Formally Verifying Security Properties for OpenTitan Boot Code with UPPAAL

Bjarke Hilmer Møller,
Magnus Winkel Pedersen,
Tobias Worm Bøgedal

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 Security Properties for OpenTitan Boot Code with Uppaal

Theme:
Security Verification and Model Checking

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

Project Group:
SV106f21

Participants:
Bjarke Hilmer Møller
Magnus Winkel Pedersen
Tobias Worm Bøgedal

Supervisors:
Danny Bøgsted Poulsen
Kim Guldstrand Larsen
René Rydhof Hansen

Number of pages: 111

Date of Completion:
June 17, 2021

Abstract:

The demand for secure computer systems increases as more information is stored and processed digitally. The ways a computer system can be attacked is limited only by the attackers' imagination. Therefore, safety-critical computer systems must be developed with this in mind. A system currently under development is the OpenTitan system. To follow the development and investigate the security mechanisms at play in the booting process of OpenTitan, we apply methods for formal verification. We do this with the hope of verifying relevant security principles of the system. To perform this task, we use the model checking tool Uppaal version 4.1.24 to model the first boot stage of OpenTitan. The model is the foundation for the verification of certain security goals that we have developed previously. With the help of Uppaal's built-in verification tool, we can successfully verify five out of eight selected goals. We further consider two of the remaining three security goals partially verified. The last security goal is not verified. This is partly due to the current state of the OpenTitan project documentation, which does not provide sufficient detail to make a qualified verification of the goal. Thus by building a model representing the OpenTitan system, we can do co-verification of the OpenTitan system with predominantly successful results.

Summary

In this project, we work with co-verification i.e. verification of software, and the hardware that is running it. Our case is the OpenTitan project. OpenTitan is an open-source chip design based on the Titan project by Google. For this project we are only interested in verifying the ROM stage of the OpenTitan chip design. During the previous semester, we have made a security analysis of OpenTitan, which resulted in a list of security policies and security goals that should be verified for the system to be considered secure. The security policies are:

1. Only code that has been validated can be executed.
2. Validation of a boot stage must only succeed if the boot environment is safe.
3. Cryptographic key leakage can not happen.
4. All data access rights follow a privilege hierarchy.

The security policies are further divided into 14 security goals. The security policies and security goals are relevant for the entire boot process. Since we choose to only look at the boot of the ROM stage, six of the security goals (G2, G4, G6, G7, G13, and G14) are not relevant for our project, so we will not try to verify them.

To verify the security goals, we have modeled the system in Uppaal. Uppaal is a verification tool that allows the user to make templates that correspond to the components in the real system. Uppaal has its own verification engine that allows its users to make queries and thus verify properties about the model.
We have modeled hardware and firmware components of the OpenTitan design in the Uppaal model. We have modeled the ROM stage like the ROM stage described in the OpenTitan documentation. The ROM stage is responsible for validating ROM_ext manifests and only transfer control to the next stage (ROM_ext) upon successful validation. The components are modeled with varying levels of abstraction. The components that are not directly related to our security goals, are modeled more abstractly. This is because a more detailed implementation would drag out the verification process, while at the same time only marginally impacting the results of the verification. For some of the aforementioned security goals we have made queries that verify them. Specifically, we were able to fully verify five security goals (G1, G9, G10, G11, and G12), partially verify two security goals (G3 and G8), and not able to verify one goal (G5) out of the security goals that we did not deem to be out of scope for the project.

To make sure that the Uppaal model works correctly, we have validated the model. This means that we have tested our assumptions about the model using Uppaal's verification tool. We have also introduced some errors in the model to see whether those errors can be seen in the results of our queries. These errors can be seen as potential bugs in the system. We introduced errors to crucial functions used for validation of the ROM_ext manifest. This includes the hashing function, the function that calculates the public key IDs, and the function responsible for validating signatures.

We reflect on our use of the Uppaal tool and OpenTitan as a case. We think that Uppaal is very well suited tool for co-verification. It might have been beneficial for us to use another chip design than OpenTitan as our case because OpenTitan is not yet fully designed.

There is another group that works with the same case as we do, but they use the C Bounded Model Checker (CBMC) tool instead. They have been able to verify properties that are roughly equivalent to four of the security goals that we have been able to verify. To do co-verification they have made C functions for the other components that are relevant for their security goals.
In the end we conclude that we have been able to verify some of the security goals with UPPAAL. We are overall satisfied with the number of security goals we have been able to verify and their importance.

Contents

Chapter 1

Introduction

The security of computer systems is of high importance. Now, in a world where more and more of our daily lives is digitalized, this becomes truer by the day. The cost of security breaches can be devastating, as seen by the NotPetya attack on Maersk in 2017 [1] which was estimated to have cost them 300 million dollars. With so much money on the line, attackers have the potential to strike gold if they succeed in ransomware attacks. Even worse for many people are data breaches, which can expose sensitive personal data from citizens who have no chance to protect themselves. An example of this is an incident with Zoom where half a million user accounts were put up for sale on the dark web [2]. One way an attacker can infiltrate an IT system is to do it during the booting process. If an attacker can somehow gain access to a system during boot, they can control what runs on the system. Such vulnerabilities in the booting process of IT systems are not just theory. In 2020 a vulnerability was reported by researchers at Eclypsium in the secure boot of the GRUB2 boot loader used by most Linux distributions. This vulnerability, dubbed "the boothole vulnerability", exploited a buffer overflow to allow for any arbitrary code to run the booting process [3]. The discovery of this vulnerability lead to similar vulnerabilities being discovered in similar types of code [4].

To ensure that newly developed systems do not suffer from such vulnerabilities, developers can thoroughly test code. However, as famously stated by E. W. Dijkstra in his Notes On Structured Programming: "Program testing can be used to show the presence of bugs, but never to show their absence!" [5]. So while testing is undoubtedly worth the effort, especially with the increasing quality and ease of using automated testing suites, we cannot guarantee that a program is free of errors based simply on testing.
A company named MAXSYT is currently working on a project where program correctness is at the forefront. For this, testing is not enough. They need proofs and guarantees on the correctness of their system.
Inspired by this project, we propose to use formal methods for verification to ensure that such a system meets these standards. These formal methods, unlike tests, give mathematical guarantees regarding the properties of IT systems. This means that if an applied formal method says that a given property of a system holds, it is mathematically guaranteed to hold. This makes formal methods a powerful tool that can be used to ensure the security of a system where normal testing cannot. However, there are many types (static analysis, theorem proving, model checking, etc.). Unlike testing, they require very diligent and time-consuming efforts to work correctly as such testing is by no means made obsolete by formal methods. However, if program correctness is of the highest priority, then formal methods are almost guaranteed to be needed during the development of a system.
In this report, we show how we can apply formal methods in the form of model checking to the case of boot code security. For this purpose, we use the model checking tool Uppaal, which is one of the leading tools within the field. We use the OpenTitan platform as a case, which is an open-source Root of Trust chip design based on Google's Titan project [6] [7]. We model a part of OpenTitan's boot process and show how such a model can verify security goals.

However, since the OpenTitan platform is still under heavy development, we have to make assumptions about some design parts. Furthermore, we have created our own boot code that takes inspiration from the available code in the OpenTitan documentation. This means that there might be slight deviations between our modeled system and the intended OpenTitan implementation.

We choose to limit the scope of the project such that we are only interested in the first stage of the OpenTitan boot process and its immediate impact on the next stage. This is done to ensure that we can give useful results in the time allotted for the project. OpenTitan is a large and complex system, and therefore we cannot realistically create a model of the entire boot process in five months. Furthermore, OpenTitan is still under development, which means that the entire chip system and the corresponding software are subject to change at any point in time. This is problematic, and the problem is exaggerated if we build a model of a larger part of the system. However, we hope to show how the fundamental process of model checking, as presented in this report, can be applied to further parts of the OpenTitan design or other systems. This leads to the following problem statement:

How can UPPAAL be used to verify the security of selected parts of the OpenTitan boot process?

Chapter 2

OpenTitan

OpenTitan is an open source reference design project of Root of Trust chips. According to [8], a Root of Trust is "A system element that provides services, including verification of system, software & data integrity and confidentiality, and data (software and information) integrity attestation between other trusted devices in a system or network."
The intention of the OpenTitan project is to make the implementation and development of Root of Trust chips more transparent, trustworthy, and secure [6]. The OpenTitan project is a collaboration between multiple tech-companies, including Google and Western Digital. The OpenTitan chip is built on a RISC-V Instruction Set Architecture[9].
The OpenTitan project is still in development. Therefore software and hardware are subject to change. The boot code that we have developed for this project (cf. section 3.1) is inspired by the booting process and software presented in the OpenTitan project[6]. Because the project is still in development, the boot code for this project will take inspiration from the boot code of the OpenTitan project available at the time of writing.
Based on [10] we define boot code to be the code that is run from the point the power on is initiated and until a boot loader takes control over the booting process.

2.1 The OpenTitan Software Stack

The OpenTitan design contains a logical security model which describes four entities: the silicon creator, the silicon owner, the application provider, and the end user. These entities all have responsibilities and roles in regards to an OpenTitan chip. E.g., the silicon creator creates the chip and provides ROM (also known as mask_ROM) and ROM_ext firmware for the chip, while the silicon owner owns the chip and provisions the boot loader and kernel. In this report, we focus primarily on the silicon creator as this entity creates the boot code for the OpenTitan chip. Information on the other entities can be found at [11].

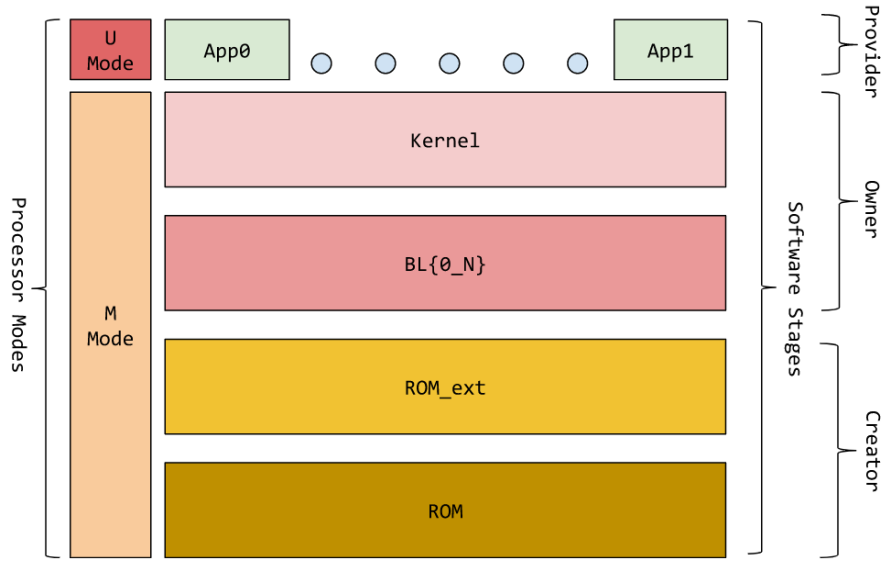


Figure 2.1.1: A graphical representation of the software stages and their owners in OpenTitan [11].

The software stack for the OpenTitan chip can be seen in Fig. 2.1.1. This figure shows the logical flow of execution for the software stages of the OpenTitan design. There are some assumptions about the execution of these stages:

- From the ROM stage until the kernel stage, the execution is one way. This means that once a stage is left, it cannot be re-executed until a system reset. Furthermore, code used during the stage can be locked through the use of PMP to ensure that it cannot be used during a later stage[11].
- When the chip has entered the kernel stage, the execution can optionally become two-way. This means that it is possible to switch between the kernel and the applications. It does not mean that the previous boot stages can be re-executed[11].

This project is concerned with the verification of security properties of boot code. Therefore we will go in-depth with ROM and ROM_ext of the OpenTitan design to describe how this boot code works.

2.1.1 ROM

This section is based on [12, Ch. 5]. ROM cannot be changed once the chip containing it has been created. The ROM is the first code that runs on startup of the system. ROM is mainly focused on finding a valid ROM_ext manifest and transferring execution to it. A ROM_ext manifest is a block of data that contains (among other things) an identifier, an image signature, the image code for the corresponding ROM_ext, and the public key that should be used to verify the signature. A more detailed description of ROM_ext manifests can be seen in appendix A.
Firstly, during the ROM stage, the Static Random Access Memory (SRAM) of the chip is cleared except for retention SRAM. After the non-retention SRAM is cleared, an initial PMP region is created (cf. section 2.6 for more on PMP). This PMP region disables writing to or executing anything in flash memory[1]. Then the boot policy is read from Flash Info. After these initial steps, ROM must decide which ROM_ext manifest it should try to boot first. To make this decision, the prioritized ROM_ext slot is derived from

[1] As of writing this report, it is not certain by looking at the documentation for OpenTitan whether this PMP region is created by hardware or ROM.

the boot policy. However, if the prioritized slot fails to load or cannot be validated, ROM can try to load another ROM_ext manifest if instructed to do so by the boot policy.

When ROM tries to validate a ROM_ext slot, the first check it performs is to see whether the slot is empty or not. Afterward, a public key ID is generated from the ROM_ext manifest's public key and checked. We could not find any documentation for how the public key ID is generated and how the ID is checked, so we opened an issue on the OpenTitan GitHub page. The respondent told us that the function for generating a public key ID does not necessitate strong cryptographic functionality, and the function is an implementation detail. He also told us that the device should have a list of trusted public keys, which is used to validate the ROM_ext manifest's public key[13]. If the slot is not empty, the manifest ID is valid, and the public key ID is valid, then ROM tries to validate the signature of the ROM_ext. If this validation succeeds then, it must hold that the Silicon Creator has created the ROM_ext image. The next step in execution is for ROM to measure system states (this process is not documented for OpenTitan as of writing this). Furthermore, ROM uses the Key Manager to derive the CreatorRootKey and locks down the Key Manager so no new key can be derived during the current session. This key is used at later stages of the boot process. The ROM_ext manifest is also used to lock down peripherals of the system at this point. Then, a new PMP region is formed for the ROM_ext that covers the ROM_ext image code. This PMP region is set to allow for the image to be read and executed and is set to be locked. With all of this done, ROM transfers execution to the newly validated ROM_ext manifest. If any of the checks or validations mentioned fail, ROM performs boot failure policy as determined by the boot policy. If the ROM_ext fails to boot correctly, then execution likewise returns to ROM so that it can perform a boot failure policy function [2]. This is because the execution before the kernel stage is one-way as described in section 2.1. If the execution returns to the ROM, the booting process has failed.

2.1.2 ROM_ext

The ROM_ext stands for Read-Only Memory extensions. This stage contains code that extends the functionality of the initial ROM. ROM_ext has several responsibilities. It must be able to apply any patches that correct critical issues in the hardware. ROM_ext also performs the remaining security configuration that was not done by ROM. This includes comparing the hashed Silicon Owner image to the hash in the Silicon Owner manifest and verifying the hash in the manifest. The hash in the manifest is verified by verifying its signature with the Owner Identity public key. Lastly, it performs Silicon Creator Identity provision and attestation. After ROM_ext has performed these tasks, it (like ROM) makes a PMP region that covers the executable region of the Silicon Owner software. This region only allows for reading and executing the covered code. The ROM_ext then transfers control to the boot loader (BL0) [11] [12].

2.1.3 Memory

The OpenTitan chips' current memory layout is illustrated in Fig. 2.1.2. The memory is split into four parts: ROM, Flash Info, Flash Bank 0, and Flash Bank 1. ROM, Flash Info, ROM_ext, and creator_cert are fixed size and are owned by the Silicon Creator. The Flash Banks are divided into six sections: ROM_ext, creator_cert, owner_cert, BL0, Kernel Apps, and Data. ROM_ext contains the ROM_ext manifests and other information related to booting the ROM_ext. creator_cert and owner_cert likely contain the certificates for the silicon creator and silicon owner, respectively. However, this is speculation on our part and is not clearly stated in the documentation. BL0 contains the first boot loader that is run after the ROM_ext has

[2] It is unclear whether there are two different boot failure policy functions or if the same is used for ROM_ext failure and not finding a valid ROM_ext.

been booted. Kernel Apps contains the programs and applications that are run in User mode. Data is used for additional storage by the Silicon Owner [14].

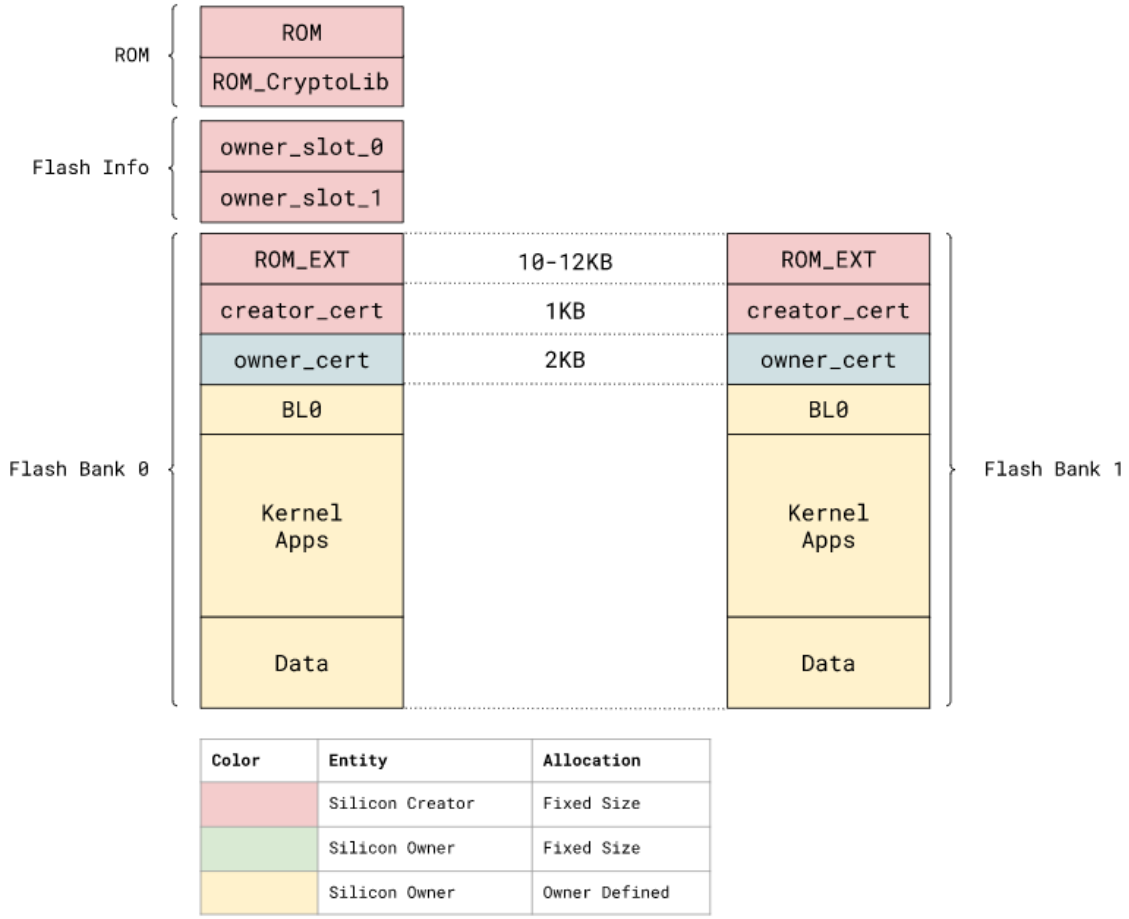


Figure 2.1.2: OpenTitan's memory layout[14].

2.2 Algorithms Used by OpenTitan

The OpenTitan design uses different algorithms for hashing, encryption, and verification. Some of these are used during the secure boot stage of the chip, and therefore, they are an important part of the booting process. According to [15] the following algorithms are used with for the purposes[3]:

- SHA256 is used as a hash algorithm.
- RSA-3072 is an asymmetric key algorithm used for signature verification during secure boot.
- ECDSA P-256 is another asymmetric key algorithm used for signature and verification handling for identity and attestation keys.
- HMAC-SHA256 (NIST FIPS 180-4 compliant) is a symmetric key algorithm used to measure the integrity of storage and in-transit data. It is also used for secure boot of the chip.
- AES (AES-CTR NIST 800-38A) is a symmetric key algorithm used to encrypt data and wrap keys stored in flash.

[3]Note that these algorithms are for the use case of the OpenTitan chip being a platform integrity module. There are slight variations in the algorithms for using it as a universal 2nd-factor security key.

Since it is stated in [15] that both RSA-3072 and HMAC-SHA256 are used during secure boot, these are explained in the following sections. RSA-3072 and HMAC-SHA256 are variations of the RSA and HMAC algorithms. As such, we try to explain the underlying structure of these algorithms.

2.2.1 The RSA Algorithm

This section is based on [16] unless otherwise stated. The RSA algorithm was invented by Ron Rivest, Adi Shamir, and Leonard Adleman. It is an asymmetric cryptography algorithm which means that it uses two keys (known as a public key and a private key) that are linked mathematically. Normally one of these keys would be used for encryption, with the other being used for decryption. However, the RSA algorithm allows for both keys to be used for both purposes. This means that the private key and the public key can both be used for encryption and decryption. In the RSA algorithm, these keys are constructed based on two prime numbers. This is because the RSA algorithm hinges on the fact that it is difficult to factor large integers that are a product of two prime numbers. If the product of the prime numbers is large enough (meaning that the prime numbers themselves are also large), then figuring out the original prime numbers from the product of them is considered infeasible.
Therefore, the first step of the algorithm is to generate these two large prime numbers (often referred to as p and q). Once this has been done, their product is also calculated. This is called the modulus (often written as n) and is used by both the public and private keys and serves as the mathematical link between the two keys. The length of the modulus in bits is called the key length. As an example, RSA-3072 is an RSA algorithm that has a key length of 3072 bits. The keys are as follows:

- The public key uses the modulus (n) and a public exponent (e) that is often set to 65537. This is often chosen due to it being a prime number that is not overly large. e can be set publicly because the public key is typically shared with everyone, thus making it unnecessary to hide the prime number. If the key is not shared with everyone, then e can be selected secretly to be some other large prime number.
- The private key uses the modulus (n) as well as a private exponent (d). d is calculated using the Extended Euclidean algorithm to find the multiplicative inverse such that $e \cdot d \equiv 1 \mod (p-1)\cdot(q-1)$ [17]. Note that $(p-1)\cdot(q-1)$ is the totient of the modulus[4].

The RSA algorithm can be used for both encryption, decryption, signing, and signature verification. The following describes signing and signature verification as these are the use cases that the RSA algorithm is used for in OpenTitan.

Digital signing with the RSA algorithm can be seen in Algorithm 1. When a sender wants to sign a message digitally and send it to someone, they have to hash their message with a hashing function. Then, the sender has to represent the hashed result as an integer v between 1 and $n-1$, where n is the modulus. The sender then uses their private key to compute the signature s with the following formula: $s \equiv m^d \mod n$. The sender then sends s, and the original message to the receiver [18].

[4]Euler's totient function to any prime number x is equal to $x-1$. Furthermore, the function is multiplicative which means that the function $\Phi(x \cdot y) = \Phi(x) \cdot \Phi(y)$.

Algorithm 1: RSA signing algorithm [18]

1: input: Message M, private key (n, d), hashing function F.
2: output: A valid signature for the message M.
3: $H \leftarrow F(M)$;
4: Represent H as an integer v between 1 and $n - 1$;
5: $s \leftarrow v^d \mod n$;
6: **return** s;

Signature verification with the RSA algorithm can be seen in Algorithm 2. When a receiver wants to verify a signature, they must first decrypt the signature using the sender's public key. This results in a hashed version of the original message encoded as an integer. Then the receiver hashes the original message that they received from the sender and encodes the resulting digest as an integer. The signature is valid iff the two encoded hashed messages are the same. This only works if the sender and the receiver use the same methods for hashing and encoding the messages as integers. If the receiver of the message confirms the signature, then they can be certain that the message is the original message that the expected sender has sent [18].

Algorithm 2: RSA signature verification algorithm [18]

1: input: Message M, signature s, sender's public key (n, e), hashing function F.
2: output: A boolean value signifying whether the signature is valid.
3: $v \leftarrow s^e \mod n$;
4: $H' \leftarrow F(M)$;
5: $v' \leftarrow$ Represent H' as an integer between 1 and $n - 1$;
6: **return** $v = v'$;

2.2.2 The HMAC Algorithm

HMAC is an abbreviation of Hash-based Message Authentication Code. The HMAC algorithm is a symmetric encryption algorithm used to ensure that a received message has not been tampered with. It uses a secret key and a hashing function to hash the key and the message first, and then hash the now hashed output and the secret key again. To use the HMAC algorithm for authentication, the secret key must be shared between the sender and the receiver of the message [19].

The algorithm can be seen in Algorithm 3 and follows the approach described in [20]. The algorithm needs a message M and secret key K as input. M consists of blocks of length b. The key must have a length of b. Therefore, it is padded with zeros, as described in lines 2 and 3 of the algorithm. This results in the padded key called K^+. In line 6, $Signature1$ is calculated based on the padded key and the bit string 00110110. The bit string must, if necessary, be repeated to also have a length of b (this is done by the function repeatedToLengthb). $Signature1$ is then concatenated with M and given as input to a hashing algorithm, here SHA256. The Temp variable of line 7 is used along with a new signature, $Signature2$, which is calculated as seen in line 8. Thus, line 9 gives the final result by concatenating $Signature2$ and Temp and giving it to the SHA256 hashing algorithm.
The output of the algorithm is a hashed message authentication code. Once a receiver receives an HMAC, it will use the secret key and the algorithm to hopefully deduce an identical hash from the message to the received hash. If the receiver succeeds in doing this, the message has not been tampered with. If the message does not match the hash computed by the receiver, some change has been made to the message between the message being sent and arriving at the receiver.

