code using CBMC. We use CBMC nondeterminism to create a C model of the boot code's hardware environment. In total, we verified that the developed boot code adheres to all 11 security properties, with most of them being derived from security goals from our previous work [1]. We also further investigate the safety of the boot code by modeling various hardware attacks and verify for their implication. We discover that the current implementation is vulnerable to attacks on flash, ROM, and the OpenTitan Big Number Accelerator, with the consequence of either executing malicious code or crashing.

Summary

OpenTitan is an open-source project for designing root-of-trust chips. OpenTitan chips perform safety-critical operations for a host and are a single point of failure for the host's security. Verifying the correctness and safety of OpenTitan chips is therefore of utmost importance. Due to the complexity and comprehensiveness of the OpenTitan chip, we narrowed down the considered system to only include the `mask_ROM` boot code, which is the initial boot stage of the OpenTitan chip. The main goal of the `mask_ROM` boot stage is to load in a `ROM_EXT` from flash, validate it, and then transfer execution to it.

Based on an extensive analysis of the OpenTitan project, we, together with SV106f21, developed C code that correspond to the `mask_ROM` boot stage. In our previous work, we established 13 security goals for the OpenTitan boot stages [1]. In this work, we derive 11 security properties from four of the security goals. Our goal is to prove the adherence of these security properties and investigate possible attacks and their implication on the security properties.

We decided to use the CBMC (C Bounded Model Checker) tool to perform formal verification of the developed boot code. CBMC is based on the formal verification technique called bounded model checking and provides guarantees for the exploration of the entire program's state space. As part of our work, we have created a theoretical and architectural overview of the CBMC tool. In addition, we have created a CBMC tutorial that describes how to annotate and verify C code using the CBMC constructs. CBMC allows expressing nondeterministic behavior, which we use to create a C model of the initial boot stage's hardware environment. Based on this, we used CBMC to successfully and formally verify that the developed C code of the `mask_ROM` boot stage satisfies the 11 security properties under reasonable assumptions. We conclude that the developed C code adheres fully to one of the four selected security goals and partially to the three others.

We also used CBMC to model potential attacks to the developed boot code to formally verify if it still adheres to the 11 security properties while being attacked. We model a total of six attacks which all can be considered hardware/physical attacks. We discover that the boot code is vulnerable to attacks on the flash, ROM, and OTBN. The consequences of these attacks are either execution of malicious code or crashing. Thus we find that the security of the developed boot code for the OpenTitan `mask_ROM` boot stage could be improved by e.g. verifying the integrity of the boot policy.

Contents

List of Figures

Listings

Introduction

The correctness and safety of the widely used and heavily relied upon software in our society depend on the correctness and safety of the underlying software and hardware responsible for booting it. Therefore, it is crucial to establish trust in the correctness and safety of the underlying booting process, both in terms of software and hardware.

We will, in our work, verify the correctness and safety of low-level C code using CBMC [2]. CBMC is a bounded model checker for static analysis of ANSI-C programs. CBMC can verify, among others, properties regarding pointer safety, array bounds, division by zero, and user-defined properties expressed as assertions. As a verification case, we will verify low-level C code from OpenTitan. Although, our findings should apply to all low-level C code.

OpenTitan is the first open-source project for designing a silicon root-of-trust (RoT) chip [3]. The purpose is to make the design and implementation of RoT chips more transparent, secure, and trustworthy [4]. The OpenTitan chip is essentially a microcontroller that performs safety-critical operations for a host device [5]. It has its own boot sequence, kernel, hardware security modules, and application layer [6] [7]. The OpenTitan chip is trusted by design and acts as a single point of failure for the host's security.

The verification case we will look into, specifically, is the code that constitutes the OpenTitan `mask_ROM` stage, which is the initial boot stage responsible for verifying and transferring execution to the subsequent boot stage `ROM_EXT`. This code is challenging to verify due to its complexity and dependency on external functionality. A simplified snippet of the `mask_ROM` boot code that we want to verify can be seen below in Listing 1. Due to OpenTitan being an ongoing **unfinished** project, the code is written by us and SV106f21, based on the OpenTitan documentation, pseudocode, and unfinished implementation (cf. Appendix A for the complete developed boot code). The group SV106f21 is doing similar work with the verification tool UPPAAL instead of CBMC [8].

```
typedef void(rom_ext_entry)(void);

extern int OTBN_RSA_VERIFY(pub_key_t pub_key, int* message, signature_t signature);

int verify_rom_ext_signature(pub_key_t rom_ext_pub_key, rom_ext_manifest_t manifest){
    return OTBN_RSA_VERIFY(rom_ext_pub_key, manifest.image_code, manifest.signature);
}

int final_jump_to_rom_ext(rom_ext_mf_t cur_rom_ext_mf){
    rom_ext_entry *rom_ext_entry = (rom_ext_entry*) cur_rom_ext_mf.entry_point;
    rom_ext_entry();
    return 0;
}
```

Listing 1: Simplified snippet of the low level C code that we will verify.

This leads to the problem statement:

Is the `mask_ROM` code correct and safe?

Chapter 1

CBMC Theory

CBMC is based on the formal verification technique called bounded model checking and uses it to formally verify properties of ANSI-C programs [2] [9]. Bounded model checking is, briefly explained, a verification technique that reduces the state space by only analyzing the subset of states reachable by a bounded number of transitions [10]. The primary technique for doing bounded model checking in CBMC is loop bounds. Loop bounds specify the max number of performed loop unrolls. CBMC can assert properties regarding pointer safety, array bounds, memory leakage, division by zero, overflow, underflow, and properties expressed as assertions by the user [2] [9]. In addition, CBMC provides guarantees that the entire state space of the ANSI-C program is explored. We define the definition of soundness and completeness for CBMC as the following [11]:

Soundness: That a tool such as CBMC is sound means that whatever it can prove for an ANSI-C program is true.

Completeness: That a tool such as CBMC is complete means that whatever is true for an ANSI-C program, it can prove.

1.1 Overall Architecture

This section is based on [12]. The general approach in CBMC is "to reduce the Model Checking Problem of C programs to determining the validity of a bit-vector equation" [9]. That is, to determine whether a model of a C program satisfies certain properties by determining the unsatisfiability of a bit-vector equation. A bit-vector equation is an equation where each variable is a bit-vector. In this case, a bit-vector represents a variable in the C program.

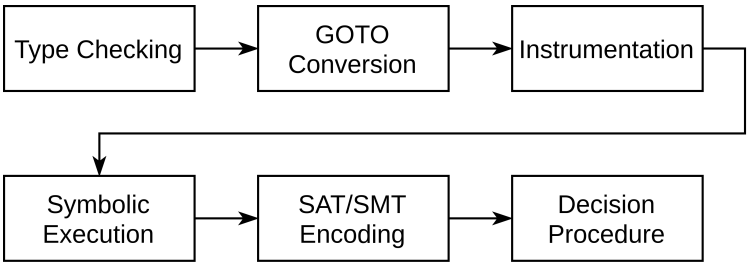

Figure 1.1.1: Overall architecture of CBMC.

The overall architecture of CBMC is illustrated in Fig. 1.1.1. We divide CBMC verification of a C program into seven steps:

1. **Preliminary**: First, the CBMC command is parsed and interpreted. Then the given program is parsed into a parse tree while creating an initial symbol table containing the identifiers.
2. **Type Checking**: Type checks the given program against type rules while decorating a symbol table with the derived type information.

3. **GOTO Conversion**: Translates the program into an equivalent GOTO program represented as an abstract syntax tree (AST) where conditional and iterative constructs are replaced by semantically equivalent GOTO statements [13].
4. **Instrumentation**: Additional transformations of the GOTO program. CBMC inserts assertions and replaces return statements with variable assignments and GOTO statements. Also, function pointer invocation is substituted by a call to the function pointed to.
5. **Symbolic Execution**: The GOTO program is unwounded and transformed into SSA form. The SSA program is then used to create two bit-vector equations denoted C and P. C expresses the constraints on the variables in the program, and P expresses the properties that should be asserted.
6. **SAT/SMT Encoding**: The equation $C\ AND\ !P$ is converted to conjunctive normal form (CNF). CBMC uses SAT by default but can also support SMT. Given a SAT/SMT formula on CNF form, a solver proves whether the properties expressed by P are satisfied under the constraints expressed by C.
7. **Decision Procedure**: Solve $C\ AND\ !P$ using a SAT/SMT third-party solver that requires input on CNF form. The default solver is MiniSAT. If a solution exists, then the assertions do not hold for all paths. If no solution exists, then all assertions hold under the specified unwinding bounds.

In the following sections, we will detail steps 3-7 of the CBMC verification.

1.2 GOTO Conversion and Instrumentation

This section is written based on [12] and [14]. In the GOTO Conversion step, a C program is taken as input and transformed into a GOTO program represented as an AST. A GOTO program is a program that only contains function definitions (can be recursive), function calls, return statements, assertions, variable declarations, assignments, labels, and guarded GOTO statements (i.e. GOTO statements which execution depends on a Boolean expression)[1]. This means that constructs such as if, switch, loops (for, while, do-while), and what they call "jumps"[2] are replaced by semantically equivalent guarded GOTO statements. In addition, a new main function is generated that; initializes CBMC and global variables, calls the original main function, and contains automatically generated postconditions (e.g. assertions for memory leaks).

After the GOTO Conversion step, additional transformations are done as part of the Instrumentation step. In this step, return statements are replaced with assignments to variables that the caller then accesses. A return statement in the middle of the function also requires a GOTO to the end of the function. The possible target of a function pointer is deduced and substituted at the place of invocation. In practice, CBMC creates if-else conditional statements that compare the function pointer to the address of the matching function definitions. Each conditional case contains a GOTO to the matching function. If the function pointer does not match a function definition, then an `ASSERT FALSE` statement is reached. That is to say that no function matches what the function pointer points to. Lastly, if specified by the user, assertions related to e.g. invalid pointers, array-out-of-bounds-access, and memory leaks are inserted (cf. Section 2.3.1 for the complete list of non-user-defined assertions). Thus, the final GOTO program, which is the output of this step, contains only function definitions (can be recursive), function calls, variable declarations, variable assignments, assertions, labels, and guarded GOTO statements that can be both forward and backward.

Example
The `--show-goto-functions` option shows the result of the GOTO Conversion and Instrumentation steps. Listing 1.1 and Listing 1.2 show an example of a transformation of a C program to a GOTO program. On the left are the C program and the associated GOTO program on the right.

[1] The GOTO program can contain other statements as well, such as `dead` and `END_FUNCTION`.
[2] Unclear what this is, as it is not function calls.

```
1 int main()
2 {
3     int x = 0;
4     if (x == 0)
5         x = 20;
6     else
7         x = 10;
8     return x;
9 }
```

Listing 1.1: A C program.

```
1  main /* main */
2          signed int x;
3          x = 0;
4          IF !(x == 0) THEN GOTO 1
5          x = 20;
6          GOTO 2
7      1: x = 10;
8      2: main#return_value = x;
9          dead x;
10         END_FUNCTION
```

Listing 1.2: A GOTO program.

1.3 Symbolic Execution

This section is written based on [15]. In the Symbolic Execution step, a GOTO program is taken as input together with a symbol table and converted to static single assignment (SSA) form. The output of the step is a set of equations in terms of constraints (denoted C) and properties (denoted P) that represents all statements and assertions in the program, respectively.

The process of performing symbolic execution consists of traversing the GOTO program, statement by statement. For each statement that is reached, initial transformations are performed and pointer deferences are removed. The transformed statement is then further transformed to SSA form. Lastly, C and P are calculated for the SSA statement. When all statements have been traversed, the resulting C and P equations are outputted.

1.3.1 Program Transformation

This section is written based on [2] [16]. The goal of the program transformation is to convert the program to SSA form as this allow for deduction of C and P.

Initial Program Transformation

The current program, namely the GOTO program, contains function definitions (can be recursive), function calls, variable declarations, variable assignments, assertions, labels, and guarded forward and backward GOTO statements, which model if statements, switch statements, and loops. The following transformations are performed for each statement in the GOTO program, if relevant:

- Loops expressed using GOTO statements are unrolled by duplicating the body n times (the bounded aspect of CBMC), where each body duplication is guarded with an if statement containing the loop condition. After unrolling n times, an assertion is inserted to assert whether enough unrolling was done. This assertion is denoted the unwinding assertion and it asserts whether the loop condition is false. Unwinding assertions are needed to prove that the analysis covers the entire state space of the program.
- Function definitions and calls are removed by replacing all function calls with the body of the function. Formal and actual parameters are handled by simply declaring the formal parameters as normal variables and assigning them the values of the actual parameters. This is known as function expansion or function inlining. An example is seen in Listing 1.3 and Listing 1.4.

```
1  int z = 0;
2
3  int foo(int x)
4  {
5      int y = 10;
6      x = y * 2;
7      return x;
8  }
9
10 int main()
11 {
12     int a = 10;
13     int y = foo(a);
14     return y + z;
15 }
16
```

Listing 1.3: Example of a C program before function inlining.

```
1  signed int foo(signed int x);
2
3  signed int z=0;
4
5  void main(void)
6  {
7    z = 0;
8    signed int a=10;
9    signed int y;
10   signed int return_value_foo;
11   signed int x=a;
12   signed int y=10;
13   x = y * 2;
14   return_value_foo = x;
15   y = return_value_foo;
16   return' = y + z;
17 }
18
```

Listing 1.4: Resulting program with functions inlined.

- Recursive function calls are unrolled, similar to loops, by duplicating the body n times and inserting an unwinding assertion asserting that no further unrolling of the recursive function is needed.

Transformation to SSA

When a program is on SSA form, each variable is assigned to only once, and all variables are defined before use. The transformation into SSA is done by creating a new version of a variable each time the variable is assigned to. SSA form allows to keep a history of the values of variables and access them by referencing the different versions of the variables. SSA form also eliminates the need for data flow analysis as the information flows directly from the SSA statements.

After the transformations above, the program consists of only variable declarations, variable assignments, forward GOTO statements, labels, and assertions. At this point, all pointer dereferences must be removed before proceeding with the transformation (cf. Section 1.3.3). After that, each statement is transformed as below:

- Variables are renamed. This step brings the statement on SSA form. It is done by introducing a new version of a variable each time it is assigned to.
- Side effects are removed from the statement. Side effects occur when using the assignment operators (+=, -=, etc.) and the pre- and post-increment/decrement operators. The side effects are removed by introducing temporary variables to "store" the side effect.
- Forward GOTO statements are expressed as equivalent if statements. This transformation step happens after the variable renaming step.

After all the statements are transformed, the final program is on SSA form and consists of only if statements, variable declarations, variable assignments, and assertions. Based on this program, C and P are computed.

1.3.2 Computing C and P

This section is written based on [16]. Given a program on SSA form, $C(p, g)$ and $P(p, g)$ are computed, where p is a statement and g is the conditions that need to be true for p to be executed/reachable. C and P are defined by the following rules based on the content of p (note that $r : S \longrightarrow S$ defines the rename function which renames a statement S):

1. **Skip**: If the statement p is empty or a skip statement there are no constraint or property, it always evaluates to true.

$$C(\text{"}skip\text{"}, g) := true \tag{1.1}$$

$$P(\text{"}skip\text{"}, g) := true \tag{1.2}$$

2. **Conditional**: If the statement p is a conditional of the form $if\ (c)\ I\ else\ I'$:

$$C(\text{"}if\ (c)\ I\ else\ I'\text{"}, g) := C(I, g \wedge r(c)) \wedge C(I', g \wedge \neg r(c)) \tag{1.3}$$

$$P(\text{"}if\ (c)\ I\ else\ I'\text{"}, g) := P(I, g \wedge r(c)) \wedge P(I', g \wedge \neg r(c)) \tag{1.4}$$

3. **Sequential Composition**: If the statement p is a sequential composition of two code blocks I and I':

$$C(\text{"}I; I'\text{"}, g) := C(I, g) \wedge C(I', g) \tag{1.5}$$

$$P(\text{"}I; I'\text{"}; g) := P(I, g) \wedge P(I', g) \tag{1.6}$$

4. **Assertion**: If the statement p is an assertion:

$$P(\text{"}assert(a)\text{"}, g) := g \implies r(a) \tag{1.7}$$

5. **Assignment**: If p is an assignment and the LHS is not an array or a struct, but simply a variable, C is calculated as:

$$C(\text{"}v = e\text{"}, g) := (v_a = (g\ ?\ r(e) : v_{a-1}) \tag{1.8}$$

   It says that v becomes equal to the renamed RHS if the guard holds. Otherwise, it becomes equal to the previous version of v.

6. **Array**: Arrays are modeled as functions using lambda notation. The function takes a single parameter and returns the value at the index equal to the parameter. If p is equal to $v[a] = e$, where v is the array and a is the index, C is computed in the following way:

$$C(\text{"}v[a] = e\text{"}, g) := v_a = \lambda i : ((g \wedge i = r(a))\ ?\ (r(e)) : (v_{a-1}[i])) \tag{1.9}$$

   The constraint says that the array v is now equal to the function that, given an index i, will return the value of e if the guard for executing the assignment is true and i matches the index assigned to, namely a. Otherwise, the function for the array should return what the previous array function did.

   They do not explain how assignments to structs are handled but hint that it is done similarly.

Example

To give an example of how C and P are computed, consider the following program:

```
1 int x = 10;
2 if(x < 10){
3     x = 2;
4 }
5 else{
6     x = 3;
7 }
8 assert(x != 2);
```

For one statement at a time, the transformations in Section 1.3.1 are performed on the statement, bringing the program on SSA form. After, the rules for C and P are applied to the statements and the assertion, respectively. This results in the following C and P:

$C :=$	$x_1 = 10 \wedge$	Line 1 and Eq. (1.8)
	$x_2 = (x_1 < 10)\ ?\ 2 : x_1 \wedge$	Line 3 and Eq. (1.3) followed by Eq. (1.8)
	$x_3 = (x_1 \geq 10)\ ?\ 3 : x_2$	Line 6 and Eq. (1.3) followed by Eq. (1.8)
$P :=$	$x_3 \neq 2$	Line 8 and Eq. (1.7)

1.3.3 Removal of Pointer Deferences

This section is written based on [16]. The removal of pointer dereferences is performed after the program has been unwounded but before variables are renamed during translation to SSA.

SSA form and SAT problems do not support pointers. Pointers point to the address of other objects and exist as an "intermediary link" to something concrete. In the case of CBMC, it is desired to be as concrete as possible. Thus pointers are replaced with the objects/values pointed to. This process consists of replacing pointer derferences with equivalent expressions. E.g. the following program: `int i; int* pi = &i; *pi = 2;` is replaced with `int i; i = 2;`. The process is done by first replacing all instances of `&*p` with `p`, as they are semantically equivalent. Next, the removal of dereferencing of pointers is done by removing them from sub-expressions before removing them from the parent expressions. There are two ways of dereferencing in C, namely `*p` and `p[i]`.

The removal of pointer dereferences is done using the function $\phi(e, g, o)$, replacing pointer dereferences with the expression pointed to. e is the expression being dereferenced, g is the guard for executing e, and o is the pointer's offset. The following cases define ϕ. There is a case for ϕ for each type of expression found in the program. In this way, ϕ can be seen as parsing/traversing the entire program and removing all pointer dereferences.

$$*e \longrightarrow \phi(e, g, 0) \tag{1.10}$$

$$e[o] \longrightarrow \phi(e, g, o) \tag{1.11}$$

1. If expression e is equal to a pointer p, and the C generated so far contains a constraint saying that $r(p) = e'$. Then derefencing p (i.e. applying ϕ to p) is the same as applying ϕ to e':

$$\phi(p, g, o) := \phi(e', g, o) \tag{1.12}$$

2. If expression e is equal to a symbol of array type, a, then e is treated as the address of the first element of a:

$$\phi(e, g, o) = \phi(\&a[0], g, o) \tag{1.13}$$

3. If expression e is equal to the address of a symbol, i.e. $e = \&s$. Then dereferencing $\&s$ with offset 0 is equal to accessing the symbol s:

$$\phi(\&s, g, 0) := s \tag{1.14}$$

4. If expression e is equal to the address of an array index, i.e. $e = \&a[i]$, with a given offset. Then deferencing e is equal to accessing the array at the total offset:

$$\phi(\&a[i], g, o) := a[i + o] \tag{1.15}$$

This rule does not detail how to replace a pointer dereference. There is no rule for replacing an array indexing. However, this is not an issue, as the array indexing can be replaced with lambda expressions in C and P (cf. Eq. (1.9) in Section 1.3.2).

5. If expression e is equal to a conditional expression, ϕ is applied to each branch:

$$\phi(c\ ?\ e' : e'', g, o) := c\ ?\ \phi(e', g \wedge c, o) : \phi(e'', g \wedge \neg c, o) \tag{1.16}$$

6. If expression e is equal to the product of a pointer, e', and an integer, i, dereferencing e is calculated as applying ϕ to e' and adding i to the total offset:

$$\phi(e' + i, g, o) := \phi(e', g, o + i) \tag{1.17}$$

7. If expression e is equal to a pointer typecast, i.e. $e = (Q*)e'$, then dereferencing e is done by calculating ϕ of e':

$$\phi((Q*)e', g, o) := \phi(e', g, o) \tag{1.18}$$

8. If e does not match the rules above, the ANSI-C standard does not specify the semantic behavior. Thus the behavior of dereferencing e is undefined. CBMC inserts an assertion to assert that e is never dereferenced. This is to prevent undefined behavior.

$$\phi(e, g, o) := \bot \tag{1.19}$$

Example

To illustrate the application of ϕ, consider the following program, where the pointer dereference on line 6 (i.e. `*p`) and array index of pointer on line 9 (i.e. `p[1]`) are removed during symbolic execution.

```
1  int x = 10;
2  int a[n], *p;
3  p = &a[5];
4
5  if(x < 10){
6      x = *p;
7  }
8  else{
9      x = p[1];
10 }
11 assert(x != 10);
```

When reaching the pointer dereference (line 6), the dereference `*p` must be transformed into something equivalent. The C constraints build until this point are as follow:

$$\begin{aligned} \mathrm{C} := \quad & x_1 = 10 \wedge \\ & a_0 = \lambda i : 0 \wedge \\ & p_1 = \&a_0[5] \end{aligned}$$

Dereference `*p` is removed in the following way:

$*p$	$= \phi(p, x_1 < 10, 0)$	Eq. (1.10)
	$= \phi(\&a_0[5], x_1 < 10, 0)$	Eq. (1.12) and $r(p) = p_1$
	$= a_0[5 + 0]$	Eq. (1.15)

After this, the constraint for x_2 can be added to C. Note that $a_0[n]$ is a call to the lambda function a_0 as described in the rule to transform an array (cf. Eq. (1.9)):

$$\begin{aligned} \mathrm{C} := \quad & x_1 = 10 \wedge \\ & a_0 = \lambda i : 0 \wedge \\ & p_1 = \&a_0[5] \wedge \\ & x_2 = x_1 < 10 \ ? \ a_0[5 + 0] : x_1 \end{aligned}$$

Similarly, the array index `p[1]` at line 9 is translated in the following way:

$p[1]$	$= \phi(p, x_1 \geq 10, 1)$	Eq. (1.11)
	$= \phi(\&a_0[5], x_1 \geq 10, 1)$	Eq. (1.12) and $r(p) = p_1$
	$= a_0[5 + 1]$	Eq. (1.15)

After this, the constraint for x_3 can be added to C:

$$\begin{aligned} \mathrm{C} := \quad & x_1 = 10 \wedge \\ & a_0 = \lambda i : 0 \wedge \\ & p_1 = \&a_0[5] \wedge \\ & x_2 = x_1 < 10 \ ? \ a_0[5 + 0] : x_1 \wedge \\ & x_3 = x_1 \geq 10 \ ? \ a_0[5 + 1] : x_2 \end{aligned}$$

The constraint for p_1 is now unnecessary. The final C and P are thus:

$$\begin{aligned} \mathrm{C} := \quad & x_1 = 10 \wedge \\ & a_0 = \lambda i : 0 \wedge \\ & x_2 = x_1 < 10 \ ? \ a_0[5 + 0] : x_1 \wedge \\ & x_3 = x_1 \geq 10 \ ? \ a_0[5 + 1] : x_2 \\ \mathrm{P} := \quad & x_3 \neq 10 \end{aligned}$$

1.3.4 __CPROVER_assume

`__CPROVER_assume` is a CBMC construct that is used to make assumptions about data. CBMC uses it during verification to reason about possible values and thus possible program paths. When performing symbolic execution, a statement in the GOTO program is only symbolically executed if the current state of the program is reachable (cf. the switch at line 623 in [17]). In addition, a statement is symbolically executed under the assumptions gathered during the symbolical execution of prior statements. We have found no documentation for how CBMC handles assumptions other than they affect future assertions. However, the option `--program-only` prints the statements that were symbolically executed, on SSA form [15]. Thus we use the `--program-only`

option to gain an insight into how the `__CPROVER_assume` statements are handled during symbolic execution.

If a `__CPROVER_assume` is symbolically executed and evaluates to false, the current state is set to be unreachable (cf. line 227 in [17]) and all later statements are therefore not symbolically executed. This is illustrated with the example in Listing 1.5. All statements after the non-true assumption, on line 4, are not included in the output of `--program-only` seen in Listing 1.6.

```
1 int main()
2 {
3     int x = 0;
4     __CPROVER_assume(x == 1);
5     if (x == 0)
6         x = 20;
7     else
8         x = 10;
9     return x;
10 }
```

Listing 1.5: Example of program with non-true assumption.

```
1 (32) x!0@1#2 == 0
2 (33) ASSUME(FALSE)
```

Listing 1.6: Output of --program-only. The program is on SSA form.

An example of where a statement is symbolically executed under a certain assumption is seen in Listing 1.7 and Listing 1.8. In the example, `x` is nondeterministically initialized but is assumed to be 1 and thus, when executing `y = x + 1`, `y` becomes 2. In this case, where the assumption restricts `x` to a single value, the assumption provides the same functionality as simply assigning `x` to 1.

```
1 int main()
2 {
3     int x;
4     __CPROVER_assume(x == 1);
5
6     int y = x + 1;
7
8     return y;
9 }
```

Listing 1.7: Example of program with a true assumption.

```
1 (32) ASSUME(x!0@1#1 == 1)
2 (33) y!0@1#2 == 2
3 (34) main#return_value!0#1 == 2
4 (35) return'!0#1 == 2
```

Listing 1.8: Output of `--program-only`. The program is on SSA form.

Listing 1.9 shows the case where an assumption specifies a range for the variable `x` which is used in a loop condition. In this case, CBMC cannot automatically determine the unwinding bound and it must be supplied in the CBMC command. Contrarily, if the assumption specified `x` to a specific value, as seen in Listing 1.10, CBMC can automatically determine the unwinding bound.

```
1 int main()
2 {
3     int x;
4     __CPROVER_assume(x > 0);
5
6     while (x < 2)
7     {
8         x = x + 1;
9     }
10
11    return x;
12 }
```

Listing 1.9: Program with a while-loop that depends on an assume.

```
1 int main()
2 {
3     int x;
4     __CPROVER_assume(x == 1);
5
6     while (x < 2)
7     {
8         x = x + 1;
9     }
10
11    return x;
12 }
```

Listing 1.10: Program with a while-loop that depends on an assume.

1.4 SAT/SMT Encoding

The Boolean satisfiability problem (SAT) is the problem of determining if a model of value assignments exists that satisfies a given propositional logic formula [18]. Solving a SAT problem is proven to be NP-complete [19]. However, certain families of SAT problems are solvable in polynomial time and SAT-solver heuristics are often effective in practice [20]. The satisfiability modulus problem (SMT) is the problem of determining if a model of value assignments exists that satisfies a given first-order logic formula [21]. In short, SMT formulas allow for a much higher level modeling language. As the CBMC papers [2] [12] [16] only describe SAT encoding we will not go into detail about SMT encoding.

The input for this step is the C and P bit-vector equations from the symbolic execution step. Before beginning the translation of the equation into a SAT problem, we simplify using the following rule:

$$(\lambda i : e)[x] \longrightarrow e[substitute\ i/x]$$

The rule denotes i as the formal parameter, e as the function body, and x as the actual parameter. By applying this rule, we substitute function calls (for array indexing) with their function body, where all instances of i are replaced with x. This substitution allows CBMC to eliminate the variables for the formal parameters and thus reduce the resulting CNF equation.

As an example consider the following C and P:

$$\begin{aligned} C := \quad & n_0 = 10 \wedge \\ & a_0 = \lambda i : 0 \wedge \\ & a_1 = \lambda i : i = (5 + 3)\ ?\ 1 : a_0[i] \\ P := \quad & a_1(5 + 3) > 0 \end{aligned}$$

Using the simplification rule above, the C and P becomes:

$$\begin{aligned} C := \quad & n_0 = 10 \\ P := \quad & ((5 + 3) = (5 + 3)\ ?\ 1 : 0) > 0 \end{aligned}$$

Which can be further simplified to:

$$\begin{aligned} C := \quad & n_0 = 10 \\ P := \quad & 1 > 0 \end{aligned}$$

The actual translation from C and P into the SAT problem on CNF form is not very well documented but is briefly mentioned in [16, p. 26]. The translation of the bit-vector Boolean operators is stated to be done by adding variables instead of applying the law of distribution. Bit-vector addition, subtraction, and relational operators are translated using a carry chain adder. Multiplication is translated into an optimized multiplication circuit. Shifting operations are translated into shifting circuits. Any remaining lambda expressions for arrays are expanded. In short, the translation into SAT is done by applying techniques from computer arithmetic.

In the end, we negate the SAT problem (corresponding to $C\ AND\ !P$) since we are more interested in validity rather than satisfiability. We are less interested in finding a model of variable assignments that can satisfy our assertions. Instead, we want to find the model of value assignment that breaks the assertions or proves that no such model exists.

1.5 Decision Procedure

CBMC uses third-party SAT/SMT solvers. By default CBMC uses the MiniSAT solver [22]. The MiniSAT solver takes as input a SAT problem on CNF form and tries to determine if the SAT problem is satisfiable.

If *C AND !P* is satisfiable, then the SAT solver returns a model that satisfies the problem [12]. The returned model can be considered mathematical proof that the assertions do not hold. Alternatively, if no satisfying solution (model) is found, then no assertion can be violated within the specified unwinding bounds [12].

Note that the proofs can only be considered sound if the underlying translation of the code is sound. The documentation does not go into detail about the soundness nor completeness of the translations. As far as we know, there are no guarantees of either.

Chapter 2

CBMC Tutorial

This chapter is written based on [22]. The content will serve as an introduction to verification with CBMC. It covers writing annotations, generating verification properties, nondeterminism, and some best modeling practices.

2.1 About and Releases

We are using release 5.28 of CBMC [23]. Note that the official page for CBMC denotes the newest CBMC release wrongfully as being 5.11 [24]. It is, in our experience, beneficial to use the newer releases of CBMC as there have been some frustrating bugs (indicating impossible errors) that have been resolved. CBMC supports "C89, C99, most of C11 and most compiler extensions provided by GCC and Visual Studio" [24]. CBMC is available for Windows, Linux, and MacOS.

The team behind CBMC has developed a CPROVER Eclipse plugin. The plugin should be beneficial to use for, among others, visualizing counterexamples. The official site states that the plugin requires Eclipse CDT Luna or newer. The plugin is, however, in our experience, incompatible with most (if not all) newer versions of Eclipse.

2.2 Annotations

Annotations are used to describe expected program behavior and to instruct the verification tool on how to verify the program. The annotations are written in the `.c` file. In this section, we will present a select few annotations.

2.2.1 *Constraints*

`__CPROVER_assume(cond)`
Assume annotations restrict the considered nondeterministic values. If values are met that do not satisfy the assumption, then these corresponding states are restricted from the search space for assertions that follow. This can result in an empty state space. Assumptions are added to the constraints C during verification. Note that assumptions are not asserted. An assumption may be illogical and never hold, but it will not fail verification.

`__CPROVER_assert(cond, mes)`
This annotation specifies an assertion. The verification will fail if the assertion does not hold for all program paths and CBMC will notify that this assertion does not hold. The assertion may take any Boolean expression as input and can consist of other CPROVER annotations. Other than the Boolean condition, it also takes a message that is used to describe the assertion. An assertion in a loop must hold for all iterations.

`assert(cond)`
The CPROVER tool can also recognize C assertions from `assert.h`. Assertions on the form `assert` are equivalent to `__CPROVER_assert` in what they can verify. CPROVER annotations may also be included in the condition.

`__CPROVER_precondition(cond, mes)`
A precondition specifies a condition that must hold at the beginning of a function body. Currently, a precondition

is simply translated into an assertion during verification. Note that a precondition must have a message describing the assertion.

`__CPROVER_postcondition(cond, mes)`
Postcondition is used to specify a condition that must hold at the end of a function body. A postcondition is translated into an assertion during verification. The postcondition must also have a descriptive message.

2.2.2 *Pointers*

`__CPROVER_POINTER_OBJECT(ptr)`
This annotation returns the ID of the object that the pointer points to. The CPROVER tool assigns a unique ID to each active object during verification.

`__CPROVER_POINTER_OFFSET(ptr)`
This annotation returns the offset of a pointer relative to the base offset of the respective object that is pointed at.

`__CPROVER_same_object(ptr1, ptr2)`
This annotation returns true if the two pointers point to the same object.

`__CPROVER_OBJECT_SIZE(ptr)`
This annotation returns the size of the object pointed to. Note that if the pointer points to a nested object it will return the size of the parent object.

`__CPROVER_DYNAMIC_OBJECT(ptr)`
This annotation returns true if the pointer points to memory allocated on the heap.

`__CPROVER_r_ok(ptr, size)`
This returns true if the memory in the range `ptr` to `ptr + size` is allocated.

`__CPROVER_w_ok(ptr, size)`
This returns true if the memory in the range `ptr` to `ptr + size` is allocated. Note that this is the exact same behavior as `__CPROVER_r_ok(ptr, size)`.

2.3 Verification

Verification is initiated by executing a command in CBMC. The commands are on the form `cbmc filename.c --arguments`. Note that multiple arguments can be applied together.

2.3.1 *Properties*

The automatically generated assertions that are verified can be seen as assertions that specify the absence of errors. If they fail, there is e.g. an out-of-bounds error. We will go into detail about some of the underlying Boolean expressions that are evaluated for some of the generated assertions that we deem most interesting. Note that the generated Boolean expressions may differ depending on the CBMC version. Although we do not expect huge differentiations.

`--bounds-check`
Generate checks related to array out-of-bounds errors. Each occurrence of an array access `arr[e]` which has array size `SIZE` results in two generated properties, as shown in Listing 2.1.

```
1 //array upper bound
2 !((signed long int)e >= arr$array_size0)
3
4 //array lower bound
5 (signed long int)e >= 0l
```

Listing 2.1: The CBMC (5.11) generated bounds checks for array accesses.

`--pointer-check`
Generate properties related to pointer errors. Each occurrence of a dereference `*ptr` results in seven generated properties, as shown in Listing 2.2.

```
1 //Pointer NULL
2 !(POINTER_OBJECT(ptr) == POINTER_OBJECT(((const char *)NULL)))
3
4 //Pointer invalid
5 !INVALID-POINTER(ptr)
6
7 //Deallocated dynamic object
8 !(POINTER_OBJECT(ptr) == POINTER_OBJECT(__CPROVER_deallocated))
9
10 //Dead object in *ptr
11 !(POINTER_OBJECT(ptr) == POINTER_OBJECT(__CPROVER_dead_object))
12
13 //pointer outside dynamic object bounds
14 POINTER_OFFSET(ptr) >= 0l && __CPROVER_malloc_size >= 1ul + (unsigned long int)
      ↪ POINTER_OFFSET(ptr) || !(POINTER_OBJECT(ptr) == POINTER_OBJECT(
      ↪ __CPROVER_malloc_object))
15
16 //pointer outside object bounds
17 OBJECT_SIZE(ptr) >= 1ul + (unsigned long int)POINTER_OFFSET(ptr) && POINTER_OFFSET(ptr) >=
      ↪  0l || DYNAMIC_OBJECT(ptr)
18
19 //Invalid integer address
20 ptr == ((const char *)NULL) || !(POINTER_OBJECT(((const char *)NULL)) == POINTER_OBJECT(
      ↪ ptr))
```

Listing 2.2: The CBMC (version 5.11) generated pointer checks for pointer dereferences.

`--signed-overflow-check`
Generates assertions related to signed arithmetic leading to overflow or underflow.

`--unsigned-overflow-check`
Generates assertions related to unsigned arithmetic leading to overflow or underflow.

`--float-overflow-check`
Generates assertions that check whether floating point numbers overflow.

`--undefined-shift-check`
Generates range checks for shift distances.

`--div-by-zero-check`
Generates division by zero checks.

`--nan-check`
Generates floating point NaN (not a number) checks.

`--pointer-primitive-check`
Generates assertions that check that the pointers given as input to the pointer annotations such as `__CPROVER_OBJECT_SIZE(ptr)` (cf. Section 2.2.2) are null or valid. The pointer is invalid if it points outside of a valid memory object and will then cause undefined behavior of the CBMC pointer annotations.

`--memory-leak-check`
Generates assertions that check that all heap allocated memory are freed before the program terminates.

`--pointer-overflow-check`
Generates assertions that check that a pointer never exceeds the bounds of the memory object it points to.

`--conversion-check`
Generates assertions that check that variables can fit the values they are assigned.

`--enum-range-check`
Generates assertions that check that values of an enum type are in the range of the enum type.

2.3.2 *Execution*

There are numerous execution options for verification. These options may be applied together. Again, the commands are on the form `cbmc filename.c --arguments`.

`--function some_function`
Specifies a function as the entry point for the verification.

`--unwind n`
The bound `n` specifies the maximum amount of loop unwinds. This bound is "global" for all occurring loops. `--unwind n` is a useful option when dealing with non-terminating verification due to seemingly endless loops. Unwinding is done in accordance with the loop unrolling described in Chapter 1.

`--depth n`
Depth limits the number of program steps to be processed for each program path. The depth `n` species how far we will search in the CFG. Note that the step criterion specifies steps in the CFG and not instructions in the source program. This option `depth` together with the option `unwind` is what makes CBMC bounded.

`--unwinding-assertions`
The `--unwinding-assertions` argument is used together with the `unwind` argument. It generates a check at the end of each unrolled loop to verify that no more loop iterations can occur. If the check fails, then subsequent checks may be based on a spurious program path.

`--unwindset L:B`
`L` denotes the loop ID and `B` denotes the bound for the loop. E.g. `--unwindset main.0:10 --unwindset main.1:20` specifies a loop bound of 10 for the first loop in main (`main.0`) and a loop bound of 20 for the second loop in main (`main.1`).

`--partial-loops`
Allows for partial execution of loops. The disadvantage of this approach is that the resulting program paths may

not exist in the actual program.

`--show-loops`
Prints a list of all loop IDs and their respective line number in the source code.

`--trace`
Prints counterexample traces. The trace describes the state numbers, the respective line numbers, and the variable values. Note that the states correspond to SSA formatted code.

`--show-properties`
Prints out all properties that are verified. An example of a property is seen in Listing 2.3.

```
Property arrayMax.assertion.1:
  file arrayMax.c line 44 function arrayMax
  Postcondition: result greater than
  result >= arr[(signed long int)n]
```

Listing 2.3: `--show-properties` output

`--no-assertions`
Ignore all user assertions during verification.

`--no-assumption`
Ignore all user assumptions during verification.

`--no-built-in-assertions`
Ignore all assertions in built-in libraries during verification.

`--assert-to-assume`
Converts all user assertions to assumptions.

2.4 Nondeterminism

It is common for programs to rely on input from an environment. The source of the input can be a file, keyboard, or network socket. To model multiple possible inputs, CBMC uses nondeterminism. This means that CBMC will consider all possible values and their respective program paths.

A function that reads from a file, keyboard, etc., is automatically treated as a source of nondeterminism. Extern variables, memory allocated with malloc, and nondeterministic functions (functions without a body) are valid sources of nondeterminism. Variables (e.g. formal parameters) that are uninitialized are also nondeterministic. As an example consider Listing 2.4. Analysis of `foo` as the verification entry point will consider all possible integer values for the formal parameter `size` and array `arr`. Note that variable `i` and function `bar` are also sources of nondeterminism.

```
int foo(int* arr, int size) { //nondeterministic formal parameters
    int result = arr[0];
    int i; // nondeterministic
    int j = bar(); //nondeterministic

    while (i < size) {
        if (arr[i] > result)
            result = arr[i];
        i += j;
    }
```

```
11      return result;
12  }
13
14  int bar(); //returns a nondeterministic integer
```

Listing 2.4: Examples of source of nondeterminism.

2.5 Loops and Boundedness

CBMC does not guarantee verification termination. CBMC is sometimes unable to determine when a loop will end if the loop condition relies on something data specific. Here, specifying a loop bound is necessary otherwise CBMC will endlessly try to verify the program.

It would seem obvious to restrict the range of the variables in the loop condition to guarantee loop termination. However, `__CPROVER_assume` will not influence the transformation of loop unrolls. Instead, it only influences the SSA program and thus the C and P equation. As a result `__CPROVER_assume` affects the succeeding state space and assertions. As an example Listing 2.5 will verify in seemingly endless time when executing `cbmc arrayMax.c --bounds-check --pointer-check`.

```
1  int arrayMax(int* arr, int size) {
2      __CPROVER_assume(5 > size && size > 0);
3      int result = arr[0];
4      int i = 0;
5
6      while (i < size) {
7          if (arr[i] > result)
8              result = arr[i];
9          i++;
10     }
11     return result;
12 }
```

Listing 2.5: Restriction of the nondeterministic `size` valuations by use of `__CPROVER_assume`.

Instead we need to use `--unwind` (or `--unwindset`) to specify a loop bound when verifying. Loop bounds should always be paired with `--unwinding-assertions` to determine if the succeeding assertions may be spurious. If we only want to consider a set of possible inputs instead of all possible inputs, we may combine loop bounds with assumptions. Executing the verification of Listing 2.5 with `cbmc arrayMax.c --bounds-check --pointer-check --unwind 5 --unwinding-assertions` resolves the termination issue.

Another solution could be to orchestrate the `main` function to call the function under verification with explicit input and then set the entry point of the verification to `main`. CBMC will then not verify the function for all inputs but only the specified input. Which is simpler to identify a loop bound for. An example is shown in Listing 2.6.

```
1  int main(int argc, char** argv){
2      int arr[6] = {1,2,3,4,5,6};
3      arrayMax(arr, 6);
4  }
```

Listing 2.6: Explicit input makes identifying loop bounds simpler for CMBC.

2.6 How to Structure a Proof

This section is based on [25]. A proof harness is the constructed verification environment that calls the function under verification. We will go into detail on how to write a perfect proof harness. It is ideal, if possible, not to alter the function under verification. To set up the data structures, nondeterminism, and assumptions, we create a proof harness function called `PROOF_HARNESS()`. Additionally, we also want to verify that our postcondition holds after the function has returned. An example of a proof harness function is seen in Listing 2.7.

```
1  void PROOF_HARNESS(){
2      unsigned int size;
3      int arr[size];
4
5      __CPROVER_assume(5 > size && size > 0);
6
7      int max = arrayMax(arr, size);
8
9      __CPROVER_postcondition(HELP_exists_in_arr(arr, size, max), "Postcondition: returned
       ↪ max exists in array");
10     __CPROVER_postcondition(HELP_element_greater_than_or_eq(arr, size, max), "
       ↪ Postcondition: returned max greater than");
11
12 }
```

Listing 2.7: Proof harness function for `arrayMax`

The `PROOF_HARNESS` function first initializes the needed data. As can be seen in Listing 2.7, `size` and `arr` are both left uninitialized and will therefore be assigned a nondeterministic value by CBMC. We then restrict the possible `size` values by stating assumptions about the data. All assumptions should be moved to the `PROOF_HARNESS` if convenient and possible. The `PROOF_HARNESS` function then makes a function call to `arrayMax`, i.e. the function we want to verify. After the function has returned, we want to verify that the returned max value satisfies our postconditions. The `arrayMax` function can be seen in Listing 2.8.

```
1  int arrayMax(int* arr, const unsigned int size) {
2      __CPROVER_precondition(5 > size && size > 0, "Precondition: assumes 5 > size > 0");
3      int result = arr[0];
4      int i = 0;
5
6      while (i < size) {
7          if (arr[i] > result)
8              result = arr[i];
9          i++;
10     }
11
12     __CPROVER_postcondition(HELP_exists_in_arr(arr, size, result), "Postcondition: result
       ↪ exists in array");
13     __CPROVER_postcondition(HELP_element_greater_than_or_eq(arr, size, result), "
       ↪ Postcondition: result greater than");
14
15     return result;
16 }
```

Listing 2.8: The `arrayMax` function that is being verified.

Note that there is a correspondence between the `PROOF_HARNESS` assumption and the `arrayMax` precondition. We assert that our assumptions hold. Firstly, this is done to ensure that CBMC has the expected behavior. All of these preconditions should, in the context of the `PROOF_HARNESS`, arbitrarily succeed. Secondly, if we verify `arrayMax` out of this particular `PROOF_HARNESS` context, we still want to assert that the preconditions hold. If they do not hold, we have no formal guarantees for the function properties.

Similarly, there is a correspondence between the `PROOF_HARNESS` postcondition and the `arrayMax` postcondition. This is because we want to verify both that the resulting behavior within the function's body is correct and that the function result is correct after its termination. In this particular case, they both succeed. However, as an example, if a pointer is returned to some local variable within the function body, both postconditions in the `PROOF_HARNESS` will fail.

We group assertions into three categories:

- Assertions that make sense to have as both a postcondition in the `PROOF_HARNESS` and an assertion in the underlying function.
- Assertions that are only correct as a postcondition in the `PROOF_HARNESS`. E.g. postconditions that verify that calls occurred in a particular order.
- Assertions that are only correct as an assertion in the underlying function. E.g. assertions that verify that the given input is ok.

When choosing where to place assertions, it is crucial to consider a number of things. First and foremost, the correctness of the placement. If both an assertion and a postcondition may be correct, then consider having both as it will provide a stronger proof as seen in the `arrayMax` case above. CBMC annotated C code can quickly become complex. Therefore similar assertions or postconditions should be grouped to improve the readability.

2.7 Guarantees

The properties that are verified are only shown when one or more fails. If all succeed, CBMC will state the number of assertions and that `SAT checker: instance is UNSATISFIABLE` meaning that no proof exists where the properties do **not** hold. An assertion fails if it is not satisfiable. An assertion succeeds if it is satisfied or if the assertion is unreachable.

Succeeding unwinding assertions guarantee that the verified properties apply for all possible program paths. Failing properties under a failing unwinding assertion may be used for bug hunting, but the failing properties are not guaranteed to exist in a "real" program path.

CBMC cannot guarantee the "total correctness" (absence of all bugs) of all programs [22]. CBMC is useful for proving specific behaviors, such as the absence of specific flaws/bugs. These guarantees for specific behaviors can contribute to the reasoning about the safety and correctness of a program. However, CBMC is not guaranteed to be without errors itself.

We have encountered properties that, in our opinion, should not fail but did. After a discussion with the developers behind CBMC, we found that there indeed was a bug in CBMC and subsequently found that it was resolved in a newer release [26]. That is to say that false negatives and false positives are not impossible in CBMC, although we have not found any in release 5.28.

2.8 Debugging CBMC

This section covers best practices for debugging errors found by CBMC. These recommendations are based on AWSLabs's findings (cf. [27]) as well as our own experiences.

2.8.1 Prioritization of Errors

We recommend the following approach on how to choose which errors to solve first: Try and fix the error discovered earliest in the CBMC verification. The reason is that the trace leading to the error is short and that fixing it could fix other errors found by CBMC. Prioritize fixing the errors found in code one understands. Since it is easier to debug known functionality. Start by fixing simple errors. The error trace for a simple error is often easier to understand than for complex errors. Solving simple errors may also fix other more complex errors.

2.8.2 Debugging Assertions

The trace provided by CBMC displays the line number and the value for each assignment, leading up to an error. However, the trace does not display the value of variables read. This means that if an error happens in a statement using variables assigned a long time ago, it can be challenging to comprehend the input to the statement. A solution is to create dummy variables before the statement and assign the input of the statement to them. In this way, it is easier to view the values going into the statement.

A practical assertion that may seem redundant at first is to introduce `__CPROVER_asssert(0)` statements. If the assertion fails, it implies that the particular line is reachable. If the assertion succeeds, the assertion has not been reached, and the line is thus unreachable.

Similarly, it may also be beneficial to introduce assert statements into the code that does not relate to a program property of interest. The assertions can test one's understanding of the program's behavior. E.g. have an assertion that asserts that a given variable is never higher than 10. This facilitates easier detection of bugs that may cause the CBMC errors.

2.8.3 Still Errors?

Lastly, if debugging is unsuccessful, the error might be due to an error in CBMC. In this case, look for issues on GitHub to see if the problem is known, create an issue, or try new/other releases.

2.9 Function Contracts

The feature for defining function contracts for functions and verify them is still under development at the time of writing (CBMC version 5.28). As there are still known bugs (cf. [28] [29] [30] [31]) we will not be using function contracts to verify the `mask_ROM` boot stage. Therefore, we will not document the use of function contracts here (cf. Appendix D for details about function contracts).

Chapter 3

Cryptographic Concepts

This chapter gives an introduction to cryptographic concepts used in OpenTitan. Among the concepts introduced are the process of creating digital signatures, verifying digital signatures, RSA encryption, HMAC, and cryptographic hash functions.

3.1 Keys, Signing, Verification, and Hash Functions

This section is written based on [32], [33], and [34]. A digital signature is data, e.g. a number, that can be distributed together with a message. The signature allows the recipients to verify the authenticity and integrity of the message. Digital signatures are based on asymmetric keys and asymmetric signing and verification algorithms.

An asymmetric key pair consists of two cryptographic keys known as the private and public key. A cryptographic key is data, such as a number, that influences the result of e.g. signing and verification algorithms [35]. The private key is used together with a message and a signing algorithm to produce a signature for that message. The public key is used together with the signature and the message to verify the signature. Thereby verifying the integrity and authenticity of the message. A signature produced by the private key should only be possible to verify by using the associated public key. In addition, it should be computationally infeasible to forge a correct signature without having the private key. Also, it should be computationally infeasible to deduce the private key based on the message, signature, and public key.

To allow the recipients to verify the authenticity and integrity of a message m (a number) digital signatures are utilized in the following way. The sender owns a public and a private key. The recipients know the public key. The sender signs the message using the private key and a signing function. The message is then distributed with the signature. The recipients can verify the signature using the public key and a verification function. Verifying the signature is done by applying the public key to the signature and asserting if the output is equal to the message. If that is the case, the recipients know that the message is from the sender associated with the public key and that the message has not been tampered during distribution.

When signing a message, the signature is often not created by signing the message but the hash of the message. The message is still distributed in its original format, but now, together with the signature of the hash instead. In this case, the recipients also have to use the same hash function on the message when verifying it. A cryptographic hash function computes the hash. A hash function takes some data as input, such as text, and outputs a value of a specific size measured in bits. A reason for using hash functions when signing is to decrease the size of the message being signed. Decreasing the input's size is of interest since asymmetric operations are computationally demanding [36].

3.2 RSA

This section is written based on [37]. OpenTitan utilizes RSA algorithms for signing and verifying the signature of the `ROM_EXT` boot stage [38]. RSA is an asymmetric encryption algorithm. This section will focus on the RSA

algorithm's use in creating and verifying digital signatures.

In RSA, the private key consists of two numbers, n and d. Likewise, the public key consists of the two numbers, n and e. n is known as the modulus and is the product of two large prime numbers. e is known as the public exponent and d is known as the private exponent. The details of how n, d, and e are computed will not be covered. The relationship between the values is that for a message (a number) m that is 0 or above and less than n (i.e. $0 \leq m < n$), the following holds:

$$m = (m^e)^d \ mod \ n$$

When RSA is used in relation to digital signatures, the signing function $Sign$ is defined as:

$$Sign(message, d, n) = HASH(message)^d \ mod \ n$$

The $HASH$ function is a hash function that is known and used both by the sender and recipients. The verification function $Verify$ is defined as:

$$Verify(message, signature, e, n) = signature^e \ mod \ n == HASH(message)$$

It outputs *true* or $false$, indicating whether the signature matches the hashed message when applying the sender's public key to the signature. If the output is true, the recipient has verified the authenticity and integrity of the message.

3.3 HMAC

A HMAC hash, also known as hash-based message authentication code, is the result of an intricate procedure for hashing a padded HMAC private key and message pair [39]. The padding is done so that it is difficult to retrieve both the message and key from the HMAC hash. HMAC provides both data integrity and data authenticity. The hash is usually computed by a SHA2 or SHA3 hash function. HMAC pesudocode can be seen below.

Algorithm 1: HMAC [39]

Input:

key:	Bytes
message:	Bytes
hash:	Function
blockSize:	Integer
outputSize:	Integer

```
// Keys longer than blockSize are shortened by hashing them
if length(key) > blockSize then
    key ← hash(key)    // key is outputSize bytes long
end

// Keys shorter than blockSize are padded to blockSize by padding with zeros on the right
if length(key) < blockSize then
    key ← Pad(key, blockSize)    // Pad key with zeros to make it blockSize bytes long
end

o_key_pad ← key xor [0x5c blockSize]    // Outer padded key
i_key_pad ← key xor [0x36 blockSize]    // Inner padded key

return hash(o_key_pad || hash(i_key_pad || message))
```

Chapter 4

System Under Verification

This chapter covers the `mask_ROM` boot stage of OpenTitan in terms of functionality, external hardware modules, and the associated boot code seen in Appendix B. Note that this source code is based on the boot code developed with SV106f21 with some slight modifications to facilitate CBMC verification. The system we are verifying is only the `mask_ROM` boot code excluding the external hardware modules such as OTBN, HMAC, etc. In addition, we present the program properties we have deduced for the `mask_ROM` boot stage and that we want to verify using CBMC.

4.1 mask_ROM

`mask_ROM` is the first boot stage in OpenTitan. The main purpose of the `mask_ROM` boot stage is to validate and transfer execution to a `ROM_EXT`. The succeeding boot stages are `ROM_EXT`, `BL0` (the initial boot loader), and `Kernel` [40]. There are four entities in the logical security model of OpenTitan: End User, Silicon Creator, Silicon Owner, and Application Provider [7]. The last three entities are responsible for supplying and signing the software for their respective stages. This is illustrated in Fig. 4.1.1, taken from [1]. The dotted line indicates that a `BL0` is not mandatory.

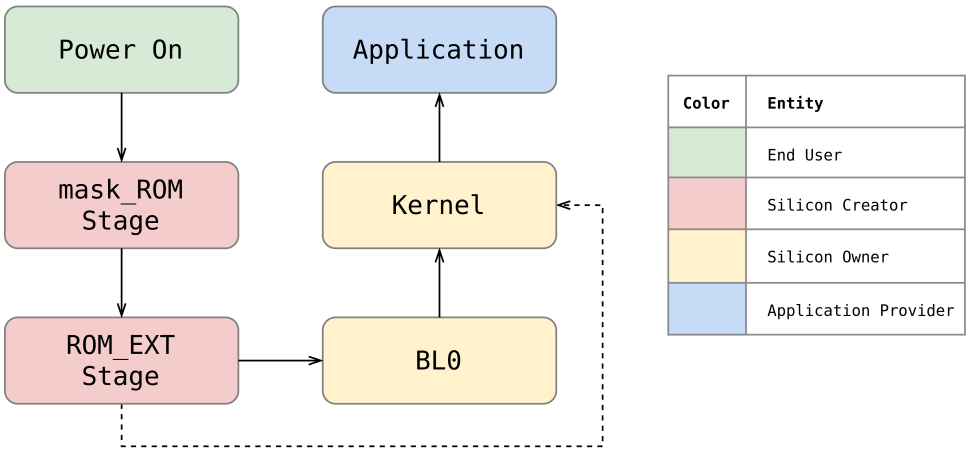

Figure 4.1.1: Stages of OpenTitan [1].

The actions performed by `mask_ROM` during secure boot is documented in [40] and [41]. The former describes the secure boot process from `Power On` to `mask_ROM` to `ROM_EXT`. The latter describes the implementation of `mask_ROM` that adheres to the description of the secure boot process for `mask_ROM`. Not all steps of the `mask_ROM` stage are mentioned here, only the ones relevant for verifying the boot code seen in Appendix B.

The first action of the `mask_ROM` is to set up the environment by disabling interrupts, cleaning the device state, and reading the boot reason [41]. Then the `mask_ROM` reads the boot policy from flash. The boot policy contains, among others, the `ROM_EXT` manifests for the `ROM_EXTs` that can be booted from. Essentially defining the `ROM_EXTs` that `mask_ROM` should try and boot from. The boot policy does also define what to do if a `ROM_EXT`

cannot be booted from. A `ROM_EXT` manifest contains e.g. a manifest identifier, `ROM_EXT` image code, a signature, and the public key used for verifying the signature (cf. [38] for the complete specification).

The second action consists of iterating through the different `ROM_EXT` manifests. For each manifest, the following is done:

1. Validate the manifest by checking the manifest identifier and if the signature is nonzero [38].
2. Retrieve the public key stored in the manifest and verify its validity against a whitelist stored in `mask_ROM` [42].
3. Use the `RSASSA-PKCS1-V1_5-VERIFY` verification function to verify the signature based on the public key and an expected message to ensure authenticity and integrity. The expected message is `system_state_value || device_usage_value || signed_area(rom_ext)` [38].
4. PMP region #0 is created and covers the memory where the `ROM_EXT` image code is stored. The PMP region is locked and permits reading and execution.
5. If all the above are successful, the execution is transferred from `mask_ROM` to the `ROM_EXT` entry point.
6. If the `ROM_EXT` returns execution to `mask_ROM`, then a fail function from the boot policy is executed.
7. If all `ROM_EXT` manifests are invalid, then a fail function from the boot policy is executed.

4.1.1 Context of System

The main hardware components used in the `mask_ROM` stage are shown in Fig. 4.1.2. Note that we do not verify the hardware components as they are not part of the considered system. Instead, we will be verifying the `mask_ROM` boot code's correct use and interaction with these.

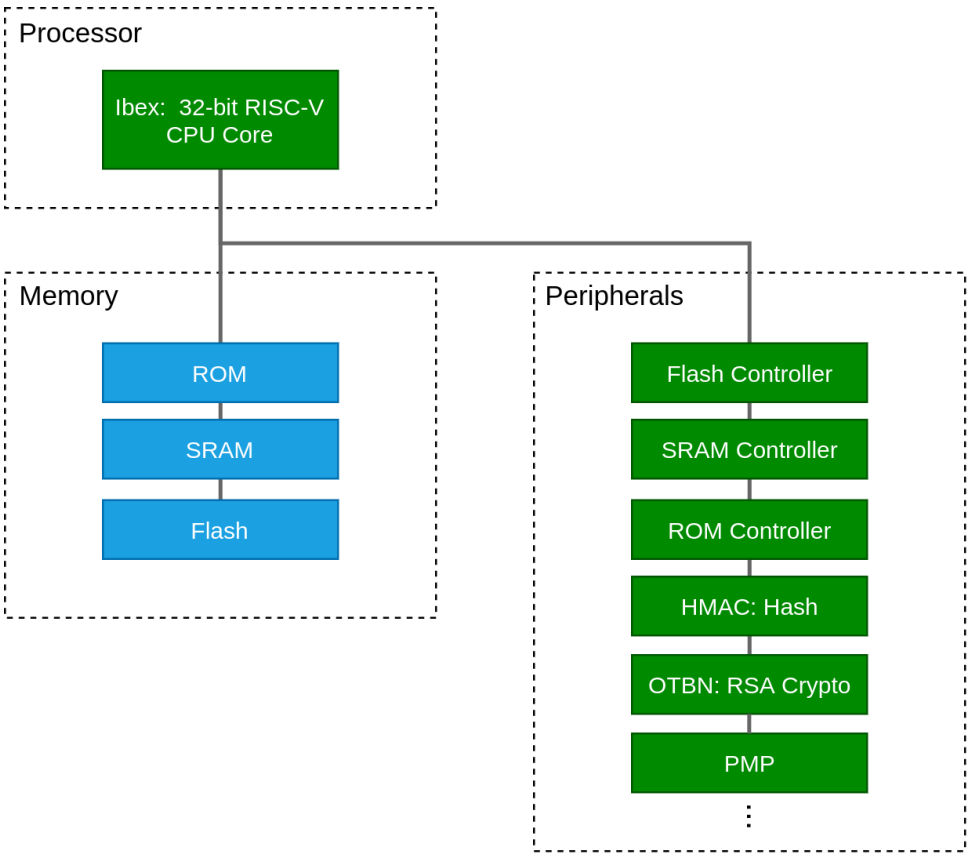

Figure 4.1.2: Overview of the hardware components related to the `mask_ROM` stage.

mask_ROM depends on the following hardware/software modules:

- **Flash/SRAM/ROM Controller**: The three controllers are responsible for defining an interface for accessing the associated memory.
- **HMAC**: The HMAC module is used within OpenTitan to compute the HMAC hash of a message and a 256-bit secret key [43]. The hash is computed as a SHA-256 hash.
- **OTBN**: The OpenTitan Big Number Accelerator (OTBN) is used within OpenTitan to perform asymmetric cryptographic operations [44]. The asymmetric cryptographic operations could be the RSA signing and verification algorithms used for signing and verifying the ROM_EXT stage.
- **PMP**: A hardware module used to restrict memory, such as flash, in terms of read, write, and execution rights [41]. When PMP restricts a memory region, it is said that a PMP region covers that memory region. A PMP region can also be locked, meaning that it can only be removed by doing a system restart.
- **Silicon Creator Public/Private keys**: There is an asymmetric key pair for each boot stage. Each pair is used for signing and verifying a stage [45]. The Silicon Creator has two asymmetric key pairs used for signing and verifying the mask_ROM and ROM_EXT stages. The keys are, at minimum, 3072-bit keys and are used together with an RSA signing and verification function [38]. Verification of the signatures is done on the OpenTitan chip. For security reasons, the private key is never stored on the OpenTitan chip. Therefore, signing is done during manufacturing and not on the OpenTitan chip itself.

4.2 mask_ROM Boot Code

This section explains the boot code in Appendix B. This is the boot code that is being verified along this report. The boot code is written based on the documentation of OpenTitan found in [38], [40], [41], and [46]. The call graph of the boot code is seen Fig. 4.2.1. The circles indicate functions and the arrows indicate function calls. The boxes indicate groups of functionality.

- mask_rom_boot : This is the entry point of the mask_ROM boot stage. It is responsible for trying to validate ROM_EXTs and upon a valid ROM_EXT it transfers execution to it [40] [41].
- FLASH_CTRL_read_boot_policy : This Flash Controller function reads the boot policy from flash [41].
- FLASH_CTRL_rom_ext_manifests_to_try : This Flash Controller function reads the ROM_EXT manifests from the boot policy stored in flash [41].
- PMP_enable_memory_protection : Calls PMP_WRITE_REGION to apply PMP region #15 which covers the entire flash, is locked, and allows for read access [40] [41].
- PMP_WRITE_REGION : This PMP function creates a PMP region based on the inputs: region ID and read, write, execute, and locked bits.
- check_rom_ext_manifest : Checks that the signature is non zero and that the identifier is non zero.
- read_pub_key : Simply retrieves the RSA-3072 public key (exponent and modulus part) from a ROM_EXT manifest [38].
- check_pub_key_valid : Verifies that a RSA-3072 public key is whitelisted, based on a whitelist stored in mask_ROM [42] [38].
- ROM_CTRL_get_whitelist : Retrieves the public key whitelist stored in mask_ROM .
- verify_rom_ext_signature : Supplies the OTBN_RSASSA_PKCS1_V1_5_VERIFY function with the RSA public key, message, and signature.

- `OTBN_RSASSA_PKCS1_V1_5_VERIFY` : This OTBN function verifies a RSA-3072 signature. It follows the `RSASSA_PKCS1_V1_5_VERIFY` specification. The supplied signature is checked to be of the correct size. The signature is then decrypted using the `OTBN_RSA_3072_DECRYPT` function. Then, the supplied data is hashed using the `HMAC_SHA2_256` function. The hash and the decrypted signature are then compared. If they are equivalent the signature is verified [38].
- `OTBN_RSA_3072_DECRYPT` : This OTBN function implements a `RSA_3072_DECRYPT` algorithm to decrypt a signature with a supplied RSA public key.
- `HMAC_SHA2_256` : This HMAC function computes a HMAC hash from a HMAC key and message. It returns the `SHA2_256` hash of a padded concatenation of the key and message.
- `PMP_unlock_rom_ext` : Calls `PMP_WRITE_REGION` to create PMP region #0 which covers the `ROM_EXT` image, is locked, and allows for reads and execution [40] [41].
- `final_jump_to_rom_ext` : Is only called if all previous verification, for the given `ROM_EXT` , passed. Transfers execution to the `ROM_EXT` entry point from the supplied `ROM_EXT` manifest [40] [41].
- `boot_failed` : Is only called if all `ROM_EXTs` fail verification. Executes a fail function supplied by the boot policy [41].
- `boot_failed_rom_ext_terminated` : Is only called if a `ROM_EXT` terminates. Executes a fail function supplied by the boot policy and supplies it with the terminated `ROM_EXT` 's manifest [40] [41].

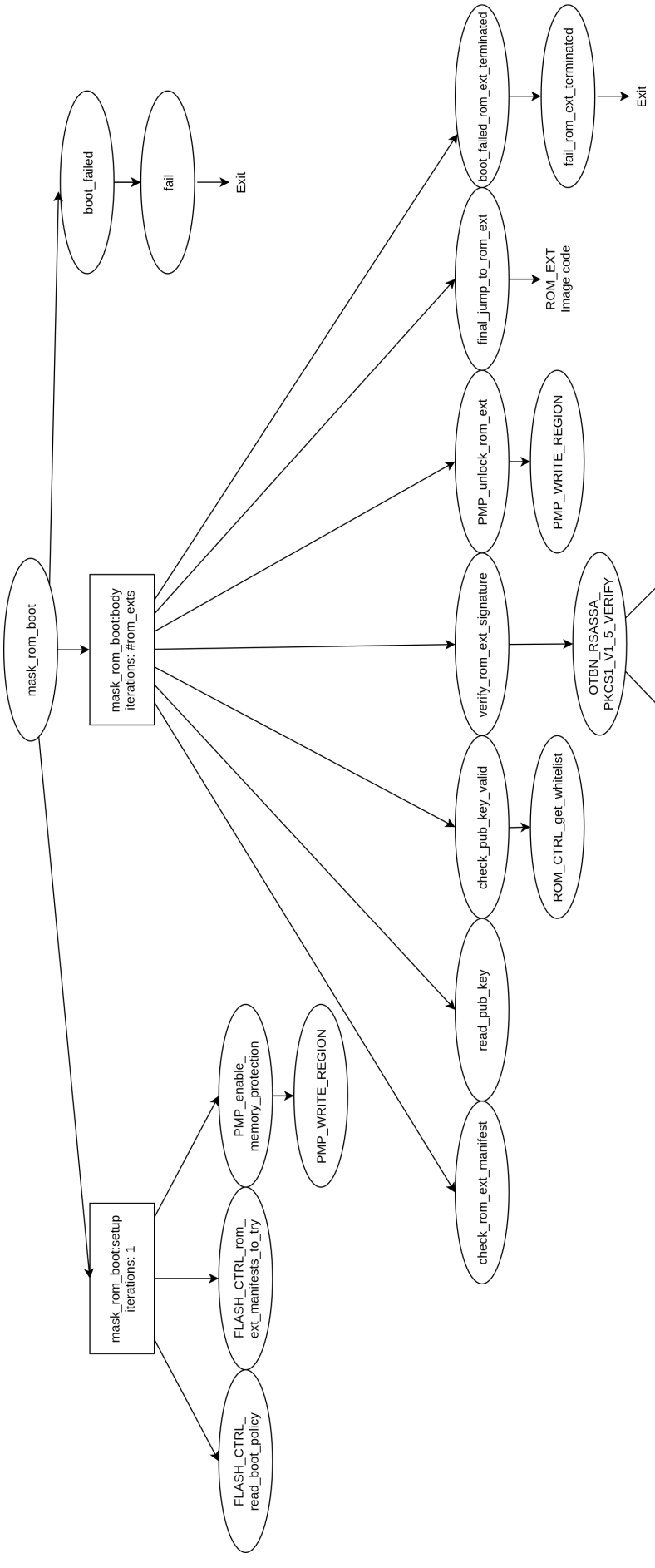

Figure 4.2.1: Call graph of the `mask_ROM` boot code.