Algorithm 3: HMAC algorithm

input: Message M consisting of blocks of length b, Key K with length $0 < |K| < b$.
output: The hash authentication code for the message M
while |K| < b **do**
    $K \leftarrow 0 \circ K$;
end while
$K^+ \leftarrow K$;
$Signature1 \leftarrow K^+ \oplus RepeatedToLengthb(00110110)$;
$Temp \leftarrow SHA256(Signature1 \circ M)$;
$Signature2 \leftarrow K^+ \oplus RepeatedToLengthb(01011100)$;
$Result \leftarrow SHA256(Signature2 \circ Temp)$;
return $Result$;

2.3 OpenTitan Hardware Components

The OpenTitan chip contains many hardware components. In this section, we will go through some of the components we find the most central for the overall design.

2.3.1 The HMAC Module

The Hash-based Message Authentication Code (HMAC) Module is an authentication code generator. Its purpose is to check the validity of signed messages. It uses a 256-bit secret key to generate the authentication codes. The resulting authentication code will be different if the secret key is different, even if the message is the same. The secret key is obtained from the key manager component (see section 2.3.3). The HMAC module executes the HMAC algorithm (described later in Algorithm 3) when called by other components. The HMAC module uses the SHA-256 hash algorithm for hashing [21].

2.3.2 The KMAC module

The Keccak Message Authentication Code (KMAC) module is similar to the HMAC module since it is also used to check the validity of signed messages. Unlike the HMAC module, the KMAC module supports SHA-224, SHA-256, SHA-384, and SHA-512, which are SHA3 algorithms. The KMAC module receives the secret key and the message needed for its computation from the key manager component (see section 2.3.3). The KMAC module implements Domain-Oriented Masking (DOM)[5] logic to protect itself against certain attacks like Side-Channel Attacks (SCA) [23].

2.3.3 Key Manager

This section is written based on [24, 25]. To transition through the boot stages in OpenTitan, a Key Manager component is needed to manage the necessary cryptographic keys. The Key Manager's responsibility is to control what software accesses critical assets and allow the software to access derived keys when necessary. The Key Manager can be in different internal states. The internal states are Reset, Initialized, CreatorRootKey, OwnerIntermediateKey, OwnerRootKey, or Disabled, which is the order in which the states are reached. Some of the states are named after their respective internal key. From the Initialized state, advancement of the stages is done through the KMAC component as described in section 2.3.2.

[5]Domain-oriented masking is explained in [22].

Advancement through the states is irreversible. Thus the only way to reach a previous state is by resetting the system. Following the Initialized state is the CreatorRootKey state. The key manager transitions to the CreatorRootKey state at the end of the ROM stage so that the Key Manager is in this state during most of the ROM_ext stage. The state contains an internal key that is essential to the other operations performed in a given state. The internal key is hidden from software and cannot be accessed by hardware controlled by software. However, other hardware, such as the KMAC module, is supplied with the internal key. In a given internal state, the key manager can generate output keys and identities. The output keys are visible by software and are needed, e.g., as an input to the HMAC module. The following internal states are only used after the booting process is in the ROM_ext phase.
To reach the CreatorRootKey state, the KMAC module is supplied with a RootKey. The RootKey is in One Time Programmable memory and is therefore read-only. The other necessary input data is HealthMeasurements, supplied by the lifecycle controller described in section 2.3.4, a DiversificationKey, a DeviceIdentifier, and a HardwareRevisionSecret. The only secret data is the DiversificationKey, although HardwareRevisionSecret has "secret" in its name. The DiversificationKey is stored in flash. The DeviceIdentifier is a unique device id. The HardwareRevisionSecret is a constant set at design time.

2.3.4 Life Cycle Controller

The Life Cycle Controller is responsible for managing the life cycle state from the point of booting. When the OpenTitan chip is powered on, the Life Cycle Controller will measure the life cycle state. Once the life cycle state is measured, the state's values are broadcast to the rest of the device. The life cycle state is, e.g., used by the Key Manager to generate keys. When running the life cycle controller can take requests from software or other hardware components to change the life cycle state[26].

2.3.5 The Big Number Accelerator Module

The OpenTitan Big Number accelerator (OTBN) is a processor that is responsible for executing cryptographic operations like RSA and Elliptic Curve Cryptography (ECC). Because the domain of cryptographic operations is typically big numbers, the OTBN module uses registers of 256 bits and instructions that can operate on words of that size. The OTBN is designed to be very secure and to avoid data leakage. This is done by separating the control flow from the data and by simplifying the design of the module[27].

2.3.6 SRAM Controller

The SRAM Controller is directly responsible for managing the SRAM, which is the volatile storage unit of the OpenTitan design. At the time of writing this report, the SRAM Controller [28] is not well documented. The documentation describes that the SRAM Controller requests scrambling keys from OTP to scramble SRAM data. It also reports SRAM integrity errors as well as hardening security when such errors are detected. We also expect that it, similarly to the Flash Controller section 2.3.7, acts as an interface between software/hardware and SRAM. However, it is not clear to us whether this is the case based on the documentation. It is stated in [28] that "The memory inside the SRAM controller can be used right away after a system reset.", but this does not explicitly say whether the SRAM Controller handles SRAM access. [28] also includes a set of steps for initializing the SRAM Controller so that the scrambling key is correctly set. These steps include requesting a key from OTP, checking if the received key is valid, (optionally) locking down the option for requesting a new scrambling key, etc. It is also stated that these steps should be performed as early in the boot process as possible by software.

2.3.7 Flash Controller

The Flash Controller can be seen as an interface between flash memory (which is the permanent storage of the OpenTitan design) and other components. It consists of both a Flash Protocol Controller and a Flash Physical Controller. The Flash Protocol Controller acts as an interface between the Key Manager, Life Cycle Controller, software, and the Flash physical controller. Therefore, the Flash Protocol Controller translates requests made by software to something understood by the Flash Physical Controller. The Flash Physical Controller is a wrapper around the actual flash. Therefore, it is responsible for dealing with scrambling as well as translating the requests made by the Flash Protocol Controller to vendor-specific signals [29]. The scrambling performed by the Flash Physical Controller does not use a scrambling history; thus, if scrambling is enabled when writing but not when reading or vice versa, the result will be incomprehensible.

2.3.8 OTP Controller

This section is based on [30]. The OTP Controller is responsible for handling the One Time Programmable memory (OTP) that is part of the OpenTitan design. The controller provides an interface between hardware/software and the OTP. Specific components such as the Life Cycle Controller (cf. section 2.3.4) and the Key Manager (cf. section 2.3.3) can directly interface with the OTP controller. It also provides software isolation if the contents of the OTP are programmable and readable and incorporates logical security protection at a high level (e.g., integrity checks and scrambling). Furthermore, the OTP controller has the following features:

- The OTP controller maintains multiple logical partitions of the OTP memory.
- The OTP controller provides persistent and periodic checks of values stored in OTP.
- The OTP controller scrambles secret partitions via a lightweight scrambling method that uses a global netlist constant.
- The OTP controller contains lightweight key derivation functions for deriving scrambling keys for the SRAM and Flash scrambling mechanisms.

The partitions into which the OTP are separated represent different functions. The isolation between these is virtual and is maintained by the OTP Controller. Each partition has certain properties that are enforceable: Confidentiality via secret partitions, Read lockability, Write lockability, and Integrity verification. Confidentiality via secret partitions pertains mostly to the fact that a partition can be marked as secret. If a partition is marked as secret, it is not readable by software when locked, and it is scrambled in storage. Read lockability is the ability to control whether a partition is readable via software. Write lockability means that a partition can be locked so that it cannot be updated in the future. Furthermore, such partitions are stored alongside digests to be used for integrity verification at a later point.

2.3.9 Entropy Source

This section is based on [31]. The Entropy Source is a noise-generating component. The generated noise is represented as bits, and these bits are used as seeds by other components. Both hardware components and firmware components can use the seeds generated by the Entropy Source component. Among the components using these seeds is the CSRNG as described in section 2.3.10. The Entropy Source component provides two sources of entropy, the first being a pseudo-random digital entropy source, Linear Feedback Shift Register. The second source of entropy is a true random number generator source, Physical True Random Number Generator, which is provided by an external interface. [31]

2.3.10 CSRNG

This section is written based on [32]. CSRNG is an abbreviation for Cryptographically Secure Random Number Generator. The component will, based on the seeds provided by the Entropy Source module described in section 2.3.9, compute random values. In combination with the Entropy Source, the CSRNG can provide deterministic random number generation and true random number generation. To perform the random number generation, the component relies on the entropy from the Entropy Source module.

2.3.11 Entropy Distribution Network

The section is written based on [33]. When random numbers have been generated by the CSRNG as described in section 2.3.10, the Entropy Distribution Network (EDN) is used to distribute the numbers to the other components in the system. Furthermore, the EDN is also responsible for managing requests to the CSRNG. Whenever a component issues a request for a random number, the EDN is used. The EDN relies on four request/acknowledge hardware interfaces to communicate with the components making the requests. The EDN uses two FIFOs to deal with commands for generating numbers and for reseeding to manage requests.

2.4 OpenTitan Scrambling

As mentioned in section 2.3.6 and section 2.3.7 OpenTitan utilizes scrambling for several components including SRAM and Flash. The scrambling performed in both components uses a PRINCE block cipher to perform the scrambling. The purpose of scrambling is to prevent leakage of secrets in the system since sensitive data can reside in both the SRAM and the Flash. Each of the respective storage components has a controller which manages the scrambling, while the PRINCE Block Cipher component performs the scrambling.

The Flash Controller does not keep any record of which addresses are scrambled. Since scrambling is not enabled for all addresses in Flash, misreading unscrambled data as scrambled will result in gibberish (cf. section 2.3.7). The same goes the other way around [29]. The SRAM controller always has scrambling enabled, and therefore it does not have the same potential for an incorrect read. The SRAM Controller and Flash Controller rely on the OTP Controller to provide them with a scrambling key. The SRAM controller is, however, provided with a predefined scrambling key used just after a reset. It will use the key until software has requested another key from the OTP Controller [28].

2.5 OpenTitan Structure

To give an overview of how the parts of the OpenTitan platform (hardware and software) interact, we have created the diagram seen in Fig. 2.5.1. This diagram shows the flow of data between some of the more prevalent hardware modules along with how they interact with the ROM boot stage described in section 2.1. As a disclaimer, it is important to note that this diagram is not complete. It does not contain all the hardware modules of OpenTitan. This is because we have tried to only show the more interesting modules and because we have tried to only show things in the diagram that we have a high degree of certainty about. We have created the diagram by reading through the available OpenTitan documentation. Furthermore, since OpenTitan is still in development, how the modules interact might change compared to how they are presented in the diagram.

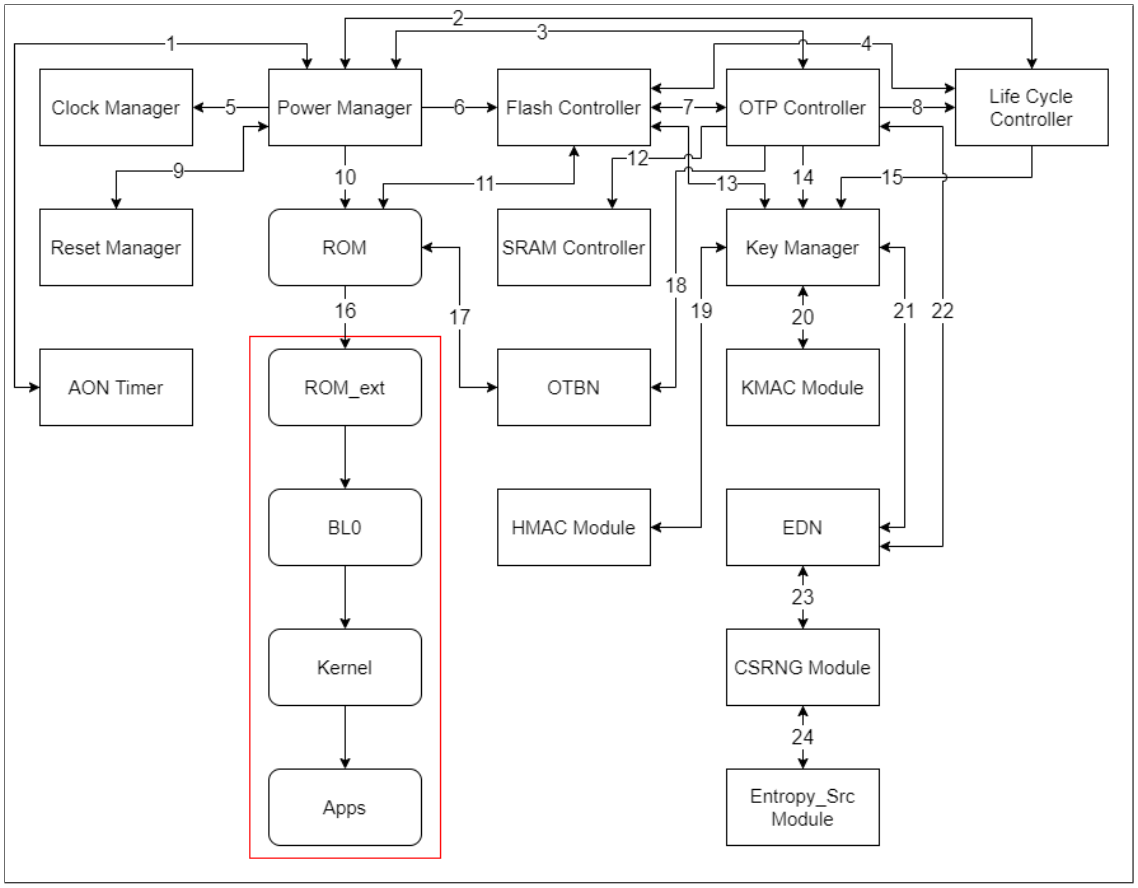


Figure 2.5.1: A diagram of how some of the modules interact in OpenTitan.

The arrows seen in Fig. 2.5.1 show the direction of communication between the modules. The numbers seen on the arrows are used to refer to specific lines of communication further into this section to describe why the modules communicate as they do. The numbers **do not** represent any order in which the modules communicate. Boxes with soft edges represent firmware/software, and boxes with hard edges represent hardware. Lastly, we have made a box (drawn in red) that encapsulates ROM_ext, BL0, Kernel, and Apps seen at the bottom left of the figure. We do not show how the firmware/software interacts with the hardware components outside the box because we are only interested in the ROM stage of the booting process. The following list describes the communication in the figure (item 1 describes the arrows in the figure labeled with 1, etc.). In this section, we do not describe each module in detail but focus on why the modules interact. A more in-depth description of some of the modules shown in Fig. 2.5.1 can be seen in section 2.3.

1. The AON (Always-ON) Timer contains two timers that count up from zero: the AON Wakeup timer and the AON Watchdog timer. It communicates to the power manager by pinging it regularly when these timers reach a threshold. Once the AON pings the power manager, it waits for acknowledgment and resets its timers [34].
2. The Power Manager initializes the Life Cycle Controller and gets an acknowledgment back that the initialization is complete [35].
3. The Power Manager initializes the OTP Controller and gets an acknowledgment back that the initialization is complete [35].

4. There are two purposes for the communication between the Flash Controller and the Life Cycle controller [29]:
    - The Life Cycle Manager can send a return material authorization (RMA) request to the Flash Controller. When the RMA entry process is complete, the Flash Controller notifies the Life Cycle Controller.
    - The Life Cycle Controller can enable an information page that holds secret seeds for the key manager.
5. It can be seen from [35] that the Power Manager communicates with the Clock Manager. However, the exact reason why is not documented.
6. The Power Manager powers up flash, but does not initialize it (this is done by ROM)[35].
7. The OTP Controller provides keys for scrambling to the Flash Controller. The OTP Controller can also enable information partition pages in flash that contain secret seeds for the Key Manager [29, 30, 36].
8. The Life Cycle Controller reads the contents of the life cycle partition in OTP and decodes the life cycle state [26].
9. The Power Manager communicates reset requests to the Reset Manager. The Reset Manager communicates reset requests from the AST to the Power Manager [35].
10. The ROM stage of the boot process starts after the system powers up (when the Power Manager starts the system).[14, 37]
11. ROM initializes flash before using it. ROM reads, writes, and executes flash [35, 38].
12. The OTP Controller derives a scrambling key that it gives to the SRAM Controller [30, 36].
13. The Flash Controller provides secret seeds to the Key Manager if the OTP Controller enables the pages containing the seeds and Life Cycle Controller [29, 24].
14. The OTP Controller provides a root key to the Key Manager [30, 36].
15. The Life Cycle Controller provides the Key Manager with health measurements (which is the current life cycle state) [24].
16. ROM transfers execution to a validated ROM_ext as described in section 2.1.
17. The ROM (mask_ROM) communicates with the OTBN to do signature verification. The OTBN is the module used to do calculations needed for the RSA algorithm [27, 38].
18. The OTP Controller derives and gives a scrambling key to the OTBN [30, 36].
19. To perform the HMAC algorithm, the HMAC module needs a key to use. This key is provided by the Key Manager [21, 24]. It is not clear to us exactly where the HMAC module is used. It is stated in [38] that it is used during signature verification, but this might not yet be implemented.
20. When a key is requested by software or hardware, the Key Manager takes the current secret key and does a KMAC operation on it. It then exposes the output of the KMAC operation as the requested key [24].
21. The Key Manager gets entropy (randomness) from the EDN when it goes from the Reset state to the Initialized state [24].
22. The OTP Controller requests entropy from the CSRNG (via the EDN) when deriving the SRAM scrambling key [30].

23. The EDN acts as an interface for the CSRNG [39].
24. The CSRNG generates random numbers with entropy from the Entropy_Src Module [40, 41].

The following is a list of hardware modules that are mentioned by OpenTitan but that we could not fit into the diagram due to not knowing where exactly they are used:

- AES [42].
- Alert Handler [43].
- NMI Generator [44].
- Interrupt Controller [45].
- Timer [46].

There are more modules described in the OpenTitan documentation that are not mentioned here. This is because we assume these modules to be basic computer components that are not exclusive to OpenTitan (GPIO, USB 2.0, etc.).

While we have added the SRAM Controller to the figure seen in Fig. 2.5.1, we cannot find any concrete documentation that gives an exhaustive overview of components that communicate with the SRAM Controller. Furthermore, the OpenTitan SRAM Controller documentation page [28] does not spend much time in explaining the fundamentals of the module or the underlying SRAM. We assume that SRAM is used like regular working memory in the system with the added feature that the SRAM Controller can scramble SRAM. This is an assumption that we use in the model of ROM stage presented in chapter 5.

2.6 Physical Memory Protection

This section is based on [47]. The OpenTitan system builds upon a RISC-V architecture. RISC is an abbreviation of Reduced Instruction Set Computer. RISC-V is also an open-source standard Instruction Set Architecture (ISA)[48].
The RISC-V ISA supports a hardware unit that limits the access that processes have to physical memory regions. The unit is called the Physical Memory Protection (PMP) unit. The PMP unit can specify whether a memory region can be accessed (i.e., read, written, or executed). Utilizing the PMP functionality allows for more secure processing. If a memory region is locked by the PMP unit, then Machine mode must also respect the access rights set in that memory region. Unlocking locked memory requires a system reset. User and Supervisor mode software cannot access the region if no fields are set in a given memory region. Thus the PMP unit can also give various access permissions to User and Supervisor modes by setting the read, write and execute bits of an unlocked region. If a process tries to access memory that will violate the restrictions set by the PMP unit, an exception will be thrown.
RISC-V allows up to 16 PMP entries of 8-bit configuration registers, and an MXLEN[6]-bit address register. The configuration registers are stored in Control State Registers (CSR's). If one of these PMP entries is implemented, then all 16 need to be implemented. However, the CSR's can be hard-coded to zero if not all need to be used. The configuration register contains parameters for whether and how the memory can be accessed. A graphical representation of the contents of the configuration register can be seen in Fig. 2.6.1. Each box in the figure corresponds to one bit. The L bit signifies whether the region is locked or unlocked. Locked regions also apply to Machine mode. 0_0 and 0_1 are unused, which means that the values are always

[6]MXLEN is the width of a register in Machine mode.

0. The A bits are used for address matching. Address matching is explained later. The X, R, and W bits are used to allow execution, reading, and writing, respectively.

Figure 2.6.1: Format of a PMP configuration register.

Whenever memory access is necessary, the PMP component must check the access right for the specific memory location. The 16 PMP entries are prioritized in numerical order. Thus PMP entry N will always have precedence over entry $N + 1$. If memory access is issued to a range of addresses, then for each PMP entry, if that PMP entry covers the whole range, that region's X, R, and L bits determine the access rights. Otherwise, the next PMP entry is queried, and so on. If access issued from Machine mode has no matching PMP region, the access will still succeed. This is not the case for User/Supervisor mode, for which access will fail if the PMP is implemented and a matching region is not found.

The range of a PMP entry can be expressed in three different address matching modes: Top of Range (TOR), Naturally Aligned four-byte (NA4), or Naturally aligned Power-Of-Two (NAPOT). This is expressed in the 'A' region in Fig. 2.6.1. A TOR region works by using the previous PMP-region's top address element as the bottom element of the range for the current region and uses the address in its address register as its top element. TOR PMP regions ensure that they do not overlap with the previous region and that there is no gap in memory after the previous PMP region. An NA4 region is consistently a 4-byte region, whereas the range of a NAPOT region is expressed as a power of 2 based on the address. The NAPOT region's smallest possible range is 8 bytes, and the largest range possible is 2^{XLEN+3}, where XLEN is the register width of the machine. The smallest possible region size across the different address matching modes is 4 bytes.

Chapter 3

Our Boot Code

In this chapter, we describe the boot code that we have created to mimic the OpenTitan design as closely as we can. Furthermore, we describe the security policies and goals that we want to verify about the OpenTitan system. This can be seen as the concrete case that we intend to model.

3.1 Boot Code Description

The work of this project builds on the boot code described in appendix B. The boot code is developed with the project group SV107f21. The developed boot code is inspired by the work of OpenTitan's open source ROM boot code available at [38]. The purpose of the ROM code is as presented in section 2.1. An excerpt of the code found in appendix B can be seen in listing 3.1.

The listing contains the `mask_rom_boot` function. This function corresponds to the ROM Boot process implemented in OpenTitan. The `mask_rom_boot` function can be thought of as the main function for the ROM stage of OpenTitan. It has the responsibility of initiating the call to all the other functions. The first task of `mask_rom_boot` is to read a boot policy from flash via the `read_boot_policy` function as seen in line 3. The boot policy will contain information necessary to determine in what way the boot shall proceed and what should happen upon failure[38]. The call to `read_boot_policy` will find the manifest in the device's flash. Accessing the flash happens through the flash controller section 2.3.7. The flash controller uses the PMP module to determine if the address is allowed to be read. If the address is not allowed to be read, an exception will be thrown. Once the boot policy is read, the next step is to find a valid ROM_ext manifest to boot. For this the function `rom_ext_manifests_to_try` is used in line 5 which takes a boot policy as input. As described in section 2.1.3, the ROM_ext manifests are to be found in the flash banks section of flash. Thus, the boot_policy determines which flash bank to investigate first and whether validating the ROM_ext manifest found in the other flash bank should be attempted.
The `mask_rom_boot` function will perform several checks for the found manifest, and it must pass all checks to take control of the boot process. The first check is done by `check_rom_ext_manifest` in line 12. This function is used to check the manifest for a specific identifier[14].

Next, we need to read the public key in line 17. The second check is done by the `check_pub_key_valid` function, which takes the public key as input. The `check_pub_key_valid` function checks if the public key in the manifest that is currently being validated is a trusted public key. This is done by checking that the key of the manifest is contained in an immutable list of trusted public keys stored in ROM. If the key is trusted and enabled, it can be used to verify the signature in the manifest. If the key is not found in the list of trusted keys, the check will fail, and the next ROM_ext manifest will be investigated. On the hardware level, the function `check_pub_key_valid` will also use the flash controller to access the flash and have the address for the access checked by the PMP module.

If the previous checks have both succeeded, the final necessary check to perform is to verify that the signature of the manifest is correct. This is done by the function `verify_rom_ext_signature`. The function relies on the OTBN to perform the RSA verification algorithm mentioned in section 2.2.1. If the signature is correct, the verification succeeds, and the ROM_ext image code is trusted. Then the PMP module must update the first PMP-region (PMP-region 0) so that addresses containing the ROM_ext image code are allowed to be executed. This is done with the call to the `pmp_unlock_rom_ext` function in line 30. In line 33 the current manifest will be used to start the next stage of the booting sequence, ROM_ext. If the function returns, the booting process has failed, which will get handled by the function `boot_failed_rom_ext_terminated` in line 35. If all the ROM_ext manifests to be tried by the boot policy failed the validation, the booting process has failed. This is handled by the function `boot_failed`.

```
1  void mask_rom_boot(void)
2  {
3      boot_policy_t boot_policy = read_boot_policy();
4
5      rom_exts_manifests_t rom_exts_to_try = rom_ext_manifests_to_try(boot_policy);
6
7      // Maybe step 2.iii
8      for (int i = 0; i < rom_exts_to_try.size; i++)
9      {
10         rom_ext_manifest_t current_rom_ext_manifest=rom_exts_to_try.rom_exts_mfs[i];
11
12         if (!check_rom_ext_manifest(current_rom_ext_manifest)) {
13           continue;
14         }
15
16         //Step 2.iii.b
17         pub_key_t rom_ext_pub_key = read_pub_key(current_rom_ext_manifest);
18
19         //Step 2.iii.b
20         if (!check_pub_key_valid(rom_ext_pub_key)) {
21             continue;
22         }
23
24         //Step 2.iii.b
25         if (!verify_rom_ext_signature(rom_ext_pub_key, current_rom_ext_manifest)) {
26             continue;
27         }
28
29         //Step 2.iii.d
30         pmp_unlock_rom_ext();
31
32         //Step 2.iii.e
33         if (!final_jump_to_rom_ext(current_rom_ext_manifest)) {
34             //Step 2.iv
35             boot_failed_rom_ext_terminated(boot_policy, current_rom_ext_manifest);
36         }
37     } // End for
38
39     //Step 2.iv
40     boot_failed(boot_policy);
41
42 }
```

Listing 3.1: Our `mask_rom_boot` function responsible for booting rom.