Note: The boxes do not indicate function calls.

4.3 Program Properties

One of the results generated from our P9 project [1] was a security analysis of OpenTitan. The security analysis contained security policies, security goals, and security mechanisms for the OpenTitan initial boot code (`mask_ROM` and `ROM_EXT`). The security policies and security goals relevant to the `mask_ROM` stage are cited below (directly taken from [1]). We will, in this section, present new program properties that are more refined and specific to the implementation level code of the `mask_ROM` stage and thus easier to translate into CBMC assertions. These program properties are derived from their respective security goal/policy. PROPERTY 0 is the exception to this as it has no direct parent goal/policy but is still fundamental for overall program correctness and safety.

"**No parent policy or goal**"

- PROPERTY 0: The `mask_ROM` boot code must be free of bugs.

"**P1: It should only be possible to execute code that has been validated (authenticity/integrity)**"

"**G1: The hash of the `ROM_EXT` image and the signature of the hash must be validated by `mask_ROM` before it is executed to ensure authenticity and integrity of the image.**"

- PROPERTY 1: The `ROM_EXT` manifest for a `ROM_EXT` must be signed with a RSA-3072 signature. If a `ROM_EXT` manifest for a `ROM_EXT` is unsigned (i.e. the signature is a sequence of zeros) the `ROM_EXT` is considered invalid to boot from.

- PROPERTY 2: The public RSA-3072 key used for the signature contained in the `ROM_EXT` manifest must be valid in order to be valid to boot from.

- PROPERTY 3: The HMAC hash must be calculated by either a SHA2-256, SHA3-256, SHA3-384, or SHA3-512 hash function.

- PROPERTY 4: The computed HMAC hash message must be calculated from `system_state_value || device_usage_value || signed_area(rom_ext)` [38].

- PROPERTY 5: The signature in the `ROM_EXT` manifest must be validated using the `RSASSA-PKCS1-V1_5-VERIFY` [47] function with inputs: public RSA-3072 key, appended message (`system_state_value || device_usage_value || signed_area(rom_ext)`), and RSA-3072 signature. If the function returns false the `ROM_EXT` is invalid to boot from.

- PROPERTY 6: If all validation steps have succeeded, then transfer execution to `ROM_EXT` by starting execution at the entry point of the `ROM_EXT` image code. If execution returns, execute the `fail_rom_ext_returned` function provided by the boot policy.

- PROPERTY 7: If at any point a `ROM_EXT` is invalidated, the `ROM_EXT` is considered unsafe to boot from and the `mask_ROM` must proceed to validate the next `ROM_EXT` .

- PROPERTY 8: If validation fails for all the `ROM_EXTs`, `mask_ROM` must execute the `fail` function provided by the boot policy.

"**P4: There is a privilege hierarchy that is respected (i.e. access rights: read, write, and execute)**"

"**G10, G11, G12: Only software with write/read/execute access to some memory section may modify/read/execute it.**"

- PROPERTY 9: The entire flash must be covered by a PMP region at the initialization of `mask_ROM`. The PMP region must be locked and restricted to read-only access.

- PROPERTY 10: If a `ROM_EXT` is validated, then `mask_ROM` must create a PMP region covering the `ROM_EXT` memory that is locked and allows for read and execution access.

Chapter 5

Verification of System

In this chapter we describe the verification and the process of verifying the program properties that we specified for the OpenTitan `mask_ROM` boot stage, described in Section 4.3. As mentioned in Section 4.1.1, we do not intend to verify the implementation of the hardware components used by `mask_ROM`. We intent to verify that the developed `mask_ROM` boot code satisfies the program properties and interacts correctly with the hardware components. The CBMC annotated boot code referenced in this chapter can be seen in full in Appendix C and on GitHub at [48].

5.1 Model of the Boot Code

This section describes the changes we have made to the boot code and how we have modeled data and hardware to verify it using CBMC and argumentation of whether these changes and models are valid.

5.1.1 Hardware and External Functionality

We have modeled most of the hardware dependencies (i.e. HMAC, OTBN, ROM Controller, and Flash Controller) depicted in Fig. 4.1.2. The functions that model the hardware are prefixed with the name of the hardware. A general approach that we have used for modeling hardware is to use nondeterminism. As a nondeterministic model is at least an over-approximation of the output, we consider it a reasonable abstraction as we only verify the use of hardware. The following functions are functions from Section 4.2 that have been modeled. The call graph of the modeled boot code is seen in Fig. 5.1.1 (Note that several of the functions are marked in red. This is to denote that they are modeled.)

- `FLASH_CTRL_read_boot_policy`: Instead of reading a boot policy from flash by interacting with the Flash Controller, it returns a nondeterministic boot policy. This model can return all possible boot policies.
- `FLASH_CTRL_rom_ext_manifest_to_try`: Instead of returning a list of `ROM_EXT` manifests from flash by interacting with the Flash Controller it returns a list of nondeterministic `ROM_EXT` manifests. This model can return all possible `ROM_EXTs`.
- `PMP_WRITE_REGION`: This function has no actual PMP like functionality, but a call to it is assumed to correctly apply a PMP region based on the inputs given to it.
- `ROM_CTRL_get_whitelist`: Instead of returning the whitelist of valid public keys from ROM, by interacting with the ROM Controller, it returns a list of nondeterministic public keys. This model can return all possible public keys.
- `OTBN_RSA_3072_DECRYPT`: Instead of implementing RSA-3072 decryption, it returns a nondeterministic 256-bit sized string representing the HMAC hash that was encrypted. This model can return all possible 256-bit sized strings.
- `HMAC_SHA2_256`: We have implemented a version of HMAC which uses SHA2-256. However, we also have a modeled version which returns a nondeterministic 256-bit sized string representing the hash. This model can return all possible 256-bit sized strings.

- `image_code`: The `image_code` of a `ROM_EXT` is modeled as an empty function that always returns. The aspect of having no functionality is a reasonable model considering that `ROM_EXT` is out of the scope of `mask_ROM`. The aspect of always returning is also reasonable as a real `ROM_EXT` will return eventually. It is also necessary for CBMC to terminate.
- `fail` and `fail_rom_ext_termininated`: The failure functions defined in the boot policy is model as empty functions that always return. The aspect of having no functionality is a reasonable model considering the lack of documentation and since the implementation is supplied by the boot policy.

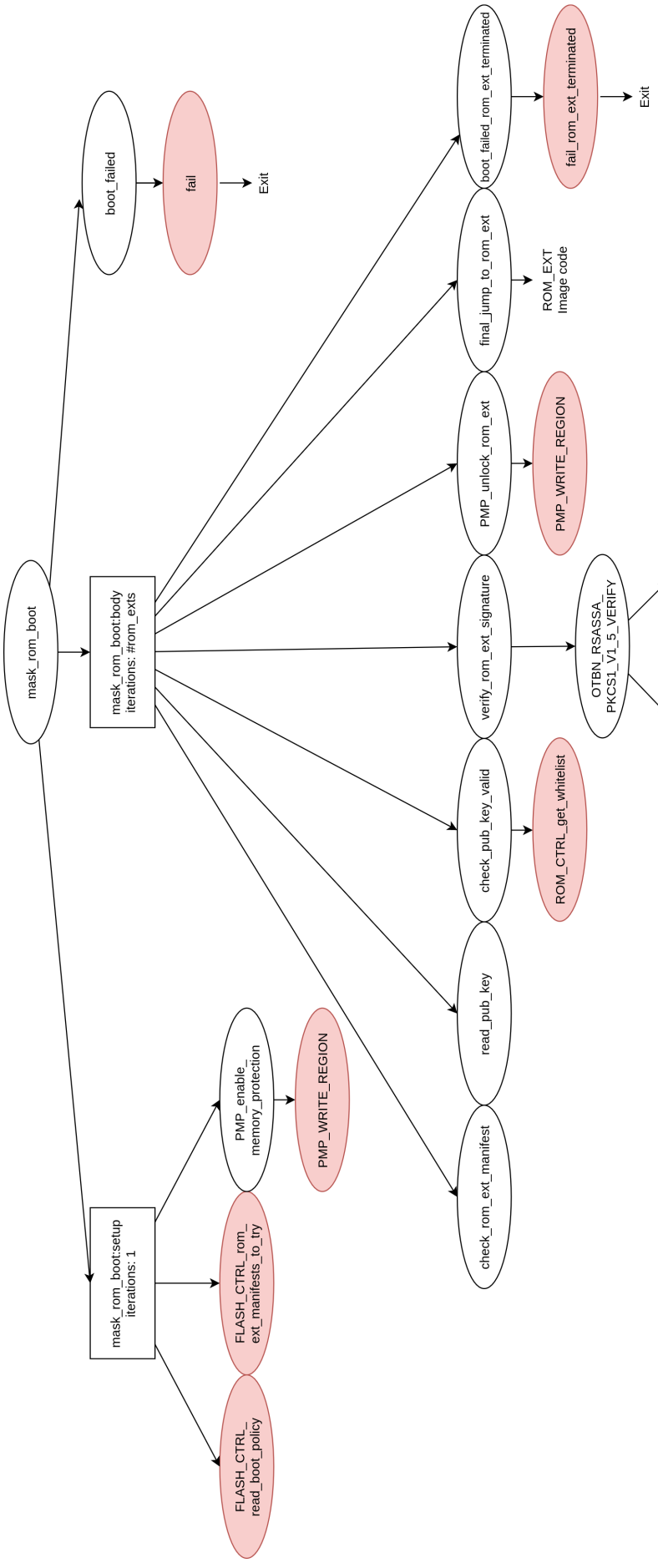

Figure 5.1.1: Call graph of the modeled `mask_ROM` boot code.

Note: The boxes do not indicate function calls. Red indicates that the function is modeled.

5.1.2 Data Structures

The most relevant data used in the `mask_ROM` boot code are the RSA-3072 key, RSA-3072 signature, SHA2-256 hash, boot policy, and `ROM_EXT` manifest. We have modeled a RSA-3072 key as a struct in C. It contains an exponent, represented as an `int32_t`. The modulus is represented as `int32_t[96]`. A RSA signature is also represented as `int32_t[96]`. A SHA2-256 hash is represented as `char[256]`. We believe these to all be reasonable representations.

The boot policy and the `ROM_EXT` manifest are modeled as structs in C. The content of the modeled boot policy and `ROM_EXT` manifest is shown in Listing 5.1 and Listing 5.2, respectively[1].

```
53 typedef struct boot_policy_t {
54     int identifier;
55
56     //which rom_ext_slot to boot
57     int rom_ext_slot;
58
59     //what to do if all ROM Ext are
       ↪ invalid
60     char* fail;
61
62     //what to do if the ROM Ext
       ↪ unexpectedly returns
63     char* fail_rom_ext_terminated;
64
65 } boot_policy_t;
```

Listing 5.1: Model of a boot policy in `mask_rom.h`.

```
33 typedef struct rom_ext_manifest_t{
34     uint32_t identifier;
35
36     signature_t signature;
37
38     //public part of signature key
39     pub_key_t pub_signature_key;
40     uint32_t image_length;
41     char* image_code;
42 } rom_ext_manifest_t;
```

Listing 5.2: Model of a `ROM_EXT` manifest in `mask_rom.h`.

We also believe the representations of the boot policy and `ROM_EXT` manifest to be reasonable, based on the documented content of the boot policy and `ROM_EXT` manifest (cf. Section 4.1 and [41], [38]).

5.2 Constraints and Assumptions

This section covers the constraints that apply to the system and the assumptions made to facilitate the verification process.

System Constraints

- There is a bound to the maximum size of the `ROM_EXT` manifest. All fields are of a fixed size except the image code [38].
- The number of `ROM_EXTs` to try and boot from are bounded.

Assumptions to Facilitate Verification

- No physical attacks to the `mask_ROM` boot code.
  - *Why is it necessary*: If we model for all possible physical attacks to the `mask_ROM` boot code then the number of program paths increase exponentially rendering verification as computationally impossible. Another reason not to model all attacks is that this chapter aims to verify if the boot code even adheres to the security properties by default.
  - *What to make it not hold*: Perform any physical attack that tampers with the memory where the `mask_ROM` boot code is stored. This form of attack is looked into in Chapter 6.

[1] In Listings that show the source code, the line numbers will match the source code.

  - *What if it does not hold*: Then malicious code can be executed.

- The `image_length` field in the `ROM_EXT` manifest corresponds to the actual image size.
  - *Why is it necessary*: This is what the boot code expects. The boot code cannot verify the correctness of the image length variable due to how the manifest is designed.
  - *What to make it not hold*: An error by the software writing the `ROM_EXT` manifest or by the people developing the `ROM_EXT` manifest. A physical attack could also alter it.
  - *What if it does not hold*: It could lead to memory errors, invalidation of a valid `ROM_EXT`, or execution of unverified code. We investigate this further in Section 6.6.

- The `image_length` field in the `ROM_EXT` manifest is assumed to be at max 10 and greater than 0.
  - *Why is it necessary*: The assumption that the `image_length` is maximum 10 and greater than 0 is necessary for decreasing the state space to facilitate faster verification.
  - *What to make it not hold*: Perform a physical attack or flash a `ROM_EXT` manifest with an `image_length` greater than 10 or less than 0.
  - *What if it does not hold*: Then the verification will require unroll loop bounds equal to the maximum positive value of an integer (because we do `memcmp` and `memcpy` of the image code), which is computationally impossible.

- The `ROM_EXT` image code always returns execution to `mask_ROM`.
  - *Why is it necessary*: This assumption is necessary to enable CBMC to terminate verification.
  - *What to make it not hold*: A `ROM_EXT` only returns if there occurs an error while booting the `BL0` or any subsequent stage. So by default, a successful boot sequence will not comply with this assumption. It can also be broken by creating a `ROM_EXT` image code that does not return or by performing a physical attack that overwrites the return address of the `ROM_EXT` image code.
  - *What if it does not hold*: Then CBMC will never terminate verification and the `ROM_EXT` will run forever.

5.3 Proof Harness

We have one proof harness, called `PROOF_HARNESS`, for verifying all the properties. The properties can be verified using a mocked SHA-256 and with an implementation of a SHA-256. Using an implementation of a SHA-256 is only necessary for properties 3 and 4 and not for the rest. It should be noted that the verification time is significantly increased when using the implementation of SHA-256. Therefore, when verifying the other properties, a mocked SHA-256 is used to decrease verification time. A call graph for `PROOF_HARNESS` can be seen in Fig. 5.3.1. Most functions are decorated with CBMC annotations. Also note that `FLASH_CTRL_boot_policy` and `FLASH_CTRL_rom_ext_manifests_to_try` is moved outside of `mask_rom_boot`. This makes the boot code deviate from the boot code in Section 4.2 but will not affect the verification as the call order and supplied arguments are equivalent. It is necessary because we make CBMC assumptions about the policy and manifests as part of our setup and we want to, if possible, move assumptions into the proof harness, as described in Section 2.6.

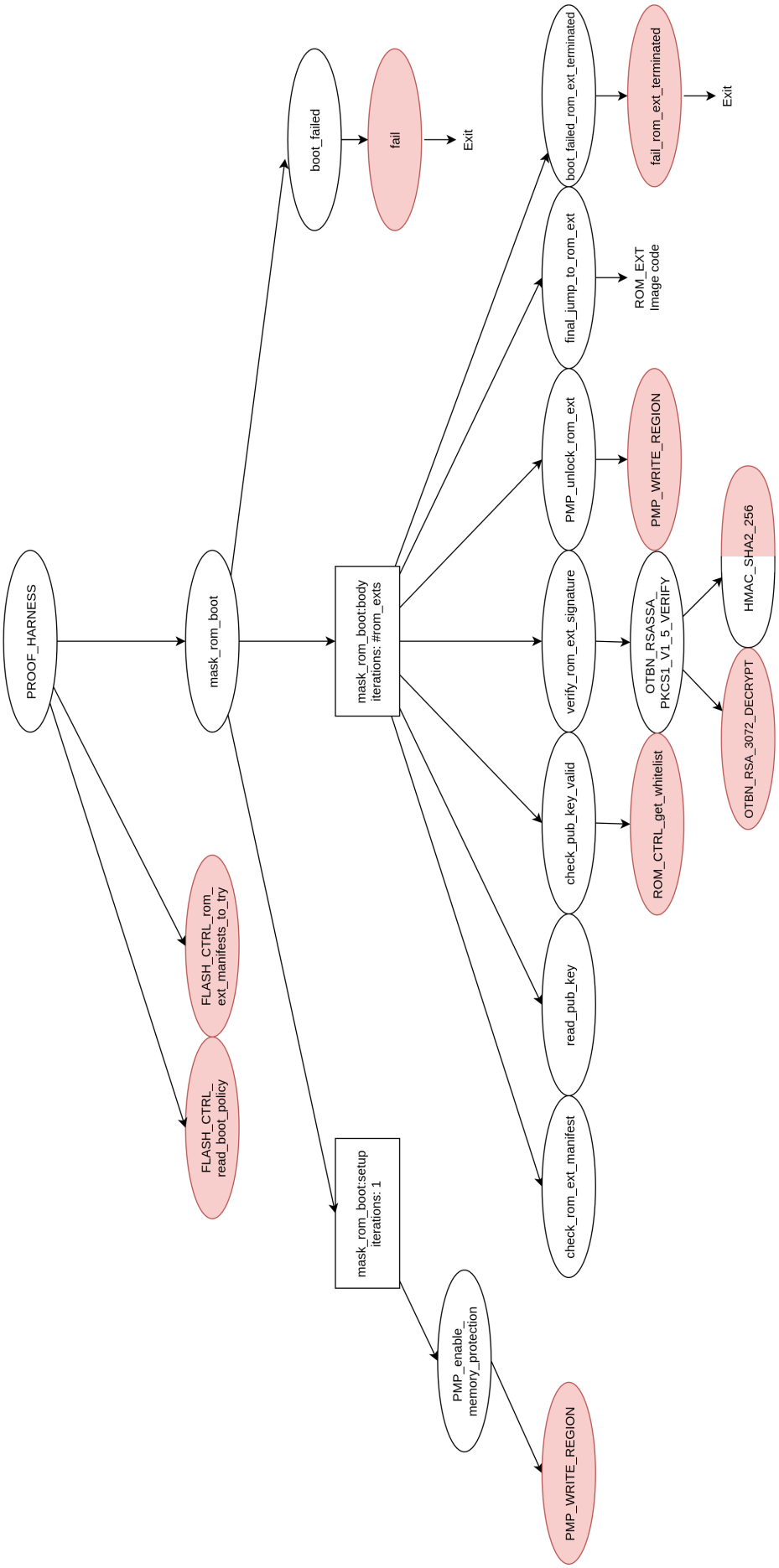

Figure 5.3.1: Call graph of `PROOF_HARNESS` for property 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10.

Note: The boxes do not indicate function calls. Red indicates that the function is modeled.

The C code for the `PROOF_HARNESS` can be seen in Listing 5.3. Note that parts of the proof harness have been left out for simplicity (denoted with three dots). The full code can be seen in Appendix C.

The setup part of the `PROOF_HARNESS` can be seen from lines 411 to 423 in Listing 5.3. The call to `mask_rom_boot` is done at line 425. All the code from line 428 to 505 is postconditions. The entire proof harness can be seen on line 411 in Appendix C.2.

```
411 void PROOF_HARNESS() {
412     boot_policy_t boot_policy = FLASH_CTRL_read_boot_policy();
413     rom_exts_manifests_t rom_exts_to_try = FLASH_CTRL_rom_ext_manifests_to_try(boot_policy
        ↪ );
414
415     __CPROVER_assume(rom_exts_to_try.size <= MAX_ROM_EXTS && rom_exts_to_try.size > 0);
416
417     __CPROVER_assume(boot_policy.fail == &__func_fail);
418     __CPROVER_assume(boot_policy.fail_rom_ext_terminated == &__func_fail_rom_ext);
419
420     for(int i = 0; i < rom_exts_to_try.size; i++){
421         __CPROVER_assume(MAX_IMAGE_LENGTH >= rom_exts_to_try.rom_exts_mfs[i].image_length
        ↪ && rom_exts_to_try.rom_exts_mfs[i].image_length > 0);
422         rom_exts_to_try.rom_exts_mfs[i].image_code = malloc(sizeof(char) * rom_exts_to_try
        ↪ .rom_exts_mfs[i].image_length);
423     }
424
425     mask_rom_boot(boot_policy, rom_exts_to_try);
426
427
428     __CPROVER_postcondition(__current_rom_ext + 1 <= rom_exts_to_try.size,
429     "Postcondition: Should never check more rom_ext than there exist");
430
431     for (int i = 0; i < rom_exts_to_try.size; i++) {
432         __CPROVER_postcondition(...);
...          ... //more postconditions
442
443         if (__validated_rom_exts[i]) { //validated - try to boot from
444             __REACHABILITY_CHECK
445
446             __CPROVER_postcondition(...);
...              ... //more postconditions
473         }
474         else { //invalidated - unsafe to boot from
475             __REACHABILITY_CHECK
476
477            __CPROVER_postcondition(...);
...              ... //more postconditions
505         }
506     }
507     __REACHABILITY_CHECK
508 }
```

Listing 5.3: Proof harness for all properties in `mask_rom.c`.

On a laptop with an Intel Core i7-4600U CPU, it takes 25 minutes to verify PROPERTY 0 and 24 minutes to verify PROPERTY 1-10. This verification time is inappropriate for developing and debugging. Therefore, when developing and debugging, we decreased RSA keys and signatures from 3072-bits to 160-bits.

Reachability Checks: We aim at having a reachability check for each branch in the execution. The goal is to ensure that the properties we verify do not just succeed because they are unreachable. The `__REACHABILITY_CHECK` is a macro for a `__CPROVER_assert(0, "message")` statement. This works, as this assertion should always

fail. Thus, if it succeeds, it means that the assertion is not reachable and instead succeeds by default (cf. Section 2.8.2).

Note: All constructs in the code that are prefixed with "__" are only used in relation to CBMC and are not part of the actual boot code. These "__" prefixed constructs are either variables to capture the internal state of the program or helper functions to aid verification.

5.4 PROPERTY 0

"*The* `mask_ROM` *boot code must be free of bugs.*"

5.4.1 Assumptions

- CBMC is, as far as we know, not guaranteed to be sound nor complete. We do, however, assume that CBMC is complete and sound in relation to the built in property checks. This also reflects what we have experienced in practice.

Verification

In order to prove the absence of bugs in the `mask_ROM` boot code we use the following CBMC property generating arguments:

- `--bounds-check`
- `--pointer-check`
- `--memory-leak-check`
- `--div-by-zero-check`
- `--signed-overflow-check`
- `--unsigned-overflow-check`
- `--pointer-overflow-check`
- `--conversion-check`
- `--undefined-shift-check`
- `--float-overflow-check`
- `--nan-check`
- `--enum-range-check`
- `--pointer-primitive-check`
- `--unwinding-assertions`

In total, CBMC generated 655 assertions for the `mask_ROM` boot code, where 1 out of the 655 assertions fails. This is caused by a memory leak of dynamically allocated memory that is never freed. However, the malloc allocation in question is used to model nondeterministic readable memory for CBMC verification and is not part of the boot code itself. As such, the failing assertion can be disregarded.

5.4.2 Results

All of the generated assertions for the boot code pass. This means that:

> CBMC is unable to prove the presence of any bugs in the `mask_ROM` boot code.

Thus the `mask_ROM` boot code is likely to be bug free given our assumptions about CBMC being sound and complete in relation to the built in property checks. This is, however, not guaranteed and cannot be considered proof.

5.5 PROPERTY 1

"The `ROM_EXT` manifest for a `ROM_EXT` must be signed with a RSA-3072 signature. If a `ROM_EXT` manifest for a `ROM_EXT` is unsigned (i.e. the signature is a sequence of zeros) the `ROM_EXT` is considered invalid to boot from."

5.5.1 Assumptions

- A RSA-3072 signature can be any given value as long as it is 3072-bits large.

5.5.2 Verification

Line 326 in `mask_rom.c` :

```
326 int __help_check_rom_ext_manifest(rom_ext_manifest_t manifest) { //used for CBMC assertion
    ↪  + postcondition
327     if (manifest.identifier == 0)
328         return 0;
329
330     signature_t signature = manifest.signature; //needed to take object size of signature
    ↪ and not entire manifest
331
332     if (__CPROVER_OBJECT_SIZE(signature.value) != 3072 / 8) //Signature must be 3072-bits
333         return 0;
334
335     for (int i = 0; i < RSA_SIZE; i++) {
336         if (manifest.signature.value[i] != 0)
337             return 1;
338     }
339     return 0;
340 }
```

The `__check_rom_ext_manifest` function is used in relation to CBMC to assert whether the identifier is nonzero and that the signature is of the correct size and nonzero. Note that the size check is trivial and could easily be done by code review. Nevertheless, we consider it good practice to verify anyway as a change to the code might break this.

Line 560 and 568 in `mask_rom.c` :

```
529 void mask_rom_boot(boot_policy_t boot_policy, rom_exts_manifests_t rom_exts_to_try){
…       ...
546     for (int i = 0; i < rom_exts_to_try.size; i++) {
…           ...
557         if (!check_rom_ext_manifest(__current_rom_ext_manifest)) {
558             __REACHABILITY_CHECK
559
```

```
560             __CPROVER_assert(!__help_check_rom_ext_manifest(current_rom_ext_manifest),
561             "PROPERTY 1: Stop verification if signature or identifier is invalid");
562
563             continue;
564         }
565
566     __REACHABILITY_CHECK
567
568     __CPROVER_assert(__help_check_rom_ext_manifest(current_rom_ext_manifest),
569     "PROPERTY 1: Continue verification if signature and identifier are valid");
...     ...
```

After the call to `check_rom_ext_manifest`, a `__CPROVER_assert`, at line 560, is used to assert that, if the function returns false, the signature or the identifier is invalid. Likewise, the `__CPROVER_assert` at line 568 asserts that if the function returns true, the signature and the identifier are valid. Both of these assertions succeed.

Line 446 in `mask_rom.c`:

```
411 void PROOF_HARNESS(){
...     ...
431     for (int i = 0; i < rom_exts_to_try.size; i++) {
...         ...
443         if (__validated_rom_exts[i]) { //validated - try to boot from
444             __REACHABILITY_CHECK
445
446             __CPROVER_postcondition(__help_check_rom_ext_manifest(rom_exts_to_try.
    ↪ rom_exts_mfs[i]),
447             "Postcondition PROPERTY 1: rom_ext VALIDATED => valid signature");
...             ...
```

The `__CPROVER_postcondition` at line 446 succeeds and asserts that if the `ROM_EXT` was validated, then the signature and identifier of the related `ROM_EXT` manifest must be valid.

5.5.3 Results

The CBMC results state that all of the three user-defined assertions pass. The reachability checks ensure that the assertions are indeed reachable. All of the unwinding assertions pass, meaning that CBMC searches the entire reachable state space. This means that:

- If the `ROM_EXT` signature is valid, it is always exactly 3072-bits.
- Verification stops if the `ROM_EXT` manifest is unsigned (equal to 0).
- Verification stops if the `ROM_EXT` manifest has an invalid identifier.
- Verification continues if the `ROM_EXT` manifest is signed (not equal to 0) and the identifier is valid.

Thus the `mask_ROM` boot code is proven to fully satisfy PROPERTY 1 under the assumptions listed in Section 5.5.1.

5.6 PROPERTY 2

"The public RSA-3072 key used for the signature contained in the `ROM_EXT` manifest, must be valid in order for the `ROM_EXT` to be considered valid to boot from."

5.6.1 Assumptions

- A public key is considered valid if it matches a key in the list of known good keys (whitelist) stored in `mask_ROM`.
- For a public key to be valid, its exponent part must be exactly 32-bits.
- For a public key to be valid, its modulus part must be exactly 3072-bits.

5.6.2 Verification

Line 343 in `mask_rom.c`:

```
343 int __help_pkey_valid(pub_key_t pkey) { //used for CBMC assertion + postcondition
344     // Public key exponent must be 32 bits.");
345     if(sizeof(pkey.exponent) * 8 != 32)
346         return 0;
347     // Public key modulus must be 3072-bits.");
348     if((sizeof(pkey) - sizeof(pkey.exponent)) * 8 != 3072)
349         return 0;
350 
351     pub_key_t* pkey_whitelist = ROM_CTRL_get_whitelist();
352 
353     for (int i = 0; i < __PKEY_WHITELIST_SIZE; i++) {
354         if (pkey_whitelist[i].exponent != pkey.exponent)
355             continue;
356 
357         int j = 0;
358         for (j = 0; j < RSA_SIZE; j++) {
359             if (pkey_whitelist[i].modulus[j] != pkey.modulus[j])
360                 break;
361         }
362 
363         //if j == RSA_SIZE, then loop ran to completion and all entries were equal
364         if (j == RSA_SIZE)
365             return 1;
366     }
367 
368     return 0;
369 }
```

The `__help_pkey_valid` function is used in relation to CBMC to assert whether a public key is valid. Note that at line 348, we exploit the fact that the `pub_key_t` structure only has two members: exponent and modulus. If this struct were to be extended, the expression at line 348 would have to be changed.

Line 578 and 586 in `mask_rom.c`:

```
529 void mask_rom_boot(boot_policy_t boot_policy, rom_exts_manifests_t rom_exts_to_try){
…       ...
546     for (int i = 0; i < rom_exts_to_try.size; i++) {
…           ...
572         pub_key_t rom_ext_pub_key = read_pub_key(current_rom_ext_manifest);
573 
574         //Step 2.iii.b
```

```
575         if (!check_pub_key_valid(rom_ext_pub_key)) {
576             __REACHABILITY_CHECK
577 
578             __CPROVER_assert(!__help_pkey_valid(rom_ext_pub_key),
579             "PROPERTY 2: Stop verification if key is invalid");
580 
581             continue;
582         }
583 
584         __REACHABILITY_CHECK
585 
586         __CPROVER_assert(__help_pkey_valid(rom_ext_pub_key),
587         "PROPERTY 2: Continue verification if key is valid");
...         ...
```

The `check_pub_key_valid` function returns false if the public key does not match a key in the known good key list (whitelist) stored in `mask_ROM`. If `check_pub_key_valid` returns false, it is asserted at line 578 that the public key was indeed invalid. Likewise, if it returns true, it is asserted at line 586 that the public key was valid. Both of these assertions succeed.

Line 449 in `mask_rom.c`:

```
411 void PROOF_HARNESS() {
...     ...
431     for (int i = 0; i < rom_exts_to_try.size; i++) {
...         ...
443         if (__validated_rom_exts[i]) { //validated - try to boot from
444             __REACHABILITY_CHECK
...             ...
449             __CPROVER_postcondition(__help_pkey_valid(rom_exts_to_try.rom_exts_mfs[i].
    ↪ pub_signature_key),
450             "Postcondition PROPERTY 2: rom_ext VALIDATED => valid key");
...             ...
```

As part of the `PROOF_HARNESS`, there is a `__CPROVER_postcondition` at line 449 that succeeds and asserts that if a `ROM_EXT` is validated, then the public key in the respective `ROM_EXT` manifest must be valid.

5.6.3 Results

The CBMC results state that all of the three user-defined assertions pass. The reachability checks ensure that the assertions are indeed reachable. All of the unwinding assertions pass, meaning that CBMC searches the entire reachable state space. This means that:

- If the `ROM_EXT` was validated, the public key exponent is exactly 32-bits.
- If the `ROM_EXT` was validated, the public key modulus is exactly 3072-bits.
- Verification stops if the public key in the `ROM_EXT` manifest is invalid.
- Verification continues if the public key in the `ROM_EXT` manifest is valid.
- If a `ROM_EXT` is fully validated, then the associated `ROM_EXT` manifest contains a valid public key.

Thus the `mask_ROM` boot code is proven to fully satisfy PROPERTY 2 under the assumptions listed in Section 5.6.1.

5.7 PROPERTY 3

"The HMAC hash must be calculated by either a SHA2-256, SHA3-256, SHA3-384, or SHA3-512 hash function."

5.7.1 Assumptions

- It is unclear how the hash in OpenTitan is implemented. For simplicity, we assume that a SHA2-256 hash function must always be used to calculate the HMAC hash.

5.7.2 Verification

It is not possible to assert that the correct hash is calculated for each input. The reason is that a mapping of input to output of the SHA2-256 function does not exist and verifying for such a mapping would be computationally impossible. However, it is possible to assert properties of the input and output. Verifying that the correct hash function is used should, however, be done through a code review.

Line 36 and 39 in `mock_hmac.c`. Equivalent assertions are also present in `hmac.c`:

```
5  BYTE* HMAC_SHA2_256(BYTE key[], BYTE mes[], int size, rom_ext_manifest_t
       ↪ __current_rom_ext_mf) {
…      ...
34     char* hash = malloc(256 / 8); //model it to be ok for PROPERTY 5
35
36     __CPROVER_assert(__CPROVER_OBJECT_SIZE(hash) == 256 / 8,
37         "PROPERTY 3: Hash is 256 bits");
38
39     __CPROVER_assert(__CPROVER_r_ok(hash, 256 / 8),
40         "PROPERTY 3: hash is in readable address");
41
42     __REACHABILITY_CHECK
43
44     return hash;
45 }
```

The `HMAC_SHA2_256` is responsible for calculating the hash of a message using external functionality. The `__CPROVER_assert` on line 36 verifies that the hash object returned from the `HMAC_SHA2_256` function is 256-bits. The `__CPROVER_assert` on line 39 verifies that the hash is in a readable address space. Both of the assertions succeed.

Line 176 and 179 in `mask_rom.c`:

```
130 int OTBN_RSASSA_PKCS1_V1_5_VERIFY(int32_t exponent, int32_t* modulus, char* message, int
       ↪ message_len, int32_t* signature, int signature_len, rom_ext_manifest_t
       ↪ __current_rom_ext_mf) {
…      ...
161    char* decrypt = OTBN_RSA_3072_DECRYPT(signature, signature_len, exponent, modulus);
162    char* hash = HMAC_SHA2_256(__hmac_key, message, message_len, __current_rom_ext_mf); //
       ↪ message_len in bytes
…      ...
176    __CPROVER_assert(__CPROVER_OBJECT_SIZE(hash) == 256 / 8,
177        "PROPERTY 3: Hash is 256 bits");
178
179    __CPROVER_assert(__CPROVER_r_ok(hash, 256 / 8),
180        "PROPERTY 3: hash is in readable address");
…      ...
```

The `OTBN_RSASSA_PKCS1_V1_5_VERIFY` is responsible for verifying a signature. It uses the `HMAC_SHA2_256` internally to compute the hash. Thus, lines 176 and 179 assert that the output of `HMAC_SHA2_256` is 256-bits and readable. Both of these assertions succeed.