3.2 What we Want to Verify

To figure out the exact security properties that we want to verify about our boot code (cf. appendix B), we refer to the security analysis that we did on OpenTitan during our P9 project. This security analysis can be seen in [12, Ch. 5]. We made the security analysis to define the system and its context. It also defines several security policies and expands these policies into several smaller and verifiable goals. Lastly, it also defines a threat model for the system to gain an insight into how different kinds of attacks (spoofing, tampering, etc.) might affect the system and how likely these attacks are.
To define realistic properties that we want to verify about our boot code, we have looked at the security policies defined in [12, Ch. 5]. These policies and their individual goals are:

- **P1** Only code that has been validated can be executed.
  - G1: Before the ROM_ext image code is executed, the signature of the ROM_ext manifest must be validated by ROM.
  - G2: Before a BL0 image is executed, the signature of the BL0 must be validated by ROM_ext.
  - G3: The ROM_ext image signature must not change and must be signed by the Silicon Creator.
  - G4: The BL0 image signature must only change upon ownership transfer, and it must only be done by the Silicon Owner.
- **P2** Validation of a boot stage must only succeed if the boot environment is safe.
  - G5: ROM must use a key dependent on the environment to validate ROM_ext. The validation must fail if the environment has changed since the code was signed.
  - G6: ROM_ext must use a key dependent on the environment to validate BL0. The validation must fail if the environment has changed since the code was signed.
- **P3** Cryptography key leakage cannot happen.
  - G7: Keys must be managed by a dedicated hardware component.
  - G8: Correct keys are only accessible to intended receivers.
  - G9: Secret information stored in memory must be cleared or scrambled after the termination of the respective boot stage.
- **P4** All data access rights follow a privilege hierarchy.
  - G10: Writing to a memory section requires writing privilege.
  - G11: Reading from a memory section requires reading privilege.
  - G12: Execution of a memory section requires execution privilege.
  - G13: Downstream software cannot re-execute a previous boot stage.
  - G14: Downstream software cannot modify code of previously executed boot stages.

3.2.1 P1

The goals of the P1 security policy covers multiple stages of the boot process. The security goals in P1 that do not fit within the scope of this project are G2 and G4. This is based on the limitation of the scope made in chapter 1. Thus the goals that we want to verify about the ROM stage are G1 and G3. Verifying G1 will ensure that the code to be executed in the following stage is the code signed by the Silicon Creator, which is a trusted entity. Goal G1 is only valid under the assumption that G3 is also fulfilled, namely that it is, in

fact, the Silicon Creator that has signed the ROM_ext image. Therefore as stated in G3, it is necessary to know that the signature must not be changed.

3.2.2 P2

The goals of the P2 security policy also cover multiple stages of the boot process. The G6 security goal falls outside the scope of the project since it concerns the BL0 stage in the boot process. However, the G5 security goal falls within the scope of our project, so we will want to verify it. When the ROM_ext manifest image code is hashed during signing, multiple register values are appended before hashing. When the signature of the ROM_ext manifest is verified, the values from the same registers are appended before hashing. If the environment is different, then the verification of the signature should fail. This means that the verification will not succeed if the environment has changed.

3.2.3 P3

For the P3 security goals, we will verify G8 and G9. While we should be able to model a key manager component that manages any cryptographic keys, it is not really used until the system begins to enter the ROM_ext stage (when it enters the CreatorRootKey state cf. section 2.3.3). As such, G7 is out of the scope of this project since we are only looking at the ROM stage. The G8 security goal states that cryptographic keys should only be received by intended receivers. This will be limited to only be considered for the ROM stage. Lastly, it should be possible to verify that any secret information in memory is wiped once ROM transfers execution to the next stage.

3.2.4 P4

Goals 10-12 should be somewhat easily verifiable with Uppaal by including a model of a PMP module (cf. section 2.6) that keeps track of the rights assigned to parts of a memory model. Likewise, we can potentially "lock down" the ROM stage so that further execution cannot execute or change it without incurring fatal boot exceptions. However, since we do not intend to model more stages than the ROM stage, it will be difficult to realistically model downstream software trying to modify or execute previous code. Therefore we believe G13 and G14 to be out of scope for this project.

3.2.5 Security Mechanisms

Now that we have selected which of the goals from the security analysis in [12, Ch. 5] that we want to verify, we can look into the part of the security analysis that concerns security mechanisms. [12, Ch. 5] mentions several security mechanisms that OpenTitan employs to ensure the outlined security goals. The rest of this section will describe the security mechanisms mentioned in [12, Ch. 5] for each of the goals that we have chosen to include. While we do not intend to model OpenTitan one-to-one, we do, to some extend, base our model on the design of OpenTitan. Therefore, most, if not all, of these security mechanisms should be represented in the developed model (or have logical equivalents) to give an accurate representation of a system similar to that of OpenTitan.

G1: Before the ROM_ext image code is executed, the signature of the ROM_ext manifest must be validated by ROM.

To ensure G1, a hashing algorithm is used to compute the digest of the ROM_ext image. The OpenTitan Big Number Accelerator (OTBN) is used to verify the signature of the ROM_ext image. This is done with the RSA-3072 algorithm and the creator identity public key. OpenTitan Secure Boot HW Support requires that the ROM_ext manifest must be validated before leaving the ROM stage[15].

G3: The ROM_ext image signature must not change and must be signed by the Silicon Creator.

The security mechanisms needed for G3 are creator identity private key, which is used when computing the signature via the RSA-3072 algorithm. The PMP is also a security mechanism for this security goal since it limits the possibility of performing any writes to manifests' signatures.

G5: ROM must use a key dependent on the environment to validate ROM_ext. The validation must fail if the environment has changed since the code was signed.

The Key Manager hardware component derives keys throughout the booting process from the RootKey, HardwareRevisionSecret, and other information (i.e., the environment) [15].

G7: Keys must be managed by a dedicated hardware component.

The G7 security goal relies on the Key Manager, which controls what software can access the generated keys. The key manager is also responsible for deriving the keys.

G8: Correct keys are only accessible to intended receivers.

OpenTitan's key derivation scheme is designed such that it is only possible for the intended entities to have access to the correct inputs to the key derivation function.

G9: Secret information stored in memory must be cleared or scrambled after the termination of the respective boot stage.

Two security mechanisms ensure that G9 is upheld. First off, the SRAM controller is used to scramble SRAM. This controller gets a new key for scrambling at each system reset. Secondly, software registers are cleared, and flash is scrambled to erase keys after the ROM_ext stage. However, the second of these mechanisms is not in the scope of our model (since we model the ROM stage).

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

To ensure these goals, OpenTitan uses the RISC-V PMP feature. This allows for the system to create PMP regions that indicate the access rights for different parts of memory (cf. section 2.6). With the built-in PMP hardware module, the system can make sure that read, write, and execute privileges are handled correctly.

We believe that we can model our boot code in a way that lets us verify some, if not all, of the policies mentioned in this section.

Chapter 4

Model Checking

This section uses [49] as the primary source. [49] describes model checking as "(...) a computer-assisted method for the analysis of dynamical systems that can be modeled by state-transition systems." One of the more prominent methodologies within model checking for verification purposes is to establish some model P of a system in the form of a state-transition graph (also known as a Kripke structure) along with a separate specification ϕ of the system such that $P \models \phi$. A Kripke structure is a finite directed labeled graph with vertices representing states and edges representing transitions. A Kripke structure is formally defined in [49] as a triple $K = \langle S, R, L \rangle$ where:

- S is a finite set of states also known as the state space.
- $R \subseteq S \times S$ is a set of transitions also known as the transition relation.
- $L : S \rightarrow 2^A$ is a function that assigns a set of atomic propositions to states. If s is a state in S, then $L(s)$ is the set of all atomic propositions that evaluate to true in s.

A path through the graph of the Kripke structure can be seen as the behavior of a dynamic system. An example of a Kripke structure for a lamp can be seen in 4.0.1.

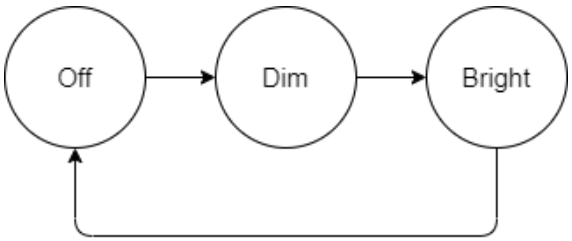


Figure 4.0.1: A simple Kripke structure for a lamp.

Early model checking was mostly done by graph algorithms that worked on these kinds of Kripke structures. However, as computer systems became more and more advanced, the gap between this approach and the systems that needed to be modeled grew. Advances within the field of model checking have helped to reduce the size of this gap. These advances can roughly be grouped into the following categories:

- The algorithmic challenge, which is concerned with designing algorithms for model checking so that they might scale to real problems.
- The modeling challenge, which is concerned with the extension of the framework of model checking so that it may go beyond Kripke structures and temporal logic.

The field of model checking encompasses many variations of the technique to deal with these challenges. Some variations are tailored for real-time systems, while others may include probability theory to better model uncertainty. Uppaal [50], PRISM [51], and Storm [52] are model checking tools that can be used to check properties about models.

Some of the more significant issues within model checking that are pervasive through all of these variants are those of "state space explosion" and what is known as "the modeling gap". State space explosion is the issue that the number of states in a model increases exponentially with its size. Therefore running out of memory becomes an issue on models of larger systems. The modeling gap refers to the fact that model checking, by its nature, uses models of systems instead of the actual systems. Therefore, the modeler of a system needs to be careful that their model carefully represents the actual system. If the gap between a model and its implementation is too large, then the model cannot be used to say anything useful/meaningful about the system.
Even with these problems and challenges, model checking is a powerful method for verification that provides a systematic approach that can potentially be fully automated. The method of model checking can also be used at different points throughout the development process. E.g., on abstract designs of the system or implemented code. Model checking also handles the concurrency of systems well due to how networks of models interact.

4.1 Timed Automata

To define timed automata we look at the definition found in [53]. In this they define a timed automaton as a quadruple of the following format:

$$(L,\ l_0,\ E,\ I)$$

With each of of the elements of the quadruple having the following meaning:

- L is a finite set of locations,
- $l_0 \in L$, is the initial location,
- E is a finite set of edges and is defined as: $E \subseteq L \times B(C) \times Act \times 2^C \times L$,
- I is a function that assigns invariants to the locations, defined as: $I : L \to B(C)$,
- C is a finite set of clocks, and $B(C)$ is a set of guards over the set of clocks,
- Act is a set of actions.

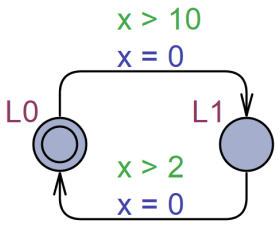


Figure 4.1.1: An example of a timed automaton in UPPAAL.

An example of a timed automaton can be seen in Fig. 4.1.1. A state of a timed automaton is a pair (l, v) where l is a location and v is a clock evaluation. A clock evaluation (v) is a function $v : C \to \mathbb{R}_{\geq 0}$ that evaluates the values of clocks. Increasing a clock valuation by a given non-negative real number d is expressed by $v + d$. Formally this is defined by $(v + d)(x) = v(x) + d$ for each $x \in C$. Resetting a subset of clocks r to 0 is expressed by $v[r]$. Formally $v[r](x)$ is defined as 0 if $x \in r$, and simply $v(x)$ otherwise. A timed automaton can change its current state either by firing an edge to go to a new location l and potentially resetting a number of clocks, or by delaying for a real number of time units. The transition relation is in [53] defined as:

$$(l, v) \xrightarrow{a} (l', v') \text{ if an edge } l \xrightarrow{a\ g\ r} l' \in E \text{ exists such that } v \models g,\ v' = v[r], \text{ and } v' \models I(l')$$

$$(l, v) \xrightarrow{d} (l, v + d) \text{ for all } d \in \mathbb{R}_{\geq 0} \text{ so that } v \models I(l) \text{ and } \forall d' : 0 \leq d' \leq d \implies v + d' \models I(l).$$

As an example, the initial state of the automaton in Fig. 4.1.1 is the pair (`L0`, x ↦ 0). Since the state does not satisfy the guard (`x > 10`), the edge to location `L1` cannot be taken. However, since the location of the state does not have an invariant, then all delays are possible. By delaying for 15 time units, the state is then (`L0`, x ↦ 15). From this state, the guard of the edge to `L1` is satisfied (i.e. (`L0`, x ↦ 15) ⊨ x > 10), and the invariant of the destination location (which is simply *true* for any clock valuation) is also satisfied. By taking that edge, the state becomes (`L1`, x ↦ 0). From this state the automaton can wait e.g. 3.2 time units to satisfy the guard of the next transition, and go back to the initial location. The following is the sequence of states above: (`L0`, x ↦ 0) $\xrightarrow{15}$ (`L0`, x ↦ 15) → (`L1`, x ↦ 0) $\xrightarrow{3.2}$ (`L1`, x ↦ 3.2) → (`L0`, x ↦ 0). Note that the last state in the sequence is the same as the first one, which implies that the sequence can technically repeat infinitely.

4.1.1 Networks of Timed Automata

This section is based on [54]. Networks of timed automata are compositions of timed automata over a common set of clocks and actions. A network of timed automata can be defined as a sextuple $A_i = (L_i, l_i^0, C, A, E_i, I_i)$, where $1 \leq i \leq n$. A location vector is a vector $\bar{l} = (l_1, \ldots, l_n)$. We use $\bar{l}[\ l_i' \ /\ l_i\ ]$ to express $\bar{l}$ where l_i has been replaced by l_i'. The invariant functions are composed into a function over location vectors $I(\bar{l}) = \bigwedge_i I_i(l_i)$. The semantics of timed automaton networks is defined as a transition system. A state of this transition system is of the form $S = (L_1 \times \ldots \times L_n) \times \mathbb{R}^C$. Initially the location vector is $(l_1^0, \ldots, l_n^0)$, and all clocks evaluate to 0 (i.e. $v(x) = 0$ for each $x \in C$). The transition relation is defined by the following:

$$(\bar{l}, v) \xrightarrow{d} (\bar{l}, v + d) \text{ if } \forall d' : 0 \leq d' \leq d \implies v + d' \models I(\bar{l}).$$

$$(\bar{l}, v) \xrightarrow{a} (\bar{l}[\ l_i' \ /\ l_i\ ], v') \text{ if there exists } l_i \xrightarrow{\tau\ g\ r} l_i' \text{ such that } v \models g, v' = v[r], \text{ and } v' \models I(\bar{l}[\ l_i' \ /\ l_i]).$$

$$(\bar{l}, v) \xrightarrow{a} (\bar{l}\ [\ l_j' \ /\ l_j,\ l_i' \ /\ l_i], v') \text{ if there exist } l_i \xrightarrow{c?\ g_i\ r_i} l_i' \text{ and } l_j \xrightarrow{c!\ g_j\ r_j} l_j' \text{ such that } v \models g_i \wedge g_j,$$
$$v' = v[r_i \cup r_j], \text{ and } v' \models I(\bar{l}\ [\ l_j' \ /\ l_j,\ l_i' \ /\ l_i]).$$

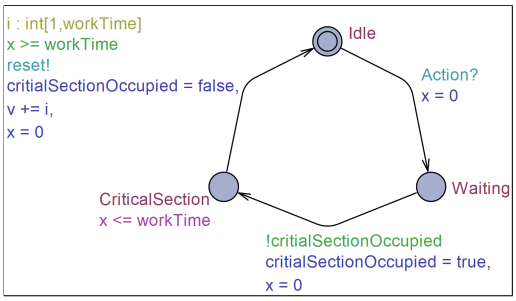
4.2 UPPAAL

Uppaal is a model checking tool used for the verification of real-time systems. The models in Uppaal are developed as timed automata, or networks thereof, and are graphically represented in the tool. The Uppaal model checker extends the standard definition of timed automata in section 4.1. The timed automata utilized in the Uppaal tool is extended with bounded integers, urgency, broadcast channels, committed locations, structured variables, user-defined types and functions, and more. The variables are part of the model's state. This means that two states are considered different, even if only their variables are different. Discrete variables can be used in the expressions, guards, or invariants of the model. Urgency is introduced both in terms of locations and channels.

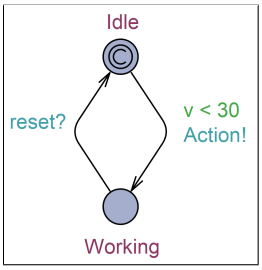
4.2.1 UPPAAL Features

We have created a simple Uppaal model to illustrate some of the more prominent features of the Uppaal formalism. This model can be seen in Fig. 4.2.2. It consists of three instances based on the two templates seen in Fig. 4.2.1. The model represents a boss and two workers in an office with only one workspace. This

means that the two workers cannot work at the same time. Furthermore, the two workers work differently. One worker can work for 5 time units, and the other can work for 10 time units. This is captured by the `Worker`'s only parameter `workTime`. Every time a worker works at the workspace, they generate some value, which is represented by the value of `v` increasing by the value of `i`, which in turn is bounded by the workers work time. The system stops when a sufficient amount of value is generated for the office, which can be seen in the fact that the `Boss` template has a guard on its transition.



(a) The `Worker` template used in the example model. Note the `workTime` parameter variable.



(b) The `Boss` template used in the example model.

Figure 4.2.1: The `Worker` and `Boss` templates.

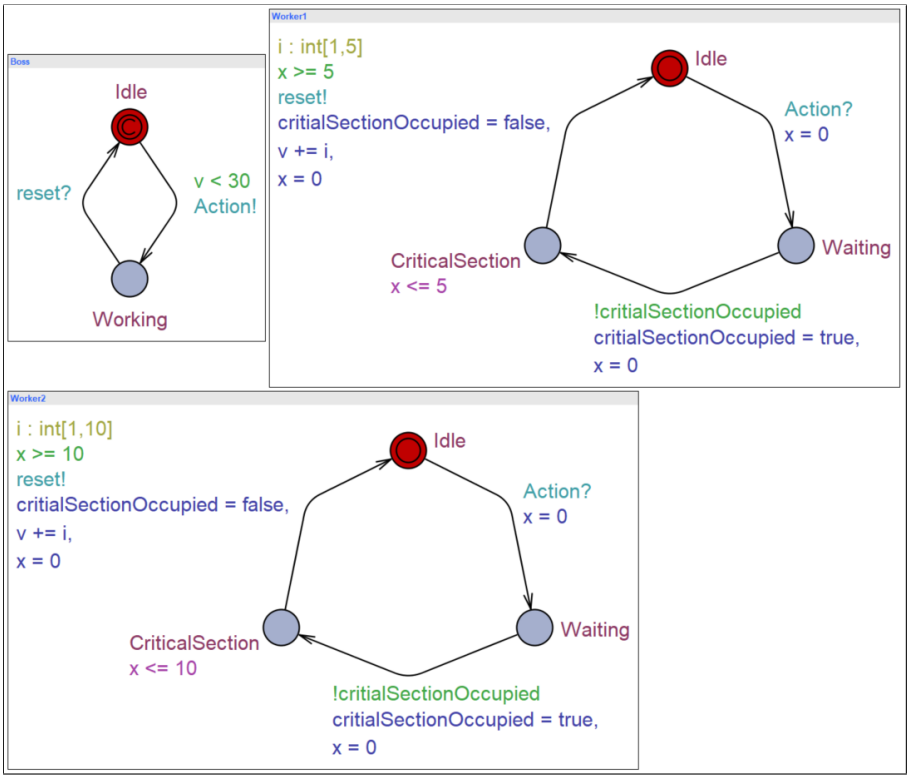


Figure 4.2.2: UPPAAL model containing three instances of templates.

UPPAAL's extension of timed automata means that guards and invariants can depend on variables and the results of user-defined functions, as well as clock valuations. Clocks are types of variables, which means that a template can have an arbitrary number of clocks. Each transition can update the variables. In the

example, criticalSectionOccupied, v, and x are variables. An enabled transition is an edge transition that is allowed to be fired in the current state concerning its guard, the invariant of the destination location, and whether another instance in the current state is in a committed location. Locations in UPPAAL templates can be marked as urgent or committed (marked with a U or C, respectively). In the example, the Controller's Idle location is committed. As long as a template instance is in an urgent location, delay transitions cannot be performed. This means that only edge transitions can be taken. As long as a template instance is in a committed location, delay transitions cannot be performed (like with urgent), and the next edge transition must be taken from a location that is committed. It is allowed to take a transition from a committed location to another committed location or even back to the same location.

For templates to communicate with each other, they use channels. Channels can be broadcast or binary, and they can be urgent or not. Binary channels have exactly one "sender" also called an "initiator" that initiates the communication, and exactly one receiver that is the recipient of the communication. An edge transition can only synchronize over one channel. A template cannot skip synchronizing if an edge synchronizes over a channel. If an urgent synchronization is enabled, the system cannot delay until the template moves to another location that does not have an enabled urgent synchronization. Broadcast channels have exactly one sender and an arbitrary number of receivers. When an edge that synchronizes over a broadcast channel fires, all potential receiving templates take the appropriate transition if the guard is satisfied. If one of these templates would enter a location whose invariant is not satisfied, none of the sending and receiving transitions can be fired. In Fig. 4.2.2 Action is a broadcast channel, and reset is a binary channel.

Transitions can have select clauses, which are used to simulate many similar edges. Select clauses generate variables to be used in a transition. Those variables exist only for the duration of the transition. The select clause generates a unique transition for each combination of values in the clause. This means that a number of transitions equal to the product of the sizes of the variable ranges are created. This means that for queries that search the state space, they will also have to check each branch of any select clause transitions. In Fig. 4.2.2, a select clause can be seen as the yellow text by the transition between the CriticalSection and Idle locations. Templates can take parameters, which are typically integers within a bound. When the templates are included in UPPAAL, a unique template is generated for each possible value and included in the model. In this example, there are two worker template instances with different values for workTime thanks to this feature.

4.2.2 UPPAAL Queries

To verify properties of developed models in UPPAAL, a subset of Timed Computation Tree Logic (TCTL) is used to formulate queries. TCTL specification requirements are written as either state formulae or path formulae, describing either a single state or the transition paths of the transition system, respectively. Single state formulae are basic expressions stating, e.g., which location a process is in. Path formulae are categorized as being either reachability, safety, or liveness properties [54]. UPPAAL queries can be in one of five forms:

- E<> phi searches the state space for a state that satisfies phi.
- E[] phi searches for a trace of transitions leading out of the initial state that all satisfy phi. The trace can either end in a deadlock or end in a loop to satisfy this.
- A<> phi searches each branch of the state space for whether phi is eventually satisfied.
- A[] phi looks through all states in the state space and returns whether all satisfy phi.

- `phi --> psi` Looks through the state space and returns whether whenever `phi` is satisfied, then `psi` is eventually satisfied during all following traces. It is equivalent to `A[] phi imply A<> psi`.

Reachability properties check if it is possible to satisfy `phi` at any point during any trace. A reachability property in UPPAAL is expressed as `E<> phi` [54]. An example of a transition system that satisfies a reachability property can be seen in Fig. 4.2.3.

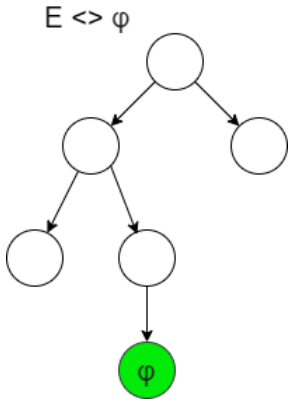


Figure 4.2.3: A transition system that satisfies the reachability property.

For the UPPAAL model shown in Fig. 4.2.2, we have made written a reachability query: `E<> v > 30`. In other words, is it possible to reach a state where the value of the variable `v` is greater than 30. By running the query, UPPAAL concludes that it is possible and gives a proof trace. The trace is that the `Worker2` instance can go into the critical section and increase `v` by 10 four times.

E[] queries check whether a property is always satisfied in at least one trace leading out of the initial state. An example of a transition system that satisfies a liveness property can be seen in Fig. 4.2.4.

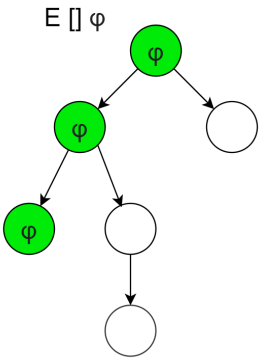


Figure 4.2.4: A transition system that satisfies an E[] property.

We have written an E[] query for the model in Fig. 4.2.2: `E[] v % 7 == 0`. The query asks whether there is a trace leading out of the initial state, such that the value of `v` is always a multiple of 7. UPPAAL provides a proof trace, which consists of taking the first transition and then waiting in that state infinitely. The value of `v` is 0 infinitely, which satisfies the query.

Liveness properties guarantee that `phi` is satisfied at some point along every path. In UPPAAL the liveness property is expressed as `A<> phi`. An example of a transition system that satisfies a liveness property can be seen in Fig. 4.2.5.

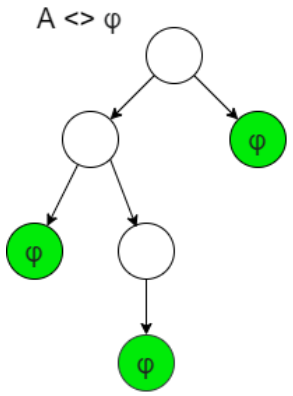


Figure 4.2.5: A transition system that satisfies the simple liveness property.

We have written the following simple liveness query for the model example in Fig. 4.2.2: `A<> Worker2.x == Worker2.workTime`, which asks whether `Worker2`'s clock `x` will ever be equal to the `workTime` parameter, which represents the clock's maximum value. UPPAAL concludes that the property is eventually true for any trace.

Safety properties check if some property `phi` is always satisfied during all traces. Thus all paths of the model must be checked if `phi` is always satisfied. If `phi` denotes a desirable property of the system, the safety property intuitively verifies whether something contradicting `phi` will ever happen. The safety property in UPPAAL is expressed as `A[] phi`. If one is only interested in ensuring that at least one path always satisfies `phi`, the syntax `E[] phi` can be used [54]. An example of a transition system that satisfies a safety property can be seen in Fig. 4.2.6.

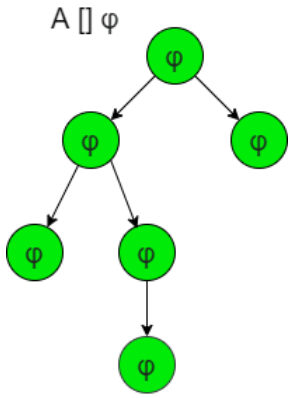


Figure 4.2.6: A transition system that satisfies the safety property.

For the model example, we have made written a safety property: `A[] not (Worker1.CriticalSection && Worker2.CriticalSection)`, that checks whether the two workers are ever in the critical section at the same time. This query is important to prove from a modeling perspective since violating the property means that there is a problem with the model. UPPAAL concludes that the property is satisfied, but is not able to provide a proof trace because the property concerns every trace, and not any trace in particular.

UPPAAL also supports an extension of the liveness property called *leads to*, expressing that if `phi` holds then another state formulae `psi` holds afterwards. The *leads to* property is expressed as `phi --> psi` [54]. An example of a transition system that satisfies a *leads to* property can be seen in Fig. 4.2.7.

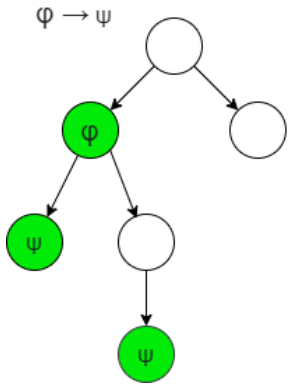


Figure 4.2.7: A transition system that satisfies the *leads to* property.

Our *leads to* property for the UPPAAL model example is the following: `Worker1.CriticalSection -->` ↪ `Worker1.Idle`. This query asks whether `Worker1` will always go to the `Idle` location whenever it enters the `CriticalSection` location. UPPAAL concludes that the property is satisfied.

Chapter 5

Modeling Boot Code in UPPAAL

In this chapter, we explain and show how we have modeled the boot code (cf. chapter 3) along with relevant OpenTitan modules in Uppaal. All modeling and verification are done using Uppaal version 4.1.24. The OpenTitan hardware components included in this model are listed below. We do not include all the components in OpenTitan due to not getting a coherent overview of how all the components interact in the first boot stage. The reason for this is the state of the documentation.

- OTP
- OTP Controller
- Flash
- Flash Controller
- SRAM
- SRAM Controller
- Entropy Source
- EDN
- CSRNG
- OTBN
- PMP
- Life Cycle Controller

Each function from the written boot code is represented as a separate template in the model. Thus channels are used to synchronize to other templates to emulate a function call and a function return. Channel names ending in "BC" are broadcast channels. In the model, we have a global variable called `NumberOfFlashBanks` which is a value that we can change to accommodate various numbers of flash banks. It is set to 1 in the current model. This is contrary to the OpenTitan design, which implements two flash banks. We have chosen to go with only having one since we want to reduce the state space of the model. Furthermore, the model generates all possible permutations of a manifest even though we only have one flash bank. Therefore we think that this abstraction is valid. However, by only having one flash bank, we will not be able to verify any of the combinations of rom_ext manifests or whether the model is suited to check the second manifest if the first one does not pass all the checks. The Uppaal model primarily passes information through global variables, but there are cases where channels are used instead (such as for reading from the `Flash` template). Global variable passing can be done atomically by setting the global variable to the value to be sent on the initiating transition (i.e., the transition with the "!"), and setting the local variable on the receiving transition (i.e., the transition with the "?"). This works because the update of the transition that initiates synchronization is executed before the update of the transition that accepts that synchronization. In the model, we clear variables regularly. This is to mitigate the state space explosion problem and should make the verification process faster. In the model, we have included a

"Zeno" template, which has one urgent location. This makes delays impossible in our model, which helps verification. This is because delays are considered a transition, and delaying infinitely is considered an infinite trace. This means that if delays are possible, then all A<> queries will fail because there is technically a trace that does not satisfy the formula. In the following sections, we will describe the templates that the model consists of rather than the instances. We do this because each template has exactly one instance in the model, and the templates do not take any parameters[1].

The figure presented in 5.0.1 displays which components interact with each other in the model and which channels they use to do so. The channel names are presented on the arrows between templates. Since the model is a representation of the OpenTitan ROM stage, this diagram resembles what is depicted in Fig. 2.5.1. Despite the similarity between the diagrams, there are some differences. Some of the components in the presented OpenTitan diagram are not incorporated in the UPPAAL model. These components are Clock Manager, Power Manager, AON Timer, Key Manager, KMAC module, HMAC module, Apps, Kernel, BL0. These components are not included because they do not seem relevant for the ROM stage and the security goals that we want to verify. Furthermore, the last three are not implemented because they represent software stages that we do not model. The UPPAAL model as seen in Fig. 5.0.1 also contains components not mentioned in Fig. 2.5.1. These components correspond to functions in the ROM stage because we have tried to divide the functions into different templates in the model. The reason for the deviations is because not all components of the OpenTitan project are sufficiently documented. Thus it is not clear how all of the components interact during the ROM stage. This has forced us to make assumptions about component interactions in some cases.

[1]The RomExt template is an exception to this. There is one instance of this template for each flash bank.

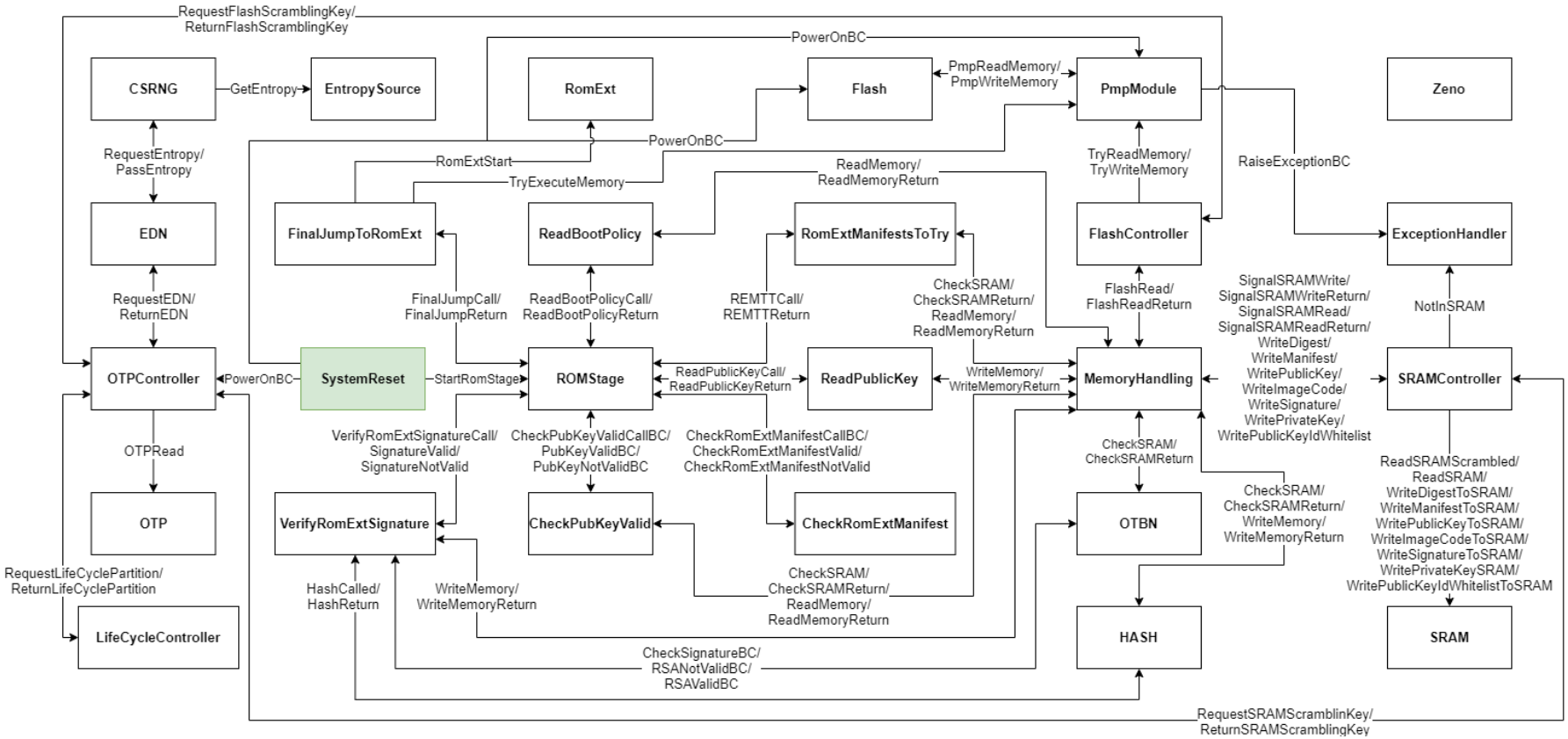


Figure 5.0.1: Diagram of the UPPAAL components and their communication. The `SystemReset` template is green because it initiates the first transition in the system.

5.1 Manifest Layout

The manifest is modeled based on OpenTitan and follows the structure seen in Fig. 5.1.1. In order of appearance in Fig. 5.1.1 the fields are:

- **Identifier**: The identifier is a string of data that identifies a given manifest. Currently in OpenTitan this is a 32-bit enumeration value [55].
- **Signature**: The signature is an RSA-3072 signature that is created using a trusted entity's private key along with a hash of the image code also found in the manifest. If a signature is valid, the system should always calculate the same signature using the public key in the manifest along with the hashed image code. This ensures that the code has not been changed since it was signed. Furthermore, if the signature is verified using a trusted public key, then the entity who created the manifest is trusted. Therefore the image code is also trusted.
- **Public Key**: This is the RSA-3072 public key used during signature verification (cf. section 2.2.1). It consists of two parts:
  - **Exponent**: This is a publicly chosen prime number (often 65537 in the case of RSA-3072).
  - **Modulus**: This is the product of two large prime numbers.
- **Image code**: The image code of the manifest is the executable ROM_ext that is associated with the manifest. This is the code that ROM transfers control to, given that the manifest is validated during ROM.
- **Entry point**: This field is one that we have added to the manifest in our model. It represents the address in the image code that needs to be jumped to once ROM_ext takes control of the booting process.

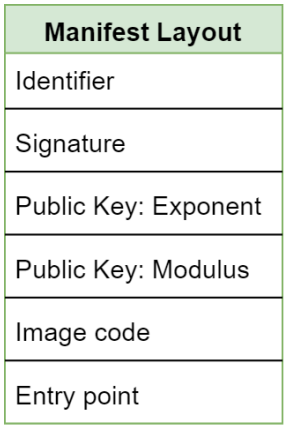


Figure 5.1.1: The modeled ROM_ext manifests.

All fields of the manifest are necessary to do the verification, except for the entry point. The entry point is used after all the checks have passed when the PMP module unlocks the address at which the image code starts to make it executable. The fields are not represented by actual values in the model, e.g., the public key exponent used in RSA encryption is not a large prime number. Instead, we assign a value of 1 to symbolize a correct value and a value of 2 to symbolize an incorrect value. The same principle goes for the other fields of the manifest, except for the entry point. The domain of the variables is limited because it is

not necessary to use realistic values for our verification. We only need to be able to classify the outcomes of a trace. E.g., for a manifest, the signature can be either correct or incorrect, leading to two possible outcomes. Furthermore, if we use large numbers and do complex calculations in the model, the state space would quickly become too large to do any verification.
The values of the fields and their meaning can be seen in table 5.1.1.

	Value	
Field	**1**	**2**
Identifier	Correct	Incorrect
Signature	Correct	Incorrect
Public key: Exponent	Correct	Incorrect
Public key: Modulus	Correct	Incorrect
Image code	Correct	Incorrect
Entry point	N/A (Address in the image code)	

Table 5.1.1: Table showing the meaning of the values of the manifest in the model. N/A means not applicable.

In the table, a correct value means that the value represents the value that is expected of that field in a valid manifest. An incorrect value means that a given field's value does not correspond to a valid manifest. E.g., a "correct" image code means that a trusted entity created the image code and that it has not been altered since signing the manifest. Incorrect in the same scenario means that the image code has been altered since signing (this is possibly done by an attacker).

We believe that the level of abstraction presented in table 5.1.1 is suitable for our model. We focus on how the correct and incorrect values impact the system. This means that we can verify the effect on the system of the manifest having been tampered with. The same goes for the other fields of the manifest. We are not interested in implementing actual values for keys and identifiers. It would dramatically increase the state space if actual keys and RSA algorithms were used and would not benefit the verification that we want to do. We are not interested in verifying the implementation of the algorithms. Furthermore, in the design of the checks, there are only two outcomes. E.g., the signature is either incorrect or correct (1 or 2). As of now, we have not found a need for more categorization of the values. If the design was made to handle signatures that are too long or too short differently, then this could also be encoded as 1 (correct), 2 (too short), and 3 (too long). However, this would require a lot of restructuring of the model. The structs that are used throughout the model, including the manifest, can be seen in appendix C.

5.2 Software Templates

The templates described in this section are all templates that correspond to functions in the boot code described in chapter 3.

5.2.1 The ROMStage Template

The most central template in our model is the `ROMStage` template as seen in Fig. 5.2.1. This template can be seen as a control flow graph (CFG) of the boot code described in section 3.1. The `ROMStage` template can be considered to be the `mask_rom_boot` function in listing 3.1. Each of the functions from the boot code described in section 3.1 are represented as separate templates, with the exceptions of *`pmp_unlock_rom_ext`*,

and *boot_failed*. The ROMStage template is connected to all the other templates that correspond to function calls and the SystemReset template.

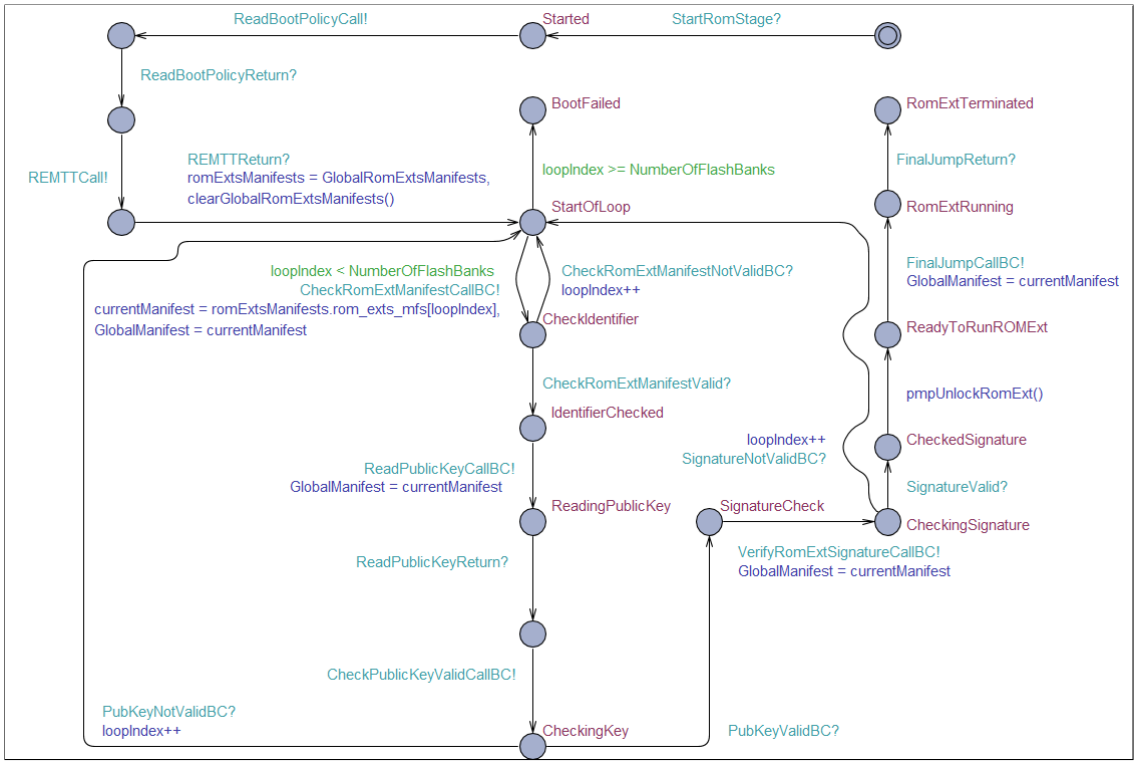


Figure 5.2.1: The ROMStage template in UPPAAL.

Variables & Functions	Descriptions
boot_policy_t bootPolicy	Stores the boot policy used to locate the manifests in Flash.
rom_exts_manifests_t romExtsManifests	Stores the manifests listed in the boot policy that the model should try to validate in order.
rom_ext_manifest_t currentManifest	The manifest that is currently being checked.
pub_key_t key	Holds the public key read from Flash.
int loopIndex	Index of the currentManifest.
void *pmpUnlockRomExt()*	Unlocks the rom ext to allow execution of it.

Table 5.2.1: The functions and variables for the ROMStage template.

The flow of the ROMStage template is initiated by synchronization over the StartRomStage channel. This leads the template to a Started location. From here, the template flow resembles that of the written boot code. The synchronizations on the edges will initiate the templates corresponding to a function from the boot code seen in section 3.1. Thereby the ROMStage template emulates a CFG where the locations are states of the boot code, and the edges are function calls. Before the StartOfLoop location is reached, the ROMStage template synchronizes with the ReadBootPolicy and RomExtManifestsToTry templates. This is done to firstly get the boot policy, and then use the boot policy to get the **rom_exts_manifests_t** from the RomExtManifestsToTry template. From the StartOfLoop all the manifests found will be iterated over until a valid manifest is found. A manifest is valid if it passes the checks that are performed

in the loop. The templates activated in the loop correspond to the checks described in section 3.1. The checks are: Validating the identifier, validating the public key, and validating the signature. First, we check the identifier, which must contain a "Magic value". This happens in the `CheckIdentifier` location. The next synchronization `ReadPublicKeyCall` will activate a template to read the public key from the current manifest and write it to `SRAM`. The synchronization initiates the public key check on the `CheckPublicKeyValidCallBC` channel. The signature check is done while in the `SignatureCheck` location. The signature is checked by several other templates. These are the `VerifyRomExtSignature`, `HASH`, and `OTBN` templates described in section 5.4. If any of the checks fail, the `StartLoop` is reached again. Otherwise the function *`pmpUnlockRomExt`* is called. Finally, the `FinalJumpCall` synchronization is fired to initiate the next stage of the boot process by synchronizing with the `FinalJumpToRomExt` template. For a description of the local functions and variables associated with `ROMStage` see table 5.2.1.

5.2.2 The ReadBootPolicy Template

The `ReadBootPolicy` template can be seen in Fig. 5.2.2. The template does not have any local variables or functions therefore there is no table associated with this template. The template is "called" by the `ROMStage` template and will synchronize with the `MemoryHandling` template. When the `ReadBootPolicy` template is in the `Called` location it will synchronize via the `ReadMemory` channel to read the given memory range [21; 29]. This is done to read a boot policy from the `Flash` template which is stored at those indices. The template is a simple representation of the *`Read_boot_policy`* function in section 3.1.

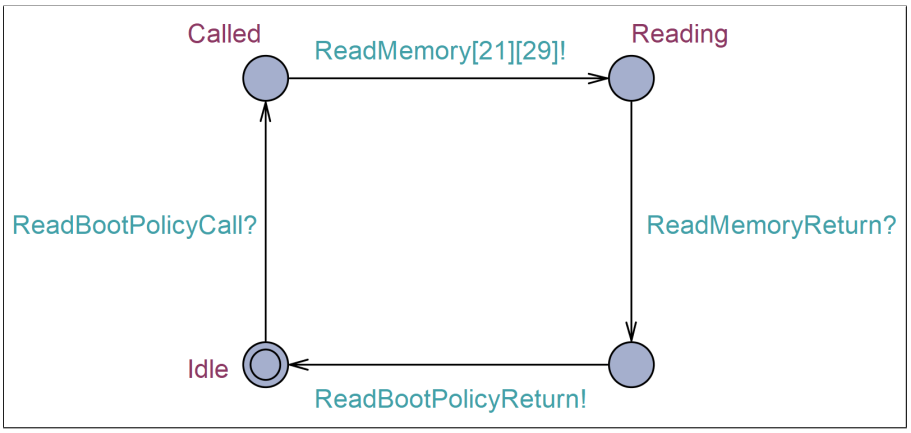


Figure 5.2.2: The `ReadBootPolicy` template in UPPAAL.

5.2.3 The RomExtManifestsToTry Template

The template seen in Fig. 5.2.3 is responsible for finding the manifests at the addresses contained in the boot policy. These manifests must be validated by the `ROMStage` template to continue the booting process. The template is modeled as a representation of the function called in line 5 of listing 3.1. After synchronization over the `REMTTCall` channel, the template is ready to check whether the boot policy is in `SRAM` to get the boot policy from which the manifests can be found. This happens by synchronizing with `CheckSRAM[0]`. After receiving the boot policy through global parameters, the template is in the `StartOfLoop` location, which will loop until `loopIndex == NumberOfFlashBanks`. In the loop the synchronization to `ReadMemory` will synchronize with the `MemoryHandling` template. As a part of this interaction with the memory handler, the manifests will later be stored in the `SRAM` template. The variables used in the `RomExtManifestsToTry` template can be seen in table 5.2.2.

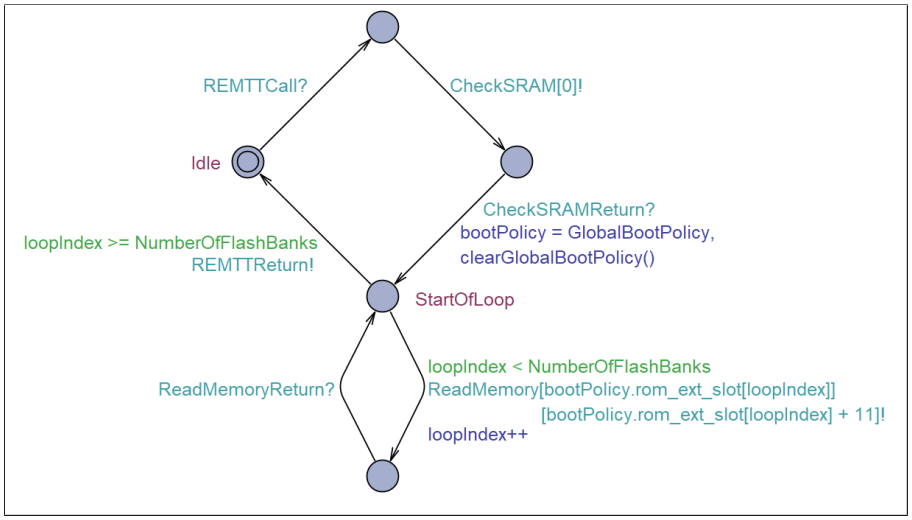


Figure 5.2.3: The `RomExtManifestsToTry` template.

Variables & Functions	Descriptions
`boot_policy_t bootPolicy`	Used to hold the local boot policy.
`int loopIndex`	Used to iterate through manifests.

Table 5.2.2: The variables for the `RomExtManifestsToTry` template.

5.2.4 The CheckRomExtManifest Template

The `CheckRomExtManifest` template can be seen in Fig. 5.2.4. This template models the `check_rom_ext_manifest` function in the boot code shown in section 3.1. The purpose of this template is to check the identifier of a given manifest. The local variables and functions used by the `CheckRomExtManifest` template can be seen in table 5.2.3.