5.7.3 Results

The CBMC results state that the four user-defined assertions pass. The reachability checks ensure that the assertions are indeed reachable. All of the unwinding assertions pass, meaning that CBMC searches the entire reachable state space. This means that:

- The HMAC hash computed by the `HMAC_SHA2_256` function is always exactly 256 bits.
- The HMAC hash computed by the `HMAC_SHA2_256` function is always in a readable address space.
- The HMAC hash returned by the `HMAC_SHA2_256` function is always exactly 256 bits.
- The HMAC hash returned by the `HMAC_SHA2_256` function is always in a readable address space.

There is no guarantee that the mapping from input to SHA2-256 output is correct. The mapping is not possible to verify using CBMC verification as this is computationally impossible. Note that this is not an issue for this property as the system under consideration is specific to how the boot code interfaces with the HMAC and not to the correctness of the modeled external hardware module.

Thus the `mask_ROM` boot code is proven to fully satisfy PROPERTY 3 under the assumptions listed in Section 5.7.1.

5.8 PROPERTY 4

"*The computed HMAC hash message must be calculated from* `system_state_value || device_usage_value || signed_area(rom_ext)` *[38].*"

5.8.1 Assumptions

- The HMAC is simply computed by hashing the appended HMAC key and message. There is no intricate padding scheme as is common in HMAC implementations. This is for simplicity.
- The message to be hashed will only be computed from the `signed_area(rom_ext)` and will not include `system_state_value` and `device_usage_value`.
- The `signed_area(rom_ext)` is restricted to only include the `public key`, `image_length`, and `image_code` for simplicity.

5.8.2 Verification

Line 10, 16, 22, 28, and 31 in `mock_hmac.c`. Equivalent assertions are present in `hmac.c`:

```
5 BYTE* HMAC_SHA2_256(BYTE key[], BYTE mes[], int size, rom_ext_manifest_t
      ↪ __current_rom_ext_mf) {
6
7     int __expected_size =
8         sizeof(__current_rom_ext_mf.pub_signature_key) + sizeof(__current_rom_ext_mf.
      ↪ image_length) + __current_rom_ext_mf.image_length;
```

```
9
10      __CPROVER_assert(cmp_key(
11          mes,
12          &__current_rom_ext_mf.pub_signature_key,
13          sizeof(__current_rom_ext_mf.pub_signature_key)) == 0,
14          "PROPERTY 4: Message contains the key");
15
16      __CPROVER_assert(cmp_image_len(
17          mes + sizeof(__current_rom_ext_mf.pub_signature_key),
18          &__current_rom_ext_mf.image_length,
19          sizeof(__current_rom_ext_mf.image_length)) == 0,
20          "PROPERTY 4: Message contains the Image length");
21
22      __CPROVER_assert(cmp_image_code(
23          mes + sizeof(__current_rom_ext_mf.pub_signature_key) + sizeof(__current_rom_ext_mf
    ↪ .image_length),
24          __current_rom_ext_mf.image_code,
25          __current_rom_ext_mf.image_length) == 0,
26          "PROPERTY 4: Message contains the Image code");
27
28      __CPROVER_assert(size == __expected_size,
29          "PROPERTY 4: Message size parameter is as expected.");
30
31      __CPROVER_assert(__CPROVER_OBJECT_SIZE(mes) == __expected_size,
32          "PROPERTY 4: Size of message is as expected.");
33
34      char* hash = malloc(256 / 8); //model it to be ok for PROPERTY 5
...     ...
```

The assertions essentially verify that the external `HMAC_SHA2_256` function is only called with an input message that constitutes the respective public key, image length, and image code. All of these five assertions succeed. The `__CPROVER_assert` at line 10 verifies that the first `n` bytes of the message (with `n` being the length of the key) is equal to the public key in the manifest. At line 16, the `__CPROVER_assert` verifies that the subsequent `n` bytes of the message (where `n` is the size of the `image_length` value type) is equal to the `image_length` field in the manifest. At line 22, the `__CPROVER_assert` verifies that the next subsequent `n` bytes of the message (where `n` is the length of the image code) is equal to the image code in the manifest. The `__CPROVER_assert` at line 28 verifies that the size provided in the formal parameter is equal to the expected size. At line 31, the `__CPROVER_assert` verifies that the size of the message object pointed to is of equal size to the expected size.

5.8.3 Results

The CBMC results state that all of the five user-defined assertions pass. The reachability checks ensure that the assertions are indeed reachable. All of the unwinding assertions pass, meaning that CBMC searches the entire reachable state space. This means that:

- The formal parameter `mes` that the hash is computed from is an appended string with the first element always being the public key exponent and modulus part.
- The second element of the appended `mes` is always the image length.
- The third element of the appended `mes` is always the image code.
- The formal parameter `size` corresponds exactly to the size of the given message and the expected size.

Thus the `mask_ROM` boot code is proven to fully satisfy PROPERTY 4 under the assumptions listed in Section 5.8.1.

5.9 PROPERTY 5

"The signature in the `ROM_EXT` manifest must be validated using the `RSASSA-PKCS1-V1_5-VERIFY` [47] function with inputs: public RSA-3072 key, appended message (`system_state_value || device_usage_value || signed_area(rom_ext)`), and RSA-3072 signature. If the function returns false the `ROM_EXT` is invalid to boot from."

5.9.1 SUBPROPERTY 5.1

The signature in the `ROM_EXT` manifest must be validated using the `RSASSA-PKCS1-V1_5-VERIFY` [47] function.

Assumptions

- The `RSASSA-PKCS1-V1_5-VERIFY` function follows the alternative implementation specified at [47] which verifies by first decoding the signature and then compares the decoded to a freshly computed hash.
- The `RSASSA-PKCS1-V1_5-VERIFY` decoding only decrypts the signature and does not do any unpadding or other decoding operations. This is done for simplicity.
- The nondeterministic signature is decrypted nondeterministically by a modeled `RSA-3072-DECRYPT` function. A `RSA-3072-DECRYPT` function is out of scope for this project and infeasible to verify using CBMC.
- The HMAC hash is calculated from a modeled SHA2-256 hash function. We choose a modeled version because it reduces verification time and in this particular case, it does not change our verification results as the `RSA-3072-DECRYPT` is also mocked.

Verification

The main function being verified in relation to PROPERTY 5 is the `verify_rom_ext_signature` function. It is not possible to assert that the verification is performed using a correct implementation of `RSASSA-PKCS1-V1_5-VERIFY`. However, it is possible to assert that the signature verification function is called when it should be. Through a code review, it would also be possible to verify that it is the correct implementation of `RSASSA-PKCS1-V1_5-VERIFY` that is called.

Line 48 in `mask_rom.c`:

```
48 int verify_rom_ext_signature(pub_key_t rom_ext_pub_key, rom_ext_manifest_t manifest) {
...     ...
55      __verify_signature_called[__current_rom_ext] = 1;
...     ...
81      int result = OTBN_RSASSA_PKCS1_V1_5_VERIFY(rom_ext_pub_key.exponent, rom_ext_pub_key.
        ↪ modulus, message, bytes, signature.value, RSA_SIZE, manifest);
82
83      return result; //0 or 1
84 }
```

The `verify_rom_ext_signature` function prepares the input to the `OTBN_RSASSA_PKCS1_V1_5_VERIFY` and calls it on line 81. In addition, `__verify_signature_called` registers that `verify_rom_ext_signature` is called while verifying the given `ROM_EXT` manifest on line 55. `__verify_signature_called` is an array of size `MAX_ROM_EXTS`, it documents whether the `verify_rom_ext_signature` function was called from

`mask_ROM` when validating the ith `ROM_EXT`. An array is necessary to allow us to track, for each `ROM_EXT`, if the `verify_rom_ext_signature` function was called or not while `mask_ROM` processed the given `ROM_EXT`.

Line 164, 167, 170, and 173 in `mask_rom.c`

```
130 int OTBN_RSASSA_PKCS1_V1_5_VERIFY(int32_t exponent, int32_t* modulus, char* message, int
        ↪ message_len, int32_t* signature, int signature_len, rom_ext_manifest_t
        ↪ __current_rom_ext_mf) {
...     ...
161     char* decrypt = OTBN_RSA_3072_DECRYPT(signature, signature_len, exponent, modulus);
162     char* hash = HMAC_SHA2_256(__hmac_key, message, message_len, __current_rom_ext_mf); //
        ↪ message_len in bytes
163
164     __CPROVER_assert(!__CPROVER_array_equal(decrypt, signature),
165         "PROPERTY 5: Decrypted signature is different from signature");
166
167     __CPROVER_assert(!__CPROVER_array_equal(hash, message),
168         "PROPERTY 5: Hash is different from original message");
169
170     __CPROVER_assert(__CPROVER_OBJECT_SIZE(decrypt) == 256 / 8,
171         "PROPERTY 5: Decrypted message is 256 bits");
172
173     __CPROVER_assert(__CPROVER_r_ok(decrypt, 256 / 8),
174         "PROPERTY 5: Decrypted message is in readable address");
...     ...
```

The `__CPROVER_assert` on line 164 succeeds and asserts that the RSA decryption algorithm used by `OTBN_RSA SSA_PKCS1_V1_5_VERIFY` returns something different from the decrypted signature. The `__CPROVER_assert` on line 167 succeeds and asserts that the used hash algorithm returns something different from the message being hashed. Lastly, the assertions on lines 170 and 173 succeed and assert, respectively, that the decrypted signature is 256-bits and in a readable address space.

Line 433, 438, and 452 in `mask_rom.c`:

```
411 void PROOF_HARNESS()
...     ...
431     for (int i = 0; i < rom_exts_to_try.size; i++) {
432
433         __CPROVER_postcondition(__imply(!__help_check_rom_ext_manifest(rom_exts_to_try.
        ↪ rom_exts_mfs[i]) ||
434                                 !__help_pkey_valid(rom_exts_to_try.rom_exts_mfs[i].
        ↪ pub_signature_key),
435                                 !__verify_signature_called[i]),
436         "Postcondition PROPERTY 5: If identifier, sign, or key is invalid then verify
        ↪ signature function is not called");
437
438         __CPROVER_postcondition(__imply(__help_check_rom_ext_manifest(rom_exts_to_try.
        ↪ rom_exts_mfs[i]) &&
439                                 __help_pkey_valid(rom_exts_to_try.rom_exts_mfs[i].
        ↪ pub_signature_key),
440                                 __verify_signature_called[i]),
441         "Postcondition PROPERTY 5: If identifier, sign, and key are valid then the
        ↪ signature verification function is called");
442
443         if (__validated_rom_exts[i]) { //validated - try to boot from
444             __REACHABILITY_CHECK
...             ...
452             __CPROVER_postcondition(__verify_signature_called[i],
453             "Postcondition PROPERTY 5: iff sign and key is valid then verify signature
        ↪ function is called");
```

... ...

The postconditions on lines 433 and 438 succeed and are checked for all `ROM_EXT` manifests. The `__CRPOVER_postcondition` on line 433 asserts that the signature verification function is not called if the identifier, signature, or key in the manifest are invalid. The `__CPROVER_postcondition` on line 438 asserts that the signature verification function should have been called if the `ROM_EXT` manifest has a valid identifier, signature, and key. The `__CPROVER_postcondition` on line 452 succeeds and asserts that the signature verification function was called if the `ROM_EXT` manifest was fully validated.

5.9.2 SUBPROPERTY 5.2

The input to the `RSASSA-PKCS1-V1_5-VERIFY` [47] function must be: public RSA-3072 key, appended message (`system_state_value || device_usage_value || signed_area(rom_ext)`), and RSA-3072 signature.

Assumptions

- A RSA-3072 key is a pair of a 32-bit exponent and a 3072-bit modulus.
- A RSA-3072 signature can be any given value as long as it is 3072-bits large.
- The signature in the `ROM_EXT` manifest computed by the Silicon Creator is only computed from the `signed_area(rom_ext)` and does not include `system_state_value` and `device_usage_value`.
- The HMAC hash of the `ROM_EXT` manifest will only be computed from the `signed_area(rom_ext)`.
- The `signed_area(rom_ext)` is restricted to only include the `public key`, `image_length`, and `image_code` for simplicity.

Verification

Line 131, 134, 137, 140, 143, and 146 in `mask_rom.c`:

```
130 int OTBN_RSASSA_PKCS1_V1_5_VERIFY(int32_t exponent, int32_t* modulus, char* message, int
        ↪ message_len, int32_t* signature, int signature_len, rom_ext_manifest_t
        ↪ __current_rom_ext_mf) {
131     __CPROVER_assert(__CPROVER_OBJECT_SIZE(message) == message_len,
132         "PROPERTY 5: Formal parameter message_len lenght matches actual message length.");
133
134     __CPROVER_assert(__CPROVER_OBJECT_SIZE(signature) == 3072 / 8,
135         "PROPERTY 5: Signature to be verified is 3072-bits.");
136
137     __CPROVER_assert(__CPROVER_OBJECT_SIZE(signature) == signature_len * sizeof(int32_t),
138         "PROPERTY 5: Formal parameter signature lenght matches actual signature length.");
139
140     __CPROVER_assert(sizeof(exponent) == 32 / 8,
141         "PROPERTY 5: Public key exponent is 32 bits.");
142
143     __CPROVER_assert((sizeof(pub_key_t) - sizeof(exponent)) == 3072 / 8,
144         "PROPERTY 5: Public key modulus is 3072-bits.");
145
146     __CPROVER_assert(__is_valid_params(exponent, modulus, message, message_len, signature,
147         signature_len, __current_rom_ext_mf),
148         "PROPERTY 5: Check that key, signature, and message matches those from the
        ↪ manifest.");
149
150     __REACHABILITY_CHECK
...     ...
```

The assertions on lines 131, 134, 137, 140, and 143 succeed and assert that the parameters match the expected size. The `__CPROVER_assert` on line 146 succeeds and asserts that `RSASSA-PKCS1-V1_5-VERIFY` is called with the correct parameters. This is done by checking that the parameters match the content of the manifest that is currently being processed. The current manifest is accessible since it is passed through the parameter named `__current_rom_ext_mf`. As the parameter is only used for CBMC purposes, we evaluate that this additional parameter does not alter the validity of the `RSASSA-PKCS1-V1_5-VERIFY` function.

5.9.3 SUBPROPERTY 5.3

If the `RSASSA-PKCS1-V1_5-VERIFY` returns false the `ROM_EXT` is invalid to boot from.

Assumptions

- The `RSASSA-PKCS1-V1_5-VERIFY` function follows the alternative implementation specified at [47].

Verification

Line 482 in `mask_rom.c`:

```
411 void PROOF_HARNESS() {
...     ...
431     for (int i = 0; i < rom_exts_to_try.size; i++) {
...         ...
443         if (__validated_rom_exts[i]) { //validated - try to boot from
...             ...
473         }
474         else { //invalidated - unsafe to boot from
475             __REACHABILITY_CHECK
...             ...
482             __CPROVER_postcondition(!__valid_signature[i],
483             "Postcondition PROPERTY 5: rom_ext INVALIDATED => signature invalid or not
    ↪ checked");
...             ...
```

The `__CPROVER_postcondition` on line 482 succeeds. It asserts that if a `ROM_EXT` is invalidated, then either the signature is invalid or some validation check before the signature check failed.

5.9.4 Results

The CBMC results state that all of the 14 user-defined assertions pass. The reachability checks ensure that the assertions are indeed reachable. All of the unwinding assertions pass, meaning that CBMC searches the entire reachable state space. This means that:

- The `OTBN_RSA_3072_DECRYPT` function returns a 256-bit value in a readable address space that is different from the value being decrypted.
- The `HMAC_SHA2_256` function returns a 256-bit hash that is different from the value being hashed.
- If the `ROM_EXT` manifest has a valid identifier, key, and signature, the `verify_rom_ext_signature` function is called.
- If the `ROM_EXT` manifest has an invalid identifier, key, or signature, the `verify_rom_ext_signature` function is **not** called.

If a `ROM_EXT` manifest is successfully validated, the `verify_rom_ext_signature` function is called.

If a `ROM_EXT` manifest is invalidated, the `verify_rom_ext_signature` function is **not** called.

The `OTBN_RSASSA_PKCS1_V1_5_VERIFY` function's parameters are always of the expected sizes.

The `OTBN_RSASSA_PKCS1_V1_5_VERIFY` function's parameters are always the expected values. I.e. the parameters correspond correctly to the current `ROM_EXT` manifest.

The mapping from input to `RSASSA-PKCS1-V1_5-VERIFY` output is not guaranteed to be correct. We will not determine the correctness of the mapping using verification as this is impossible with a mocked RSA-3072 decrypt function. Even if we used a real implementation of RSA-3072 decrypt, we believe it would be computationally impossible to verify the correctness of the `RSASSA-PKCS1-V1_5-VERIFY` output. Note that this is not an issue for this property as the system under consideration is specific to how the boot code interfaces with the `RSASSA-PKCS1-V1_5-VERIFY` function and not to the correctness of this modeled external hardware module.

Thus the `mask_ROM` boot code is proven to fully satisfy PROPERTY 5, as it satisfies SUBPROPERTY 5.1, SUBPROPERTY 5.2, and SUBPROPERTY 5.3 under the associated assumptions.

5.10 PROPERTY 6

"If all validation steps have succeeded then transfer execution to `ROM_EXT` by starting execution at the entry point of the `ROM_EXT` image code. If execution returns, execute the `fail_rom_ext_returned` function provided by the boot policy."

5.10.1 PROPERTY 6.1

If all validation steps have succeeded then transfer execution to `ROM_EXT` by starting execution at the entry point of the `ROM_EXT` image code.

Assumptions

- A `ROM_EXT` is valid if the associated `ROM_EXT` manifest contains an identifier that is nonzero, a valid signature, a valid public key, and if the verification of the signature succeeds using the `RSASSA-PKCS1-V1_5-VERIFY` function.
- The entry point of the `ROM_EXT` image code is equal to the `__some_entry_func` function.
- `__some_entry_func` is a reasonable abstraction of an actual `ROM_EXT` entry point function considering that the execution of the `ROM_EXT` is out of scope of this project.

Verification

Line 286 in `mask_rom.c` :

```
276 void __some_entry_func() { __rom_ext_called[__current_rom_ext] = 1; /*for CBMC PROPERTY 6
      ↪ */ }
277
278
279 int final_jump_to_rom_ext(rom_ext_manifest_t current_rom_ext_manifest) { // Returns a
      ↪ boolean value.
...     ...
```

```
282     current_rom_ext_manifest.image_code = &__some_entry_func;
283     //Execute rom ext code step 2.iii.e
284     rom_ext_boot_func* rom_ext_entry = (rom_ext_boot_func*)current_rom_ext_manifest.
        ↪ image_code;
285
286     __CPROVER_assert(rom_ext_entry == current_rom_ext_manifest.image_code,
287     "PROPERTY 6: Correct entry point address.");
288
289     __REACHABILITY_CHECK
290
291     rom_ext_entry();
292
293     __rom_ext_returned[__current_rom_ext] = 1; //for CBMC PROPERTY 6
294
295     //if rom_ext returns, we should return false
296     //and execute step 2.iv.
297     return 0;
298 }
```

The `__CPROVER_assert` at line 286 succeeds and asserts that the entry point function is equivalent to the entry point of the `ROM_EXT` image code. In this case, it is always the `__some_entry_func` that is called. The `__rom_ext_called` array is used to register if the `ROM_EXT` is called. In this case, `__some_entry_func` is needed to model a potential entry point function for `ROM_EXT`, otherwise, CBMC will not accept line 284 as a valid operation and the verification cannot register if the `ROM_EXT` is called.

Line 458 in `mask_rom.c`:

```
411 void PROOF_HARNESS() {
...     ...
431     for (int i = 0; i < rom_exts_to_try.size; i++) {
...         ...
443         if (__validated_rom_exts[i]) { //validated - try to boot from
444             __REACHABILITY_CHECK
...             ...
458             __CPROVER_postcondition(__rom_ext_called[i],
459             "Postcondition PROPERTY 6: rom_ext VALIDATED => rom ext code inititated");
...             ...
```

The `__CPROVER_postcondition` at line 458 succeeds and asserts that if the `ROM_EXT` was validated, then it was also called.

5.10.2 PROPERTY 6.2

If execution returns, execute the `fail_rom_ext_returned` function provided by the boot policy.

Assumptions

- The image code for the `ROM_EXT` always returns. Otherwise, CBMC is not able to verify the program.
- The `fail_rom_ext_returned` function in the boot policy is equal to the `__func_fail_rom_ext`.
- The `__func_fail_rom_ext` function is a reasonable abstraction of the `fail_rom_ext_returned` function.

```
void __func_fail_rom_ext(rom_ext_manifest_t _) {
    __rom_ext_fail_func[__current_rom_ext] = 1;
}
```

Verification

Line 461 and 464 in `mask_rom.c`:

```
411 void PROOF_HARNESS() {
…       ...
431     for (int i = 0; i < rom_exts_to_try.size; i++) {
…           ...
443         if (__validated_rom_exts[i]) { //validated - try to boot from
444             __REACHABILITY_CHECK
…               ...
461             __CPROVER_postcondition(__imply(__rom_ext_returned[i], __rom_ext_fail_func[i])
        ↪ ,
462             "Postcondition PROPERTY 6: (valid rom _ext and rom_ext code return) => that
        ↪ rom_ext term func is called");
463
464             __CPROVER_postcondition(__imply(!__rom_ext_returned[i], !__rom_ext_fail_func[i
        ↪ ]),
465                 "Postcondition PROPERTY 6: (valid rom _ext and rom_ext code !return) =>
        ↪ that rom_ext term func not called");
…               ...
```

The `__CPROVER_postcondition` at line 461 succeeds and asserts that if the `ROM_EXT` was validated, called, and returned, the `fail_rom_ext_terminated` function from the boot policy should have been called. The `__CPROVER_postcondition` at line 464 succeeds and asserts that if the `ROM_EXT` was validated and did not return, then the fail function should not have been called. However, note that in the current model of the boot code, the `ROM_EXT` is modeled to always return. This means that despite this postcondition succeeding, no guarantees can be made about the behavior that should happen if `ROM_EXT` does not return. The postcondition succeeds by default as `!__rom_ext_returned[i]` is false for all states.

Line 485 in `mask_rom.c`:

```
411 void PROOF_HARNESS() {
…       ...
431     for (int i = 0; i < rom_exts_to_try.size; i++) {
…           ...
443         if (__validated_rom_exts[i]) { //validated - try to boot from
…               ...
473         }
474         else { //invalidated - unsafe to boot from
475             __REACHABILITY_CHECK
…               ...
485             __CPROVER_postcondition(!__rom_ext_fail_func[i],
486             "Postcondition PROPERTY 6: invalid rom_ext => that rom_ext term func not
        ↪ called");
…               ...
```

The `__CPROVER_postcondition` at line 485 succeeds and asserts that if the `ROM_EXT` was not validated then the `fail_rom_ext_terminated` function from the boot policy should also not have been called.

5.10.3 Results

The CBMC results state that all of the five user-defined assertions succeed. The reachability checks ensure that the assertions are indeed reachable. All of the unwinding assertions pass, meaning that CBMC searches the entire reachable state space. This means that:

> If the `ROM_EXT` is successfully validated, in terms of signature and public key, then execution is transferred from `mask_ROM` to `ROM_EXT`.

The correct entry point function is called when executing the `ROM_EXT`.

If a `ROM_EXT` is invalidated then the `fail_rom_ext_terminated` function is not called for the respective `ROM_EXT`.

The `fail_rom_ext_terminated` function in the boot policy is called when a validated `ROM_EXT` returns after being called.

Thus the `mask_ROM` boot code is proven to fully satisfy PROPERTY 6, as it satisfies SUBPROPERTY 6.1 and SUBPROPERTY 6.2 under the associated assumptions.

5.11 PROPERTY 7

"*If at any point a `ROM_EXT` is invalidated the `ROM_EXT` is considered unsafe to boot from and the `mask_ROM` must proceed to validate the next `ROM_EXT`.*"

5.11.1 SUBPROPERTY 7.1

If at any point a `ROM_EXT` is invalidated the `ROM_EXT` is considered unsafe to boot.

Assumptions

- A `ROM_EXT` is valid if the associated `ROM_EXT` manifest contains an identifier that is nonzero, a valid signature, a valid public key, and if the verification of the signature succeeds using the `RSASSA-PKCS1-V1_5-VERIFY` function.

Verification

Line 488 in `mask_rom.c`:

```
411 void PROOF_HARNESS() {
... ...
431     for (int i = 0; i < rom_exts_to_try.size; i++) {
...         ...
443         if (__validated_rom_exts[i]) { //validated - try to boot from
...             ...
473         }
474         else { //invalidated - unsafe to boot from
475             __REACHABILITY_CHECK
...             ...
488             __CPROVER_postcondition(!__rom_ext_called[i],
489             "Postcondition PROPERTY 7: rom_ext INVALIDATED => rom ext code not executed");
...             ...
```

The `__CPROVER_postcondition` at line 488 succeeds and asserts that if the `ROM_EXT` was invalid, then the `ROM_EXT` was not executed.

5.11.2 SUBPROPERTY 7.2

If at any point a `ROM_EXT` is invalidated the `mask_ROM` must proceed to validate the next `ROM_EXT`.

Assumptions

- A `ROM_EXT` is valid if the associated `ROM_EXT` manifest contains an identifier that is nonzero, a valid signature, a valid public key, and if the verification of the signature succeeds using the `RSASSA-PKCS1-V1_5-VERIFY` function.
- If a `ROM_EXT` is validated, executed, and then returns, `mask_ROM` does not try to validate and execute another `ROM_EXT`. Instead `mask_ROM` executes the `fail_rom_ext_terminated` function in the boot policy and then terminates.

Verification

Line 491 in `mask_rom.c`:

```
411 void PROOF_HARNESS() {
...     ...
431     for (int i = 0; i < rom_exts_to_try.size; i++) {
...         ...
443         if (__validated_rom_exts[i]) { //validated - try to boot from
...             ...
473         }
474         else { //invalidated - unsafe to boot from
475             __REACHABILITY_CHECK
...             ...
491             __CPROVER_postcondition(__current_rom_ext > i || (i + 1) == rom_exts_to_try.
    ↪ size || __boot_policy_stop,
492             "Postcondition PROPERTY 7: rom_ext INVALIDATED => we check the next rom_ext if
    ↪  any left and no boot policy instructed stop");
...             ...
```

The `__current_rom_ext` is the `ROM_EXT` that was last tried to be validated by `mask_ROM`. The `__boot_policy_stop` (set at line 614 in `mask_rom.c`) is a Boolean indicating whether a `ROM_EXT` was validated, called, and then returned, resulting in `mask_ROM` returning as well.

The `__CPROVER_postcondition` succeeds and asserts that if the ith `ROM_EXT` was invalid, then `mask_ROM` tried to validate a later `ROM_EXT` (this is the first clause). However, `mask_ROM` did not have to try and validate a later `ROM_EXT` if all `ROM_EXT` were invalid (i.e. the ith `ROM_EXT` is the last and is invalid) since that is not possible (this is the second clause). But `mask_ROM` should also not continue and try to validate a later `ROM_EXT` if some earlier `ROM_EXT` was validated, called, and returned since `mask_ROM` should then return (this is the last clause).

5.11.3 Results

The CBMC results state that the two user-defined assertions succeed. The reachability checks ensure that the assertions are indeed reachable. All of the unwinding assertions pass, meaning that CBMC searches the entire reachable state space. This means that:

> An invalidated `ROM_EXT` is never called.

> `mask_ROM` will proceed to validate the next `ROM_EXT` if the current is invalid and it is not the last `ROM_EXT`.

> `mask_ROM` does not try to validate the next `ROM_EXT` if the current was validated and called. Even if the `ROM_EXT` returns.

Thus the `mask_ROM` boot code is proven to fully satisfy PROPERTY 7, as it satisfies SUBPROPERTY 7.1 and SUBPROPERTY 7.2 under the associated assumptions.

5.12 PROPERTY 8

"If validation fails for all the `ROM_EXTs`, `mask_ROM` must execute the `fail` function provided by the boot policy."

5.12.1 Assumptions

- A `ROM_EXT` is valid if the associated `ROM_EXT` manifest contains an identifier that is nonzero, a valid signature, a valid public key, and if the verification of the signature succeeds using the `RSASSA-PKCS1-V1_5-VERIFY` function.
- The `fail` function in the boot policy is equal to the `__func_fail`.
- The `__func_fail` function is a reasonable abstraction of the `fail` function.

```
void __func_fail() {
    __boot_failed_called[__current_rom_ext] = 1;
} //used for CBMC
```

5.12.2 Verification

Line 494 in `mask_rom.c`:

```
411 void PROOF_HARNESS() {
... ...
431     for (int i = 0; i < rom_exts_to_try.size; i++) {
...         ...
443         if (__validated_rom_exts[i]) { //validated - try to boot from
...             ...
473         }
474         else { //invalidated - unsafe to boot from
475             __REACHABILITY_CHECK
...             ...
494             __CPROVER_postcondition(__imply(i < __current_rom_ext, !__boot_failed_called[i
    ↪ ]),
495             "Postcondition PROPERTY 8: A rom_ext (not the last one) fails => fail func is
    ↪ not called");
...             ...
```

The `__CPROVER_postcondition` at line 494 succeeds and asserts that if the `ROM_EXT` is invalidated, and the `ROM_EXT` is not the last `ROM_EXT` being processed by `mask_ROM`, then the `fail` function in the boot policy should not have been called, as a later `ROM_EXT` might succeed validation. `__boot_failed_called` is an array of size `MAX_ROM_EXTS`, it documents whether the fail function was called from `mask_ROM` when validating the ith `ROM_EXT`.

Line 497 in `mask_rom.c`:

```
411 void PROOF_HARNESS() {
... ...
431     for (int i = 0; i < rom_exts_to_try.size; i++) {
```

```
...         ...
443         if (__validated_rom_exts[i]) { //validated - try to boot from
...             ...
473         }
474         else { //invalidated - unsafe to boot from
475             __REACHABILITY_CHECK
...             ...
497             __CPROVER_postcondition(__imply(i == __current_rom_ext, __boot_failed_called[i
        ↪ ]),
498             "Postcondition PROPERTY 8: Last rom_ext fail => fail func has been called");
...             ...
```

The `__CPROVER_postcondition` at line 497 succeeds and asserts that if the last `ROM_EXT` being processed by `mask_ROM` was invalidated, the `fail` function in the boot policy was called after `mask_ROM` processed that `ROM_EXT`.

5.12.3 Results

The CBMC results state that all of the two user-defined assertions succeed. The reachability checks show that all of the assertions are indeed reachable. All of the unwinding assertions pass, meaning that CBMC searches the entire reachable state space. This means that:

> The `fail` function in the boot policy is guaranteed to be executed when all `ROM_EXTs` are invalidated.

> The `fail` function in the boot policy is guaranteed to only be executed when all `ROM_EXTs` are invalidated.

Thus the `mask_ROM` boot code is proven to fully satisfy PROPERTY 8 under the assumptions in Section 5.12.1.

5.13 PROPERTY 9

"The entire flash must be covered by a PMP region at the initialization of `mask_ROM`. The PMP region must be locked and restricted to read-only access."

5.13.1 Assumptions

- The external `PMP_WRITE_REGION` function is out of scope of the `mask_ROM` boot code and thus we model it to mock a PMP region based on the inputs given.
- If `PMP_WRITE_REGION` is called with 15 as the first parameter, it will create a PMP region covering the entire flash.

5.13.2 Verification

Line 245 in `mask_rom.c`:

```
245 void PMP_enable_memory_protection() {
246     //Apply PMP region 15 to cover entire flash
247     PMP_WRITE_REGION(       15,          1,          0,          0,          1);
248     //                   Region        Read       Write     Execute     Locked
249
250     __register_pmp_region(-1, 15, 1, 0, 0, 1);
251     __REACHABILITY_CHECK
252 }
```

The `PMP_enable_memory_protection` function applies PMP region #15 that is read-only and locked and covers the entire flash. When it is called by `mask_ROM`, it is registered that PMP region #15 has been applied in the process of validating a given `ROM_EXT` (line 250). The applied PMP regions are registered in an array.

Line 540 in `mask_rom.c`:

```
529 void mask_rom_boot(boot_policy_t boot_policy, rom_exts_manifests_t rom_exts_to_try){
…       …
540     __CPROVER_precondition(__help_all_pmp_inactive(),
541     "Precondition PROPERTY 9: All PMP regions should be unset at beginning of mask_rom.");
…       …
```

The `__CPROVER_precondition` at line 540 succeeds and asserts that at the initialization of `mask_ROM` before it begins validating `ROM_EXTs`, no PMP regions have been applied.

Line 548 in `mask_rom.c`:

```
529 void mask_rom_boot(boot_policy_t boot_policy, rom_exts_manifests_t rom_exts_to_try){
…       …
543     PMP_enable_memory_protection();
544
545     //Step 2.iii
546     for (int i = 0; i < rom_exts_to_try.size; i++) {
547
548         __CPROVER_assert(__help_check_pmp_region(i, 15, 1, 0, 0, 1),
549         "PROPERTY 9: PMP region 15 should be R and L.");
…           …
```

The `__CPROVER_assert` at line 548 succeeds and asserts that before `mask_ROM` starts validating `ROM_EXTs`, PMP region #15 has been applied correctly.

Line 467 and 500, in `mask_rom.c`:

```
411 void PROOF_HARNESS() {
…       …
431     for (int i = 0; i < rom_exts_to_try.size; i++) {
…           …
443         if (__validated_rom_exts[i]) { //validated - try to boot from
444             __REACHABILITY_CHECK
…               …
467             __CPROVER_postcondition(__help_check_pmp_region(i, 15, 1, 0, 0, 1),
468             "Postcondition PROPERTY 9: PMP region 15 should be R and L, when rom_ext was
        ↪ validated.");
…               …
473         }
474         else { //invalidated - unsafe to boot from
475             __REACHABILITY_CHECK
…               …
500             __CPROVER_postcondition(__help_check_pmp_region(i, 15, 1, 0, 0, 1),
501             "Postcondition PROPERTY 9: PMP region 15 should be R and L. Even if rom_ext
        ↪ was invalidated.");
…               …
```

The postconditions at line 467 and 500 assert that independently of the `ROM_EXT` being validated or not, PMP region #15 should have been applied in the process of validating them. Both of these succeed.

5.13.3 Results

The CBMC results state that all of the four user-defined assertions pass. The reachability checks ensure that the assertions are indeed reachable. All of the unwinding assertions pass, meaning that CBMC searches the entire reachable state space. This means that:

> No PMP regions are applied before executing `mask_ROM`.

> PMP region #15 is applied correctly to be locked and only facilitates reading before starting to validate `ROM_EXT` manifests.

> PMP region #15 is applied correctly to be locked and only facilitates reading in the process of validating all `ROM_EXT` manifest, independently of their validity.

Thus the `mask_ROM` boot code is proven to fully satisfy PROPERTY 9 under the assumptions in Section 5.13.1.

5.14 PROPERTY 10

> "*If a `ROM_EXT` is validated then `mask_ROM` must create a PMP region covering the `ROM_EXT` memory, that is locked and that allows for read and execution access.*"

5.14.1 Assumptions

- A `ROM_EXT` is valid if the associated `ROM_EXT` manifest contains an identifier that is nonzero, a valid signature, a valid public key, and if the verification of the signature succeeds using the `RSASSA-PKCS1-V1_5-VERIFY` function.
- The external `PMP_WRITE_REGION` function is out of scope of the `mask_ROM` boot code and thus we model it to mock a PMP region based on the inputs given.
- If `PMP_WRITE_REGION` is called with 0 as the first parameter, it will create a PMP region covering the memory of the `ROM_EXT` image.

5.14.2 Verification

Line 236 in `mask_rom.c`:

```
236 void PMP_unlock_rom_ext() {
237     //Read, Execute, Locked the address space of the ROM extension image
238     PMP_WRITE_REGION(       0,          1,          0,          1,          1);
239     //                  Region        Read       Write     Execute     Locked
240     __register_pmp_region(__current_rom_ext, 0, 1, 0, 1, 1);
241     __REACHABILITY_CHECK
242 }
```

The `PMP_unlock_rom_ext` function applies PMP region #0 that is read-only, executable, and locked and covers the `ROM_EXT` image. When it is called by `mask_ROM`, it is registered that PMP region #0 has been applied in the process of validating a given `ROM_EXT`. This happens at line 240.

Line 605 in `mask_rom.c`:

```
529 void mask_rom_boot(boot_policy_t boot_policy, rom_exts_manifests_t rom_exts_to_try){
...     ...
546     for (int i = 0; i < rom_exts_to_try.size; i++) {
```

```
...         ...
600           __validated_rom_exts[i] = 1; //for CBMC
601
602         //Step 2.iii.d
603         PMP_unlock_rom_ext();
604
605         __CPROVER_assert(__help_check_pmp_region(i, 0, 1, 0, 1, 1),
606         "PROPERTY 10: PMP region 0 should be R, E, and L.");
...         ...
```

The `__CPROVER_assert` at line 605 succeeds and asserts that when the given `ROM_EXT` has been validated and `PMP_unlock_rom_ext` has been called, PMP region #0 has been applied correctly.

Line 470 in `mask_rom.c`:

```
411 void PROOF_HARNESS() {
...     ...
431     for (int i = 0; i < rom_exts_to_try.size; i++) {
...         ...
443         if (__validated_rom_exts[i]) { //validated - try to boot from
444             __REACHABILITY_CHECK
...             ...
470             __CPROVER_postcondition(__help_check_pmp_region(i, 0, 1, 0, 1, 1),
471             "Postcondition PROPERTY 10: If rom_ext was valided, then PMP region 0 should
    ↪ be R, E, and L.");
...             ...
```

The `__CPROVER_postcondition` at line 470 succeeds and asserts that if the given `ROM_EXT` was validated, then PMP region #0 has been applied correctly.

Line 503 in `mask_rom.c`:

```
411 void PROOF_HARNESS() {
...     ...
431     for (int i = 0; i < rom_exts_to_try.size; i++) {
...         ...
443         if (__validated_rom_exts[i]) { //validated - try to boot from
...             ...
473         }
474         else { //invalidated - unsafe to boot from
475             __REACHABILITY_CHECK
...             ...
503             __CPROVER_postcondition(__help_check_pmp_region(i, 0, 0, 0, 0, 0),
504             "Postcondition PROPERTY 10: If rom_ext was invalid, PMP region 0 should not be
    ↪  R, E, W, and L.");
...             ...
```

The `__CPROVER_postcondition` at line 503 succeeds and asserts that if the given `ROM_EXT` was invalidated, PMP region #0 was not applied in the process of verifying that `ROM_EXT`.

5.14.3 Results

The CBMC results state that all of the three user-defined assertions succeed. The reachability checks ensure that the assertions are indeed reachable. All of the unwinding assertions pass, meaning that CBMC searches the entire reachable state space. This means that:

> PMP region #0 is applied correctly by `mask_ROM` to be locked and only allow for read and execution if the validation of the given `ROM_EXT` succeeds.

> PMP region #0 is not applied in the process of validating an invalid `ROM_EXT`.

Thus the `mask_ROM` boot code is proven to fully satisfy PROPERTY 10 under the assumptions in Section 5.14.1.

5.15 Conclusion

The verification results apply to the model of the boot code described in Section 5.1 and under the assumptions listed throughout Chapter 5. First of all, the results show that the `mask_ROM` boot code is free of bugs that can be automatically detected by CBMC (PROPERTY 0). Note that this cannot be considered proof as it does not guarantee the absence of all bugs. In addition, the results show that PROPERTY 1-10 listed in Section 4.3 are verified to be satisfied.

The goal *"G1: The hash of the `ROM_EXT` image and the signature of the hash must be validated by `mask_ROM` before it is executed to ensure authenticity and integrity of the image."* can be considered satisfied as all the derivable security properties (PROPERTY 1-8) are proven to be satisfied.

The goals *"G10, G11, G12: Only software with write/read/execute access to some memory section may modify/read/execute it."* cannot be considered fully satisfied as the derived properties (PROPERTY 9 and 10) are not exhaustive but those relevant to the system under verification. We believe that full verification of G10, G11, and G12, must take into consideration a larger system under verification that requires a detailed model of the memory, CPU, and PMP hardware component.

Chapter 6

System Attacks

A part of our P9 project [1] was to create a threat analysis of OpenTitan based on the STRIDE model [49]. In this chapter, we will enumerate potential attacks on the `mask_ROM` boot code and model them in CBMC to see their implication on the security properties.

We will only consider attacks that require physical access to the OpenTitan chip. We assume that while the `mask_ROM` boot code is executed, the OpenTitan chip cannot be accessed by other means than physical. We consider it to be impossible to replace hardware components of the OpenTitan chip with malicious hardware components. The attacks considered are only the attacks that are possible by reading memory, modifying memory, exploiting a backdoor, or by affecting the hardware modules. The physical attacks could be performed by alpha particle blasting, hitting the device, heating, etc.

6.1 Attack on the HMAC Module

The HMAC module is responsible for calculating the hash used during the signature verification process performed by the OTBN module. In this section, we will investigate the consequences of a successful attack on the HMAC module that results in wrong outputs. The means of such an attack could be physical attacks or exploitation of a backdoor. Such an attack requires physical access to the OpenTitan at some point. Additionally, it would require knowing how and where to attack the HMAC physically or how to exploit a possible backdoor. The consequence of such an attack could be that it returns, wrongfully, what corresponds to the decrypted signature. In order to succeed, the HMAC must be altered to return precisely the value of the decrypted signature. However, acquiring a valid key and signature pair is not impossible, as it is present in all of OpenTitan's `ROM_EXT` manifest stored in flash. Additionally, for malicious code to be executed, it requires flashing a malicious `ROM_EXT` with a valid public key. If the attacker succeeds in making the HMAC module return a hash that matches the decrypted signature, it is possible to make a `ROM_EXT` manifest with an incorrect signature pass the signature verification step.

The attack is modeled by having the HMAC return either a nondeterministic hash or the decrypted signature. The model of the attack is seen in Listing 6.1.

```
char* __compromised_decrypt;

char* OTBN_RSA_3072_DECRYPT(int32_t* signature, int signature_len, int32_t exponent,
    ↪ int32_t* modulus) {
    char* decrypt = malloc(256 / 8); //model it to be ok for PROPERTY 5
    __compromised_decrypt = decrypt;
    return decrypt;
}

BYTE* HMAC_SHA2_256(BYTE mes[], int size, rom_ext_manifest_t __current_rom_ext_mf) {
    int __nondet_int;
    switch (__nondet_int)
    {
    case 1:
        return malloc(256 / 8);
    default:
```

```
            return __compromised_decrypt; //Attacker spoofs the hash, knowing the public key (
        ↪ and thus target decrypt)
        }
    }
```

Listing 6.1: Model of HMAC occasionally returning the decrypted signature.

Results: Verifying the attacked `mask_ROM` boot code produces the same CBMC output as verifying the non-attacked `mask_ROM` boot code. Thus, the assertions we have included so far do not detect such an attack on the HMAC nor indicate a change in functionality of the `mask_ROM` boot code. This is because we have no assertions about the correctness of the hash (as those would be computationally impossible) and are thus unable to prove or disprove the correctness of the HMAC output with CBMC (cf. PROPERTY 3).

6.2 Attack on the OTBN Module

The OTBN is responsible for verifying the signature in the `ROM_EXT` manifest against a public key and an expected message. In this section, we will investigate the consequences of a successful attack on the OTBN module that results in wrong outputs. The means of such an attack could be physical attacks or exploitation of a backdoor. Such an attack requires physical access to the OpenTitan at some point. The likeliness of such an attack to be successful is low since it would require knowing how and where to attack the OTBN physically or how to exploit a possible backdoor. The consequence of this attack could be that it falsely validates an invalid `ROM_EXT` manifest or invalidates a valid `ROM_EXT` manifest.

This attack is modeled by having the OTBN module occasionally validate an invalid `ROM_EXT` manifest and occasionally invalidate a valid `ROM_EXT` manifest. The model of the attack is seen in Listing 6.2.

```
1  int OTBN_RSASSA_PKCS1_V1_5_VERIFY(int32_t exponent, int32_t* modulus, char* message, int
       ↪ message_len, int32_t* signature, int signature_len, rom_ext_manifest_t
       ↪ __current_rom_ext_mf) {
...        ...
3      if (cmp_hash_decrypt(hash, decrypt, 256 / 8) == 0){
4          __valid_signature[__current_rom_ext] = 1;
5          int __nondet_int;
6          switch (__nondet_int)
7          {
8          case 1:
9              return 0;
10         default:
11             return 1;
12         }
13     }
14     else{
15         __valid_signature[__current_rom_ext] = 0;
16         int __nondet_int;
17         switch (__nondet_int)
18         {
19         case 1:
20             return 0;
21         default:
22             return 1;
23         }
24     }
25 }
```

Listing 6.2: Model of Faulty OTBN.

Results: When verifying the attacked `mask_ROM` boot code, the following assertions now fail:

- Postcondition PROPERTY 5: rom_ext VALIDATED => signature valid
- Postcondition: rom_ext INVALIDATED => signature or key is invalid
- Postcondition PROPERTY 5: rom_ext INVALIDATED => signature invalid or not checked
- PROPERTY 5: Stop verification if signature is invalid
- PROPERTY 5: Continue verification if signature is valid

Thus, when the OTBN can falsely invalidate a valid `ROM_EXT` manifest and falsely validate an invalid `ROM_EXT` manifest, the CBMC verification shows that:

> A validated `ROM_EXT` manifest can have an invalid signature.

> A `ROM_EXT` manifest can be invalidated even though it has a valid key and a valid signature.

> Verification may continue even if a `ROM_EXT` manifest has an invalid signature.

> Verification may stop even if the signature in the `ROM_EXT` manifest is valid.

Thus, according to our assertions, the `mask_ROM` boot code is vulnerable to an attack on the OTBN module that would compromise PROPERTY 5.

6.3 Attack on the Whitelist

This attack consists of adding a key to the whitelist defining the set of valid public keys. The consequence of this attack is that it is possible to execute a malicious `ROM_EXT` that is signed with the new key. To successfully perform this attack would require physical access to the OpenTitan chip, knowledge of where the whitelist is stored, and being able to flash a new `ROM_EXT` manifest.

The attack is modeled by creating a new tampered whitelist which the boot code uses for verification. This whitelist is modeled to always contain the public key for a given `ROM_EXT` manifest. In this way, both malicious and benevolent `ROM_EXTs` pass the whitelist check. The CBMC assertions will, however, still use the original whitelist. The model of the attack is seen in Listing 6.3.

```
1  static pub_key_t __pkey_whitelist[MAX_ROM_EXTS];
2  static pub_key_t __tampered_pkey_whitelist[MAX_ROM_EXTS];
…  …
4
5  pub_key_t* ROM_CTRL_get_whitelist() {
6      return __tampered_pkey_whitelist;
7  }
…  …
9
10 extern int check_pub_key_valid(pub_key_t rom_ext_pub_key){ //assumed behavior behavior of
       ↪ check func
11     pub_key_t* pkey_whitelist = ROM_CTRL_get_whitelist();
…      …
13 }
14
15 int __help_pkey_valid(pub_key_t pkey) {
…      …
```

```
17      pub_key_t* pkey_whitelist = __pkey_whitelist; //Note it uses the original whitelist (
        ↪ untampered)
...     ...
19  }
...     ...
21
22  void PROOF_HARNESS() {
23      boot_policy_t boot_policy = FLASH_CTRL_read_boot_policy();
24      rom_exts_manifests_t rom_exts_to_try = FLASH_CTRL_rom_ext_manifests_to_try(boot_policy
        ↪ );
...     ...
26      for(int i = 0; i < MAX_ROM_EXTS; i++){
27          __tampered_pkey_whitelist[i] = rom_exts_to_try.rom_exts_mfs[i].pub_signature_key;
        ↪ //WHITELIST TAMPERING ATTACK
28          }
...     ...
```

Listing 6.3: Model of the tampered whitelist attack.

Results:
When verifying the attacked `mask_ROM` boot code, the following assertions fail:

- PROPERTY 2: Continue verification if key is valid
- Postcondition PROPERTY 2: rom_ext VALIDATED => valid key
- Postcondition PROPERTY 5: If sign or key is invalid then verify signature function is not called

In addition, the `check_pub_key_valid` function never returns false. Therefore, the assertion: "PROPERTY 2: Stop verification if key is invalid", is never reached and succeeds by default.

This means that when the whitelist has been tampered, the CBMC verification shows that:

> It is not guaranteed that verification is only continued if the key in the `ROM_EXT` manifest is valid.

> It is not guaranteed that verification is not stopped if a `ROM_EXT` manifest contains an invalid key.

> It is not guaranteed that all validated `ROM_EXT` manifest contains a valid key.

> It is not guaranteed that the signature verification function is only called if the signature and key of a `ROM_EXT` manifest are valid.

Thus, according to our assertions, the `mask_ROM` boot code is vulnerable to an attack on the whitelist, which would compromise PROPERTY 2 and 5.

6.4 Attack on the Boot Policy Failure Functions

The failure functions in the boot policy define the functionality that should be executed if all `ROM_EXT` manifests fail validation or if a `ROM_EXT` returns. The first fail function is called `fail`, the second is called `fail_rom_ext_terminated`. Modifying the failure functions could lead to executing malicious code. Below is a list of reasonable ways of guaranteeing that the validation process always fails for all `ROM_EXT` manifests. Thereby guaranteeing that the `fail` is always executed:

- Overwriting the whitelist in mask ROM. This way, none of the `ROM_EXT` will contain a valid public key.
- Change the public key in all `ROM_EXT` manifests to an invalid key.

- Change the signature in all `ROM_EXT` manifests to an invalid signature.

For performing any of these attacks, the attacker needs physical access to the OpenTitan chip, knowledge of where to flash a new boot policy to, and be able to alter the whitelist or the `ROM_EXT` manifests.

Note that the fact that a malicious boot policy can go undetected is a security risk in itself. The reason is that even without performing any of the three attacks mentioned above, the failure functions could be executed, leading to the execution of malicious code.

The fail functions attack is modeled by making `fail` point to `dangerFunctionALL` and `fail_rom_ext_terminated` point to `dangerFunctionRETURN`. The danger functions represent malicious code.

The consequence of this attack is that if all `ROM_EXT` manifest naturally fail validation or a validated `ROM_EXT` returns, then malicious code will be executed. The model of the attack is seen in Listing 6.4.

```
1  void dangerFunctionALL() {
2      __REACHABILITY_CHECK
3  }
4
5  void dangerFunctionRETURN(rom_ext_manifest_t _) {
6      __REACHABILITY_CHECK
7  }
8
9  void PROOF_HARNESS() {
10     boot_policy_t boot_policy = FLASH_CTRL_read_boot_policy();
11     rom_exts_manifests_t rom_exts_to_try = FLASH_CTRL_rom_ext_manifests_to_try(boot_policy
       ↪ );
...    ...
13     __CPROVER_assume(boot_policy.fail == &dangerFunctionALL);
14     __CPROVER_assume(boot_policy.fail_rom_ext_terminated == &dangerFunctionRETURN);
...    ...
16     mask_rom_boot(boot_policy, rom_exts_to_try);
...    ...
```

Listing 6.4: Overwriting failure functions in boot policy.

Results: When verifying the attacked `mask_ROM` boot code, the following assertions fail:

- Precondition: Assumes boot_policy.fail has ok address
- Precondition: Assumes boot_policy.fail_rom_ext_terminated has ok address
- Postcondition PROPERTY 6: (valid rom_ext and rom_ext code return) => that rom_ext term func is called
- Postcondition PROPERTY 8: Last rom_ext fail => fail func has been called

All of these assertions fail for trivial reasons. They all assert for a hardcoded address value, which we now have changed by assumption.

The reachability checks in `dangerFunctionALL` and `dangerFunctionRETURN` are triggered. Thus the tampered fail functions can be executed. An important thing to note is the absence of failing reachability checks in the boot code. This means that the attack is not caught by the boot code. This verifies that the boot policy can be altered without the boot code halting verification.

When the failure functions in the boot policy have been attacked, the CBMC verification shows that:

If all `ROM_EXT` manifests fail validation or a `ROM_EXT` returns, then malicious code is executed.

A tampered boot policy is not caught by any boot code integrity checks.

Thus, according to our assertions, the `mask_ROM` boot code is vulnerable to an attack on the boot policy, which compromises the security of the OpenTitan chip and PROPERTY 6 and 8.

6.5 Attack on the PMP Module

An attack on the PMP module would consist of making it malfunction by performing a physical attack on it. A possible consequence of such an attack would be to alter the PMP regions to have other properties than originally, e.g. to wrongly allow for execution. The current abstraction level of the boot code is not affected by the PMP regions. Therefore, the effect of modeling a PMP module attack is not detectable. For this reason, the attack is not modeled.

6.6 Attack on the Image Code Length

This attack consists of tampering the `ROM_EXT` manifest by modifying the `image_length` variable. This could lead to e.g. loading more than the `ROM_EXT` image code resulting in executing malicious code or making all `ROM_EXT` manifests fail validation.

There are two different scenarios when tampering the `image_length` variable:

- If the `ROM_EXT` manifest is signed with an `image_length` greater than the actual length of the `image_code` then the `mask_ROM` boot code will access memory out of `image_code` bounds but may still succeed verification. It may however cause some memory errors.
- If the `ROM_EXT` manifest is signed with an `image_length` less than the actual length of the `image_code` then the message to be verified against the signature will exclude the last part of the `image_code`. This will allow the last part of the `image_code` to be altered and still pass verification.

The attack is modeled in the `PROOF_HARNESS` by allocating image code memory of length `__image_actual_size` which can be different from the `image_length` field in the manifest. We will first consider the case where the `image_length` is greater than the actual `image_code` length in Listing 6.5.

```
1  void PROOF_HARNESS() {
...    ...
3      for(int i = 0; i < rom_exts_to_try.size; i++){
4          __CPROVER_assume(MAX_IMAGE_LENGTH >= rom_exts_to_try.rom_exts_mfs[i].image_length
       ↪ && rom_exts_to_try.rom_exts_mfs[i].image_length > 0);
5
6          int __image_actual_size;
7
8          __CPROVER_assume(__image_actual_size <= MAX_IMAGE_LENGTH);
9
10         __CPROVER_assume(__image_actual_size < rom_exts_to_try.rom_exts_mfs[i].
       ↪ image_length);
11
12         rom_exts_to_try.rom_exts_mfs[i].image_code = malloc(sizeof(char) *
       ↪ __image_actual_size);
...        ...
```

Listing 6.5: Model of `image_length` attack where `image_length` is greater than the actual size.

Results: When verifying the attacked `mask_ROM` boot code the following assertions fail:

- Precondition: Assumes rom ext image code is readable
- memcpy source region readable
- pointer outside dynamic object bounds

The last two of these assertions are automatically generated as part of PROPERTY 0. The precondition is the only user-defined assertion that fails. CBMC only captures this attack as giving memory errors.

When the `image_length` is greater than the actual length of the `image_code`, the CBMC verification shows that:

> An image length greater than the actual size of the image code causes memory errors.

We will now consider the case where `image_length` is less than the actual `image_code` length in Listing 6.6.

```
1  void PROOF_HARNESS() {
…      ...
3      for(int i = 0; i < rom_exts_to_try.size; i++){
4          __CPROVER_assume(MAX_IMAGE_LENGTH >= rom_exts_to_try.rom_exts_mfs[i].image_length
       ↪ && rom_exts_to_try.rom_exts_mfs[i].image_length > 0);
5
6          int __image_actual_size;
7
8          __CPROVER_assume(__image_actual_size <= MAX_IMAGE_LENGTH);
9
10         __CPROVER_assume(__image_actual_size > rom_exts_to_try.rom_exts_mfs[i].
       ↪ image_length);
11
12         rom_exts_to_try.rom_exts_mfs[i].image_code = malloc(sizeof(char) *
       ↪ __image_actual_size);
…          ...
```

Listing 6.6: Model of `image_length` attack where `image_length` is less than the actual size.

Results: When verifying the attacked `mask_ROM` boot code with `image_length` less than the actual `image_code` length no assertions fail.

None of our assertions fail because the reachability has not changed and because our model does not label image code as either benevolent or malicious. The image code, signature, hash, and decrypt are all nondeterministic and thus, all values are considered. Therefore, we can not distinguish malicious data from benevolent data and assertion checking for this case of malicious code execution is not possible with our model. Note that none of the reachability checks are unreachable as a result of the attack. Therefore the boot code can validate a `ROM_EXT` manifest with tampered image code and image length.

When the `image_length` is less than the actual length of the `image_code`, the CBMC verification shows that:

> A `ROM_EXT` manifest with tampered image code and with image length less than the actual size of the image code can pass verification.

> The `mask_ROM` boot code is unable to validate the image code length.

Thus, according to our assertions, the `mask_ROM` boot code is vulnerable to an attack on the `ROM_EXT` manifests' `image_length`.

Chapter 7

Discussion

In this chapter, we present a discussion of the CBMC tool and the CBMC verification. In addition we will compare our work with CBMC to SV106f21's work with UPPAAL.

7.1 Verifying C using C

We have previously worked with the tool UPPAAL. In UPPAAL, a system is verified by expressing it in terms of a timed automata, that is, in a language different from the one used to implement the system. It is usually desirable that the verification should be easier or less error prone than implementing the system. However, in CBMC, the verification of C code often requires adding C code. We have experienced issues with this. As an example, we have a function named `__help_check_rom_ext_manifest` which purpose is to help verify the correctness of `check_rom_ext_manifest`. The case is that the code constituting the `check_rom_ext_manifest` function is a subset of the code that constitutes `__help_check_rom_ext_manifest`. This means that we verify if unverified code works by asserting if it behaves the same as a, more or less, copy of itself. A solution could be that a function such as `check_rom_ext_manifest` is verified against a function contract instead, as this would require writing the expected behavior with a different syntax/semantic[1].

7.2 Modeling of Hardware and Co-verification

Co-verification of software and hardware is about verifying software and hardware together [50]. The purpose is to verify that the implementation of the software and the hardware works as expected when running together. Co-verification is achieved by creating a software model of the hardware and running the software together with that.

There is a significant difference in the necessary degree of detail for verifying the use of hardware versus the hardware implementation. If the goal is to verify the use of hardware, like it is in our case, then the best approach is to abstract away most hardware details. In our experience, it is beneficial to use nondeterminism as an abstraction for hardware functionality. Nondeterminism allows for an over-approximation of the output. This means that if the software satisfies the properties using this model of a hardware component, then it should also do it using a more detailed model with a "narrower output range". Thus a more detailed model would not create a stronger proof for this case. Instead, it would increase the verification complexity and increase the risk of modeling errors.

If the goal is to verify meaningful properties for a hardware component, the model must be precise and accurately represent the implementation. I.e. there should be a small modeling gap to create a strong proof. To create such a model it is necessary to know possible inputs, outputs, and internal implementation. In reality, hardware is very complex as its domain is within the physical world. As an example, consider the notion of bits. In the digital world, a bit is either 0 or 1. In the world of transistor-transistor logic, a bit is either no higher than 0.4 volts (represents 0) or no less than 2.6 volts (represents 1) [51]. This is not to say that modeling of hardware is impossible and should never be done. Instead, we argue that modeling of hardware is difficult, and one needs first to consider

[1] Function contracts is, however, not fully implemented in CBMC

what needs to be verified and then determine what is necessary to detail and what can be abstracted away.

We have in our work experimented with two different approaches for hardware modeling. These are by CBMC nondeterminism and by creating a C model. Most models of hardware will require both approaches to some degree. CBMC nondeterminism is fast to implement, creates an over-approximation, and does not require any implementation knowledge. It does, however, inhibit verifying any properties about the hardware itself. It should be possible to model most hardware (if not all) as a C model. C models are of varying detail in our experience and it is difficult to determine if the model is correct. C models allow for both over-, under-, and exact-approximation. Under-approximation should be avoided as this allows for false negatives. In retrospect, our C models abstract away all hardware details and most of them include some nondeterminism.

An approach that we have not tried is to create a model from the hardware Verilog or SystemVerilog. The OpenTitan GitHub consists of 65.2% SystemVerilog and is thus assumed to have a considerable part of the hardware implementation in SystemVerilog.

A tool such as v2c [52] or Verilator [53] can generate a C/C++ model of the SystemVerilog code. The v2c tool (developed by the people behind EBMC/CBMC) converts Verilog into ANSI-C. The Verilator tool converts Verilog into C++. We have no experience ourselves with the output of these tools but the authors of [52] denote that the Verilator output code is large. A concern could be that this auto generated code is complex and unreadable making it unsuited for manual integration into the model of the software.

The EBMC verification tool takes either Verilog or SystemVerilog as input and computes a C model from it using v2c. The EBMC input can be decorated with C code to model higher level abstractions as well. The EBMC engine uses the CBMC engine for SAT encoding and satisfiability checking [54]. We have no experience with EBMC but the tool sounds promising for co-verification of hardware properties.

7.3 Modeling Cryptographic Functionality

Cryptographic functions like hashing, encryption, and decryption are particularly difficult functions to verify. Most cryptographic functions that are used for security are computationally heavy. The set of possible inputs can be enormous. As an example, the HMAC hash in OpenTitan takes as input the entire image code. Thus the possible inputs that verification needs to consider can create a large state space.

In order to verify that the cryptographic functions are correct, we would need to verify that the mapping of input to output is correct. Verifying that a cryptographic output is correct is hard. Cryptographic functions used for security purposes are designed to have an unpredictable mathematical link between input and output. So in order to verify the output, one would need to compare it to the output of a function guaranteed to be correct. For such a function to exist, it would need to be verified to be correct, which is just as hard to verify.

Another approach to verify a cryptographic function would be to verify the output against a known valid test vector. This approach is closer to testing than verification. For a cryptographic function to be considered correct with this approach, the test vector will need to be complete. For most cryptographic functions, such a test vector does not exist as the existence of such would be in direct conflict with the general design goals of cryptographic functions.

7.4 Correctness of CBMC

CBMC's internal representation of the C code can be considered a model. The model is either incorrect or more abstract if, at any point, the translation to or verification of this internal model is incorrect or loses detail. As mentioned in Section 1.5, the verification performed by CBMC is only valid as long as the transformation from C code to GOTO, SSA, and SAT instance is correct. In addition, the used SAT solver (currently MiniSAT) also has to be correct in order for the verification results to be considered proof. We have not found any documentation containing proof of the correctness of CBMC.

7.5 Our Experience with CBMC

In the previous semester, we worked with Frama-C to verify C programs. We experienced that there was a steep learning curve for learning how to properly use Frama-C. We believe this to be caused by how detailed function contracts and loop invariants must be for Frama-C to perform meaningful verification. In contrast, we experienced that it was relatively easy to verify the same or similar properties using CBMC. This might be attributed to either that assertions are easier to express in CBMC, that CBMC does not need loop invariants to perform verification, or that the experience gained from Frama-C made it easier to use CBMC.

There are still some difficult parts of using CBMC and certain things to be aware of. It was not easy to fully understand how all constructs work in CBMC, e.g. precisely what effects `__CPROVER_assume` has. Documentation about the underlying CBMC theory is, in our opinion, lackluster or in some cases nonexistent. When performing verification, it takes some effort to understand the output of `--trace` when debugging errors. It is important to have a reachability check on all branches with assertions to avoid false negatives (false assertions that succeed). Also, certain constructs do not behave intuitively, e.g. the output of `__CPROVER_OBJECT_SIZE` on a pointer inside a struct will return the size of the struct and not the memory object pointed to by the pointer.

Lastly, we lacked the feature to individually specify loop bounds for different executions of the same loop. E.g. consider the case where a program calls `memcmp` 10 times. In 9 of the cases, it is enough to unroll the loop in `memcmp` 10 times, but it requires 700 unrolls in 1 of the cases. This results in that all 10 cases have to be unrolled 700 times, significantly increasing the verification time. What we did to solve this was that we created our own `memcmp` functions, a version for each call to `memcmp` (cf. memory_compare.c in Appendix C.7). This allowed us to individually specify a bound for each comparison.

7.6 Comparison with SV106f21

SV106f21 has also been working on formally verifying the security and correctness of the co-developed C boot code for the `mask_ROM` boot stage [8]. In this section, we will compare our results to theirs in terms of the overall model of the system and verification results.

UPPAAL is a tool that is arguably easier to use for abstract modeling. This can be seen in the work of the SV106f21 group who have modeled a substantially larger part of the hardware components used in relation to the `mask_ROM` boot stage than we have.

As mentioned in Section 5.15 we have been able to fully verify G1 and partially G10, G11, and G12. Compared to SV106f21, they have been able to fully verify G1 and G9 and partially G3, G8, G10, G11, and G12. We believe that this is in part due to their modeled system being larger as it includes more hardware components.

CBMC verification is done directly on the C code. This suggests that there is less of a modeling gap compared to a more abstract UPPAAL model. This allows us to consider more security properties and attacks concerning G1, suggesting that our verification is in a smaller scope but in greater detail. In addition, we are also able to verify the absence of software bugs.

Chapter 8

Conclusion

In this chapter, we will conclude on our work with verifying the developed OpenTitan `mask_ROM` boot code seen in Appendix B using the formal verification tool CBMC. The conclusions are made with the problem statement in mind:

Is the `mask_ROM` code correct and safe?

To solve this problem, we first investigated the formal verification tool CBMC. The documentation of CBMC is sparse. As part of our project, we have made efforts to formalize a near complete description of all the theoretical steps of CBMC verification, which can be seen in Chapter 1. We go into detail about the GOTO conversions, translation of CBMC constructs, unwinding, SSA transformation, computation of bit-vector equation C and P, removal of pointer dereferences, conversion to SAT formula on CNF form, and solving of SAT problems.

We created a tutorial describing the practical aspect of CBMC in Chapter 2. In the tutorial, we cover CBMC annotations, verification arguments, execution arguments, CBMC modeling techniques, best practices, etc.

To evaluate the safety and correctness of the OpenTitan `mask_ROM` boot code, we specified 11 security properties (PROPERTIES 0-10 cf. Section 4.3) derived from the security goals from our previous work [1]. These security properties cover various aspects of safety and correctness, such as the absence of errors and compliance to the OpenTitan design. Note that these properties are not exhaustive for the security goals.

To verify the `mask_ROM` boot code, we make a set of assumptions that allow us to verify these security properties. Under these assumptions, we find that all of the security properties (PROPERTIES 0-10) are satisfied. Thus by CBMC proof, we can state that the `mask_ROM` boot code fully satisfies security goal *"G1: The hash of the `ROM_EXT` image and the signature of the hash must be validated by `mask_ROM` before it is executed to ensure authenticity and integrity of the image."* and partly satisfies security goals *"G10, G11, G12: Only software with write/read/execute access to some memory section may modify/read/execute it."*.

In order to further evaluate the safety of the `mask_ROM` boot code, we investigate the implications of a series of hardware/physical attacks. As a result, we found four vulnerabilities in the boot code. An attack on the OTBN module can comprise PROPERTY 5. An attack on the whitelist can compromise PROPERTY 2 and 5. An attack on the boot policy failure functions can compromise PROPERTY 6 and 8. An attack on the `ROM_EXT` manifest image length can compromise PROPERTY 0 and can lead to the execution of malicious code, but this is not caught by any of the security properties.

In conclusion, we find CBMC to be an adequate tool for verifying security properties of low level C code even though we found no proof of the soundness and completeness. We find that improvements can be made to the OpenTitan `mask_ROM` boot code to mitigate the implications of the hardware/physical attacks investigated.