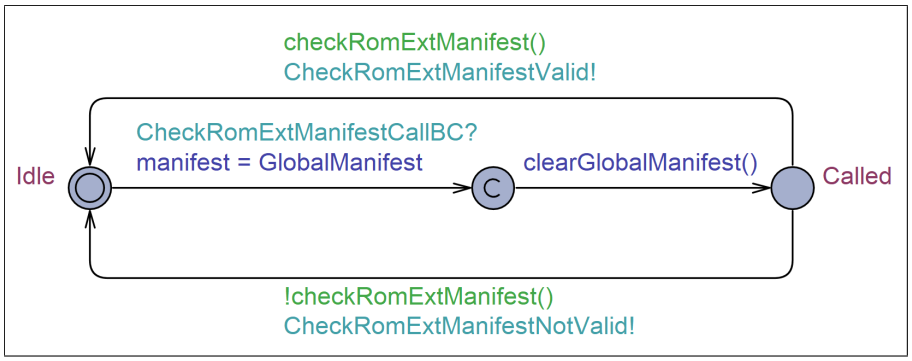


Figure 5.2.4: The `CheckRomExtManifest` template in UPPAAL.

Variables & Functions	Descriptions
`rom_ext_manifest_t manifest`	Variable used to hold the local manifest.
`bool checkRomExtManifest()`	Function used to check if the manifest's identifier is correct (has a value of 1).

Table 5.2.3: The variables and functions that are locally declared in the `CheckRomExtManifest` template.

The `CheckRomExtManifest` template stays in the `Idle` location until the `ROMStage` template synchronizes via the `CheckRomExtManifestCallBC` channel. This template receives the `currentManifest` from the `ROMStage` template via global variable value passing. From the `Called` location, there are two transitions which both lead back to the `Idle` location. Only one of these transitions is active at a time, since their guards are negations of each other. The choice depends on the result of calling the *checkRomExtManifest* function, which returns a boolean value. In the OpenTitan context, the function returns whether the manifest's identifier contains some magic header value[14]. In our implementation, the *checkRomExtManifest* function returns whether the manifest's identifier is equal to 1. If the *checkRomExtManifest* function returns true, then this template synchronizes with the `ROMStage` template via the `CheckRomExtManifestValid` channel. Otherwise, it does so via the `CheckRomExtManifestNotValid` channel. The `ROMStage` template takes different transitions based on the channel.

5.2.5 The ReadPublicKey Template

The `ReadPublicKey` template can be seen in Fig. 5.2.5. This template is responsible for reading the public key of a given manifest and putting this public key into SRAM so that it can be used by *CheckPubKeyValid* (cf. section 5.2.6). `ReadPublicKey` synchronizes with the `ROMStage` template, which "starts" the functionality in the template, and `MemoryHandling` which is used to write a public key to SRAM. The template is made to model the function `read_pub_key` in the boot code described in section 3.1. table 5.2.4 shows the locally used variable.

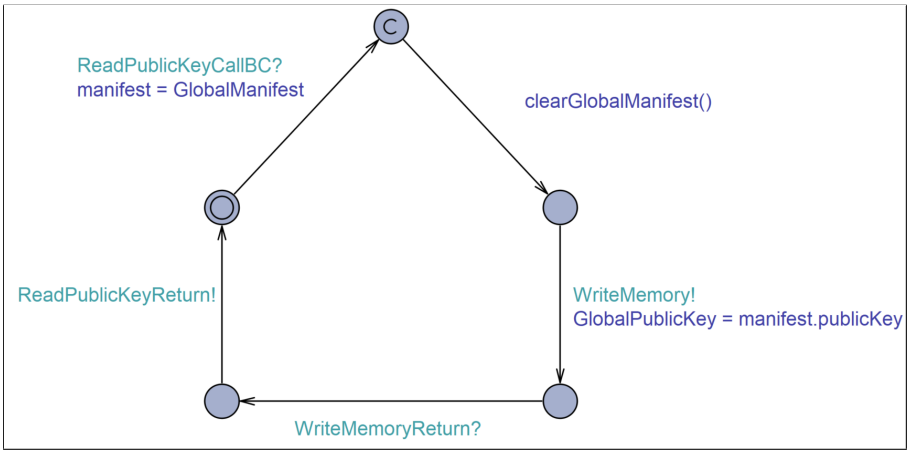


Figure 5.2.5: The `ReadPublicKey` template in UPPAAL.

Variables & Functions	Description
`rom_ext_manifest_t manifest`	Used to keep a local copy of the current manifest.

Table 5.2.4: Local variables and functions used in the `ReadPublicKey` template.

The template starts in the `Idle` location and waits for `ROMStage` to synchronize with it over the `ReadPublicKeyCall` channel. Once this synchronization happens, the `GlobalManifest` (which is set to be the current manifest by the `ROMStage` template) is copied into the local variable called `manifest`. Next, `ReadPublicKey` initiates synchronization with `MemoryHandling` via the `WriteMemory` channel and puts the current manifest's public key into the `GlobalPublicKey` variable and moves to the `WritingSRAM` location. The template stays in this location until `MemoryHandling` has written the public key of the current manifest to `SRAM`. Once `MemoryHandling` has done this, it initiates synchronization to `ReadPublicKey` over the `WriteMemoryReturn` channel. Once this synchronization is complete, `ReadPublicKey` synchronizes with the `ROMStage` template over the `ReadPublicKeyReturn` channel to indicate that it has successfully completed its tasks and moves back to the `Idle` location.

5.2.6 The CheckPubKeyValid Template

The `CheckPubKeyValid` template can be seen in Fig. 5.2.6. It corresponds to the function `check_pub_key_valid` described in section 3.1. This template is responsible for checking that the public key of the current manifest is a valid public key. In this case, a valid public key refers to a key for which the calculated ID can be found in a whitelist in immutable memory. The `CheckPubKeyValid` template synchronizes with the `ROMStage`, and `MemoryHandling` templates. The local variables and functions used in the template can be seen in table 5.2.5.

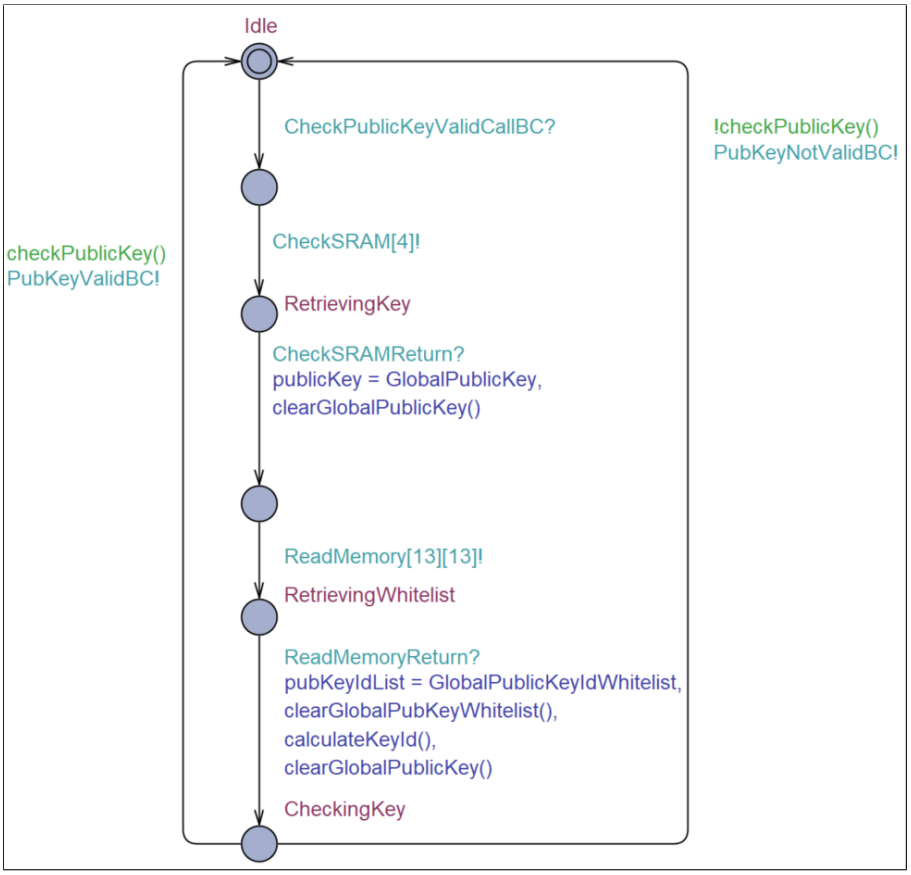


Figure 5.2.6: The `CheckPubKeyValid` template in UPPAAL.

Variables & Functions	**Description**
`pub_key_t publicKey`	Used to keep a local copy of the current public key
`int[0,NumberOfKeysInWhitelist] pubKeyIdList[NumberOfKeysInWhitelist]`	Used to keep a local copy of the key ID whitelist from flash.
`int[0,1] currentPubKeyId`	Used to keep the resulting ID from a public key.
`void calculateKeyId()`	Function for calculating ID for a locally stored public key. Implemented to set `currentPubKeyId` to 1 if the `publicKey` modulus and exponent are both 1 and 0 otherwise.
`bool checkPublicKey()`	Function for checking if the `publicKey` is valid. Returns true if the calculated `currentPubKeyId` is in the `pubKeyIdList`. False otherwise.

Table 5.2.5: Local variables and functions used in the `CheckPubKeyValid` template.

The `CheckPubKeyValid` template is in the `Idle` location until the `ROMStage` template synchronizes with it over the `CheckPubKeyValidBC` channel. Once this happens, `CheckPubKeyValid` synchronizes with `MemoryHandling` to read a public key from `SRAM`. While it waits for the public key to be available it is in the `RetrievingKey` location. Once `MemoryHandling` synchronizes over the `CheckSRAMReturn` chan-

nel, CheckPubKeyValid stores the given public key locally. After this, it once again synchronizes with MemoryHandling (over a different channel) to read the whitelist of key ID's from flash and moves to the RetrievingWhitelist location. When MemoryHandling synchronizes over the ReadMemoryReturn channel to signal that the whitelist is ready to be read CheckPubKeyValid stores the whitelist locally, calculates the ID for the current public key[2], and moves to the CheckingKey location. From this location there is only one enabled transition back to Idle. This is achieved with the guard checkPubKeyValid() on one transition and its negation on the other. This function returns true if the calculated ID for the current public key is in the whitelist and false otherwise. If the function evaluates to true then CheckPubKeyValid initiates synchronization over the PubKeyValidBC channel and moves back to the Idle. If not then it synchronizes over PubKeyNotValidBC and moves to the Idle location.

5.2.7 The VerifyRomExtSignature Template

The VerifyRomExtSignature template, seen in Fig. 5.2.7, models the function verify_rom_ext_signature in the boot code described in section 3.1. This function in the boot code simply returns the output of the RSA_VERIFY function on the current manifest's public key, hashed image code (the hashing is done just before evaluating RSA_VERIFY), and signature. The VerifySignature template synchronizes with the ROMStage, HASH, and OTBN templates. The template only implements one local variable. This can be seen in table 5.2.6.

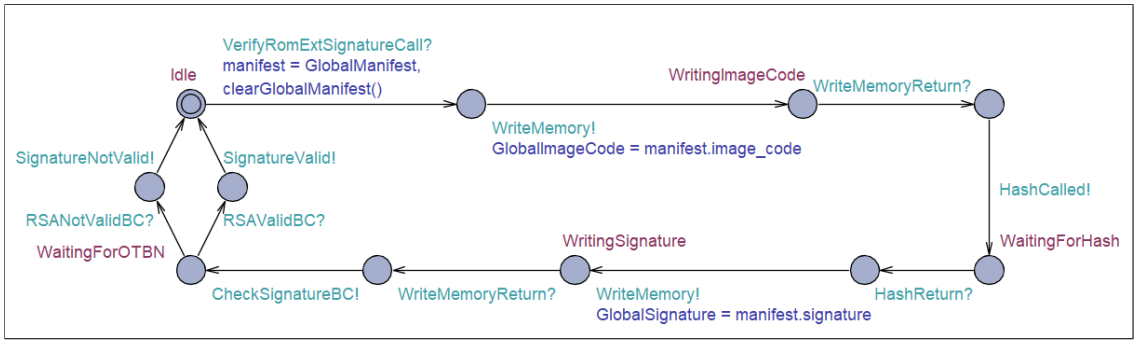


Figure 5.2.7: The VerifyRomExtSignature template in UPPAAL.

Variables & Functions	Description
rom_ext_manifest_t manifest	Used to keep a local copy of the current manifest.

Table 5.2.6: The local variables and functions for the VerifyRomExtSignature template.

The template is in the Idle location until it is "called" by the ROMStage template via synchronization over the VerifyRomExtSignatureCall channel. Once this happens, the VerifyRomExtSignature template takes what is in the GlobalManifest variable and copies it into the manifest local variable. While we normally go through SRAM to get such data, we have decided that the currentManifest found in the ROMStage template can be sent to other software templates via global variables. This can happen when ROMStage itself has read it from SRAM via other functions, and only when passing data between software templates as parameters. After this, the VerifyRomExtSignature template requests to write

[2]This is only done symbolically. If the public key exponent and modulus are both 1 then the ID will be set to 1. If not then it is set to 0.

the image code of the current manifest to SRAM. This is done by synchronizing over the WriteMemory channel and setting the GlobalImageCode to be equal to manifest.image_code and then moving to the WritingImageCode location. Once the MemoryHandling template signals that the image code has been successfully written to SRAM by synchronizing over the WriteMemoryReturn channel, VerifyRomExtSignature signals to the HASH template that it should start hashing. This is done by synchronizing over the HashCalled channel and moving to the WaitingForHash location. Once HASH signals that it has hashed the image code by synchronizing over the HashReturn channel, VerifyRomExtSignature writes the signature of the current manifest to SRAM in the same manner as it does with image code. Lastly, VerifyRomExtSignature signals to the OTBN that it needs to perform RSA signature verification by synchronizing over the CheckSignatureBC channel and moves to the WaitingForOTBN location. Once the OTBN has performed verification it synchronizes over either the RSAValidBC or RSANotValidBC channels depending on whether the signature is correct or not respectively. Then the VerifyRomExtSignature forwards this response to the ROMStage template and moves back to the Idle location.

5.2.8 The FinalJumpToRomExt template

The FinalJumpToRomExt template models the final_jump_to_rom_ext function seen in section 3.1. In a successful booting path, the last synchronization the ROMStage template will make is FinalJumpCall, this synchronizes with the FinalJumpToRomExt template. The template will receive the current manifest through a global parameter. The manifest contains an entry point to the image code, which is checked by the PMP module via the synchronization over TryExecuteMemory. If the execution is allowed, no exception is thrown. The next task is to start the ROM_ext stage. This happen with the synchronization RomExtStart. If a failure should happen while in the ROMExtRunning location, the ROM_ext stage will return with RomExtFail and then FinalJumpToRomExt can synchronize over FinalJumpReturn. The return FinalJumpReturn is used to emulate the behavior of the boot code described in section 3.1. It does so by having the ROMStage, described in section 5.2.1, go to its final failure state via the synchronization.

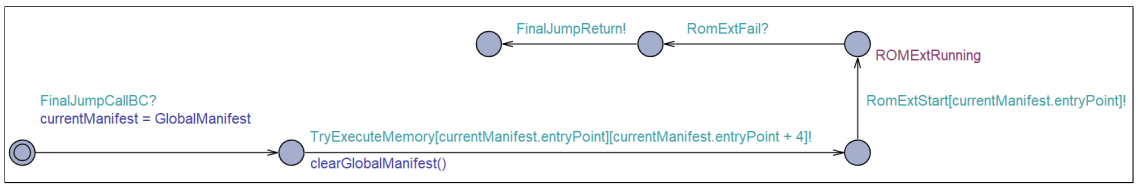


Figure 5.2.8: The template for FinalJumpToRomExt

5.2.9 The RomExt Template

The RomExt template is a placeholder for a given ROM_ext. Since the ROM_ext boot stage is out of scope for this project, we only model the ROM_ext to transfer execution to it. The template does not model anything from the OpenTitan ROM_ext boot stage.

Figure 5.2.9: The RomExt template in UPPAAL.

The template has no functions or variables; thus, we do not present a table for it. The template symbolizes the stage after the ROMStage. The template is included to mark the end of the first part of the booting

process. The only template that `RomExt` communicates with is the `FinalJumpToRomExt` template. The `id` parameter is a template parameter, indicating the ID of the manifest. The value of the identifier is multiplied by 35 due to the way we have modeled the memory where the beginning of each manifest resides 35 indices apart.

5.3 Memory Templates

The templates described in this section have to do with the memory-related aspects of the OpenTitan design. These are flash, SRAM, and OTP memory. Furthermore, their controllers are also described in this section. The PMP module is also described here as it sits between flash and the flash controller. We have also included a special template called `MemoryHandling` in this section which acts as an interface between memory-related operations and any template attempting to perform it.

5.3.1 The MemoryHandling Template

The `MemoryHandling` template (seen in Fig. 5.3.1) is a special template. It is one of few templates that does not have a direct counterpart in either our boot code (cf. chapter 3) or the OpenTitan design (cf. chapter 2). The other examples are the `Zeno` and `SystemReset` templates. We have created this template to centralize all memory operations in the model. In theory we could eliminate this template and move the functionalities of it out to other templates. However, this would create redundancy as we would need to have the functionality for reading and writing seen in Fig. 5.3.1 in multiple templates. We have chosen to avoid this redundancy so that changing how memory operations are done is easier. However, this limits the number of interleavings that are possible in the model since every memory operation is bottle-necked by the `MemoryHandling` template. The `MemoryHandling` template is responsible for handling flash reads, SRAM reads, and SRAM writes. It synchronizes with the `ROMStage`, `ReadBootPolicy`, `RomExtManifestsToTry`, `CheckPubKeyValid`, `VerifyRomExtSignature`, `HASH`, `OTBN`, `FlashController`, and `SRAMController` templates. The local variables and functions used in this template can be seen in table 5.3.1.

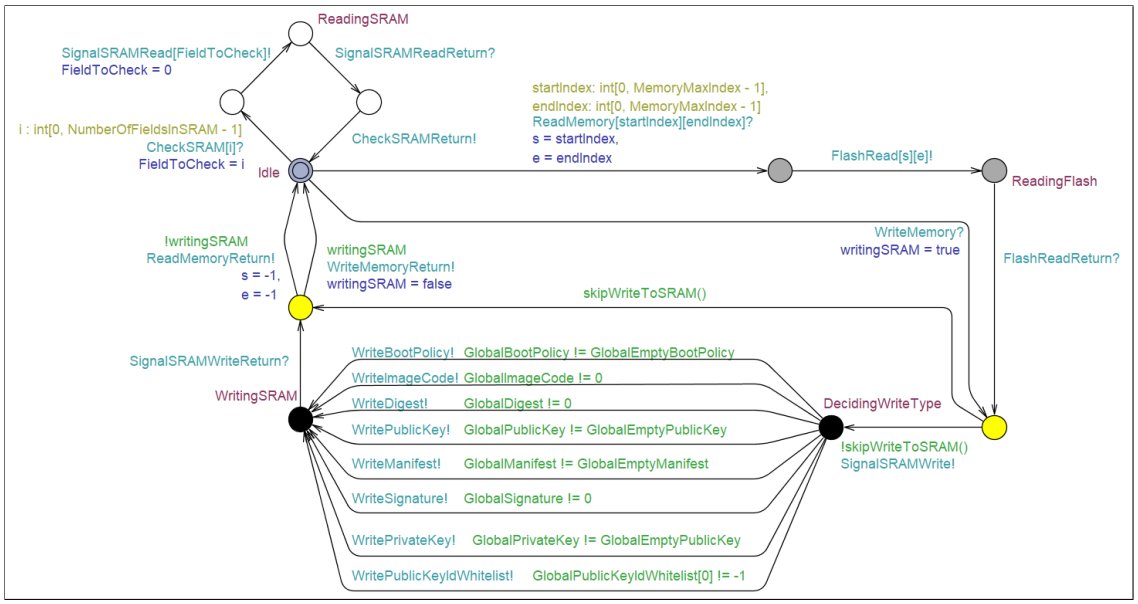


Figure 5.3.1: The `MemoryHandling` template in UPPAAL.

Variables & Functions	Description
`int s`	Used to indicate the starting index for a flash read operation.
`int e`	Used to indicate the end index for a flash read operation.
`int fieldToCheck`	Used to indicate which data field in SRAM that needs to be read.
`bool writingSRAM`	Used to indicate whether we need to write to SRAM or not.
`bool skipWriteToSRAM()`	Used to check if all global placeholder variables are empty. If true, then we need to skip writing to SRAM.

Table 5.3.1: Local variables and functions used in the `MemoryHandling` template.

Since the `MemoryHandling` template has multiple responsibilities, we have tried to partition it into its three main functions with colored locations. Reading flash uses the gray and yellow locations, reading SRAM uses the white locations, and writing SRAM uses the black and yellow locations.

Reading Flash

If the `MemoryHandling` template is in the `Idle` location and receives a synchronization over the `ReadMemory[startIndex][endIndex]` channel, it will initiate reading from flash memory. This is done by initiating synchronization with the `FlashController` over the `FlashRead[s][e]` channel[3] and moving to the `ReadingFlash` location. Once the `FlashController` initiates synchronization over `FlashReadReturn` we know that there should be a read response ready. After this, the model chooses to either put the read data into `SRAM` or circumvent this process. The template only circumvents writing to `SRAM`, if there is nothing in the global placeholder variables that can be written to `SRAM`. In the current model the template always writes the response from the `FlashController` template to the `SRAM` template.

Reading SRAM

`MemoryHandling` can handle read requests for the `SRAM` template as well as the `Flash` template. When in the `Idle` location it is possible for another template to initiate synchronization over the `CheckSRAM[i]` channel. Here `i` indicates the data field in `SRAM` that is supposed to be read. Once such a synchronization happens, the `MemoryHandling` template signals to the `SRAMController` that it wants to read from `SRAM` along with what it wants to read by synchronizing over the `SignalSRAMRead[fieldToCheck]` channel and moving to the `ReadingSRAM` location. In this location the `MemoryHandling` template waits for the `SRAMController` template to synchronize over the `SignalSRAMReadReturn` channel which indicates that the read request has been processed. After this the `MemoryHandling` template synchronizes over the `CheckSRAMReturn` channel to indicate to the initiator of the read request that it has been processed and returns to the `Idle` location.

[3] In this case $startIndex = s$ and $endIndex = e$. This means that we forward the indexes that we need to read to the `FlashController`.

Writing SRAM

As mentioned in the previous section the MemoryHandling template normally writes something to the SRAM template when it has been read from the Flash template. Once MemoryHandling has received synchronization over the FlashReadReturn channel as described in the previous section and the skipWriteToSRAM() guard evaluates to false, MemoryHandling synchronizes with the SRAMController template. This is done over the SignalSRAMWrite channel. Afterwards, MemoryHandling moves to the DecidingWriteType location. From this location there is one transition for each possible data type that can be written to the SRAM template in the model. Only one of these transitions is active when MemoryHandling is in the DecidingWriteType location due to the guards that state that the global placeholder variables used for passing values may not be 0 (or an equivalent empty value for structs). E.g. if what we are trying to write to the SRAM template is a boot policy, then the global variable called GlobalBootPolicy should be set to a non-empty boot policy. This enables the top-most transition which means that the MemoryHandling template will synchronize with the SRAMController template over the WriteBootPolicy channel and write the boot policy into SRAM. Once such a transition is fired, the MemoryHandling template moves to the WritingSRAM location and waits for the FlashController to synchronize over the SignalSRAMWriteReturn channel, which indicates that the data has been written to the SRAM template. Lastly, the MemoryHandling template synchronizes over the ReadMemoryReturn channel indicating to the template that initiated reading from flash that the process is complete.
It is also possible to write to SRAM without reading from Flash from the Idle location. This happens if another template initiates synchronization over the WriteMemory channel. After this synchronization the process of writing to the SRAM template is the same as already described in this section. The only exception is that the MemoryHandling template initiates synchronization over the WriteMemoryReturn channel instead of ReadMemoryReturn.

5.3.2 The Flash Controller Template

The FlashController template is as seen in Fig. 5.3.2. It resembles the OpenTitan Flash controller described in section 2.3.7 but does so at a high level of abstraction. It is not divided into two controllers (protocol and physical), but it interfaces with the same components in the model that it does in the OpenTitan system. Furthermore, it contains functionality needed during the ROM stage of the booting process that is similar to what is described in section 2.3.7. The local variables and functions used in the template can be seen in table 5.3.2. The template is responsible for managing all requests to read/write to and from the flash. Besides managing the requests to the Flash template, the FlashController template is also responsible for performing scrambling of data in the Flash template. The FlashController template needs a scrambling key, which it requests from the OTPController via the RequestScramblingKey channel. The process of getting the scrambling key starts in the initial location and ends at the Idle location. The FlashController template will forward all the requests it gets from the MemoryHandling template to the Flash template. From the Reading location, one of two edges is fired. The guard controlling this is GlobalMemoryIsScrambled. This variable is a bool indicating whether the FlashController template scrambled the most recently read value or not.