Chapter 9

Future Work

At the time of writing, the OpenTitan project is not finished. The current model of the `mask_ROM` boot code is mainly based on a pseudo code implementation of the `mask_ROM` boot code created by OpenTitan. As with other aspects of OpenTitan, the `mask_ROM` boot code is subject to change. This means that if the documentation about the `mask_ROM` boot code is significantly changed, our model and the verification results become outdated. Despite this, we believe that it would be interesting and beneficial to keep a model of OpenTitan up to date. The reason is that OpenTitan is not a simple project and such a model could be beneficial to detect bugs. E.g. we detected the need to validate the boot policy, which is not done in the current version of the boot code. The end goal would be to have a model of the finished OpenTitan chip and thereby be able to formally verify properties about the final version of OpenTitan. Verifying a model of the entire OpenTitan chip could lead to scalability/performance issues as the produced SAT instance becomes large. A possible solution would be to try out an experimental but scalable version of CBMC [55, app. 1].

The OpenTitan project consists of 65.2% SystemVerilog. We therefore also believe that a reasonable activity could be to try and use the EBMC tool to formally verify properties about the SystemVerilog code. Another approach for verifying the SystemVerilog code, could be to write C models of the SystemVerilog programs, test the C models, and verify if the C models and SystemVerilog programs are consistent, using CBMC [16]. The EBMC tool uses the tool v2c to make a trustworthy translation of Verilog or SystemVerilog code to ANSI-C code [52]. Therefore, an approach could also be to use the v2c tool on the SystemVerilog code of OpenTitan and verify the output ANSI-C code together with the developed boot code.

As mentioned in Appendix D, the CBMC team is in the process of developing the feature to write function contracts. When that feature is finished, the current C code should be annotated. There are certain clauses of the function contract that cannot easily be asserted using standard assertions, e.g. the `__CPROVER_assigns` clause. Using function contracts could also alleviate the problem of verifying C code using C code, mentioned in Section 7.1. In addition, there are other features of CBMC that we have not explored, such as options for specifying details about the platform in terms of architecture and operating system and a feature for generating a test-suite.

Bibliography

[1] Bjarke Hilmer Møller et al. *Evaluation of Tools for Formal Verification of OpenTitan Boot Code*. URL: https://projekter.aau.dk/projekter/da/studentthesis/evaluering-af-vaerktoejer-for-formel-verifikation-af-opentitan-bootcode(53463018-b393-45bb-a428-b442ff4b1c3d).html (visited on 04/12/2021).

[2] Edmund Clarke, Daniel Kroening, and Flavio Lerda. "A Tool for Checking ANSI-C Programs". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*. Ed. by Kurt Jensen and Andreas Podelski. Vol. 2988. Lecture Notes in Computer Science. Springer, 2004, pp. 168–176. ISBN: 3-540-21299-X.

[3] OpenTitan. *OpenTitan*. URL: https://opentitan.org/ (visited on 02/17/2021).

[4] OpenTitan. *GitHub - lowRISC/OpenTitan*. URL: https://github.com/lowRISC/opentitan (visited on 02/17/2021).

[5] OpenTitan. *OpenTitan: Use Cases*. URL: https://docs.opentitan.org/doc/security/use_cases/ (visited on 02/17/2021).

[6] OpenTitan. *OpenTitan: IP cores*. URL: https://docs.opentitan.org/hw/ip/ (visited on 02/17/2021).

[7] OpenTitan. *OpenTitan: Logical Security Model*. URL: https://docs.opentitan.org/doc/security/logical_security_model/ (visited on 02/17/2021).

[8] Bjarke Hilmer Møller, Magnus Winkel Pedersen, and Bøgedal Tobias Worm. "Formally Verifying Security Properties for OpenTitan Boot Code with UPPAAL". To Appear. MA thesis. AAU, 2021.

[9] Daniel Kroening, Edmund Clarke, and Karen Yorav. "Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking". In: *Design Automation Conference (DAC)*. ACM Press, 2003, pp. 368–371. ISBN: 1-58113-688-9.

[10] Wikipedia. *Model checking*. URL: https://en.wikipedia.org/wiki/Model_checking (visited on 03/31/2021).

[11] Wikipedia. *Soundness*. URL: https://en.wikipedia.org/wiki/Soundness (visited on 05/13/2021).

[12] Daniel Kroening and Michael Tautschnig. "CBMC – C Bounded Model Checker". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Vol. 8413. LNCS. Springer, 2014, pp. 389–391. ISBN: 978-3-642-54861-1.

[13] Martin Brain. *CBMC*. URL: http://cprover.diffblue.com/group__cbmc.html (visited on 03/31/2021).

[14] Kareem Khazem and Martin Brain. *goto-programs*. URL: http://cprover.diffblue.com/group__goto-programs.html (visited on 03/26/2021).

[15] Kareem Khazem and Martin Brain. *goto-symex*. URL: http://cprover.diffblue.com/group__goto-symex.html (visited on 03/26/2021).

[16] *E. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of C and Verilog pro-grams using Bounded Model Checking. Technical Report CMU-CS-03-126, Carnegie Mellon University, School of Computer Science, 2003*. URL: http://reports-archive.adm.cs.cmu.edu/anon/2003/CMU-CS-03-126.pdf.

[17] Diffblue. *symex_main.cpp*. URL: https://github.com/diffblue/cbmc/blob/develop/src/goto-symex/symex_main.cpp (visited on 04/07/2021).

[18] Wikipedia. *SAT*. URL: https://en.wikipedia.org/wiki/SAT (visited on 03/31/2021).

[19] Wikipedia. *Cook–Levin theorem*. URL: https://en.wikipedia.org/wiki/Cook%E2%80%93Levin_theorem (visited on 03/31/2021).

[20] Doina Bucur. *SAT-based bounded model checking*. 2015. URL: http://doina.net/AR15/AR15-L6.pdf (visited on 03/31/2021).

[21] Wikipedia. *Satisfiability modulo theories*. URL: https://en.wikipedia.org/wiki/Satisfiability_modulo_theories (visited on 03/31/2021).

[22] cprover.org. *The CPROVER Manual*. URL: http://www.cprover.org/cprover-manual/ (visited on 02/15/2021).

[23] Diffblue. *GitHub Repository: diffblue/cbmc*. URL: https://github.com/diffblue/cbmc (visited on 02/22/2021).

[24] cprover.org. *CBMC*. URL: http://www.cprover.org/cbmc/ (visited on 02/15/2021).

[25] Amazon Web Services - Labs. *How to Write a CBMC Proof*. URL: https://github.com/awslabs/aws-templates-for-cbmc-proofs/blob/master/training-material/PROOF-WRITING.md (visited on 02/22/2021).

[26] Diffblue. *Github Issue: Unexpected Pointer Out Of Bounds FAILURE #5843*. URL: https://github.com/diffblue/cbmc/issues/5843#issuecomment-782371955 (visited on 02/23/2021).

[27] AWSLabs. *Debugging CBMC Issues*. URL: https://github.com/awslabs/aws-templates-for-cbmc-proofs/blob/master/training-material/DEBUG-CBMC.md (visited on 02/19/2021).

[28] Diffblue Limited. *Bug in the Exists Clause*. URL: https://github.com/diffblue/cbmc/tree/develop/regression/contracts/quantifiers-exists-ensures-02 (visited on 04/29/2020).

[29] Diffblue Limited. *Bug in the Exists Clause*. URL: https://github.com/diffblue/cbmc/tree/develop/regression/contracts/quantifiers-exists-both-02 (visited on 04/29/2020).

[30] Diffblue Limited. *Bug in the Exists Clause*. URL: https://github.com/diffblue/cbmc/tree/develop/regression/contracts/quantifiers-exists-requires-02 (visited on 04/29/2020).

[31] Diffblue Limited. *Bug in the Forall Clause*. URL: https://github.com/diffblue/cbmc/tree/develop/regression/contracts/quantifiers-forall-both-02 (visited on 04/29/2020).

[32] Wikipedia. *Digital Signature*. URL: https://en.wikipedia.org/wiki/Digital_signature (visited on 03/03/2021).

[33] Wikipedia. *Public-key cryptography*. URL: https://en.wikipedia.org/wiki/Public-key_cryptography (visited on 03/04/2021).

[34] Wikipedia. *Cryptographic hash function*. URL: https://en.wikipedia.org/wiki/Cryptographic_hash_function (visited on 03/04/2021).

[35] Wikipedia. *Key (cryptography)*. URL: https://en.wikipedia.org/wiki/Key_(cryptography) (visited on 03/04/2021).

[36] Thomas Pornin. *Security StackExchange: SHA, RSA and the relation between them*. URL: https://security.stackexchange.com/questions/9260/sha-rsa-and-the-relation-between-them#answer-9265 (visited on 12/20/2020).

[37] Wikipedia. *RSA (cryptosystem)*. URL: https://en.wikipedia.org/wiki/RSA_(cryptosystem) (visited on 03/03/2021).

[38] OpenTitan. *Reference ROM_EXT Manifest Format*. URL: https://github.com/lowRISC/opentitan/blob/master/sw/device/rom_exts/docs/manifest.md (visited on 02/22/2021).

[39] Wikipedia. *HMAC*. URL: https://en.wikipedia.org/wiki/HMAC (visited on 12/09/2020).

[40] OpenTitan. *OpenTitan Secure Boot*. URL: https://docs.opentitan.org/doc/security/specs/secure_boot/ (visited on 02/22/2021).

[41] OpenTitan. *Reference Mask ROM: Secure Boot Description*. URL: https://github.com/lowRISC/opentitan/blob/master/sw/device/mask_rom/docs/index.md (visited on 02/22/2021).

[42] OpenTitan. *OpenTitan Issue about Validation of public keys in ROM_EXT manifests*. URL: https://github.com/lowRISC/opentitan/issues/5671 (visited on 04/09/2020).

[43] OpenTitan. *HMAC HWIP Technical Specification*. URL: https://github.com/lowRISC/opentitan/blob/master/hw/ip/hmac/doc/_index.md (visited on 05/05/2020).

[44] OpenTitan. *OpenTitan Big Number Accelerator (OTBN) Technical Specification*. URL: https://github.com/lowRISC/opentitan/blob/master/hw/ip/otbn/doc/_index.md (visited on 02/24/2021).

[45] OpenTitan. *OpenTitan Issue about Documentation*. URL: https://github.com/lowRISC/opentitan/issues/4315#event-4071904280 (visited on 12/04/2020).

[46] OpenTitan. *mask_rom.c*. URL: https://github.com/lowRISC/opentitan/blob/master/sw/device/silicon_creator/mask_rom/mask_rom.c (visited on 05/25/2021).

[47] J. Jonsson and B. Kaliski. *RFC3447: Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1*. 2003. URL: https://tools.ietf.org/html/rfc3447#section-8.2.2.

[48] *Jacob G. Søndergaard, Kristoffer S. Jensen. Github Repository: CBMC - Boot Code.* URL: https://github.com/KVISDAOWNER/CBMC-b00t-c0d3.

[49] Danny Bøgsted Poulsen and René Rydhof Hansen. *Del 1 Sikkerhed? - SikSoft 01*. Oct. 23, 2020.

[50] Embedded Staff. *HW/SW co-verification basics: Part 1 – Determining what & how to verify*. URL: https://www.embedded.com/hw-sw-co-verification-basics-part-1-determining-what-how-to-verify/ (visited on 06/08/2021).

[51] Wikipedia. *Bit*. URL: https://en.wikipedia.org/wiki/Bit (visited on 06/08/2021).

[52] Rajdeep Mukherjee, Daniel Kroening, and Tom Melham. "Hardware Verification using Software Analyzers". In: *IEEE Computer Society Annual Symposium on VLSI*. IEEE, 2015, pp. 7–12. ISBN: 978-1-4799-8719-1.

[53] Wilson Snyder. *Verilator*. URL: https://github.com/verilator/verilator (visited on 06/08/2021).

[54] Diffblue. *hw-cbmc*. URL: https://github.com/diffblue/hw-cbmc (visited on 05/28/2021).

[55] Niklas Büscher. "Compilation for More Practical Secure Multi-Party Computation". PhD thesis. Darmstadt: Technische Universität, Nov. 2018. URL: http://tuprints.ulb.tu-darmstadt.de/8495/.

[56] *Bjarke H. Møller, Jacob G. Søndergaard, Kristoffer S. Jensen, Magnus W. Pedersen, Tobias W. Bøgedal. Github Repository: Boot Code.* URL: https://github.com/KVISDAOWNER/b00t-c0d3.

[57] Diffblue Limited. *Tests for Function Contracts*. URL: https://github.com/diffblue/cbmc/tree/develop/regression/contracts (visited on 04/29/2020).

[58] Diffblue Limited. *Bug in the Assigns Clause*. URL: https://github.com/diffblue/cbmc/tree/develop/regression/contracts/assigns_replace_05 (visited on 04/29/2020).

Appendix A

Developed Boot Code

The entire C code for the `mask_ROM` boot stage, developed by us and group SV106f21, can be seen below in Listing A.1. The boot code is based on information found in [38], [41], [40], [46] . The code can also be found on GitHub at [56].

```
/*
EARLY DRAFT
Not compiled or otherwise tested for ANSI C compliance

Written based on:
sw/device/rom_ext/docs/manifest.md
sw/device/mask_rom/mask_rom.c
sw/device/mask_rom/docs/index.md
doc/security/specs/secure_boot/index.md

*/
#include <string.h>
#include <stdint.h>

// The identifier that a correct manifest must contain.
// Based on https://github.com/lowRISC/opentitan/blob/master/sw/device/silicon_creator/
    ↪ mask_rom/mask_rom.c
static const uint_32 expectedRomExtIdentifier = 0x4552544F;

//Represents a public key
typedef struct pub_key_t{
    int32_t modulus[96];
    int32_t exponent;
    //something else
} pub_key_t;

//Struct representing rom_ext_manifest
typedef struct rom_ext_manifest_t{
    uint32_t identifier;

    //address of entry point
    //note: not part of the doc on the rom_ext_manifest, but included based on code seen
    ↪ in mask_rom.c
    int* entry_point;

    int32_t signature[96];

    //public part of signature key
    pub_key_t pub_signature_key;
    char image_code[];
} rom_ext_manifest_t;


//Returned by rom_ext_manifests_to_try
typedef struct rom_exts_manifests_t{
    int size;
```

```
45      rom_ext_manifest_t rom_exts_mfs[];
46  } rom_exts_manifests_t;
47
48
49  //Represents boot policy
50  typedef struct boot_policy_t{
51      int identifier;
52
53      //which rom_ext_slot to boot
54      int rom_ext_slot;
55
56      //what to do if all ROM Ext are invalid
57      void (*fail) ();
58
59      //what to do if the ROM Ext unexpectedly returns
60      void (*fail_rom_ext_terminated) (rom_ext_manifest_t);
61
62  } boot_policy_t;
63
64
65
66  typedef void(rom_ext_boot_func)(void); // Function type used to define function pointer to
        ↪  the entry of the ROM_EXT stage.
67
68
69  extern int* READ_FLASH(int start, int end);
70
71  boot_policy_t read_boot_policy()
72  {
73      int* data = READ_FLASH(0, sizeof(boot_policy_t));
74
75      boot_policy_t boot_policy;
76
77      memcpy(&boot_policy.identifier, data, sizeof(boot_policy.identifier));
78      memcpy(&boot_policy.rom_ext_slot, data + 1, sizeof(boot_policy.rom_ext_slot));
79      memcpy(&boot_policy.fail, data + 2, sizeof(boot_policy.fail));
80
81      return boot_policy;
82  }
83
84  rom_exts_manifests_t rom_ext_manifests_to_try(boot_policy_t boot_policy) {}
85
86  pub_key_t read_pub_key(rom_ext_manifest_t current_rom_ext_manifest) {
87      return current_rom_ext_manifest.pub_signature_key;
88  }
89
90  int check_pub_key_valid(pub_key_t rom_ext_pub_key); // returns a boolean value
91
92  extern char* HASH(char* message);
93
94  extern int RSA_VERIFY(pub_key_t pub_key, char* message, int32_t* signature);
95
96  int verify_rom_ext_signature(pub_key_t rom_ext_pub_key, rom_ext_manifest_t manifest) {
97      return RSA_VERIFY(rom_ext_pub_key, HASH(manifest.image_code), manifest.signature); //0
        ↪  or 1
98  }
99
100 extern void WRITE_PMP_REGION(uint8_t reg, uint8_t r, uint8_t w, uint8_t e, uint8_t l);
101
102 void pmp_unlock_rom_ext() {
103     //Read, Execute, Locked the address space of the ROM extension image
```

```
104     WRITE_PMP_REGION(       0,          1,          0,          1,          1);
105     //                   Region        Read       Write     Execute     Locked
106 }
107
108 int final_jump_to_rom_ext(rom_ext_manifest_t current_rom_ext_manifest) { // Returns a
    ↪ boolean value.
109     //Execute rom ext code step 2.iii.e
110     rom_ext_boot_func* rom_ext_entry = (rom_ext_boot_func*)current_rom_ext_manifest.
    ↪ entry_point;
111
112     rom_ext_entry();
113
114     //if rom_ext returns, we should return false
115     //and execute step 2.iv.
116     return 0;
117 }
118
119 void boot_failed(boot_policy_t boot_policy) {
120     boot_policy.fail();
121 }
122
123 void boot_failed_rom_ext_terminated(boot_policy_t boot_policy, rom_ext_manifest_t
    ↪ current_rom_ext_manifest) {
124     boot_policy.fail_rom_ext_terminated(current_rom_ext_manifest);
125 }
126
127
128 int check_rom_ext_manifest(rom_ext_manifest_t manifest) {
129     return manifest.identifier == expectedRomExtIdentifier; // If the identifier !=
    ↪ expectedRomExtIdentifier, the manifest is invalid.
130 }
131
132 void mask_rom_boot(void)
133 {
134     boot_policy_t boot_policy = read_boot_policy();
135
136     rom_exts_manifests_t rom_exts_to_try = rom_ext_manifests_to_try(boot_policy);
137
138     //MÃěske step 2.iii
139     for (int i = 0; i < rom_exts_to_try.size; i++)
140     {
141         rom_ext_manifest_t current_rom_ext_manifest = rom_exts_to_try.rom_exts_mfs[i];
142
143         if (!check_rom_ext_manifest(current_rom_ext_manifest)) {
144           continue;
145         }
146
147         //Step 2.iii.b
148         pub_key_t rom_ext_pub_key = read_pub_key(current_rom_ext_manifest);
149
150         //Step 2.iii.b
151         if (!check_pub_key_valid(rom_ext_pub_key)) {
152             continue;
153         }
154
155         //Step 2.iii.b
156         if (!verify_rom_ext_signature(rom_ext_pub_key, current_rom_ext_manifest)) {
157             continue;
158         }
159
160         //Step 2.iii.d
```

```
161         pmp_unlock_rom_ext();
162
163         //Step 2.iii.e
164         if (!final_jump_to_rom_ext(current_rom_ext_manifest)) {
165             //Step 2.iv
166             boot_failed_rom_ext_terminated(boot_policy, current_rom_ext_manifest);
167         }
168     } // End for
169
170     //Step 2.iv
171     boot_failed(boot_policy);
172 }
```

Listing A.1: The code corresponds to the `mask_ROM` stage.

Appendix B

Developed Boot Code for CBMC Verification

This chapter contains the boot code we verify in this report. It can be found at [48] on the branch "base-boot-code". This boot code is based on the boot code developed by us and SV106f21 described in Appendix A. The boot code that is verified in this report is contained in the following four files: `mask_rom.h`, `mask_rom.c`, `hmac.h`, and `hmac.c`.

B.1 mask_rom.h

```
#ifndef MASK_ROM_H
#define MASK_ROM_H

#include <string.h>
#include <stdint.h>
#include <malloc.h>
#include <stdlib.h>
#include <memory.h>

#define MAX_ROM_EXTS 1
#define RSA_SIZE 96
#define PMP_REGIONS 16
#define MAX_IMAGE_LENGTH 2


//Represents a signature.
typedef struct signature_t{
    int32_t value[RSA_SIZE];
    //something else
} signature_t;


//Represents a public key
typedef struct pub_key_t{
    int32_t exponent;
    int32_t modulus[RSA_SIZE];
    //something else
} pub_key_t;


//Struct representing rom_ext_manifest
typedef struct rom_ext_manifest_t{
    uint32_t identifier;

    signature_t signature;

    //public part of signature key
    pub_key_t pub_signature_key;
```

```
39      uint32_t image_length;
40      char* image_code;
41  } rom_ext_manifest_t;
42
43
44  //Returned by rom_ext_manifests_to_try
45  typedef struct rom_exts_manifests_t {
46      int size;
47      rom_ext_manifest_t rom_exts_mfs[MAX_ROM_EXTS];
48  } rom_exts_manifests_t;
49
50
51  //Represents boot policy
52  typedef struct boot_policy_t {
53      int identifier;
54
55      //which rom_ext_slot to boot
56      int rom_ext_slot;
57
58      //what to do if all ROM Ext are invalid
59      char* fail;
60
61      //what to do if the ROM Ext unexpectedly returns
62      char* fail_rom_ext_terminated;
63
64  } boot_policy_t;
65
66  #endif
```

Listing B.1: The content of the mask_rom.h file

B.2 mask_rom.c

```
1  /*
2  OpenTitan bootcode,
3  written based on:
4  sw/device/rom_ext/docs/manifest.md
5  sw/device/mask_rom/mask_rom.c
6  sw/device/mask_rom/docs/index.md
7  doc/security/specs/secure_boot/index.md
8  */
9
10 #include "hmac.h"
11 #include "mask_rom.h"
12
13 #define PKEY_WHITELIST_SIZE 5
14
15 BYTE hmac_key[HMAC_KEY_SIZE];
16
17 // Function type used to define function pointer to the entry of the ROM_EXT stage.
18 typedef void(rom_ext_boot_func)(void);
19
20 // Function type for entry point of boot policy fail function
21 typedef void(fail_func)(void);
22
23 // Function type for entry point of boot policy fail rom ext terminated function.
24 typedef void(fail_rom_ext_terminated_func)(rom_ext_manifest_t);
25
26
```

```
27 extern boot_policy_t FLASH_CTRL_read_boot_policy();
28
29
30 extern rom_exts_manifests_t FLASH_CTRL_rom_ext_manifests_to_try(boot_policy_t boot_policy)
       ↪ ;
31
32
33 extern char* OTBN_RSA_3072_DECRYPT(int32_t* signature, int signature_len, int32_t exponent
       ↪ , int32_t* modulus);
34
35
36 extern pub_key_t* ROM_CTRL_get_whitelist();
37
38
39 extern void PMP_WRITE_REGION(uint8_t reg, uint8_t r, uint8_t w, uint8_t e, uint8_t l);
40
41
42 int verify_rom_ext_signature(pub_key_t rom_ext_pub_key, rom_ext_manifest_t manifest) {
43
44     int bytes =
45         sizeof(manifest.pub_signature_key) + sizeof(manifest.image_length) + manifest.
       ↪ image_length;
46
47     char message[bytes];
48
49     memcpy(
50         message,
51         &manifest.pub_signature_key,
52         sizeof(manifest.pub_signature_key)
53     );
54     memcpy(
55         message + sizeof(manifest.pub_signature_key),
56         &manifest.image_length,
57         sizeof(manifest.image_length)
58     );
59     memcpy(
60         message + sizeof(manifest.pub_signature_key) + sizeof(manifest.image_length),
61         manifest.image_code,
62         manifest.image_length
63     );
64
65     signature_t signature = manifest.signature;
66
67     int result = OTBN_RSASSA_PKCS1_V1_5_VERIFY(rom_ext_pub_key.exponent, rom_ext_pub_key.
       ↪ modulus, message, bytes, signature.value, RSA_SIZE, manifest);
68
69     return result; //0 or 1
70 }
71
72
73 int OTBN_RSASSA_PKCS1_V1_5_VERIFY(int32_t exponent, int32_t* modulus, char* message, int
       ↪ message_len, int32_t* signature, int signature_len) {
74
75     if (signature_len != RSA_SIZE) {
76             return 0;
77     }
78
79     char* decrypt = OTBN_RSA_3072_DECRYPT(signature, signature_len, exponent, modulus);
80     char* hash = HMAC_SHA2_256(hmac_key, message, message_len); //message_len in bytes
81
82
```

```
83      if (memcmp(hash, decrypt, 256 / 8) == 0){
84          return 1; //verified
85      }
86      else{
87          return 0;
88      }
89  }
90
91
92  pub_key_t read_pub_key(rom_ext_manifest_t current_rom_ext_manifest) {
93      return current_rom_ext_manifest.pub_signature_key;
94  }
95
96
97  int check_pub_key_valid(pub_key_t rom_ext_pub_key){ //assumed behavior behavior of check
        ↪ func
98      pub_key_t* pkey_whitelist = ROM_CTRL_get_whitelist();
99
100     for (int i = 0; i < PKEY_WHITELIST_SIZE; i++) {
101         if (pkey_whitelist[i].exponent != rom_ext_pub_key.exponent)
102             continue;
103
104         int j = 0;
105         for (j = 0; j < RSA_SIZE; j++) {
106             if (pkey_whitelist[i].modulus[j] != rom_ext_pub_key.modulus[j])
107                 break;
108         }
109
110         //if j == RSA_SIZE, then loop ran to completion and all entries were equal
111         if (j == RSA_SIZE)
112             return 1;
113     }
114
115     return 0;
116 }
117
118
119 void PMP_unlock_rom_ext() {
120     //Read, Execute, Locked the address space of the ROM extension image
121     PMP_WRITE_REGION(        0,        1,        0,        1,        1);
122     //                  Region      Read     Write    Execute     Locked
123 }
124
125
126 void PMP_enable_memory_protection() {
127     //Apply PMP region 15 to cover entire flash
128     PMP_WRITE_REGION(       15,        1,        0,        0,        1);
129     //                  Region      Read     Write   Execute    Locked
130 }
131
132
133 int final_jump_to_rom_ext(rom_ext_manifest_t current_rom_ext_manifest) { // Returns a
        ↪ boolean value.
134     //Execute rom ext code step 2.iii.e
135     rom_ext_boot_func* rom_ext_entry = (rom_ext_boot_func*)current_rom_ext_manifest.
        ↪ image_code;
136
137     rom_ext_entry();
138
139     //if rom_ext returns, we should return false
140     //and execute step 2.iv.
```

```
141     return 0;
142 }
143
144
145 void boot_failed(boot_policy_t boot_policy) {
146     fail_func* fail_func_entry = (fail_func*)boot_policy.fail;
147     fail_func_entry();
148 }
149
150
151 void boot_failed_rom_ext_terminated(boot_policy_t boot_policy, rom_ext_manifest_t
    ↪ current_rom_ext_manifest) {
152     fail_rom_ext_terminated_func* fail_func_entry = (fail_rom_ext_terminated_func*)
    ↪ boot_policy.fail_rom_ext_terminated;
153     fail_func_entry(current_rom_ext_manifest);
154 }
155
156
157 int check_rom_ext_manifest(rom_ext_manifest_t manifest) {
158     if (manifest.identifier == 0)
159         return 0;
160     for (int i = 0; i < RSA_SIZE; i++) {
161         if (manifest.signature.value[i] != 0)
162             return 1; // If the signature[i] != 0 for one i, the manifest is valid.
163     }
164     return 0;
165 }
166
167
168 void mask_rom_boot(boot_policy_t boot_policy, rom_exts_manifests_t rom_exts_to_try ){
169
170     boot_policy_t boot_policy = FLASH_CTRL_read_boot_policy();
171     rom_exts_manifests_t rom_exts_to_try = FLASH_CTRL_rom_ext_manifests_to_try(boot_policy
    ↪ );
172
173     PMP_enable_memory_protection();
174
175     //Step 2.iii
176     for (int i = 0; i < rom_exts_to_try.size; i++) {
177
178         rom_ext_manifest_t current_rom_ext_manifest = rom_exts_to_try.rom_exts_mfs[i];
179
180         signature_t signature = current_rom_ext_manifest.signature;
181
182         if (!check_rom_ext_manifest(current_rom_ext_manifest)) {
183             continue;
184         }
185
186         //Step 2.iii.b
187         pub_key_t rom_ext_pub_key = read_pub_key(current_rom_ext_manifest);
188
189         //Step 2.iii.b
190         if (!check_pub_key_valid(rom_ext_pub_key)) {
191             continue;
192         }
193
194         //Step 2.iii.b
195         if (!verify_rom_ext_signature(rom_ext_pub_key, current_rom_ext_manifest)) {
196             continue;
197         }
198
```

```
199         //Step 2.iii.d
200         PMP_unlock_rom_ext();
201 
202         //Step 2.iii.e
203         if (!final_jump_to_rom_ext(current_rom_ext_manifest)) {
204 
205             //Step 2.iv
206             boot_failed_rom_ext_terminated(boot_policy, current_rom_ext_manifest);
207             return;
208         }
209     } // End for
210 
211     //Step 2.iv
212     boot_failed(boot_policy);
213 }
```

Listing B.2: The content of the mask_rom.c file

B.3 hmac.h

```
1  /*********************************************************************
2  * Filename:   sha256.h
3  * Author:     Brad Conte (brad AT bradconte.com)
4  * Copyright:
5  * Disclaimer: This code is presented "as is" without any guarantees.
6  * Details:    Defines the API for the corresponding SHA1 implementation.
7  *********************************************************************/
8 
9  #ifndef SHA256_H
10 #define SHA256_H
11 
12 /************************* HEADER FILES ************************/
13 #include <stddef.h>
14 #include "mask_rom.h"
15 
16 /**************************** MACROS ******************************/
17 #define SHA2_256_BLOCK_SIZE 32            // SHA256 outputs a 32 byte digest
18 #define HMAC_KEY_SIZE 32                  // HMAC key is 32 bytes
19 /**************************** DATA TYPES ****************************/
20 typedef unsigned char BYTE;             // 8-bit byte
21 typedef unsigned int WORD;              // 32-bit word, change to "long" for 16-bit
       ↪ machines
22 
23 typedef struct {
24     BYTE data[64];
25     WORD datalen;
26     unsigned long long bitlen;
27     WORD state[8];
28 } SHA2_256_CTX;
29 
30 /*********************** FUNCTION DECLARATIONS **********************/
31 void HMAC_SHA2_256_init(SHA2_256_CTX *ctx);
32 void HMAC_SHA2_256_update(SHA2_256_CTX *ctx, const BYTE data[], size_t len);
33 void HMAC_SHA2_256_final(SHA2_256_CTX *ctx, BYTE hash[]);
34 BYTE* HMAC_SHA2_256(BYTE key[], BYTE mes[], int size);
35 
36 #endif  // SHA256_H
```

Listing B.3: The content of the hmac.h file

B.4 hmac.c

```
/*********************************************************************
////////////////////// ORIGINAL FILE ////////////////////////////////
* Filename:   sha2_256.c
* Original
  Author:     Brad Conte (brad AT bradconte.com)
* Copyright:
* Disclaimer: This code is presented "as is" without any guarantees.
* Details:    Implementation of the SHA-256 hashing algorithm.
              SHA-256 is one of the three algorithms in the SHA2
              specification. The others, SHA-384 and SHA-512, are not
              offered in this implementation.
              Algorithm specification can be found here:
               * http://csrc.nist.gov/publications/fips/fips180-2/fips180-2
    ↪ withchangenotice.pdf
              This implementation uses little endian byte order.

* Modified By: Jacob Gosch and Kristoffer Jensen
*********************************************************************/

/************************** HEADER FILES ***************************/
#include <stdlib.h>
#include <memory.h>
#include "hmac.h"
#include "memory_compare.h"
/****************************** MACROS ******************************/
#define ROTLEFT(a,b) (((a) << (b)) | ((a) >> (32-(b))))
#define ROTRIGHT(a,b) (((a) >> (b)) | ((a) << (32-(b))))

#define CH(x,y,z) (((x) & (y)) ^ (~(x) & (z)))
#define MAJ(x,y,z) (((x) & (y)) ^ ((x) & (z)) ^ ((y) & (z)))
#define EP0(x) (ROTRIGHT(x,2) ^ ROTRIGHT(x,13) ^ ROTRIGHT(x,22))
#define EP1(x) (ROTRIGHT(x,6) ^ ROTRIGHT(x,11) ^ ROTRIGHT(x,25))
#define SIG0(x) (ROTRIGHT(x,7) ^ ROTRIGHT(x,18) ^ ((x) >> 3))
#define SIG1(x) (ROTRIGHT(x,17) ^ ROTRIGHT(x,19) ^ ((x) >> 10))

/**************************** VARIABLES *****************************/
static const WORD k[64] = {
    0x428a2f98,0x71374491,0xb5c0fbcf,0xe9b5dba5,0x3956c25b,0x59f111f1,0x923f82a4,0
    ↪ xab1c5ed5,
    0xd807aa98,0x12835b01,0x243185be,0x550c7dc3,0x72be5d74,0x80deb1fe,0x9bdc06a7,0
    ↪ xc19bf174,
    0xe49b69c1,0xefbe4786,0x0fc19dc6,0x240ca1cc,0x2de92c6f,0x4a7484aa,0x5cb0a9dc,0
    ↪ x76f988da,
    0x983e5152,0xa831c66d,0xb00327c8,0xbf597fc7,0xc6e00bf3,0xd5a79147,0x06ca6351,0
    ↪ x14292967,
    0x27b70a85,0x2e1b2138,0x4d2c6dfc,0x53380d13,0x650a7354,0x766a0abb,0x81c2c92e,0
    ↪ x92722c85,
    0xa2bfe8a1,0xa81a664b,0xc24b8b70,0xc76c51a3,0xd192e819,0xd6990624,0xf40e3585,0
    ↪ x106aa070,
    0x19a4c116,0x1e376c08,0x2748774c,0x34b0bcb5,0x391c0cb3,0x4ed8aa4a,0x5b9cca4f,0
    ↪ x682e6ff3,
    0x748f82ee,0x78a5636f,0x84c87814,0x8cc70208,0x90befffa,0xa4506ceb,0xbef9a3f7,0
    ↪ xc67178f2
};

/*********************** FUNCTION DEFINITIONS ***********************/
void HMAC_SHA2_256_transform(SHA2_256_CTX *ctx, const BYTE data[])
{
```

```
50      WORD a, b, c, d, e, f, g, h, i, j, t1, t2, m[64];
51
52      for (i = 0, j = 0; i < 16; ++i, j += 4)
53          m[i] = (data[j] << 24) | (data[j + 1] << 16) | (data[j + 2] << 8) | (data[j + 3]);
54      for ( ; i < 64; ++i)
55          m[i] = SIG1(m[i - 2]) + m[i - 7] + SIG0(m[i - 15]) + m[i - 16];
56
57      a = ctx->state[0];
58      b = ctx->state[1];
59      c = ctx->state[2];
60      d = ctx->state[3];
61      e = ctx->state[4];
62      f = ctx->state[5];
63      g = ctx->state[6];
64      h = ctx->state[7];
65
66      for (i = 0; i < 64; ++i) {
67          t1 = h + EP1(e) + CH(e,f,g) + k[i] + m[i];
68          t2 = EP0(a) + MAJ(a,b,c);
69          h = g;
70          g = f;
71          f = e;
72          e = d + t1;
73          d = c;
74          c = b;
75          b = a;
76          a = t1 + t2;
77      }
78
79      ctx->state[0] += a;
80      ctx->state[1] += b;
81      ctx->state[2] += c;
82      ctx->state[3] += d;
83      ctx->state[4] += e;
84      ctx->state[5] += f;
85      ctx->state[6] += g;
86      ctx->state[7] += h;
87  }
88
89  void HMAC_SHA2_256_init(SHA2_256_CTX *ctx)
90  {
91      ctx->datalen = 0;
92      ctx->bitlen = 0;
93      ctx->state[0] = 0x6a09e667;
94      ctx->state[1] = 0xbb67ae85;
95      ctx->state[2] = 0x3c6ef372;
96      ctx->state[3] = 0xa54ff53a;
97      ctx->state[4] = 0x510e527f;
98      ctx->state[5] = 0x9b05688c;
99      ctx->state[6] = 0x1f83d9ab;
100     ctx->state[7] = 0x5be0cd19;
101 }
102
103 void HMAC_SHA2_256_update(SHA2_256_CTX *ctx, const BYTE data[], size_t len)
104 {
105     WORD i;
106
107     for (i = 0; i < len; ++i) {
108         ctx->data[ctx->datalen] = data[i];
109         ctx->datalen++;
110         if (ctx->datalen == 64) {
```

```
111             HMAC_SHA2_256_transform(ctx, ctx->data);
112             ctx->bitlen += 512;
113             ctx->datalen = 0;
114         }
115     }
116 }
117 
118 void HMAC_SHA2_256_final(SHA2_256_CTX *ctx, BYTE hash[])
119 {
120     WORD i;
121 
122     i = ctx->datalen;
123 
124     // Pad whatever data is left in the buffer.
125     if (ctx->datalen < 56) {
126         ctx->data[i++] = 0x80;
127         while (i < 56)
128             ctx->data[i++] = 0x00;
129     }
130     else {
131         ctx->data[i++] = 0x80;
132         while (i < 64)
133             ctx->data[i++] = 0x00;
134         HMAC_SHA2_256_transform(ctx, ctx->data);
135         memset(ctx->data, 0, 56);
136     }
137 
138     // Append to the padding the total message's length in bits and transform.
139     ctx->bitlen += ctx->datalen * 8;
140     ctx->data[63] = ctx->bitlen;
141     ctx->data[62] = ctx->bitlen >> 8;
142     ctx->data[61] = ctx->bitlen >> 16;
143     ctx->data[60] = ctx->bitlen >> 24;
144     ctx->data[59] = ctx->bitlen >> 32;
145     ctx->data[58] = ctx->bitlen >> 40;
146     ctx->data[57] = ctx->bitlen >> 48;
147     ctx->data[56] = ctx->bitlen >> 56;
148     HMAC_SHA2_256_transform(ctx, ctx->data);
149 
150     // Since this implementation uses little endian byte ordering and SHA uses big endian,
151     // reverse all the bytes when copying the final state to the output hash.
152     for (i = 0; i < 4; ++i) {
153         hash[i]      = (ctx->state[0] >> (24 - i * 8)) & 0x000000ff;
154         hash[i + 4]  = (ctx->state[1] >> (24 - i * 8)) & 0x000000ff;
155         hash[i + 8]  = (ctx->state[2] >> (24 - i * 8)) & 0x000000ff;
156         hash[i + 12] = (ctx->state[3] >> (24 - i * 8)) & 0x000000ff;
157         hash[i + 16] = (ctx->state[4] >> (24 - i * 8)) & 0x000000ff;
158         hash[i + 20] = (ctx->state[5] >> (24 - i * 8)) & 0x000000ff;
159         hash[i + 24] = (ctx->state[6] >> (24 - i * 8)) & 0x000000ff;
160         hash[i + 28] = (ctx->state[7] >> (24 - i * 8)) & 0x000000ff;
161     }
162 }
163 
164 BYTE* HMAC_SHA2_256(BYTE key[], BYTE mes[], int mes_size){
165 
166     BYTE* buff = malloc(SHA2_256_BLOCK_SIZE * sizeof(BYTE));
167     SHA2_256_CTX ctx;
168 
169     BYTE* key_mes_pad = malloc(HMAC_KEY_SIZE * sizeof(BYTE) + mes_size * sizeof(BYTE)); //
    ↪ key âĹě mes
170     memcpy(
```

```
171         key_mes_pad,
172         key,
173         HMAC_KEY_SIZE
174     );
175     memcpy(
176         key_mes_pad + HMAC_KEY_SIZE,
177         mes,
178         mes_size
179     );
180 
181     HMAC_SHA2_256_init(&ctx);
182     HMAC_SHA2_256_update(&ctx, key_mes_pad, HMAC_KEY_SIZE + mes_size);
183     HMAC_SHA2_256_final(&ctx, buff);
184 
185     return buff;
186 }
```

Listing B.4: The content of the hmac.c file

Appendix C

CBMC Annotated Boot Code

This chapter contains the boot code, CBMC annotations, and proof harness used for verifying property 0-10 mentioned in Section 4.3. It is based on the boot code in Appendix A. The content of this chapter is the files found on GitHub at [48].

C.1 mask_rom.h

```
#ifndef MASK_ROM_H
#define MASK_ROM_H

#include <string.h>
#include <stdint.h>
#include <malloc.h>
#include <stdlib.h>
#include <memory.h>

#define __REACHABILITY_CHECK __CPROVER_assert(0, "Reachability check, should always
    ↪ \033[0;31mFAIL\033[0m");
#define MAX_ROM_EXTS 1
#define RSA_SIZE 96
#define PMP_REGIONS 16
#define MAX_IMAGE_LENGTH 2 //necessary constraint in order to terminate CBMC verification


//Represents a signature. Needed for CBMC OBJECT_SIZE to see if signature is of ok size
typedef struct signature_t{
    int32_t value[RSA_SIZE];
    //something else
} signature_t;


//Represents a public key
typedef struct pub_key_t{
    int32_t exponent;
    int32_t modulus[RSA_SIZE];
    //something else
} pub_key_t;


//Struct representing rom_ext_manifest
typedef struct rom_ext_manifest_t{
    uint32_t identifier;

    signature_t signature;

    //public part of signature key
    pub_key_t pub_signature_key;
    uint32_t image_length;
    char* image_code;
} rom_ext_manifest_t;
```

```
43
44
45 //Returned by rom_ext_manifests_to_try
46 typedef struct rom_exts_manifests_t {
47     int size;
48     rom_ext_manifest_t rom_exts_mfs[MAX_ROM_EXTS];
49 } rom_exts_manifests_t;
50
51
52 //Represents boot policy
53 typedef struct boot_policy_t {
54     int identifier;
55
56     //which rom_ext_slot to boot
57     int rom_ext_slot;
58
59     //what to do if all ROM Ext are invalid
60     char* fail;
61
62     //what to do if the ROM Ext unexpectedly returns
63     char* fail_rom_ext_terminated;
64
65 } boot_policy_t;
66
67
68
69 //Represents a pmp region
70 typedef struct __PMP_region_t {
71     int identifier;
72
73     //Locked, Read, Write, Execute
74     int R;
75     int W;
76     int E;
77     int L;
78
79 } __PMP_region_t;
80
81
82 typedef struct __PMP_regions_t {
83     //There are 16 PMP regions (0...15)
84     __PMP_region_t pmp_regions[PMP_REGIONS];
85 } __PMP_regions_t;
86
87 #endif
```

Listing C.1: The content of the mask_rom.h file

C.2 mask_rom.c

```
1 /*
2 CBMC Verification of OpenTitan bootcode,
3 written based on:
4 sw/device/rom_ext/docs/manifest.md
5 sw/device/mask_rom/mask_rom.c
6 sw/device/mask_rom/docs/index.md
7 doc/security/specs/secure_boot/index.md
8 */
9
```

```
10 #include "hmac.h"
11 #include "mask_rom.h"
12 #include "memory_compare.h"
13
14 //HMAC key in OTP/Keymanager
15 BYTE __hmac_key[HMAC_KEY_SIZE];
16
17 //Whitelist in ROM
18 #define __PKEY_WHITELIST_SIZE 1
19 pub_key_t __pkey_whitelist[__PKEY_WHITELIST_SIZE];
20
21 //for CBMC
22 int __current_rom_ext = 0;
23 rom_ext_manifest_t __current_rom_ext_mf;
24 int __boot_policy_stop = 0;
25 int __rom_ext_called[MAX_ROM_EXTS] = { }; //used for CBMC postcondition
26 int __rom_ext_fail_func[MAX_ROM_EXTS] = { }; //for CBMC PROPERTY 6
27 int __boot_failed_called[MAX_ROM_EXTS] = { };
28 int __validated_rom_exts[MAX_ROM_EXTS] = { }; //used for CBMC postcondition
29 int __rom_ext_returned[MAX_ROM_EXTS] = { }; //used for CBMC postcondition
30 int __verify_signature_called[MAX_ROM_EXTS] = { };
31 int __imply(int a, int b) { return a ? b : 1; }
32 int __valid_signature[MAX_ROM_EXTS] = { }; //result of verify_rom_ext_signature
33
34
35 //The configured PMP regions for each rom ext.
36 __PMP_regions_t __rom_ext_pmp_region[MAX_ROM_EXTS];
37
38 // Function type used to define function pointer to the entry of the ROM_EXT stage.
39 typedef void(rom_ext_boot_func)(void);
40
41 // Function type for entry point of boot policy fail function
42 typedef void(fail_func)(void);
43
44 // Function type for entry point of boot policy fail rom ext terminated function.
45 typedef void(fail_rom_ext_terminated_func)(rom_ext_manifest_t);
46
47
48 int verify_rom_ext_signature(pub_key_t rom_ext_pub_key, rom_ext_manifest_t manifest) {
49     __CPROVER_precondition(MAX_IMAGE_LENGTH >= manifest.image_length && manifest.
    ↪ image_length > 0,
50         "Precondition: Assumes rom ext image code is < 10 and > 0");
51
52     __CPROVER_precondition(__CPROVER_r_ok(manifest.image_code, manifest.image_length),
53         "Precondition: Assumes rom ext image code is readable");
54
55     __verify_signature_called[__current_rom_ext] = 1;
56
57     int bytes =
58         sizeof(manifest.pub_signature_key) + sizeof(manifest.image_length) + manifest.
    ↪ image_length;
59
60     char message[bytes];
61
62     memcpy(
63         message,
64         &manifest.pub_signature_key,
65         sizeof(manifest.pub_signature_key)
66     );
67     memcpy(
68         message + sizeof(manifest.pub_signature_key),
```

```
69          &manifest.image_length,
70          sizeof(manifest.image_length)
71      );
72      memcpy(
73          message + sizeof(manifest.pub_signature_key) + sizeof(manifest.image_length),
74          manifest.image_code,
75          manifest.image_length
76      );
77
78      //Otherwise OBJECT_SIZE returns size of manifest and not signature.
79      signature_t signature = manifest.signature;
80
81      int result = OTBN_RSASSA_PKCS1_V1_5_VERIFY(rom_ext_pub_key.exponent, rom_ext_pub_key.
        ↪ modulus, message, bytes, signature.value, RSA_SIZE, manifest);
82
83      return result; //0 or 1
84  }
85
86
87  int __is_valid_params(int32_t exponent, int32_t* modulus, char* message, int message_len,
        ↪ int32_t* signature, int signature_len, rom_ext_manifest_t __current_rom_ext_mf) {
88
89      if (exponent != __current_rom_ext_mf.pub_signature_key.exponent)
90          return 0;
91
92      if (cmp_modulus(modulus,
93          __current_rom_ext_mf.pub_signature_key.modulus,
94          RSA_SIZE*sizeof(int32_t)) != 0)
95          return 0;
96
97      if (cmp_signature(signature,
98          __current_rom_ext_mf.signature.value,
99          RSA_SIZE * sizeof(int32_t)) != 0)
100         return 0;
101
102     //Message is: pkey+image_length+image_code
103     if (cmp_key(message,
104         &__current_rom_ext_mf.pub_signature_key,
105         sizeof(__current_rom_ext_mf.pub_signature_key)) != 0)
106         return 0;
107
108     if (cmp_image_len(
109         message + sizeof(__current_rom_ext_mf.pub_signature_key),
110         &__current_rom_ext_mf.image_length,
111         sizeof(__current_rom_ext_mf.image_length)) != 0)
112         return 0;
113
114     if (cmp_image_code(
115         message + sizeof(__current_rom_ext_mf.pub_signature_key) + sizeof(
        ↪ __current_rom_ext_mf.image_length),
116         __current_rom_ext_mf.image_code,
117         __current_rom_ext_mf.image_length) != 0)
118         return 0;
119
120
121     return 1;
122 }
123
124
125 char* OTBN_RSA_3072_DECRYPT(int32_t* signature, int signature_len, int32_t exponent,
        ↪ int32_t* modulus) {
```

```
126     char* decrypt = malloc(256 / 8); //model it to be ok for PROPERTY 5
127     return decrypt;
128 }
129
130 int OTBN_RSASSA_PKCS1_V1_5_VERIFY(int32_t exponent, int32_t* modulus, char* message, int
        ↪ message_len, int32_t* signature, int signature_len, rom_ext_manifest_t
        ↪ __current_rom_ext_mf) {
131     __CPROVER_assert(__CPROVER_OBJECT_SIZE(message) == message_len,
132         "PROPERTY 5: Formal parameter message_len lenght matches actual message length.");
133
134     __CPROVER_assert(__CPROVER_OBJECT_SIZE(signature) == 3072 / 8,
135         "PROPERTY 5: Signature to be verified is 3072-bits.");
136
137     __CPROVER_assert(__CPROVER_OBJECT_SIZE(signature) == signature_len * sizeof(int32_t),
138         "PROPERTY 5: Formal parameter signature lenght matches actual signature length.");
139
140     __CPROVER_assert(sizeof(exponent) == 32 / 8,
141         "PROPERTY 5: Public key exponent is 32 bits.");
142
143     __CPROVER_assert((sizeof(pub_key_t) - sizeof(exponent)) == 3072 / 8,
144         "PROPERTY 5: Public key modulus is 3072-bits.");
145
146     __CPROVER_assert(__is_valid_params(exponent, modulus, message, message_len, signature,
147         signature_len, __current_rom_ext_mf),
148         "PROPERTY 5: Check that key, signature, and message matches those from the
        ↪ manifest.");
149
150     __REACHABILITY_CHECK
151
152     if (signature_len != RSA_SIZE) {
153         __CPROVER_assert(signature_len * 32 != 3072,
154             "PROPERTY 5: Length checking: If the length of the signature is not 3072-bits,
        ↪  stop.");
155         __REACHABILITY_CHECK // Not reachable atm
156
157             return 0;
158     }
159     __REACHABILITY_CHECK
160
161     char* decrypt = OTBN_RSA_3072_DECRYPT(signature, signature_len, exponent, modulus);
162     char* hash = HMAC_SHA2_256(__hmac_key, message, message_len, __current_rom_ext_mf); //
        ↪ message_len in bytes
163
164     __CPROVER_assert(!__CPROVER_array_equal(decrypt, signature),
165         "PROPERTY 5: Decrypted signature is different from signature");
166
167     __CPROVER_assert(!__CPROVER_array_equal(hash, message),
168         "PROPERTY 5: Hash is different from original message");
169
170     __CPROVER_assert(__CPROVER_OBJECT_SIZE(decrypt) == 256 / 8,
171         "PROPERTY 5: Decrypted message is 256 bits");
172
173     __CPROVER_assert(__CPROVER_r_ok(decrypt, 256 / 8),
174         "PROPERTY 5: Decrypted message is in readable address");
175
176     __CPROVER_assert(__CPROVER_OBJECT_SIZE(hash) == 256 / 8,
177         "PROPERTY 3: Hash is 256 bits");
178
179     __CPROVER_assert(__CPROVER_r_ok(hash, 256 / 8),
180         "PROPERTY 3: hash is in readable address");
181
```

```
182     if (cmp_hash_decrypt(hash, decrypt, 256 / 8) == 0){
183         __valid_signature[__current_rom_ext] = 1;
184         return 1; //verified
185     }
186     else{
187         __valid_signature[__current_rom_ext] = 0;
188         return 0;
189     }
190 }
191 
192 
193 boot_policy_t FLASH_CTRL_read_boot_policy() {}
194 
195 
196 rom_exts_manifests_t FLASH_CTRL_rom_ext_manifests_to_try(boot_policy_t boot_policy) {}
197 
198 
199 pub_key_t read_pub_key(rom_ext_manifest_t current_rom_ext_manifest) {
200     return current_rom_ext_manifest.pub_signature_key;
201 }
202 
203 //Mocked function for reading pkey whitelist from maskrom.
204 pub_key_t* ROM_CTRL_get_whitelist() {
205     return __pkey_whitelist;
206 }
207 
208 
209 extern int check_pub_key_valid(pub_key_t rom_ext_pub_key){ //assumed behavior behavior of
    ↪ check func
210     pub_key_t* pkey_whitelist = ROM_CTRL_get_whitelist();
211 
212     for (int i = 0; i < __PKEY_WHITELIST_SIZE; i++) {
213         if (pkey_whitelist[i].exponent != rom_ext_pub_key.exponent)
214             continue;
215 
216         int j = 0;
217         for (j = 0; j < RSA_SIZE; j++) {
218             if (pkey_whitelist[i].modulus[j] != rom_ext_pub_key.modulus[j])
219                 break;
220         }
221 
222         //if j == RSA_SIZE, then loop ran to completion and all entries were equal
223         if (j == RSA_SIZE)
224             return 1;
225     }
226 
227     return 0;
228 }
229 
230 
231 extern void PMP_WRITE_REGION(uint8_t reg, uint8_t r, uint8_t w, uint8_t e, uint8_t l){
232     __REACHABILITY_CHECK
233 }
234 
235 
236 void PMP_unlock_rom_ext() {
237     //Read, Execute, Locked the address space of the ROM extension image
238     PMP_WRITE_REGION(        0,        1,        0,        1,        1);
239     //                   Region      Read     Write    Execute     Locked
240     __register_pmp_region(__current_rom_ext, 0, 1, 0, 1, 1);
241     __REACHABILITY_CHECK
```

```
242 }
243
244
245 void PMP_enable_memory_protection() {
246     //Apply PMP region 15 to cover entire flash
247     PMP_WRITE_REGION(       15,         1,         0,         0,         1);
248     //                   Region      Read     Write    Execute    Locked
249
250     __register_pmp_region(-1, 15, 1, 0, 0, 1);
251     __REACHABILITY_CHECK
252 }
253
254
255 void __register_pmp_region(int rom_ext, int pmp_id, int r, int w, int e, int l){
256     if (rom_ext == -1) {
257         //register PMP region for all rom exts.
258         for (int i = 0; i < MAX_ROM_EXTS; i++) {
259             __rom_ext_pmp_region[i].pmp_regions[pmp_id].identifier = pmp_id;
260             __rom_ext_pmp_region[i].pmp_regions[pmp_id].R = r;
261             __rom_ext_pmp_region[i].pmp_regions[pmp_id].W = w;
262             __rom_ext_pmp_region[i].pmp_regions[pmp_id].E = e;
263             __rom_ext_pmp_region[i].pmp_regions[pmp_id].L = l;
264         }
265     }
266     else {
267         __rom_ext_pmp_region[rom_ext].pmp_regions[pmp_id].identifier = pmp_id;
268         __rom_ext_pmp_region[rom_ext].pmp_regions[pmp_id].R = r;
269         __rom_ext_pmp_region[rom_ext].pmp_regions[pmp_id].W = w;
270         __rom_ext_pmp_region[rom_ext].pmp_regions[pmp_id].E = e;
271         __rom_ext_pmp_region[rom_ext].pmp_regions[pmp_id].L = l;
272     }
273 }
274
275
276 void __some_entry_func() { __rom_ext_called[__current_rom_ext] = 1; /*for CBMC PROPERTY 6
    ↪ */ }
277
278
279 int final_jump_to_rom_ext(rom_ext_manifest_t current_rom_ext_manifest) { // Returns a
    ↪ boolean value.
280     //This assumption causes reachability checks to succeed. If == is =, then they fail.
281     //__CPROVER_assume(current_rom_ext_manifest.image_code == &__some_entry_func); //for
    ↪ cbmc
282     current_rom_ext_manifest.image_code = &__some_entry_func;
283     //Execute rom ext code step 2.iii.e
284     rom_ext_boot_func* rom_ext_entry = (rom_ext_boot_func*)current_rom_ext_manifest.
    ↪ image_code;
285
286     __CPROVER_assert(rom_ext_entry == current_rom_ext_manifest.image_code,
287     "PROPERTY 6: Correct entry point address.");
288
289     __REACHABILITY_CHECK
290
291     rom_ext_entry();
292
293     __rom_ext_returned[__current_rom_ext] = 1; //for CBMC PROPERTY 6
294
295     //if rom_ext returns, we should return false
296     //and execute step 2.iv.
297     return 0;
298 }
```

```
299
300
301 void boot_failed(boot_policy_t boot_policy) {
302     __REACHABILITY_CHECK
303     fail_func* fail_func_entry = (fail_func*)boot_policy.fail;
304     fail_func_entry();
305 }
306
307
308 void boot_failed_rom_ext_terminated(boot_policy_t boot_policy, rom_ext_manifest_t
    ↪ current_rom_ext_manifest) {
309     __REACHABILITY_CHECK
310     fail_rom_ext_terminated_func* fail_func_entry = (fail_rom_ext_terminated_func*)
    ↪ boot_policy.fail_rom_ext_terminated;
311     fail_func_entry(current_rom_ext_manifest);
312 }
313
314
315 int check_rom_ext_manifest(rom_ext_manifest_t manifest) {
316     if (manifest.identifier == 0)
317         return 0;
318     for (int i = 0; i < RSA_SIZE; i++) {
319         if (manifest.signature.value[i] != 0)
320             return 1; // If the signature[i] != 0 for one i, the manifest is valid.
321     }
322     return 0;
323 }
324
325
326 int __help_check_rom_ext_manifest(rom_ext_manifest_t manifest) { //used for CBMC assertion
    ↪  + postcondition
327     if (manifest.identifier == 0)
328         return 0;
329
330     signature_t signature = manifest.signature; //needed to take object size of signature
    ↪ and not entire manifest
331
332     if (__CPROVER_OBJECT_SIZE(signature.value) != 3072 / 8) //Signature must be 3072-bits
333         return 0;
334
335     for (int i = 0; i < RSA_SIZE; i++) {
336         if (manifest.signature.value[i] != 0)
337             return 1;
338     }
339     return 0;
340 }
341
342
343 int __help_pkey_valid(pub_key_t pkey) { //used for CBMC assertion + postcondition
344     // Public key exponent must be 32 bits.");
345     if(sizeof(pkey.exponent) * 8 != 32)
346         return 0;
347     // Public key modulus must be 3072-bits.");
348     if((sizeof(pkey) - sizeof(pkey.exponent)) * 8 != 3072)
349         return 0;
350
351     pub_key_t* pkey_whitelist = ROM_CTRL_get_whitelist();
352
353     for (int i = 0; i < __PKEY_WHITELIST_SIZE; i++) {
354         if (pkey_whitelist[i].exponent != pkey.exponent)
355             continue;
```

```
356
357         int j = 0;
358         for (j = 0; j < RSA_SIZE; j++) {
359             if (pkey_whitelist[i].modulus[j] != pkey.modulus[j])
360                 break;
361         }
362
363         //if j == RSA_SIZE, then loop ran to completion and all entries were equal
364         if (j == RSA_SIZE)
365             return 1;
366     }
367
368     return 0;
369 }
370
371
372 int __help_check_pmp_region(int rom_ext, int pmp_id, int r, int w, int e, int l) {
373     if (rom_ext == -1) {
374         //When we need to check a PMP region for all rom exts
375         for (int i = 0; i < MAX_ROM_EXTS; i++) {
376             // If something is wrong return 0
377             if (__rom_ext_pmp_region[i].pmp_regions[pmp_id].R != r    ||
378                 __rom_ext_pmp_region[i].pmp_regions[pmp_id].W != w      ||
379                 __rom_ext_pmp_region[i].pmp_regions[pmp_id].E != e      ||
380                 __rom_ext_pmp_region[i].pmp_regions[pmp_id].L != l)
381                 return 0;
382         }
383         return 1;
384     }
385     else {
386         return __rom_ext_pmp_region[rom_ext].pmp_regions[pmp_id].R == r &&
387             __rom_ext_pmp_region[rom_ext].pmp_regions[pmp_id].W == w &&
388             __rom_ext_pmp_region[rom_ext].pmp_regions[pmp_id].E == e &&
389             __rom_ext_pmp_region[rom_ext].pmp_regions[pmp_id].L == l;
390     }
391 }
392
393
394 int __help_all_pmp_inactive(){
395     //Inactive if all fields are 0.
396     for (int i = 0; i < MAX_ROM_EXTS; i++) {
397         for (int j = 0; j < PMP_REGIONS; j++) {
398             if (__rom_ext_pmp_region[i].pmp_regions[j].R != 0 ||
399                 __rom_ext_pmp_region[i].pmp_regions[j].W != 0 ||
400                 __rom_ext_pmp_region[i].pmp_regions[j].E != 0 ||
401                 __rom_ext_pmp_region[i].pmp_regions[j].L != 0)
402                 return 0;
403         }
404     }
405     return 1;
406 }
407
408 void __func_fail() { __boot_failed_called[__current_rom_ext] = 1; } //used for CBMC
409 void __func_fail_rom_ext(rom_ext_manifest_t _) { __rom_ext_fail_func[__current_rom_ext] =
    ↪ 1; } //used for CBMC
410
411 void PROOF_HARNESS() {
412     boot_policy_t boot_policy = FLASH_CTRL_read_boot_policy();
413     rom_exts_manifests_t rom_exts_to_try = FLASH_CTRL_rom_ext_manifests_to_try(boot_policy
    ↪ );
414
```

```
415     __CPROVER_assume(rom_exts_to_try.size <= MAX_ROM_EXTS && rom_exts_to_try.size > 0);
416
417     __CPROVER_assume(boot_policy.fail == &__func_fail);
418     __CPROVER_assume(boot_policy.fail_rom_ext_terminated == &__func_fail_rom_ext);
419
420     for(int i = 0; i < rom_exts_to_try.size; i++){
421         __CPROVER_assume(MAX_IMAGE_LENGTH >= rom_exts_to_try.rom_exts_mfs[i].image_length
    ↪ && rom_exts_to_try.rom_exts_mfs[i].image_length > 0);
422         rom_exts_to_try.rom_exts_mfs[i].image_code = malloc(sizeof(char) * rom_exts_to_try
    ↪ .rom_exts_mfs[i].image_length);
423     }
424
425     mask_rom_boot(boot_policy, rom_exts_to_try);
426
427
428     __CPROVER_postcondition(__current_rom_ext + 1 <= rom_exts_to_try.size,
429     "Postcondition: Should never check more rom_ext than there exist");
430
431     for (int i = 0; i < rom_exts_to_try.size; i++) {
432
433         __CPROVER_postcondition(__imply(!__help_check_rom_ext_manifest(rom_exts_to_try.
    ↪ rom_exts_mfs[i]) ||
434                                 !__help_pkey_valid(rom_exts_to_try.rom_exts_mfs[i].
    ↪ pub_signature_key),
435                                 !__verify_signature_called[i]),
436         "Postcondition PROPERTY 5: If identifier, sign, or key is invalid then verify
    ↪ signature function is not called");
437
438         __CPROVER_postcondition(__imply(__help_check_rom_ext_manifest(rom_exts_to_try.
    ↪ rom_exts_mfs[i]) &&
439                                 __help_pkey_valid(rom_exts_to_try.rom_exts_mfs[i].
    ↪ pub_signature_key),
440                                 __verify_signature_called[i]),
441         "Postcondition PROPERTY 5: If identifier, sign, and key are valid then the
    ↪ signature verification function is called");
442
443         if (__validated_rom_exts[i]) { //validated - try to boot from
444             __REACHABILITY_CHECK
445
446             __CPROVER_postcondition(__help_check_rom_ext_manifest(rom_exts_to_try.
    ↪ rom_exts_mfs[i]),
447             "Postcondition PROPERTY 1: rom_ext VALIDATED => valid signature");
448
449             __CPROVER_postcondition(__help_pkey_valid(rom_exts_to_try.rom_exts_mfs[i].
    ↪ pub_signature_key),
450             "Postcondition PROPERTY 2: rom_ext VALIDATED => valid key");
451
452             __CPROVER_postcondition(__verify_signature_called[i],
453             "Postcondition PROPERTY 5: iff manifest is valid then verify signature
    ↪ function is called");
454
455             __CPROVER_postcondition(__valid_signature[i],
456             "Postcondition PROPERTY 5: rom_ext VALIDATED => signature valid");
457
458             __CPROVER_postcondition(__rom_ext_called[i],
459             "Postcondition PROPERTY 6: rom_ext VALIDATED => rom ext code inititated");
460
461             __CPROVER_postcondition(__imply(__rom_ext_returned[i], __rom_ext_fail_func[i])
    ↪ ,
462             "Postcondition PROPERTY 6: (valid rom _ext and rom_ext code return) => that
    ↪ rom_ext term func is called");
```

```
463
464             __CPROVER_postcondition(__imply(!__rom_ext_returned[i], !__rom_ext_fail_func[i
    ↪ ]),
465             "Postcondition PROPERTY 6: (valid rom _ext and rom_ext code !return) => that
    ↪ rom_ext term func not called");
466
467             __CPROVER_postcondition(__help_check_pmp_region(i, 15, 1, 0, 0, 1),
468             "Postcondition PROPERTY 9: PMP region 15 should be R and L, when rom_ext was
    ↪ validated.");
469
470             __CPROVER_postcondition(__help_check_pmp_region(i, 0, 1, 0, 1, 1),
471             "Postcondition PROPERTY 10: If rom_ext was valided, then PMP region 0 should
    ↪ be R, E, and L.");
472
473         }
474         else { //invalidated - unsafe to boot from
475             __REACHABILITY_CHECK
476
477             __CPROVER_postcondition(!__help_check_rom_ext_manifest(rom_exts_to_try.
    ↪ rom_exts_mfs[i]) ||
478                                     !__help_pkey_valid(rom_exts_to_try.rom_exts_mfs[i].
    ↪ pub_signature_key) ||
479                                     !__valid_signature[i],
480             "Postcondition: rom_ext INVALIDATED => identifier, signature, or key is
    ↪ invalid");
481
482             __CPROVER_postcondition(!__valid_signature[i],
483             "Postcondition PROPERTY 5: rom_ext INVALIDATED => signature invalid or not
    ↪ checked");
484
485             __CPROVER_postcondition(!__rom_ext_fail_func[i],
486             "Postcondition PROPERTY 6: invalid rom_ext => that rom_ext term func not
    ↪ called");
487
488             __CPROVER_postcondition(!__rom_ext_called[i],
489             "Postcondition PROPERTY 7: rom_ext INVALIDATED => rom ext code not executed");
490
491             __CPROVER_postcondition(__current_rom_ext > i || (i + 1) == rom_exts_to_try.
    ↪ size || __boot_policy_stop,
492             "Postcondition PROPERTY 7: rom_ext INVALIDATED => we check the next rom_ext if
    ↪  any left and no boot policy instructed stop");
493
494             __CPROVER_postcondition(__imply(i < __current_rom_ext, !__boot_failed_called[i
    ↪ ]),
495             "Postcondition PROPERTY 8: A rom_ext (not the last one) fails => fail func is
    ↪ not called");
496
497             __CPROVER_postcondition(__imply(i == __current_rom_ext, __boot_failed_called[i
    ↪ ]),
498             "Postcondition PROPERTY 8: Last rom_ext fail => fail func has been called");
499
500             __CPROVER_postcondition(__help_check_pmp_region(i, 15, 1, 0, 0, 1),
501             "Postcondition PROPERTY 9: PMP region 15 should be R and L. Even if rom_ext
    ↪ was invalidated.");
502
503             __CPROVER_postcondition(__help_check_pmp_region(i, 0, 0, 0, 0, 0),
504             "Postcondition PROPERTY 10: If rom_ext was invalid, PMP region 0 should not be
    ↪  R, E, W, and L.");
505         }
506     }
507     __REACHABILITY_CHECK
```

```
508 }
509
510 /*
511 PROPERTY 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 (mocked hmac)
512
513 RSA_SIZE = 96
514 Run:
515 cbmc mask_rom.c mock_hmac.c memory_compare.c --function PROOF_HARNESS --unwind 97 --
      ↪ unwindset cmp_key.0:390 --unwindset cmp_image_len.0:5 --unwindset cmp_image_code
      ↪ .0:3 --unwindset cmp_modulus.0:385 --unwindset cmp_signature.0:385 --unwindset
      ↪ cmp_has_decrypt.0:33 --unwindset mask_rom_boot.0:2 --unwindset PROOF_HARNESS.0:2 --
      ↪ unwinding-assertions --pointer-check --bounds-check --object-bits 9
516
517 RSA_SIZE = 5 (SPEEDS UP VERIFICAITON) (note. remember to set in mask_rom.h)
518 Run:
519 cbmc mask_rom.c mock_hmac.c memory_compare.c --function PROOF_HARNESS --unwind 33 --
      ↪ unwindset memcmp.0:25 --unwindset mask_rom_boot.0:2 --unwindset PROOF_HARNESS.0:2
      ↪ --unwinding-assertions --pointer-check --bounds-check
520
521
522 PROPERTY 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
523 RSA_SIZE = 96
524 Run:
525 cbmc mask_rom.c hmac.c memory_compare.c --function PROOF_HARNESS --unwind 97 --unwindset
      ↪ cmp_key.0:390 --unwindset cmp_image_len.0:5 --unwindset cmp_image_code.0:3 --
      ↪ unwindset cmp_modulus.0:385 --unwindset cmp_signature.0:385 --unwindset
      ↪ cmp_has_decrypt.0:33 --unwindset mask_rom_boot.0:2 --unwindset HMAC_SHA2_256_update
      ↪ .0:431 --unwindset PROOF_HARNESS.0:2 --unwinding-assertions --pointer-check --
      ↪ bounds-check --object-bits 9
526 */
527
528
529 void mask_rom_boot(boot_policy_t boot_policy, rom_exts_manifests_t rom_exts_to_try ){
530     __CPROVER_precondition(rom_exts_to_try.size <= MAX_ROM_EXTS && rom_exts_to_try.size >
      ↪ 0,
531     "Precondition: Assumes MAX_ROM_EXTS >= rom_exts > 0");
532
533     __CPROVER_precondition(boot_policy.fail == &__func_fail,
534     "Precondition: Assumes boot_policy.fail has ok address");
535
536     __CPROVER_precondition(boot_policy.fail_rom_ext_terminated == &__func_fail_rom_ext,
537     "Precondition: Assumes boot_policy.fail_rom_ext_terminated has ok address");
538
539     //All pmp regions should be inactive at this point
540     __CPROVER_precondition(__help_all_pmp_inactive(),
541     "Precondition PROPERTY 9: All PMP regions should be unset at beginning of mask_rom.");
542
543     PMP_enable_memory_protection();
544
545     //Step 2.iii
546     for (int i = 0; i < rom_exts_to_try.size; i++) {
547
548         __CPROVER_assert(__help_check_pmp_region(i, 15, 1, 0, 0, 1),
549         "PROPERTY 9: PMP region 15 should be R and L.");
550
551         __current_rom_ext = i;
552         rom_ext_manifest_t current_rom_ext_manifest = rom_exts_to_try.rom_exts_mfs[i];
553
554         signature_t __signature = current_rom_ext_manifest.signature; //needed for
      ↪ __CPROVER_OBJECT_SIZE
555
```

```
556
557         if (!check_rom_ext_manifest(current_rom_ext_manifest)) {
558             __REACHABILITY_CHECK
559
560             __CPROVER_assert(!__help_check_rom_ext_manifest(current_rom_ext_manifest),
561             "PROPERTY 1: Stop verification if signature or identifier is invalid");
562
563             continue;
564         }
565
566         __REACHABILITY_CHECK
567
568         __CPROVER_assert(__help_check_rom_ext_manifest(current_rom_ext_manifest),
569         "PROPERTY 1: Continue verification if signature and identifier are valid");
570
571         //Step 2.iii.b
572         pub_key_t rom_ext_pub_key = read_pub_key(current_rom_ext_manifest);
573
574         //Step 2.iii.b
575         if (!check_pub_key_valid(rom_ext_pub_key)) {
576             __REACHABILITY_CHECK
577
578             __CPROVER_assert(!__help_pkey_valid(rom_ext_pub_key),
579             "PROPERTY 2: Stop verification if key is invalid");
580
581             continue;
582         }
583
584         __REACHABILITY_CHECK
585
586         __CPROVER_assert(__help_pkey_valid(rom_ext_pub_key),
587         "PROPERTY 2: Continue verification if key is valid");
588
589         //Step 2.iii.b
590         if (!verify_rom_ext_signature(rom_ext_pub_key, current_rom_ext_manifest)) {
591             __REACHABILITY_CHECK
592             __CPROVER_assert(!__valid_signature[i],
593             "PROPERTY 5: Stop verification if signature is invalid");
594             continue;
595         }
596         __REACHABILITY_CHECK
597
598         __CPROVER_assert(__valid_signature[i],
599         "PROPERTY 5: Continue verification if signature is valid");
600         __validated_rom_exts[i] = 1; //for CBMC
601
602         //Step 2.iii.d
603         PMP_unlock_rom_ext();
604
605         __CPROVER_assert(__help_check_pmp_region(i, 0, 1, 0, 1, 1),
606         "PROPERTY 10: PMP region 0 should be R, E, and L.");
607
608         //Step 2.iii.e
609         if (!final_jump_to_rom_ext(current_rom_ext_manifest)) {
610             __REACHABILITY_CHECK
611
612             //Step 2.iv
613             boot_failed_rom_ext_terminated(boot_policy, current_rom_ext_manifest);
614             __boot_policy_stop = 1;
615             return;
616         }
```

```
617     } // End for
618
619     __REACHABILITY_CHECK
620
621     //Step 2.iv
622     boot_failed(boot_policy);
623 }
```