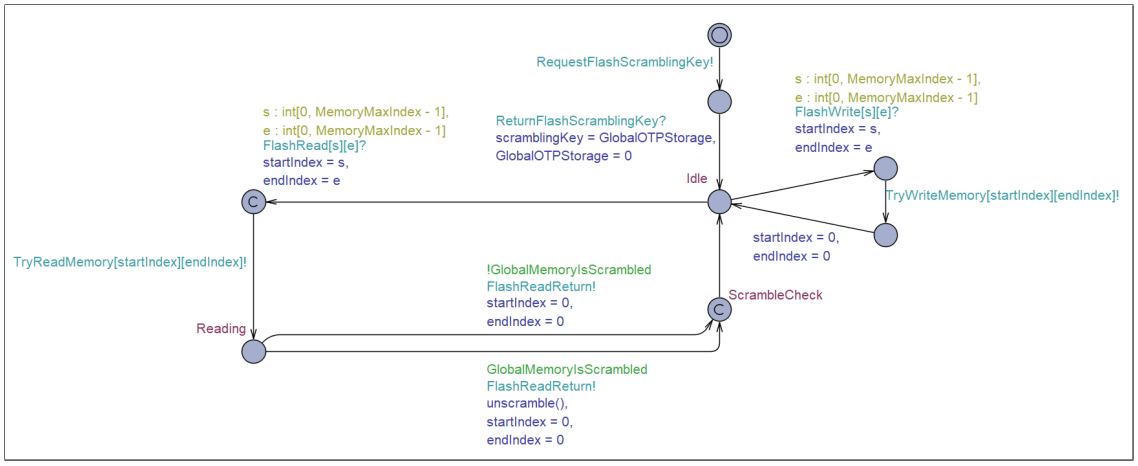


Figure 5.3.2: The `FlashController` template in UPPAAL.

Variables & Functions	Descriptions
`int startIndex`	Used as first index of a flash read.
`int endIndex`	Used as last index of a flash read.
`int scramblingKey`	Used to hold the scrambling key received from OTP.
`void unscramble()`	Function used to unscramble data if the scrambling key is provided.

Table 5.3.2: Functions and variables in the `FlashController` template.

5.3.3 The Flash Template

The `Flash` template is triggered by the `SystemReset` template synchronizing via the `PowerOnBC` channel. The `DecideManifests` location is included in the `Flash` template to allow for the "generation" of different manifests. To do this we use the select statements for choosing a value for `i`, `j`, `k`, `l`, `m`, which are assigned to fields of a manifest. This gives us the opportunity to try different combinations of field values in a manifest, without making any changes in the model. This models the **`rom_ext_manifest_t`**s already being in the `Flash` when the system starts. An alternative would be to have a specific manifest hard coded in the declarations of the `Flash` template. But this would require changing the code if different values should be verified. Once the guard `loopIndex >= NumberOfFlashBanks` holds, the `Flash` template gets to the final location. From this location the template receives reading and writing requests exclusively from the `PmpModule` template. This is done via the `PmpReadMemory` and `PmpWriteMemory` channels.
The OpenTitan memory layout (which can be seen in section 2.1.3) can be compared to the layout in Fig. 5.3.4a. A precise layout of the fields of the ROM_ext manifests can be seen in Fig. 5.3.4b.

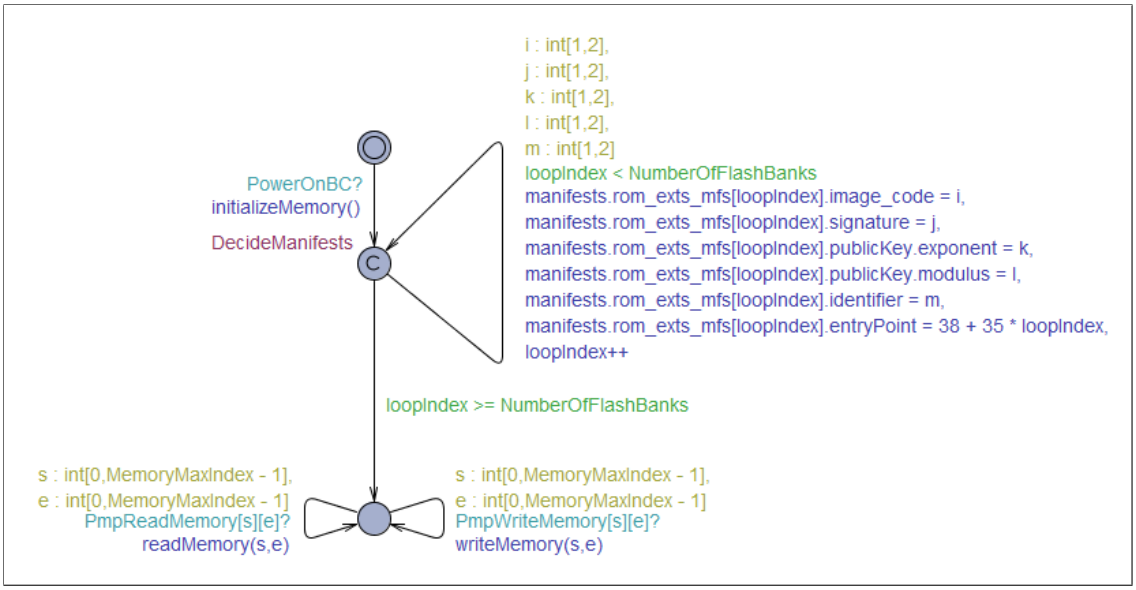


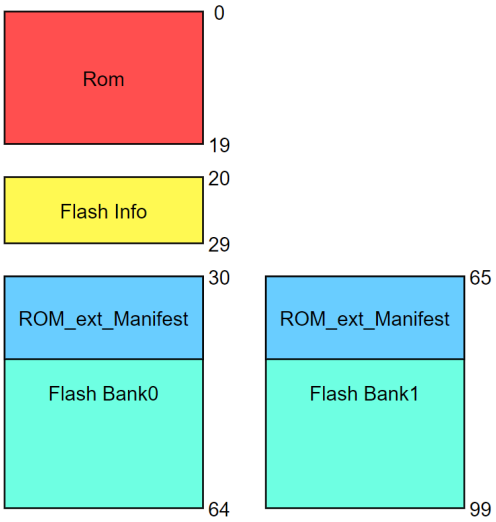
Figure 5.3.3: The `Flash` template in UPPAAL.

Variables & Functions	**Descriptions**
`rom_exts_manifests_t manifests`	This variable holds the manifest generated in this template.
`int[0, NumberOfKeysInWhitelist] pubKeyWhitelist[NumberOfKeysInWhitelist]`	This array holds the valid public key IDs.
`void InitializeMemory()`	This function is called in the first transition. It loads the `pubKeyWhitelist` and it loads a new manifest into manifests.
`void readMemory(int startIndex, int endIndex)`	This function copies hard coded values into the global variables depending on the parameters.
`void writeMemory(int startIndex, int endIndex)`	This function is supposed to write data passed from other templates to its internal memory, but for now it is just for show.

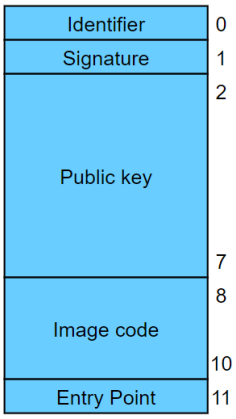
Table 5.3.3: A list of the functions and variables local to the Flash template.

The `Flash` template works as a secondary memory for the model. It contains and exposes data that is used by the other templates.

The first transition in this template is responsible for generating the manifest that is checked in the `ROMStage` template. The transition contains five select clauses that decide the manifest's image code, signature, public



(a) The layout of memory in the `Flash` template. The numbers on the right of sections denote indices.



(b) The layout of a ROM_ext manifest in UPPAAL. The numbers to the right of sections denote index offsets.

Figure 5.3.4: Layout of flash memory in the UPPAAL model.

key, and identifier. We chose to model it like this to reflect that the manifest values can have many different combinations of values (as shown in section 5.1), and only one combination of these values should be able to pass all the checks. From the `Idle` location, the `Flash` template can accept read and write requests made from the `PmpModule` template and react to them. The parameters `s` and `e` correspond to the start and end indices, respectively, of the memory range to be read or written.

5.3.4 The PmpModule Template

The `PmpModule` template represents the PMP hardware module that exists on RISC-V platforms. It is responsible for checking whether a memory region can be read, written, or executed. This means that every time something in the model needs to access flash memory, it has to go through the PMP module. The UPPAAL template can be seen in Fig. 5.3.5.

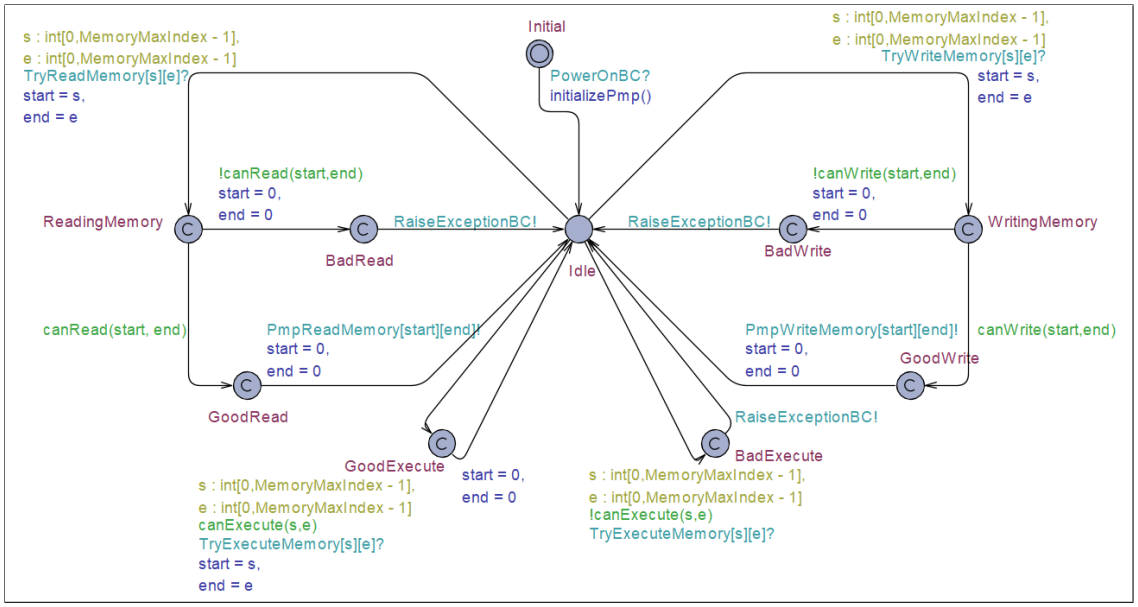


Figure 5.3.5: The `PmpModule` template in UPPAAL.

Variables & Functions	**Descriptions**
`int start`	This variable temporarily holds the beginning index for the read / write / execute request.
`int end`	This variable temporarily holds the ending index for the read / write / execute request.
`void initializePmp()`	This function is used on the first transition. It sets the last PMP region to allow reading the entire flash memory.
`bool canRead(int startIndex, int endIndex)`	This function returns whether the memory in the specified range is allowed to be read according to the PMP regions.
`bool canWrite(int startIndex, int endIndex)`	This function returns whether the memory in the specified range is allowed to be written according to the PMP regions.
`bool canExecute(int startIndex, int endIndex)`	This function returns whether the memory in the specified range is allowed to be executed according to the PMP regions.

Table 5.3.4: A list of the functions and variables local to the `PmpModule` template.

The PmpModule template initializes when a synchronization on the PowerOnBC channel is made. This happens upon system boot. The *initializePmp()* function sets the 16th PMP region (the last PMP region as described in section 2.6) to cover all of memory with the flags set so that the memory is readable and locked. The code for the *initializePmp* function can be seen in listing 5.1. This is done to adhere to the description of the OpenTitan boot process [14]. Once this is done, the template goes to the Idle location, where it can handle requests to read, write, and execute parts of memory. The PmpModule template only receives requests from the FlashController. If the PMP regions validate the request, the PmpModule template forwards the request to the Flash template. Otherwise, this template communicates with the ExceptionHandler template. The following sections will describe how the template handles the types of requests.

```
void initializePmp(){
    PmpRegions[15].startAddress = 0;
    PmpRegions[15].endAddress = MemoryMaxIndex;
    PmpRegions[15].write = false;
    PmpRegions[15].read = true;
    PmpRegions[15].execute = false;
    PmpRegions[15].locked = true;
}
```

Listing 5.1: The code for initializing the PMP module.

Reading Memory

The left part of the template seen in Fig. 5.3.5 is the part that deals with read requests. When the FlashController forwards a reading request by synchronizing over the TryReadMemory[s][e] channel, then the template goes to the ReadingMemory location. The s and e indices in the channel indicate the **s**tart and the **e**nd indices of the memory range which the read request concerns. s and e are locally saved during the transition to the ReadingMemory location. This is another way of passing values over channels. The ReadingMemory location is committed to make an atomic operation over multiple locations. Once in the ReadingMemory location, there are two transitions. Only one of these will be able to fire at a time since the guard on the transitions is the evaluation of the *canRead* function and its negation, respectively. The code for this function can be seen in listing 5.2. If the memory can be read, then the template takes the transition that synchronizes over the PmpReadMemory[start][end] to the GoodRead location. If the memory cannot be read, then this template takes the transition that synchronizes over the RaiseException channel to the BadRead location.

```
bool canRead(int startIndex, int endIndex){
    int i;
    for(i = 0; i < 16;i++)
        if (PmpRegions[i].startAddress <= startIndex &&
    ↪ PmpRegions[i].endAddress >= endIndex)
            if (PmpRegions[i].read)
                return true;
    return false;
}
```

Listing 5.2: The code for the *canRead* function.

Writing Memory

The right part of the template seen in Fig. 5.3.5 deals with write requests. This part works almost entirely similar to the one for reading memory. The difference is that it uses the relevant channels and functions for writing instead of reading. Furthermore, the guards on the transitions from the `WritingMemory` location to `GoodWrite` is a function that checks whether the memory range from `start` to `end` can be written. The only difference between the functions *`canRead`*, *`canWrite`* and *`canExecute`* is the field that is checked in line 5 in listing 5.2.

Executing Memory

The middle-bottom part of the template seen in Fig. 5.3.5 handles execute requests. While this part looks different from the ones handling read and write requests, it works very similarly to them. Here the guard over the transitions is the function *`canExecute`* and its negation, respectively. This function checks that the memory range from `s` to `e` can be executed. The reason that this part of the template looks different from the other two is that this template does not need to synchronize with the `Flash` template.

5.3.5 The OTPController Template

The `OTPController` template is depicted in Fig. 5.3.6. As described in section 2.3.8 the template is responsible for managing all the requests to the `OTP` template. However, in this template, we have only implemented the features of the OTP Controller that pertain to deriving scrambling keys for SRAM and Flash as well as a readable life cycle partition (which is read by the Life Cycle Controller as described in section 5.5.2). This is because the other features are not necessary for the verification that we need to perform. From the initial location the edge with the channel `OTPRead[RootKeyIndex]` is fired. This synchronization is necessary to give the `KeyManager` template the rootKey. Once the `GetRootKey` synchronization is done, the `OTPController` template is ready to handle requests for flash scrambling keys and SRAM scrambling keys. Each key is derived from the respective key seed in OTP. When requesting an SRAM scrambling key, the `OTPController` template receives a source of entropy from the `EDN` template. The variables and functions of the `OTPController` is described in table 5.3.5.

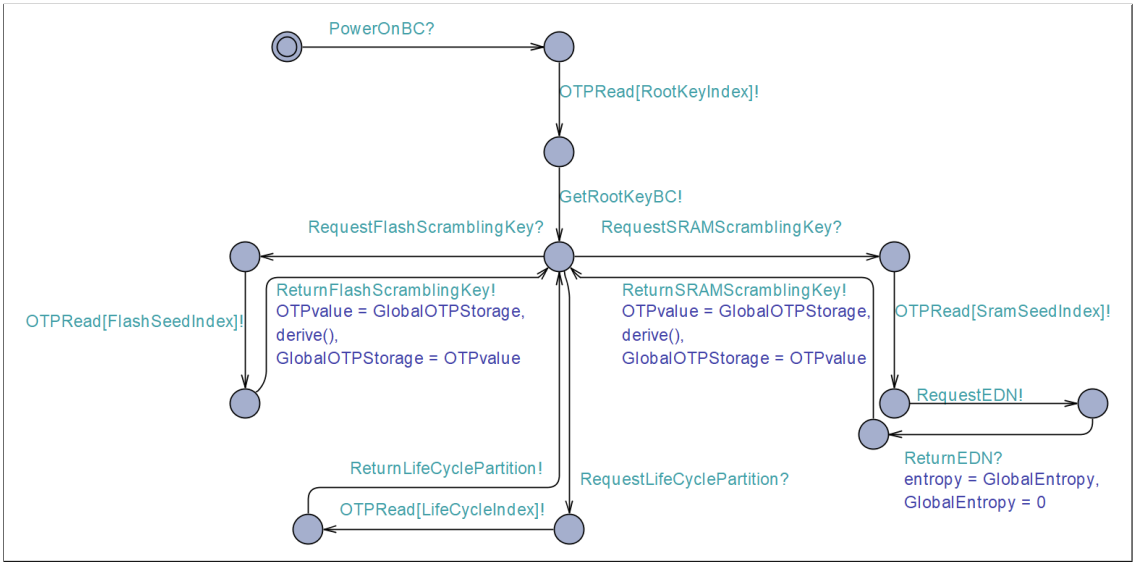


Figure 5.3.6: The `OTPController` template in UPPAAL.

Variables & Functions	Descriptions
`int OTPvalue`	Variable is used to store a seed that can be incremented to a key.
`int entropy`	Variable used to hold entropy used when generating SRAM scrambling key.
`void derive()`	Function used to derive a key from a seed. This is done by incrementing the OTP value representing a seed by 1.

Table 5.3.5: Functions and variables in the `OTPController` template.

5.3.6 The OTP Template

The `OTP` template can be seen in Fig. 5.3.7. It is the memory component that is controlled by the OTP Controller (cf. section 2.3.8). Since OTP is One Time Programmable memory, the template does not have any write functionality and all the local variables are declared as being const as depicted in table 5.3.6. All the interactions to the `OTP` template go through the `OTPController` template. It passes the read values via global variables back to `OTPController`.

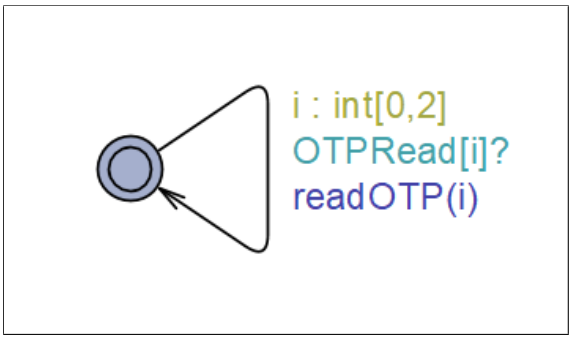


Figure 5.3.7: The template for `OTP` template in UPPAAL.

Variables & Functions	Descriptions
`const int rootKey`	Variable used to symbolize the rootKey stored in OTP.
`const int sramScramblingKeySeed`	Variable used as seed for generating SRAM scramble key.
`const int flashScramblingKeySeed`	Variable used as seed for generating flash Scrambling key.
`void readOTP(int index)`	Function with index parameter to read a specific address in OTP. The variables that can be read are the ones mentioned in this table. The implementation can be seen in listing 5.3.

Table 5.3.6: Functions and variables in the `OTP` template.

```
void readOTP(int index){
    if (index == RootKeyIndex)
        GlobalOTPStorage = rootKey;
    if (index == SramSeedIndex)
        GlobalOTPStorage = sramScramblingKeySeed;
    if (index == FlashSeedIndex)
        GlobalOTPStorage = flashScramblingKeySeed;
    if (index == LifeCycleIndex)
        GlobalOTPStorage = lifeCycleState;
}
```

Listing 5.3: The implementation of the *readOTP* function.

5.3.7 The SRAMController Template

The SRAMController as seen in Fig. 5.3.8 is responsible for managing all the read/write requests forwarded to it by MemoryHandling and pass them to the SRAM template via channels. All read/write requests are initiated from the MemoryHandling template. The SRAMController template is also responsible for managing the scrambling of the SRAM template. From the Initial location the SRAMController template can request a key to use for scrambling. The scrambling key is gathered via the OTPController. This key is used to scramble data before storing it in the SRAM template as well as unscrambling data after a read from the SRAM template. In the Idle location, this template can receive either a signal to write to the SRAM template or to read from it. A signal to write to the SRAM template will enable a choice of one of eight transitions, each corresponding to one of the possible values which can be stored in the SRAM template, e.g., the boot policy. If a read request is issued while SRAMController is in the Idle location, the Reading location is reached. From here, the read will be performed as a scrambled read or unscrambled depending on whether the data being read from the SRAM template is scrambled. This choice is controlled by a guard in the SRAM template. Once in the HitOrMiss location, either a transition will be fired to symbolize that the content was not to be found in the SRAM template or that the content was there by the channels NotInSRAM and SignalSRAMReadReturn respectively. The guards that control which edge is fired take into account the contents of the global variables. If one of the global variables is not 0, the SignalSRAMReadReturn is fired. If all the variables are zero, the content is not in the SRAM template. The functions and variables used in the SRAMController are described in table 5.3.7.

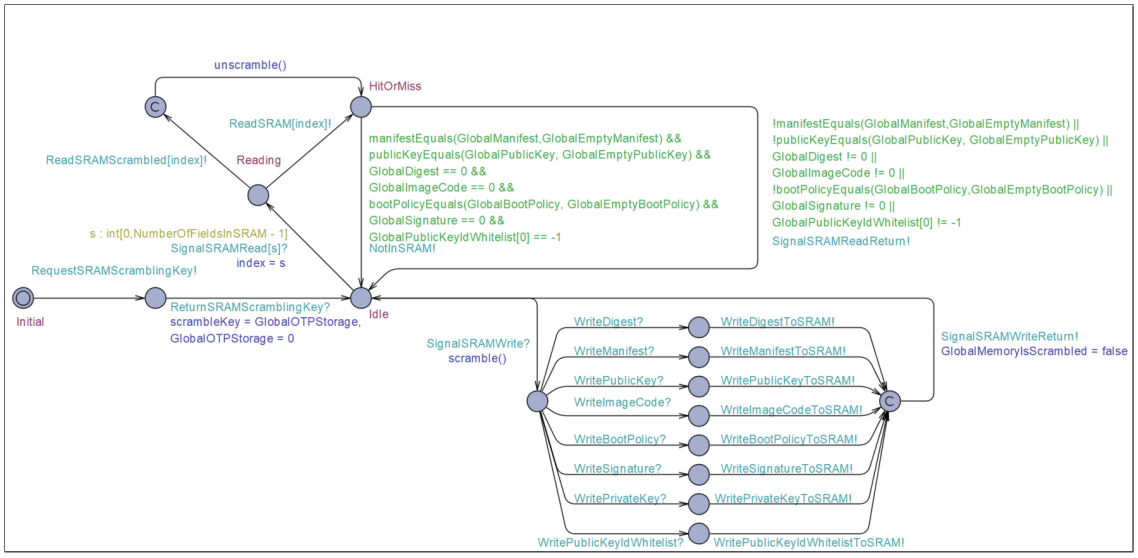


Figure 5.3.8: The template for the `SRAMController` template in UPPAAL

Variables & Functions	Descriptions
`int scrambleKey`	Used to hold the key acquired from OTP.
`int index`	Used to control what memory in SRAM is read.

`bool manifestEquals(`
`rom_ext_manifest_t m1,`
`rom_ext_manifest_t m2)`
	Used to compare each field of two manifests with respect to equality.

`bool bootPolicyEquals(`
`boot_policy_t bp1,`
`boot_policy_t bp2)`
	Used to compare each field of two boot policies with respect to equality.

`bool publicKeyEquals(`
`pub_key_t key1,`
`pub_key_t key2)`
	Used to compare each field of two public keys with respect to equality.

Table 5.3.7: Functions and variables in the `SRAMController` template.

5.3.8 The SRAM Template

The `SRAM` template is seen in Fig. 5.3.9. The template is responsible for storing the variables described in table 5.3.8. The `SRAM` template can perform a total of eight different write requests, one for each of the variables. Additionally, it can take two different read transitions; either a scrambled read with channel `ReadSRAMScrambled` or a not scrambled read with `ReadSRAM`. Which of these edges is fired depends on the local array `scrambled` which denotes for each of the variables in table 5.3.8 whether it is scrambled. The template only synchronizes with `SRAMController`. Thus the template is used whenever the `SRAMController` forwards a request to it. Initially, the SRAM template is empty, however as the software makes requests of data stored in `Flash` this data is put into the `SRAM` template.

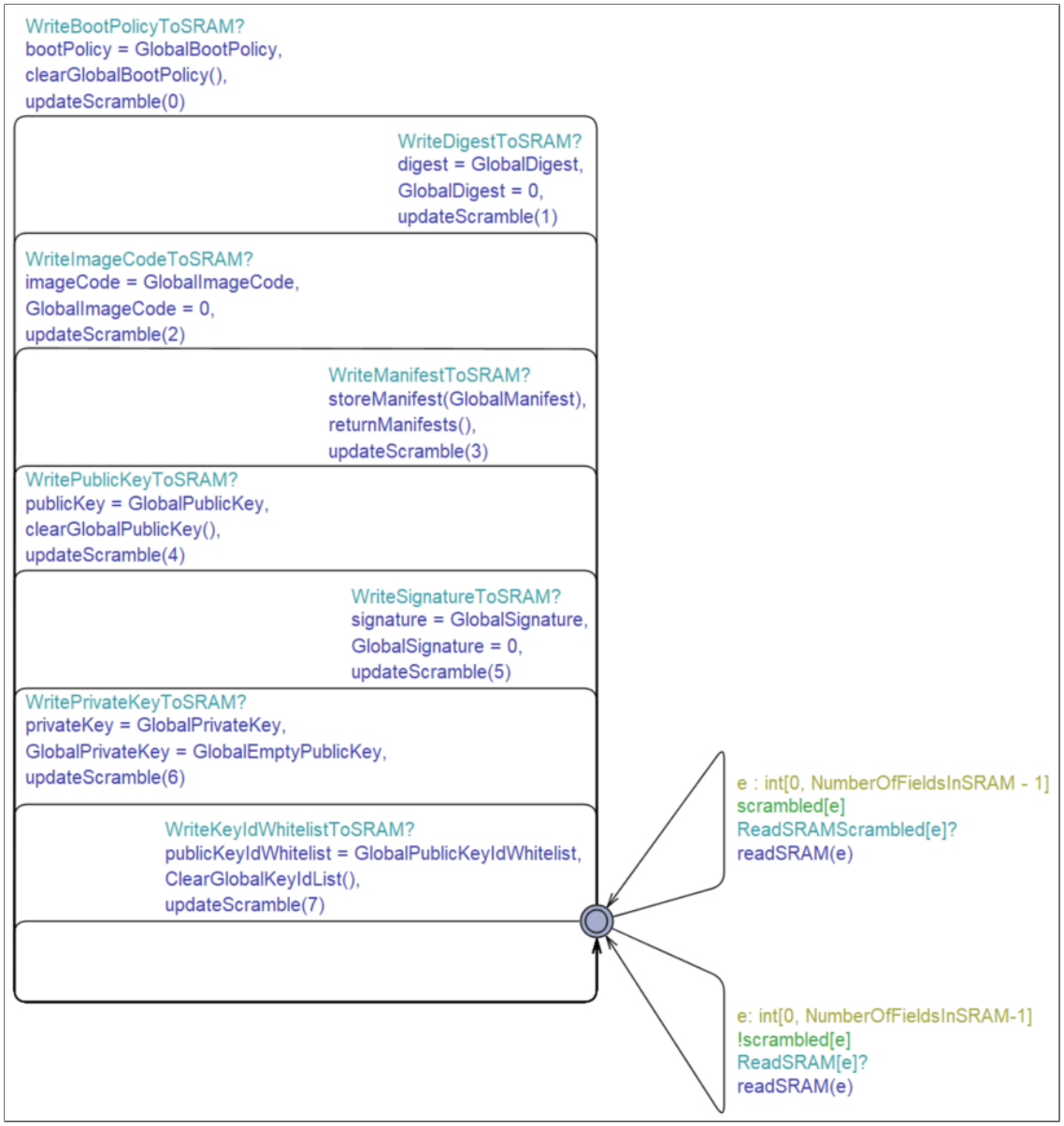


Figure 5.3.9: The template for the SRAM in UPPAAL.

Variables & Functions	Descriptions
`int publicKey`	Used to store the public key.
`int privateKey`	Used to store a private key.
`int[-1,1] publicKeyIdWhitelist[1]`	Stores the whitelist of key IDs.
`int digest`	Used to store the digest.
`int imageCode`	Used to store the image code.
`boot_policy_t bootPolicy`	Used to store the boot policy.
`rom_exts_manifests_t manifests`	Used to store the manifests.
`int signature`	Used to store the signature.
`bool scrambled[8]`	Array used to keep track of which fields are scrambled.
`void readSRAM(int i)`	Function used to read the i'th element in the modeled SRAM.
`void returnManifests()`	Function used to assign the `GlobalRomExtsManifests` the value of the local `manifests` variable.
`void storeManifest(rom_ext_manifest_t manifest)`	Function used to store the manifest in the `SRAM` template.
`void updateScramble(int i)`	Function used to set the i'th value of the scrambled array to true.
`bool indexIsEmpty(int i)`	Function used to tell whether the i'th value of the array is empty.

Table 5.3.8: Functions and variables in the `SRAM` template.

The way `SRAM` is modeled, it is only able to hold data in the specified local fields. This choice is made to give a simple implementation of SRAM. It abstracts the usage of addresses which is otherwise ubiquitous in computer memory. However, for our model, we do not necessarily care about where in SRAM things are placed. We just care about what is in `SRAM`. Therefore, abstracting away from addresses and having variables that represent what is in `SRAM` and not an array to say where they are should be sufficient. Furthermore, when passing variables between templates in UPPAAL, we need to know exactly which data type is sent. Therefore, it is challenging to create a generic array of addresses to store any data. A disadvantage of this solution is that edges have to be added for each of the fields in `SRAM`. This effect carries over to other templates as well. Thus both `MemoryHandling` and `SRAMController` also have several edges, each corresponding only to one variable in `SRAM`.

5.4 Cryptographic Templates

This section is used to describe the templates of the model used for cryptographic operations or are strongly tied to cryptographic components of the system.

5.4.1 The HASH Template

The `HASH` template, seen in section 5.4.1, models the hashing function that is used during signature verification in the boot code described in section 3.1. We have modeled it to be a simple abstraction of a SHA-256 algorithm which takes a message from `SRAM` and hashes it. The digest is then put into `SRAM` so that it can

be used by the OTBN template. The HASH template synchronizes with the VerifyRomExtSignature and MemoryHandling templates. The local variables and functions used by the HASH template can be seen in table 5.4.1.

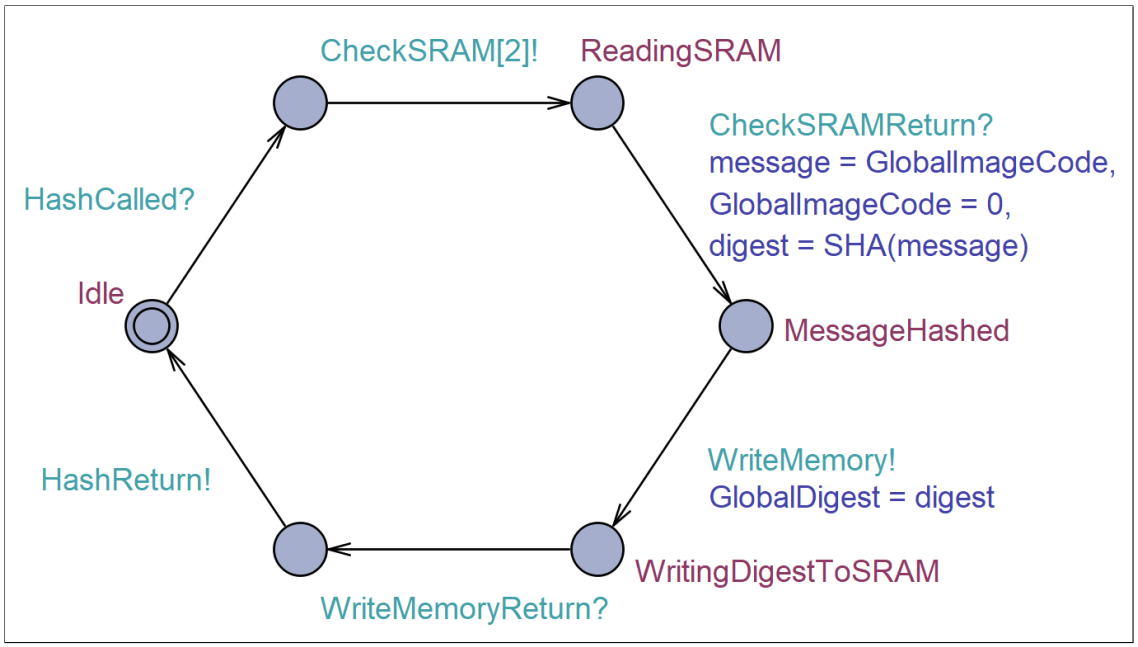


Figure 5.4.1: The HASH template in UPPAAL.

Variables & Functions	Description
`int message`	Used to keep a local copy of the message that needs to be hashed.
`int digest`	Used to keep a local copy of the digest.
`int SHA(int msg)`	Used to hash a given message.

Table 5.4.1: Functions and variables in the HASH template.

The HASH template is in the Idle location until synchronization over the HashCalled channel is initiated by the VerifyRomExtSignature template. Once this happens, the HASH template tries to read a message from SRAM that needs to be hashed and moves to the ReadingSRAM location until SRAM returns the message. Afterwards, HASH hashes the message by calling the local function *SHA* with the message as parameter. Once this is done, HASH initiates a write to put the digest into SRAM and moves to the WritingDigestToSRAM location. Once this write operation is completed, as signaled by synchronization over the WriteMemoryReturn channel, the HASH template initiates synchronization over the HashReturn channel to signal that it has completed its task.

5.4.2 The OTBN Template

The OTBN template can be seen in Fig. 5.4.2. It is meant to model the RSA-3072 signature verification portion of the OTBN module found in OpenTitan (cf. section 2.3.5). This means that the OTBN that is responsible for checking whether a given ROM_ext manifest's signature is valid. The OTBN synchronizes with the VerifyRomExtSignature, MemoryHandling, and ROMStage templates. The local variables and functions used by the OTBN can be seen in table 5.4.2.

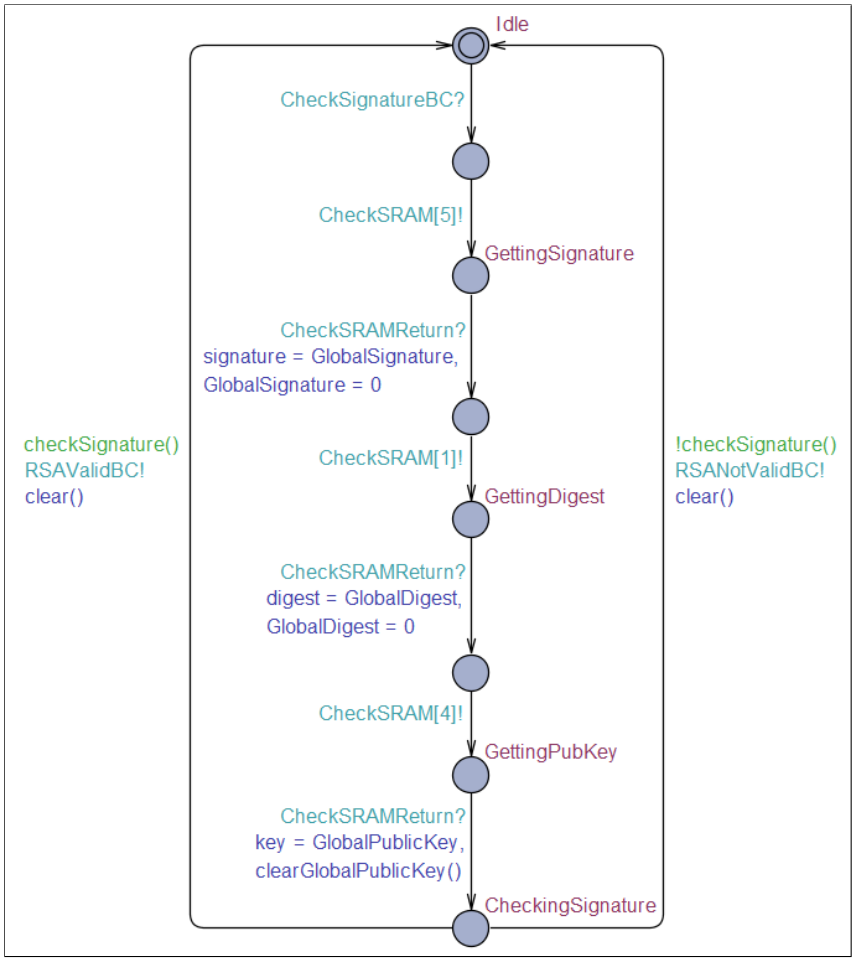


Figure 5.4.2: The OTBN template in UPPAAL.

Variables & Functions	Description
int signature	Stores a local copy of the signature of the current manifest.
int digest	Stores a local copy of the image code digest for the current manifest.
pub_key_t Key	Stores a local copy of the public key of the current manifest.
bool *checkSignature()*	Used to perform an abstracted version of the RSA-3072 algorithm on the signature, digest, and public key. Outputs true if the signature, digest, and public key are the expected values (signature = 1, digest = 3, key.modulus = 1, key.exponent = 1).
void *clear()*	Used to clear the local variables.

Table 5.4.2: Functions and variables in the OTBN template.

In the Idle location the OTBN waits for synchronization from the VerifyRomExtSignature over the CheckSignatureBC channel. Once this happens, the OTBN starts to read the data needed for signature verification from SRAM (via the MemoryHandling template). First, the OTBN reads the signature for the current manifest while in the GettingSignature location. After this is done, it reads the digest and the public key by moving to the GettingDigest and GettingPubKey locations respectively. Once all the data has been collected into local variables the OTBN moves to the CheckingSignature location. Here it synchro-

nizes with ROMStage using either the RSAValidBC or the RSANotValidBC channel depending on whether the signature is correct or not. To decide whether the signature is correct we use the *checkSignature()* function which evaluates to a boolean value. This function is used as a guard on the transitions (seen as the checkSignature() on the transition that synchronizes over the RSAValidBC channel and the negated guard on the other transition). Once the validation of the signature passes or fails the OTBN moves back to the Idle location.

5.4.3 The EDN Template

The EDN template, as seen in Fig. 5.4.3, only has a single local variable. Therefore it will not be presented as a table. The local variable is an **int** called entropy used for storing the entropy. The EDN template is responsible for managing requests for entropy. Therefore from the Initial location, the template will synchronize and afterward forward the request for entropy to the CSRNG template. When in the Awaiting location, the entropy will get passed back to the EDN template once the synchronization is made. Finally the template will initiate the synchronization over ReturnEDN to the OTPController template. The template only communicates with the CSRNG and OTPController templates. The template resembles the design of the EDN described in section 2.3.11 in that it communicates with the same components (OTP Controller and CSRNG) and that all requests must go through this template.

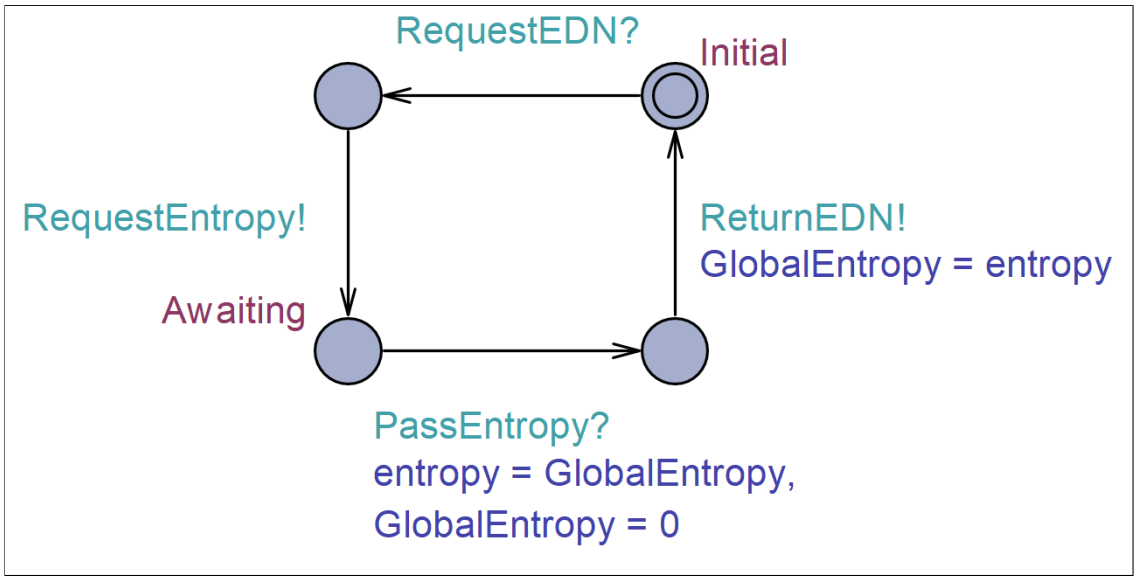


Figure 5.4.3: The EDN template in UPPAAL.

5.4.4 The CSRNG Template

The CSRNG template as depicted in Fig. 5.4.4 is responsible for generating random numbers based on the seed provided to it from the EntropySource template. It does so with the *RNG* function described in table 5.4.3. From the initial location the template awaits a request for entropy from EDN, once received it fires a GetEntropy request to the EntropySource. The final edge also calls the function that computes a random number from a seed. The template only communicates with EDN and EntropySource.

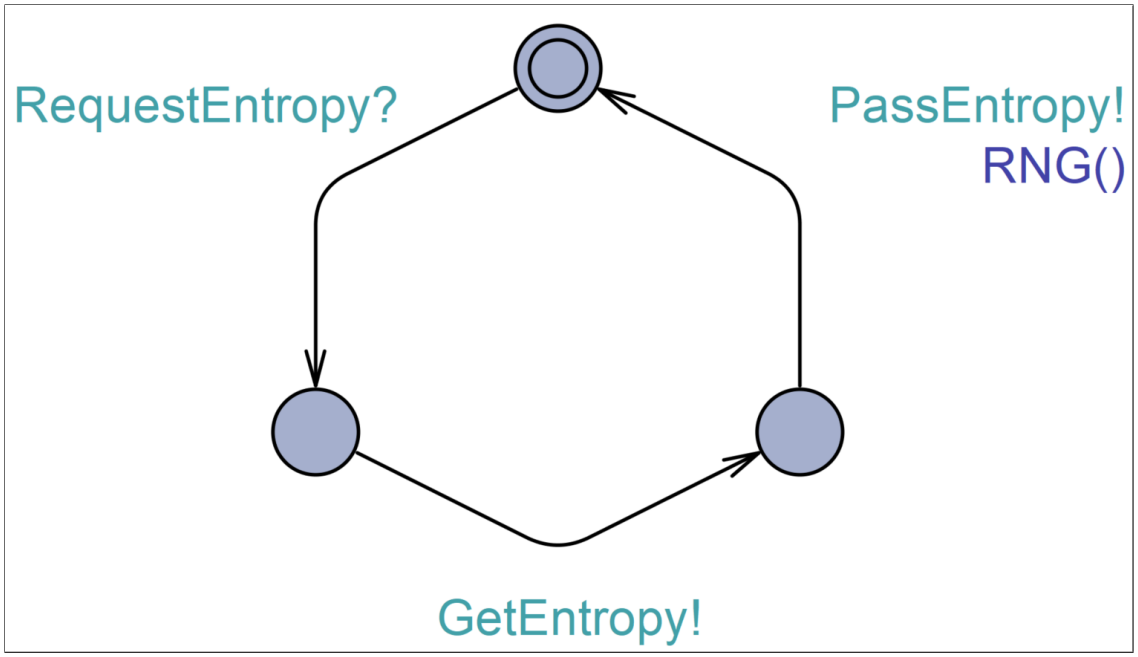


Figure 5.4.4: The CSRNG template in UPPAAL.

Variables & Functions	Descriptions
`void RNG()`	Used to increment the `GlobalEntropy` variable by 1. Thus symbolizing the generation of a random number from the previous value, a seed.

Table 5.4.3: The function for the CSRNG template.

5.4.5 The Entropy Source Template

The entropy source template (seen in Fig. 5.4.5) will upon synchronization over `GetEntropy` assign `GlobalEntropy = 1`. The assignment correspond to a seed, which the CSRNG (as described in section 5.4.4) will increment to emulate it becoming a random number. The template does not have any local variables or functions, therefore no table is presented.

Figure 5.4.5: The EntropySource template in UPPAAL.

5.5 Miscellaneous Templates

The templates described in this section are templates that do not clearly fall under one of the other categories mentioned in this chapter.

5.5.1 System Reset

The `SystemReset` template is used to initiate the booting process. Therefore the first edge initiates a synchronization which is used to initiate the PMP module and the key manager via the `PowerOnBC` channel. The last task of system reset is to synchronize with `ROMStage` via `StartRomStage`.

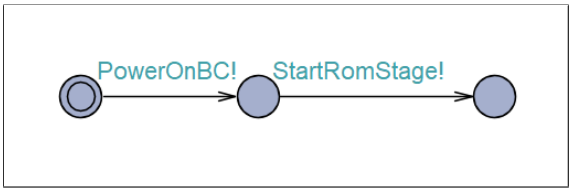


Figure 5.5.1: The template for the `SystemReset`.

5.5.2 The LifeCycleController Template

The `LifeCycleController` template as seen in Fig. 5.5.2 is responsible for providing measurement of the system's state to the other components. This template is modeled to imitate the Life Cycle Controller found in OpenTitan as described in section 2.3.4. The model is first interacted with by synchronizing over `PowerOnBC`, which brings the template to its `RAW` location. From this location the template fires an edge to synchronize over `RequestLifeCyclePartition` with the `OTPController` template, which is the only template that the `LifeCycleController` communicates with. When requesting the life cycle partition from the `OTPController` the template will receive a value representing the partition. The `LifeCycleController` will then derive a specific system state from the value with the *`deriveLifeCycleState`* function described in table 5.5.1. In the model, this function simply checks if the `LifeCycleController` has received data from the `OTPController` when it tries to derive the LifeCycleState. If data has been received, then it derives the correct state (by assigning the `currentState` to be `correctState`), if not then nothing happens and the state remains empty. From the `Idle` location the template can accept and return incoming requests for life cycle states. These requests will come from the Key Manager, despite it not currently being a part of our model. The synchronizations are kept in the template so that few to no changes are needed in the `LifeCycleController` template if a Key Manager is implemented during future work.

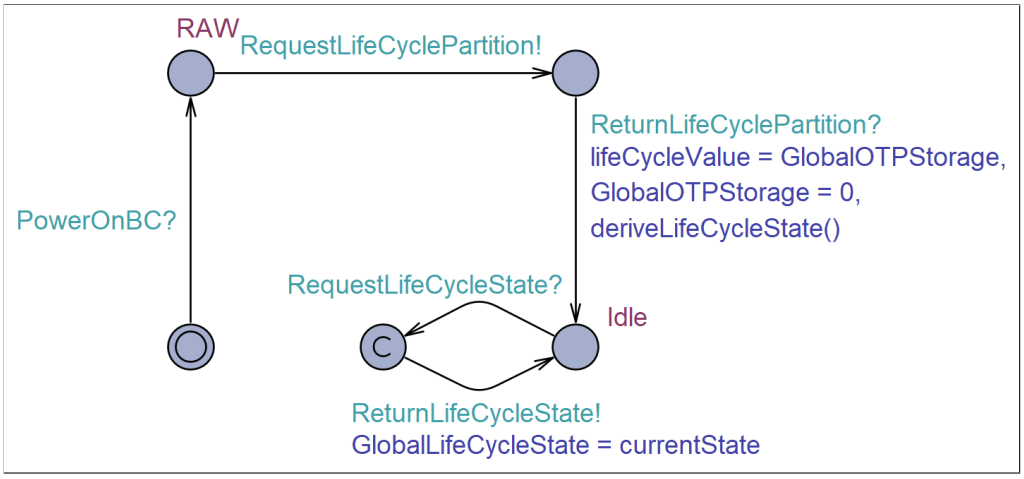


Figure 5.5.2: The template for the `LifeCycleController`.

Variables & Functions	Descriptions
`life_cycle_state_t currentState`	Used as a representation for the current life cycle state.
`life_cycle_state_t correctState`	A hard coded representation of the life cycle state. The variable is only used in the *deriveLifeCycleState* function. Multiple values constitute the state.
`int lifeCycleValue`	Value used to derive the correctState.
`void deriveLifeCycleState()`	Will assign the currentState variable the value stored in the correctState variable, based on the value of lifeCycleValue.

Table 5.5.1: The functions and variables for the `LifeCycleController` template.

5.5.3 The ExceptionHandler Template

The `ExceptionHandler` template, seen in Fig. 5.5.3, is not something that we have found directly in the OpenTitan documentation. However, some components can fail in some way. E.g., if a piece of flash memory is read without the PMP module allowing for such an action, then the PMP module must express this violation somehow. Because of this, we have modeled a simple exception handler that stops the execution of the model if an exception is raised. The `ExceptionHandler` template communicates with the `PmpModule` and `SRAMController` templates. These two templates are the only ones we have modeled to potentially throw exceptions. This is an arbitrary choice and could be expanded to other templates in future work if needed. The `ExceptionHandler` template does not have any local variables or functions.

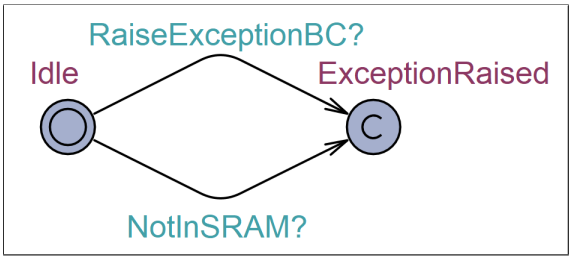


Figure 5.5.3: The template for the `ExceptionHandler` in UPPAAL.

As can be seen from Fig. 5.5.3, the `ExceptionHandler` template is a simple template. It moves from the `Idle` location to the `ExceptionRaised` location if any template synchronizes via the `RaiseExceptionBC` or `NotInSRAM` channel. Since the `ExceptionRaised` location is committed, any of these synchronizations effectively stop any other transition in the model from happening. This models that the system halts or breaks in some fashion due to an exception.

5.6 Deviations and Assumptions

During the building of the model based on the OpenTitan project, we have made certain assumptions about how OpenTitan works based on vague or missing documentation. The assumptions are made to ease the modeling task.

Another assumption about the model is regarding the manifests. We have assumed that each manifest has the same size. The components of a manifest can also be found at the same offset from the manifest's start index. Furthermore, we have also assumed that each of the manifest fields has the same size across manifests. E.g., the image code will always have a size of four entries in our modeled memory (cf. section 5.3.3).

Another important assumption we have made is regarding the scrambling of SRAM and memory. The documentation clearly states that the flash is scrambled [29]. We assumed that the scrambling that is done on the flash is reversible, and therefore that it is more like encrypting the memory rather than wiping the memory.