Listing C.2: The content of the mask_rom.c file

C.3 hmac.h

```
1  /*********************************************************************
2  * Filename:   sha256.h
3  * Author:     Brad Conte (brad AT bradconte.com)
4  * Copyright:
5  * Disclaimer: This code is presented "as is" without any guarantees.
6  * Details:    Defines the API for the corresponding SHA1 implementation.
7  *********************************************************************/
8
9  #ifndef SHA256_H
10 #define SHA256_H
11
12 /************************* HEADER FILES ************************/
13 #include <stddef.h>
14 #include "mask_rom.h"
15
16 /*************************** MACROS ****************************/
17 #define SHA2_256_BLOCK_SIZE 32            // SHA256 outputs a 32 byte digest
18 #define HMAC_KEY_SIZE 32                  // HMAC key is 32 bytes
19 /*************************** DATA TYPES ************************/
20 typedef unsigned char BYTE;             // 8-bit byte
21 typedef unsigned int WORD;              // 32-bit word, change to "long" for 16-bit
       ↪ machines
22
23 typedef struct {
24     BYTE data[64];
25     WORD datalen;
26     unsigned long long bitlen;
27     WORD state[8];
28 } SHA2_256_CTX;
29
30 /*********************** FUNCTION DECLARATIONS **********************/
31 void HMAC_SHA2_256_init(SHA2_256_CTX *ctx);
32 void HMAC_SHA2_256_update(SHA2_256_CTX *ctx, const BYTE data[], size_t len);
33 void HMAC_SHA2_256_final(SHA2_256_CTX *ctx, BYTE hash[]);
34 BYTE* HMAC_SHA2_256(BYTE key[], BYTE mes[], int size, rom_ext_manifest_t
       ↪ __current_rom_ext_mf);
35
36 #endif  // SHA256_H
```

Listing C.3: The content of the hmac.h file

C.4 hmac.c

```
1  /*********************************************************************
2  ////////////////////// ORIGINAL FILE ////////////////////////////////
```

```
3  * Filename:   sha2_256.c
4  * Original
5    Author:     Brad Conte (brad AT bradconte.com)
6  * Copyright:
7  * Disclaimer: This code is presented "as is" without any guarantees.
8  * Details:    Implementation of the SHA-256 hashing algorithm.
9                SHA-256 is one of the three algorithms in the SHA2
10               specification. The others, SHA-384 and SHA-512, are not
11               offered in this implementation.
12               Algorithm specification can be found here:
13                * http://csrc.nist.gov/publications/fips/fips180-2/fips180-2
       ↪ withchangenotice.pdf
14               This implementation uses little endian byte order.
15
16 * Modified By: Jacob Gosch and Kristoffer Jensen
17 *****************************************************************/
18
19 /************************* HEADER FILES ************************/
20 #include <stdlib.h>
21 #include <memory.h>
22 #include "hmac.h"
23 #include "memory_compare.h"
24 /************************** MACROS *****************************/
25 #define ROTLEFT(a,b) (((a) << (b)) | ((a) >> (32-(b))))
26 #define ROTRIGHT(a,b) (((a) >> (b)) | ((a) << (32-(b))))
27
28 #define CH(x,y,z) (((x) & (y)) ^ (~(x) & (z)))
29 #define MAJ(x,y,z) (((x) & (y)) ^ ((x) & (z)) ^ ((y) & (z)))
30 #define EP0(x) (ROTRIGHT(x,2) ^ ROTRIGHT(x,13) ^ ROTRIGHT(x,22))
31 #define EP1(x) (ROTRIGHT(x,6) ^ ROTRIGHT(x,11) ^ ROTRIGHT(x,25))
32 #define SIG0(x) (ROTRIGHT(x,7) ^ ROTRIGHT(x,18) ^ ((x) >> 3))
33 #define SIG1(x) (ROTRIGHT(x,17) ^ ROTRIGHT(x,19) ^ ((x) >> 10))
34
35 /************************* VARIABLES ***************************/
36 static const WORD k[64] = {
37     0x428a2f98,0x71374491,0xb5c0fbcf,0xe9b5dba5,0x3956c25b,0x59f111f1,0x923f82a4,0
       ↪ xab1c5ed5,
38     0xd807aa98,0x12835b01,0x243185be,0x550c7dc3,0x72be5d74,0x80deb1fe,0x9bdc06a7,0
       ↪ xc19bf174,
39     0xe49b69c1,0xefbe4786,0x0fc19dc6,0x240ca1cc,0x2de92c6f,0x4a7484aa,0x5cb0a9dc,0
       ↪ x76f988da,
40     0x983e5152,0xa831c66d,0xb00327c8,0xbf597fc7,0xc6e00bf3,0xd5a79147,0x06ca6351,0
       ↪ x14292967,
41     0x27b70a85,0x2e1b2138,0x4d2c6dfc,0x53380d13,0x650a7354,0x766a0abb,0x81c2c92e,0
       ↪ x92722c85,
42     0xa2bfe8a1,0xa81a664b,0xc24b8b70,0xc76c51a3,0xd192e819,0xd6990624,0xf40e3585,0
       ↪ x106aa070,
43     0x19a4c116,0x1e376c08,0x2748774c,0x34b0bcb5,0x391c0cb3,0x4ed8aa4a,0x5b9cca4f,0
       ↪ x682e6ff3,
44     0x748f82ee,0x78a5636f,0x84c87814,0x8cc70208,0x90befffa,0xa4506ceb,0xbef9a3f7,0
       ↪ xc67178f2
45 };
46
47 /*********************** FUNCTION DEFINITIONS **********************/
48 void HMAC_SHA2_256_transform(SHA2_256_CTX *ctx, const BYTE data[])
49 {
50     WORD a, b, c, d, e, f, g, h, i, j, t1, t2, m[64];
51
52     for (i = 0, j = 0; i < 16; ++i, j += 4)
53         m[i] = (data[j] << 24) | (data[j + 1] << 16) | (data[j + 2] << 8) | (data[j + 3]);
54     for ( ; i < 64; ++i)
```

```
55          m[i] = SIG1(m[i - 2]) + m[i - 7] + SIG0(m[i - 15]) + m[i - 16];
56
57      a = ctx->state[0];
58      b = ctx->state[1];
59      c = ctx->state[2];
60      d = ctx->state[3];
61      e = ctx->state[4];
62      f = ctx->state[5];
63      g = ctx->state[6];
64      h = ctx->state[7];
65
66      for (i = 0; i < 64; ++i) {
67          t1 = h + EP1(e) + CH(e,f,g) + k[i] + m[i];
68          t2 = EP0(a) + MAJ(a,b,c);
69          h = g;
70          g = f;
71          f = e;
72          e = d + t1;
73          d = c;
74          c = b;
75          b = a;
76          a = t1 + t2;
77      }
78
79      ctx->state[0] += a;
80      ctx->state[1] += b;
81      ctx->state[2] += c;
82      ctx->state[3] += d;
83      ctx->state[4] += e;
84      ctx->state[5] += f;
85      ctx->state[6] += g;
86      ctx->state[7] += h;
87  }
88
89  void HMAC_SHA2_256_init(SHA2_256_CTX *ctx)
90  {
91      ctx->datalen = 0;
92      ctx->bitlen = 0;
93      ctx->state[0] = 0x6a09e667;
94      ctx->state[1] = 0xbb67ae85;
95      ctx->state[2] = 0x3c6ef372;
96      ctx->state[3] = 0xa54ff53a;
97      ctx->state[4] = 0x510e527f;
98      ctx->state[5] = 0x9b05688c;
99      ctx->state[6] = 0x1f83d9ab;
100     ctx->state[7] = 0x5be0cd19;
101 }
102
103 void HMAC_SHA2_256_update(SHA2_256_CTX *ctx, const BYTE data[], size_t len)
104 {
105     WORD i;
106
107     for (i = 0; i < len; ++i) {
108         ctx->data[ctx->datalen] = data[i];
109         ctx->datalen++;
110         if (ctx->datalen == 64) {
111             HMAC_SHA2_256_transform(ctx, ctx->data);
112             ctx->bitlen += 512;
113             ctx->datalen = 0;
114         }
115     }
```

```
116 }
117
118 void HMAC_SHA2_256_final(SHA2_256_CTX *ctx, BYTE hash[])
119 {
120     WORD i;
121
122     i = ctx->datalen;
123
124     // Pad whatever data is left in the buffer.
125     if (ctx->datalen < 56) {
126         ctx->data[i++] = 0x80;
127         while (i < 56)
128             ctx->data[i++] = 0x00;
129     }
130     else {
131         ctx->data[i++] = 0x80;
132         while (i < 64)
133             ctx->data[i++] = 0x00;
134         HMAC_SHA2_256_transform(ctx, ctx->data);
135         memset(ctx->data, 0, 56);
136     }
137
138     // Append to the padding the total message's length in bits and transform.
139     ctx->bitlen += ctx->datalen * 8;
140     ctx->data[63] = ctx->bitlen;
141     ctx->data[62] = ctx->bitlen >> 8;
142     ctx->data[61] = ctx->bitlen >> 16;
143     ctx->data[60] = ctx->bitlen >> 24;
144     ctx->data[59] = ctx->bitlen >> 32;
145     ctx->data[58] = ctx->bitlen >> 40;
146     ctx->data[57] = ctx->bitlen >> 48;
147     ctx->data[56] = ctx->bitlen >> 56;
148     HMAC_SHA2_256_transform(ctx, ctx->data);
149
150     // Since this implementation uses little endian byte ordering and SHA uses big endian,
151     // reverse all the bytes when copying the final state to the output hash.
152     for (i = 0; i < 4; ++i) {
153         hash[i]      = (ctx->state[0] >> (24 - i * 8)) & 0x000000ff;
154         hash[i + 4]  = (ctx->state[1] >> (24 - i * 8)) & 0x000000ff;
155         hash[i + 8]  = (ctx->state[2] >> (24 - i * 8)) & 0x000000ff;
156         hash[i + 12] = (ctx->state[3] >> (24 - i * 8)) & 0x000000ff;
157         hash[i + 16] = (ctx->state[4] >> (24 - i * 8)) & 0x000000ff;
158         hash[i + 20] = (ctx->state[5] >> (24 - i * 8)) & 0x000000ff;
159         hash[i + 24] = (ctx->state[6] >> (24 - i * 8)) & 0x000000ff;
160         hash[i + 28] = (ctx->state[7] >> (24 - i * 8)) & 0x000000ff;
161     }
162 }
163
164 BYTE* HMAC_SHA2_256(BYTE key[], BYTE mes[], int mes_size, rom_ext_manifest_t
    ↪ __current_rom_ext_mf){
165     int __expected_size =
166     sizeof(__current_rom_ext_mf.pub_signature_key)+sizeof(__current_rom_ext_mf.
    ↪ image_length)+__current_rom_ext_mf.image_length;
167
168     __CPROVER_assert(cmp_key(
169         mes,
170         &__current_rom_ext_mf.pub_signature_key,
171         sizeof(__current_rom_ext_mf.pub_signature_key)) == 0,
172         "PROPERTY 4: Message contains the key");
173
174     __CPROVER_assert(cmp_image_len(
```

```
175         mes + sizeof(__current_rom_ext_mf.pub_signature_key),
176         &__current_rom_ext_mf.image_length,
177         sizeof(__current_rom_ext_mf.image_length)) == 0,
178         "PROPERTY 4: Message contains the Image length");
179
180     __CPROVER_assert(cmp_image_code(
181         mes + sizeof(__current_rom_ext_mf.pub_signature_key) + sizeof(__current_rom_ext_mf
    ↪ .image_length),
182         __current_rom_ext_mf.image_code,
183         __current_rom_ext_mf.image_length) == 0,
184         "PROPERTY 4: Message contains the Image code");
185
186     __CPROVER_assert(mes_size == __expected_size,
187     "PROPERTY 4: Message size parameter is as expected.");
188
189     __CPROVER_assert(__CPROVER_OBJECT_SIZE(mes) == __expected_size,
190     "PROPERTY 4: Size of message is as expected.");
191
192
193     BYTE* buff = malloc(SHA2_256_BLOCK_SIZE * sizeof(BYTE));
194     SHA2_256_CTX ctx;
195
196     BYTE* key_mes_pad = malloc(HMAC_KEY_SIZE * sizeof(BYTE) + mes_size * sizeof(BYTE)); //
    ↪ key âĹě mes
197     memcpy(
198         key_mes_pad,
199         key,
200         HMAC_KEY_SIZE
201     );
202     memcpy(
203         key_mes_pad + HMAC_KEY_SIZE,
204         mes,
205         mes_size
206     );
207
208     HMAC_SHA2_256_init(&ctx);
209     HMAC_SHA2_256_update(&ctx, key_mes_pad, HMAC_KEY_SIZE + mes_size);
210     HMAC_SHA2_256_final(&ctx, buff);
211
212
213     __CPROVER_assert(__CPROVER_OBJECT_SIZE(buff)==256/8,
214     "PROPERTY 3: Hash is 256 bits");
215
216     __CPROVER_assert(__CPROVER_r_ok(buff, 256/8),
217     "PROPERTY 3: hash is in readable address");
218
219     __REACHABILITY_CHECK
220
221     return buff;
222 }
```

Listing C.4: The content of the hmac.c file

C.5 mock_hmac.c

```
1 #include "hmac.h"
2 #include "memory_compare.h"
3
4
```

```
BYTE* HMAC_SHA2_256(BYTE key[], BYTE mes[], int size, rom_ext_manifest_t
    ↪ __current_rom_ext_mf) {

    int __expected_size =
        sizeof(__current_rom_ext_mf.pub_signature_key) + sizeof(__current_rom_ext_mf.
    ↪ image_length) + __current_rom_ext_mf.image_length;

    __CPROVER_assert(cmp_key(
        mes,
        &__current_rom_ext_mf.pub_signature_key,
        sizeof(__current_rom_ext_mf.pub_signature_key)) == 0,
        "PROPERTY 4: Message contains the key");

    __CPROVER_assert(cmp_image_len(
        mes + sizeof(__current_rom_ext_mf.pub_signature_key),
        &__current_rom_ext_mf.image_length,
        sizeof(__current_rom_ext_mf.image_length)) == 0,
        "PROPERTY 4: Message contains the Image length");

    __CPROVER_assert(cmp_image_code(
        mes + sizeof(__current_rom_ext_mf.pub_signature_key) + sizeof(__current_rom_ext_mf
    ↪ .image_length),
        __current_rom_ext_mf.image_code,
        __current_rom_ext_mf.image_length) == 0,
        "PROPERTY 4: Message contains the Image code");

    __CPROVER_assert(size == __expected_size,
        "PROPERTY 4: Message size parameter is as expected.");

    __CPROVER_assert(__CPROVER_OBJECT_SIZE(mes) == __expected_size,
        "PROPERTY 4: Size of message is as expected.");

    char* hash = malloc(256 / 8); //model it to be ok for PROPERTY 5

    __CPROVER_assert(__CPROVER_OBJECT_SIZE(hash) == 256 / 8,
        "PROPERTY 3: Hash is 256 bits");

    __CPROVER_assert(__CPROVER_r_ok(hash, 256 / 8),
        "PROPERTY 3: hash is in readable address");

    __REACHABILITY_CHECK

    return hash;
}
```

Listing C.5: The content of the mock_hmac.c file

C.6 memory_compare.h

```
#ifndef MEMORY_COMPARE_H
#define MEMORY_COMPARE_H

int cmp_key(const void* buf1, const void* buf2, unsigned int size);

int cmp_image_len(const void* buf1, const void* buf2, unsigned int size);

int cmp_image_code(const void* buf1, const void* buf2, unsigned int size);

int cmp_modulus(const void* buf1, const void* buf2, unsigned int size);
```

```
11
12 int cmp_signature(const void* buf1, const void* buf2, unsigned int size);
13
14 int cmp_hash_decrypt(const void* buf1, const void* buf2, unsigned int size);
15
16 #endif  // MEMORY_COMPARE_H
```

Listing C.6: The content of the memory_compare.h file

C.7 memory_compare.c

```
1  #include "mask_rom.h"
2
3  int cmp_key(const void* buf1, const void* buf2, unsigned int size) {
4
5      __CPROVER_assert(size == (3072 + 32)/8,
6      "Assert: Size should be equal to size of modulus and exponent");
7
8      const char* cbuf1 = (char*)buf1;
9      const char* cbuf2 = (char*)buf2;
10
11     int mismatch = 0;
12     for (int i = 0; i < size; i++)
13     {
14         if (*cbuf1 != *cbuf2)
15         {
16             mismatch = 1;
17             break;
18         }
19         cbuf1++;
20         cbuf2++;
21     }
22
23     return mismatch; //0 is equal, 1 is not equal.
24 }
25
26
27 int cmp_image_len(const void* buf1, const void* buf2, unsigned int size) {
28
29     __CPROVER_assert(size == 4,
30     "Assert: Size should be equal to size of image_len variable type");
31
32     const char* cbuf1 = (char*)buf1;
33     const char* cbuf2 = (char*)buf2;
34
35     int mismatch = 0;
36     for (int i = 0; i < size; i++)
37     {
38         if (*cbuf1 != *cbuf2)
39         {
40             mismatch = 1;
41             break;
42         }
43         cbuf1++;
44         cbuf2++;
45     }
46
47     return mismatch; //0 is equal, 1 is not equal.
48 }
```

```
49
50 int cmp_image_code(const void* buf1, const void* buf2, unsigned int size) {
51
52     __CPROVER_assert(size <= MAX_IMAGE_LENGTH && size > 0,
53     "Assert: Size should be less than or equal to MAX_IMAGE_LENGTH");
54     const char* cbuf1 = (char*)buf1;
55     const char* cbuf2 = (char*)buf2;
56
57     int mismatch = 0;
58     for (int i = 0; i < size; i++)
59     {
60         if (*cbuf1 != *cbuf2)
61         {
62             mismatch = 1;
63             break;
64         }
65         cbuf1++;
66         cbuf2++;
67     }
68
69     return mismatch; //0 is equal, 1 is not equal.
70 }
71
72
73 int cmp_modulus(const void* buf1, const void* buf2, unsigned int size) {
74
75     __CPROVER_assert(size == 3072/8,
76     "Assert: Size should be equal to size of modulus");
77
78     const char* cbuf1 = (char*)buf1;
79     const char* cbuf2 = (char*)buf2;
80
81     int mismatch = 0;
82     for (int i = 0; i < size; i++)
83     {
84         if (*cbuf1 != *cbuf2)
85         {
86             mismatch = 1;
87             break;
88         }
89         cbuf1++;
90         cbuf2++;
91     }
92
93     return mismatch; //0 is equal, 1 is not equal.
94 }
95
96
97 int cmp_signature(const void* buf1, const void* buf2, unsigned int size) {
98
99     __CPROVER_assert(size == 3072/8,
100    "Assert: Size should be equal to size of signature");
101
102    const char* cbuf1 = (char*)buf1;
103    const char* cbuf2 = (char*)buf2;
104
105    int mismatch = 0;
106    for (int i = 0; i < size; i++)
107    {
108        if (*cbuf1 != *cbuf2)
109        {
```

```
110             mismatch = 1;
111             break;
112         }
113         cbuf1++;
114         cbuf2++;
115     }
116 
117     return mismatch; //0 is equal, 1 is not equal.
118 }
119 
120 
121 int cmp_hash_decrypt(const void* buf1, const void* buf2, unsigned int size) {
122 
123     __CPROVER_assert(size == 256/8,
124     "Assert: Size should be equal to size of hash");
125 
126     const char* cbuf1 = (char*)buf1;
127     const char* cbuf2 = (char*)buf2;
128 
129     int mismatch = 0;
130     for (int i = 0; i < size; i++)
131     {
132         if (*cbuf1 != *cbuf2)
133         {
134             mismatch = 1;
135             break;
136         }
137         cbuf1++;
138         cbuf2++;
139     }
140 
141     return mismatch; //0 is equal, 1 is not equal.
142 }
```

Listing C.7: The content of the memory_compare.c file

Appendix D

Function Contracts

This chapter is based on the test cases found in [57] and the insights gained from using the `--show-goto-functions` and `--program-only` flags. The implementation of function contracts is not finished at the time of writing (CBMC version 5.28). However, we still believe it to be relevant to document the use of function contracts. For each of the different constructs, we will mention whether or not they work as expected.

A function contract can contain the following three constructs:

- `__CPROVER_assigns(params)` : This construct asserts that the function assigns to the variables in params. `__CPROVER_assigns` is used on global variables and pointers. Including local parameters in params do not have any effect.
- `__CPROVER_ensures(cond)` : This construct is used to assert if a Boolean condition is met at the end of a function. E.g. that the function returns a specific value.
- `__CPROVER_requires(cond)` : This is used to define assumptions about variables at the entry of the function in terms of a Boolean condition.

It is possible to use constructs such as `__CPROVER_exists` and `__CPROVER_forall` in `__CPROVER_ensures` and `__CPROVER_requires` clauses. They have the following format:

```
1 __CPROVER_ensures(__CPROVER_forall{
2   int i;
3   (0 <= i && i < 5) ==> Boolean Expression
4     })
```

However, there are known bugs with the `__CPROVER_exists` and `__CPROVER_forall` constructs (cf. [28] [29] [30] [31]) and they will therefore not be covered further. But, a `__CPROVER_requires` or `__CPROVER_ensures` clause that use either a `__CPROVER_exists` or a `__CPROVER_forall` construct can also be modeled equivalently as a for-loop with `__CPROVER_assume` or `__CPROVER_assert` .

Normally, when performing verification using CBMC, the command is on the following form: `cbmc file.c --flags` . However, the flags for verifying function contracts are not implemented in the cbmc command. Therefore, when verifying the function contracts in a file called file.c the build process, for Windows, is:

- `goto-cl file.c`
- `goto-instrument --flags file.exe outputfile`
- `cbmc outputfile`

And for Linux:

- `goto-cc file.c -o file.gb`
- `goto-instrument --flags file.gb file2.gb`
- `cbmc file2.gb`

The two flags used when verifying function contracts are the `--enforce-all-contracts` and the `replace-all-calls-with-contracts` flag. The two flags has the following functionality:

- `--enforce-all-contracts`: In this case assumptions and assertions are inserted in all functions with contracts. The assertions assert whether the function contracts is satisfied by the functions.
- `--replace-all-calls-with-contracts`: All calls to functions with function contracts are replaced with behavior matching the function contracts. Such as `__CPROVER_assume`, `__CPROVER_assert`, or nondeterministic values.

Thus, the `--enforce-all-contracts` flag (which has a singular version) is used when the goal is to assert whether the functions adhere to their function contracts. The `--replace-all-calls-with-contracts` (which also has a singular version) is used when e.g. verifying a program containing a computationally demanding function. In this case, all calls to the demanding function can be replaced by its contract and thereby speed up the verification process.

D.0.1 __CPROVER_assigns

For illustrating the `__CPROVER_assigns` construct consider the program in Listing D.1 below:

```
1  int global_var = 0;
2
3  int foo(int* ptr1, int *ptr2)
4  __CPROVER_assigns(global_var, *ptr2)
5  {
6      global_var = 2;
7      *ptr2 = 5;
8      return *ptr1 * global_var;
9  }
10
11 int main()
12 {
13     int i = 10;
14     int j = 10;
15     int p = foo(&i, &j);
16     return 0;
17 }
```

Listing D.1: Example with `__CPROVER_assigns`.

When verified using the `--enforce-all-contracts` flag the verification succeeds as `foo` only assigns to the variables `global_var` and `*ptr2`. Essentially, it means that the function may or may not assign to any of the variables in the assigns clause, but if it assigns to a global variable or a pointer which is not part of the assigns clause then the verification fails.

The `--enforce-all-contracts` flag inserts assertions in `foo` asserting whether the addresses of the global variables and pointers that are assigned to is equal to any of the addresses of the variables in the assign clause.

Below is a list of examples for the assigns clause and outcome:

- `__CPROVER_assigns(*ptr1)`: Verification fails as expected. `foo` does not assign to `*ptr1` but `foo` does assign to other variables than `*ptr1`.
- `__CPROVER_assigns(global_var)`: This fails as expected, as `foo` also assigns to `*ptr2`, which is not in the `__CPROVER_assigns` clause.

- `__CPROVER_assigns(global_var, *ptr2, *ptr1)` : This succeeds as expected as `foo` only assigns to the global variables and pointers listed in the `__CPROVER_assigns` clause. It is not mandatory to assign to all the variables in the `__CPROVER_assigns` clause.

When verified using the `--replace-all-calls-with-constracts` flag no assertions nor assumptions are generated. The call to foo is replaced by assigning `global_var` , `j` and `p` to a nondeterministic integer value. As `i` is not in the `__CPROVER_assigns` is remains equal to 10. On CBMC's GitHub there have been documented bugs with using `__CPROVER_assigns` together with `__CPROVER_ensures` when verifying using the `--replace-all-calls-with-constracts` flag (cf. [58]).

In conclusion, the functionality of `__CPROVER_assigns` is that:

- It fails if a global variable or pointer not in the `__CPROVER_assigns` clause is assigned to.
- It does not fail if a variable in the `__CPROVER_assigns` clause is not assigned to.

To our knowledge, there is not a build-in way of saying that a function does not assign anything. However, it could be done by having a dummy global variable named "nothing". Then have the function assign to that and the function contract to contain `__CPROVER_assign(nothing)` . In this way, if the function assigns to other variables (global variables or pointers) the function does not adhere to function contract and CBMC will indicate so.

D.0.2 __CPROVER_ensures

For illustrating the `__CPROVER_ensures` construct consider the program in Listing D.2 below:

```
1  int foo(int var1, int var2, int* ptr3)
2  __CPROVER_ensures(__CPROVER_return_value == var1 * var2)
3  {
4      *ptr3 = var1;
5      return var1 * var2;
6  }
7
8
9  int main()
10 {
11     int i = 5;
12     int j = foo(10, 10, &i);
13     __CPROVER_assert(j == 100, "Expected behavior of foo.");
14     __CPROVER_assert(i == 10, "Expected behavior of foo.");
15 }
```

Listing D.2: Example with `__CPROVER_ensures` .

The `__CPROVER_return_value` is equal to the return value of the function. When verified using the `--enforce-all-contracts` flag the verification succeeds as expected. The fulfillment of the function contract does not influence the value of `i` and `j` in `main` . The use of the `--enforce-all-contracts` flag inserts an assertion before the return statement in `foo` , asserting whether the temporary variable storing the return value is equal to `var1 * var2` .

Below is a list of examples for the ensures clause and outcome:

- `__CPROVER_ensures(__CPROVER_return_value == var1 * var2 && *ptr3 == var1)` : Succeeds as expected.
- `__CPROVER_ensures(__CPROVER_return_value == var1)` : The function contract fails as expected and the assertions in `main` still succeeds.

- `__CPROVER_ensures(__CPROVER_return_value == var1 && *ptr3 == var1)` : The function contract fails as expected and the assertions in `main` still succeeds.

When verified using the `--replace-all-calls-with-contracts` flag, the verification fails. The reason is that the call to foo is replaced by a `__CPROVER_assume` assuming the function contract, namely that the return value of `foo` is 10 * 10. `j` is asserted to be 100, which succeeds. However, as the function contract did not describe what `foo` did to `i`, the assertion of whether `i` is 10 fails, as `i` is still 5. If the ensure clause is `__CPROVER_ensures(__CPROVER_return_value == var1 * var2 && *ptr3 == var1)` the verification succeeds as the `__CPROVER_assume` also assumes that `i` is 10.

D.0.3 __CPROVER_requires

A `__CPROVER_requires` clause can only be used if a `__CPROVER_assigns` or a `__CPROVER_ensures` clause is in the function contract as well. For illustrating the `__CPROVER_requires` construct consider the program in Listing D.3 below:

```
1  int foo(int var1, int var2)
2  __CPROVER_requires(var1 > 0 && var1 > 0)
3  __CPROVER_ensures(__CPROVER_return_value == var1*var2 && __CPROVER_return_value > 0)
4  {
5      __CPROVER_assert(0, "Reachability Check");
6      return var1 * var2;
7  }
8
9
10 int main()
11 {
12     int j = foo(10, 10);
13     __CPROVER_assert(j == 100, "Expected behavior of foo.");
14     __CPROVER_assert(0, "Reachability Check");
15 }
```

Listing D.3: Example with `__CPROVER_requires` and `__CPROVER_ensures`.

When verified using the `--enforce-all-contracts` flag the verification succeeds. When using the `--enforce-all-contracts` flag the `__CPROVER_requires` clause is transformed into an `__CPROVER _ assume` in `foo`, assuming that `var1 > 0 && var2 > 0` and the `__CPROVER_ensures` clause (which is transformed to an assertion) is evaluated under that assumption. However, as with normal assumptions, if the assumption cannot be true, e.g. that it assumed that `var1 > 0 && var2 < 0`, then all future assertions will succeed by default.

When the above code is verified using the `--replace-all-calls-with-contracts` the call to `foo` is replaced with its contract. The `__CPROVER_requires` clause is transformed into an `__CPROVER_assert` which asserts that the actual parameters are above 0. The `__CPROVER_ensures` is transformed into a `__CPROVER_ assume` as described earlier. Do note that the reachability check in `foo` fails as `foo` is never called.

D.0.4 Loop Invariants

It is also possible annotate loops with loop invariants and verify whether the loops adhere to them, using the `--enforce-all-contracts` flag. A loop invariant is a condition that should hold before the loop is entered and after each iteration of the loop. However, the feature for loop invariants is faulty. To illustrate, consider the example in Listing D.4 below:

```
1  int main()
2  {
3      int r = 10;
```

```
4       while (r > 0)
5           __CPROVER_loop_invariant(r > 9);
6       {
7           r--;
8       }
9   }
```

Listing D.4: Example of a loop invarant.

The verification of the loop invariant succeeds, which should not be the case as `r > 9` does not hold after each iteration of the loop.

While the `__CPROVER_loop_invariant` does not work, it is possible to achieve the same functionality by having an assertion before the loop, at the end of the loop body, and after the loop.