In the OpenTitan system, a system reset will clear all the contents of SRAM, except for some retention SRAM which will not be cleared[14]. This is a feature not incorporated in our model. The `SRAM` template in our model is always empty from the initial model state. Therefore we do not incorporate this clearing in our model and thus deviate from the actual booting process. It should not be challenging to implement clearing of memory in future work if we feel that it is necessary for the model.

The documentation states that the ROM_ext manifest is used to lock down certain peripherals of the system. Although we have modeled some of the peripherals, we have not implemented any lockdown functionality of these components. This is because it was not clear from the documentation which peripherals would be locked by the manifest. Also, how a peripheral is locked is not stated.

Chapter 6

Verification and Results

This chapter presents the verification that we have done on the model described in chapter 5. We present the individual goals as they are described in section 3.2 and show the queries that we have made to verify those goals. We then discuss whether or not we see the goals as fulfilled in the model. If a query fails, we use the built-in feature of UPPAAL to get a diagnostic trace whenever possible that shows why a property does not hold by providing a counterexample consisting of a trace where the property is not upheld[1]. These traces are too large to feasibly show in this report, but we document how we use them to figure out exactly why queries fail as this is an essential part of verification with UPPAAL.
We have chosen not to include the amount of time each query takes in the tables presented throughout this chapter. This is because most of our queries are `A[]` queries, which take roughly the same amount of time to verify given that they pass (due to the same state space having to be calculated). If an `A[]` property fails, then it does so in at most the same time as if it were to pass, and we are primarily interested in documenting how long verification potentially takes. Since we have designed most of our `A[]` queries to pass, we think that the inclusion of verification time in the tables would be mostly redundant.
Running the queries on an AMD Ryzen 3700x CPU in UPPAAL means that each `A[]` query that passes takes around 12 minutes and 37 seconds to verify. We have included a table in appendix E which includes the time it takes to verify queries that are not passing `A[]` queries. In this table, it can be seen that our overall longest verification time for any query is roughly 35 minutes. All of the verification done for the model is done with one flash bank (cf. chapter 5). Running the command line version of UPPAAL via the verifyta.exe file, we have performed an `A[] true` query to compute the total state space of the model. We use `A[] ↪ true` because the query will force UPPAAL to calculate the entire state space. Using the -u option in the command line, we can see that the total number of possible states in the model is 26,474.
The verification results in this chapter depend upon the fact that the boot code is compiled with a correct C compiler that does not introduce any errors or malignant code.

In the tables throughout this chapter, we denote a query that passes in UPPAAL with a ✓. Conversely, we denote a query that does not pass with a ×. A table containing all queries can be seen in appendix D.

6.1 Model Validation

Before we start verifying the security goal described in section 3.2 we want to ensure that the model functions as we expect it to. For this, we can use the verification tool in UPPAAL. We believe that this form of validation is crucial to ensure that the model is "correct" in terms of our modeling practices. Like with any piece of code, we might have introduced errors in the model that is not due to the design of OpenTitan. In this section, we describe the queries that are used to validate the model. A list of all queries used for model validation can be seen in table 6.1.1. These queries are not necessarily a complete list of queries that we could use to validate the model. However, we believe that it is sufficient when taken together with

[1] Unless otherwise stated we use the option to get "some trace" when finding a diagnostic trace.

the rest of the queries in this chapter. Furthermore, this section highlights how model validation is done in general so that the method can be applied elsewhere.

ID	Name	Query	Passes?
D1	Possibly Never Fail	`E[] not( ROMStage.BootFailed || ROMStage.RomExtTerminated)`	✓
D2	Guaranteed Never Fail	`A[] not( ROMStage.BootFailed || ROMStage.RomExtTerminated)`	×
D3	Success Possible	`E<> FinalJumpToRomExt.ROMExtRunning`	✓
D4	Success Guaranteed	`A<> FinalJumpToRomExt.ROMExtRunning`	×
D5	Exception Never Raised	`A[] !ExceptionHandler.ExceptionRaised`	✓
D6	Memory Unscramble Urgent	`A[] FinalJumpToRomExt.ROMExtRunning imply GlobalMemoryIsScrambled == false`	✓
D7	Flash Immediately Unscrambled	`A[] FlashController.ScrambleCheck imply !GlobalMemoryIsScrambled`	✓
D8	SRAM Immediately Unscrambled	`A[] SRAMController.ScrambleCheck imply !GlobalMemoryIsScrambled`	✓
D9	Exactly One Choice	`A[]MemoryHandling.ChooseType imply (((GlobalBootPolicy != GlobalEmptyBootPolicy) + (GlobalImageCode != 0) + (GlobalDigest != 0) + (GlobalPublicKey != GlobalEmptyPublicKey) + (GlobalManifest != GlobalEmptyManifest) + (GlobalSignature != 0) + (GlobalPrivateKey != GlobalEmptyPublicKey) + (GlobalPublicKeyIdWhitelist[0] != -1)) == 1)`	✓
D10	Writing Manifest is Disallowed	`A[] !PmpModule.canWrite (30 + 35 * ROMStage.loopIndex, 41 + 35 * ROMStage.loopIndex)`	✓

Table 6.1.1: Queries for validating that the model functions as expected.

6.1.1 D1: Possibly Never Fail

```
E[] not( ROMStage.BootFailed || ROMStage.RomExtTerminated)
```

This query states that there exists a path such that it always holds that the `ROMStage` template never enters one of the two failure locations: `BootFailed` or `RomExtTerminated`. When run on the current model, it passes, which we expect of a correct model.

6.1.2 D2: Guaranteed Never Fail

```
A[] not( ROMStage.BootFailed || ROMStage.RomExtTerminated)
```

This query states the same as D1, except that for this query to be true, it must hold for all paths due to the use of `A[]`. This query does not pass in the current model, which we expect since not all ROM_ext manifests should be valid. When we ask for a diagnostic trace in UPPAAL, we get a counterexample in the form of a trace where the `ROMStage` template ends in the `BootFailed` location due to an invalid manifest.

6.1.3 D3: Success Possible

```
E<> FinalJumpToRomExt.ROMExtRunning
```

While D1 and D2 together state that some but not all traces through the model should lead to a failure state, they do not express that any traces should lead to a success state. For this, we use D3 and D4. D3 is a reachability property that states that there exists a path such that the state of the system eventually contains the `ROMExtRunning` location of the `FinalJumpToRomExt` template. This is a state where `ROMStage` has successfully transferred execution to the `RomExt` template. This query passes for the current model, which we expect because there should be a valid manifest leading to this outcome.

6.1.4 D4: Success Guaranteed

```
A<> FinalJumpToRomExt.ROMExtRunning
```

This is a simple liveness property. It says that independently of which transitions are taken, the `FinalJumpToRomExt` template will eventually be in the `ROMExtRunning` location, which is our version of our success state. This query does not pass, which means that the model does not always reach this state. This is what we expect, as not all manifests are valid. The diagnostic trace that we can get from UPPAAL confirms that there is indeed a counterexample trace, which leads to the `BootFailed` location in the `ROMStage` template.

6.1.5 D5: Exception Never Raised

```
A[] !ExceptionHandler.ExceptionRaised
```

This is a safety property. It says that the `ExceptionHandler` is never able to reach the `ExceptionRaised` location. This will happen if the `PmpModule` template receives a bad request for reading, writing, or executing, or if the `SRAMController` template receives a read request for something that is not in SRAM. The query passes, which means that the current model never raises an exception. This is what we expect of the model.

6.1.6 D6: Memory Unscramble Urgent

```
A[] FinalJumpToRomExt.ROMExtRunning imply GlobalMemoryIsScrambled == false
```

This is a safety property. It says that whenever the `FinalJumpToRomExt` template is in the `ROMExtRunning` location, then the global variable `GlobalMemoryIsScrambled` must be false. This query passes for the current model, which means that unscrambling memory that has been read cannot be delayed until the end of the `ROMStage` template. This is the result that we expect to see.

6.1.7 D7: Flash Immediately Unscrambled

```
A[] FlashController.ScrambleCheck imply !GlobalMemoryIsScrambled
```

This is a safety property. It says that whenever the `FlashController` template is in the `ScrambleCheck` location, then the global variable `GlobalMemoryIsScrambled` must be false. The query passes as expected, which further validates that the flash memory that is read is unscrambled immediately.

6.1.8 D8: SRAM Immediately Unscrambled

```
A[] SRAMController.ScrambleCheck imply !GlobalMemoryIsScrambled
```

This is a safety property very similar to D7. This query checks whether the SRAM memory that is read is immediately unscrambled. This query passes, which means that the model behaves as expected.

6.1.9 D9: Exactly One Choice

```
A[] MemoryHandling.ChooseType imply
(((GlobalBootPolicy != GlobalEmptyBootPolicy) +
(GlobalImageCode != 0) +
(GlobalDigest != 0) +
(GlobalPublicKey != GlobalEmptyPublicKey) +
(GlobalManifest != GlobalEmptyManifest) +
(GlobalSignature != 0) +
(GlobalPrivateKey != GlobalEmptyPublicKey) +
(GlobalPublicKeyIdWhitelist[0] != -1)) == 1)
```

This safety property says that whenever the `MemoryHandling` template is in the `ChooseType` location, then exactly one of the eight transitions must be active. We have made this query to verify that it is only possible to choose a single transition when reading from the `SRAM`. To verify this, we sum the guards and check whether it is equal to 1. The guards are boolean, but their values are represented as 0 or 1. If their sum is equal to one, then exactly one of the guards must be true. Running this query on the model, we can see that it is indeed the case that only one of the transitions is active at once. This is because it passes. This is the behavior that we want so that there is no nondeterminism in this part of the model.

6.1.10 D10: Writing Manifest Is Disallowed

```
A[] !PmpModule.canWrite(30 + 35 * ROMStage.loopIndex, 41 + 35 * ROMStage.loopIndex)
```

This is a safety property. It says that according to the `PmpModule` template, it is never possible to write anything to the specific range of memory where the `currentManifest` is placed. This query passes, which verifies that it is impossible to change the manifests by writing to flash. This is the intended behavior.

6.2 Verifying P1

The P1 security goal ensures that only code that has been validated can be executed. This means that it should not be possible to execute a ROM_ext manifest that has not passed the three validation checks made in the ROM stage. The P1 security policy contains the two security goals G1, G2, G3, and G4. Because of the limitation of scope mentioned in section 3.2, we are only interested in verifying G1 and G3.

6.2.1 Verifying G1

G1 states that "*Before the ROM_ext image code is executed, the signature of the ROM_ext manifest must be validated by ROM.*"
This goal concerns the three checks on the manifests that are made in the boot code (cf. section 3.1) and the `ROMStage` template: The identifier must be correct, the public key in the manifest must be a trusted key, and the manifest's signature must be correct with respect to the manifest's image code and public key. These checks can be seen in the `ROMStage` template (cf. section 5.2.1). If the three checks pass, the manifest must be valid. In this section, we verify that the checks work as intended and are modeled such that an invalid manifest cannot be executed by the `ROMStage` template.

Queries

To verify G1, we have created the queries that can be seen in table 6.2.1.

ID	Name	Query	Passes?
Q1	Checking Manifest Identifier	`A[] ROMStage.IdentifierChecked imply ROMStage.currentManifest.identifier == 1`	✓
Q2	Checking Valid Manifest Identifier	`A[] (CheckRomExtManifest.manifest.identifier == 1) == CheckRomExtManifest.checkRomExtManifest()`	✓
Q3	Checking Public Key	`A[] ROMStage.CheckingSignature imply (ROMStage.currentManifest.identifier == 1 && ROMStage.currentManifest.publicKey.modulus == 1 && ROMStage.currentManifest.publicKey.exponent == 1)`	✓
Q4	Checking Signature	`A[] ROMStage.CheckedSignature imply (ROMStage.currentManifest.identifier == 1 && ROMStage.currentManifest.publicKey.modulus == 1 && ROMStage.currentManifest.publicKey.exponent == 1 && ROMStage.currentManifest.signature == 1)`	✓

ID	Name	Query	Passes?
Q5	PMP Execute	`A[] ROMStage.ReadyToRunROMExt imply (PmpRegions[0].execute && PmpRegions[0].startAddress <= ROMStage.currentManifest.entryPoint && PmpRegions[0].endAddress >= ROMStage.currentManifest.entryPoint + 4)`	✓
Q6	Valid Key ID	`A[] (CheckPubKeyValid.currentPubKeyId == 1) == CheckPubKeyValid.checkPublicKey()`	✓
Q7	Valid Key Leads to Valid Key ID	`(CheckPubKeyValid.publicKey.exponent == 1 && CheckPubKeyValid.publicKey.modulus == 1) --> CheckPubKeyValid.currentPubKeyId == 1`	✓
Q8	Valid Signature	`A[] (OTBN.signature == 1 && OTBN.digest == 3 && OTBN.key.modulus == 1 && OTBN.key.exponent == 1) == OTBN.checkSignature()`	✓
Q9	Valid Manifest Leads to Rom Ext Running	`EqualManifestContents(validManifest, ROMStage.currentManifest) --> (FinalJumpToRomExt.ROMExtRunning && EqualManifestContents(validManifest, ROMStage.currentManifest))`	✓
Q10	Invalid Manifest Leads to Failure	`!EqualManifestContents(validManifest, ROMStage.currentManifest) --> (ROMStage.StartOfLoop || ROMStage.BootFailed)`	✓
Q11	Invalid Key Leads To Invalid Key ID	`(CheckPubKeyValid.publicKey.exponent != 1 || CheckPubKeyValid.publicKey.modulus != 1) --> CheckPubKeyValid.currentPubKeyId == 0`	✓

Table 6.2.1: The queries associated with the G1 security goal.

Q1: Checking Manifest Identifier

`A[] ROMStage.IdentifierChecked imply ROMStage.currentManifest.identifier == 1`

This query is a safety property that verifies that the `ROMStage` template can only enter the `IdentifierChecked` location if the identifier of the current manifest is valid (a valid identifier in the model is represented with a 1). This query passes, which is what we expect.

Q2: Checking valid manifest identifier

```
A[] (CheckRomExtManifest.manifest.identifier == 1) ==
CheckRomExtManifest.checkRomExtManifest()
```

This safety property says that the locally defined *checkRomExtManifest* function will return true, if and only if the CheckRomExtManifest template's local manifest has an identifier of 1. This query verifies the *checkRomExtManifest* function. The query passes, which means that the function behaves as expected. Furthermore, the query also verifies that 1 is a correct value for the ROM_ext manifest's identifier. This is verified by the fact that the query uses boolean equivalence to conclude that the result of the *checkRomExtManifest* function is dependent on the value of the manifest's identifier.

Q3: Checking Public Key

```
A[] ROMStage.CheckingSignature imply
(ROMStage.currentManifest.identifier == 1 &&
ROMStage.currentManifest.publicKey.modulus == 1 &&
ROMStage.currentManifest.publicKey.exponent == 1)
```

This query is a safety property that works similarly to the one described before. The difference here is that we add another part to the query regarding checking the public key of the manifest. Here we state that if the ROMStage template is in the CheckingSignature location, then the current manifest must have a valid identifier and a valid public key. Note that a valid public key in the model is represented as having its modulus and exponent equal to 1. For the current model, this query passes, which is the desired result.

Q4: Checking Signature

```
A[] ROMStage.CheckedSignature imply
(ROMStage.currentManifest.identifier == 1 &&
ROMStage.currentManifest.publicKey.modulus == 1 &&
ROMStage.currentManifest.publicKey.exponent == 1 &&
ROMStage.currentManifest.signature == 1)
```

This query is also a safety property that is similar to the two previous ones. This one adds that if the ROMStage template is in the CheckedSignature location, then the signature of the current manifest must be valid as well as the identifier and public key (a valid signature is represented as 1 in the model). This query also passes for the current model as expected.

Q5: PMP Execute

```
A[] ROMStage.ReadyToRunROMExt imply
(PmpRegions[0].execute &&
PmpRegions[0].startAddress <= ROMStage.currentManifest.entryPoint &&
PmpRegions[0].endAddress >= ROMStage.currentManifest.entryPoint + 4)
```

This query is a safety property that states that if the ROMStage template is in the ReadyToRunROMExt location, then the first PMP region must allow for the ROM_ext manifest image code to be executed (cf. section 3.1). This query passes, which means that we never execute something that is not allowed to be executed. This is the desired result.

Q6: Valid Key ID

```
A[] (CheckPubKeyValid.currentPubKeyId == 1) ==
CheckPubKeyValid.checkPublicKey()
```

This safety property states that whenever the `CheckPubKeyValid`'s `currentPubKeyId` is equal to 1 (i.e. a correct public key ID), then the locally declared *`checkPublicKey`* will return true. This query passes, which verifies the *`checkPublicKey`* function, so we know it behaves like we expect it to behave.

Q7: Valid Key Leads to Valid Key ID

```
(CheckPubKeyValid.publicKey.exponent == 1 &&
CheckPubKeyValid.publicKey.modulus == 1) --> 
CheckPubKeyValid.currentPubKeyId == 1
```

This is a liveness property. It says that whenever the public key is correct (i.e., both fields in the struct are equal to 1), the system eventually reaches a state where the ID is equal to 1. This query passes, which verifies the *`calculateKeyId`* function that is responsible for calculating the public key ID.

Q8 : Valid Signature

```
A[] (OTBN.signature == 1 && OTBN.digest == 3 &&
OTBN.key.modulus == 1 && OTBN.key.exponent == 1) ==
OTBN.checkSignature()
```

This is a safety property. It says that the `OTBN` template has the correct signature, key, and digest values, if and only if the locally declared *`checkSignature`* function will evaluate to true. This query passes, which verifies the *`checkSignature`* function in the positive and negative case.

Q9: Valid Manifest Leads to Rom Ext Running

```
EqualManifestContents(validManifest,ROMStage.currentManifest) --> 
(FinalJumpToRomExt.ROMExtRunning &&
EqualManifestContents(validManifest,ROMStage.currentManifest))
```

This query is a liveness query. It says that whenever the current manifest in the `ROMStage` template is equal to a hypothetically correct manifest, then the `FinalJumpToRomExt` template will eventually be in the `ROMExtRunning` location, and the current manifest in the `ROMStage` template will still be equal to the correct manifest. The variable `validManifest` is a global variable of the type **`rom_ext_manifest_t`**, which is considered correct with respect to the three checks made by the `ROMStage` template. The function *`EqualManifestContents`* is a globally defined boolean function that returns whether two given **`rom_ext_manifest_t`**s have the same identifier, public key, and signature. This query passes as expected.

Q10: Invalid Manifest Leads to Failure

```
!EqualManifestContents(validManifest,ROMStage.currentManifest) --> 
(ROMStage.StartOfLoop || ROMStage.BootFailed)
```

This liveness property says that if the current manifest in the `ROMStage` template is different from `validManifest`, which is the only values of manifests that can pass all the checks in the `ROMStage` template, then the `ROMStage` template will eventually go back to the `StartOfLoop` or `BootFailed` locations. This means

that the manifest has failed at least one check. With the current model, this query passes, which is what we expect.

Q11: Invalid key Leads to Invalid Key ID

```
(CheckPubKeyValid.publicKey.exponent != 1 ||
CheckPubKeyValid.publicKey.modulus != 1) --->
CheckPubKeyValid.currentPubKeyId == 0
```

This liveness property can be seen as a complement to Q7. It says that whenever the public key has a public key where either the modulus or the exponent is different from 0, then the resulting public key ID will be 0. Q11 passes as expected when run on the current model. Together with Q7, this query says that the calculated public key ID depends on the value of the given public key.

Results

Q1, Q3, and Q4 together show that values of the manifest are what we expect them to be throughout the locations in the `ROMStage` template. Q2, Q6, and Q8 verify that the values we define as correct are considered correct values in the model. Q5 shows that our PMP regions are correctly updated if we get to a point where we have verified a manifest and are ready to transfer execution to the next stage. Q7 and Q11 show that we calculate a correct key ID from a given public key. Q9 and Q10 show that we can successfully initiate the next stage (ROM_ext) and that not every manifest allows us to do this. Running Q1-Q11 through the verification tool in UPPAAL on the model as described in chapter 5 shows that they all pass. We believe that these queries combined show that the current model satisfies G1 fully.

6.2.2 Verifying G3

G3 states that "*The ROM_ext image signature must not change and must be signed by the Silicon Creator*". The signing by the Silicon creator is not in the scope of this project, but we can verify that the ROM_ext image signature does not change during the boot of the ROM stage. To verify this, we have made an observer template called the `ManifestObserver` template. The template can be seen in Fig. 6.2.1. The observer template reacts to broadcast synchronizations made by the `ROMStage` template. The `ManifestObserver` template verifies that the current manifest does not change between checks made by the `ROMStage` template.

In the transition from the `Initial` location, this template copies and saves the manifest that the `ROMStage` template has just passed to the `CheckRomExtManifest` template. When the `ROMStage` template is about to check the manifest's key, the observer template copies the key and checks whether that key is the same as the key in the manifest that was copied in the first transition. If it is not, the observer template goes to the `Failure` location. When the `ROMStage` template is about to check the manifest's image code signature, the observer template copies the image code and signature. It then checks whether those values are equal to the corresponding values in the original manifest copied in the first transition. If at least one of them is not equal, the observer template goes to the `Failure` location. If both are equal, the observer template goes to the `AwaitingManifest` location. From there, the `ManifestObserver` waits for the `ROMStage` template to hopefully pass the last check. When it does, it passes the manifest to the `FinalJumpToRomExt` and `ManifestObserver` templates. If the manifest that the `ManifestObserver` template receives is different from the one it received in the first transition, this template goes to the `Failure` location. Otherwise, it goes to the `Success` location.

If the `currentManifest` in the `ROMStage` template fails any of the checks and is therefore not validated, then this template returns to the `Initial` location. In case there is more than one flash bank, the `ManifestObserver` can be used to verify that none of the manifests change.

The `ManifestObserver` template being in the `Failure` location means that between the `ROMStage` receiving the manifest from the `RomExtManifestsToTry` template to passing the manifest to the `FinalJumpToRomExt` template, the manifest has changed. The `ManifestObserver` template being in the `Success` location means that in that same time frame, the manifest in the `ROMStage` template has not changed. It also means that all three checks performed in the `ROMStage` have passed.

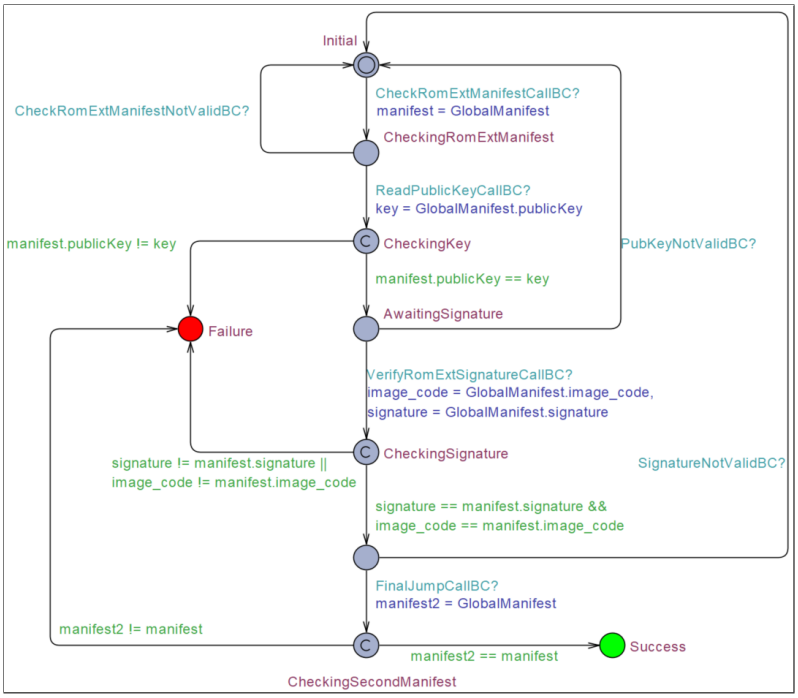


Figure 6.2.1: The `ManifestOberver` template in UPPAAL.

Queries

To verify G3, we have created two queries which can be seen in table 6.2.2.

ID	Name	Query	Passes?
Q12	Never Failure	`A[] !ManifestObserver.Failure`	✓
Q13	Verify Observer	`A[] FinalJumpToRomExt.ROMExtRunning imply ManifestObserver.Success`	✓

Table 6.2.2: The queries relevant for the G3 security goal.

Q12: Never Failure

```
A[] !ManifestObserver.Failure
```

This query says that the `ManifestObserver` template never reaches the `Failure` location. This query passes, which means that we know that the manifest (including its signature and image code fields) never changes between passed checks. This property does not verify that the `FinalJumpToRomExt` template re-

ceives the same manifest, since there might be a deadlock before `ManifestObserver` template reaches the `Failure` location.

Q13: Verify Observer

```
A[] FinalJumpToRomExt.ROMExtRunning imply ManifestObserver.Success
```

This query says that as long as the `FinalJumpToRomExt` template is in the `ROMExtRunning` location, then the `ManifestObserver` template must be in the `Success` location. This query passes, meaning that whenever the ROM_ext is running, then the three checks required for the ROM to boot are passed, and the manifest has not changed in the process.

Results

The results of queries Q12 and Q13 verify that the manifest in the `ROMStage` never changes between checks or before executing the ROM_ext image. This verifies the part of the security goal that says that the image signature must not change. The part of the security goal that says that the image signature must be signed by the Silicon Creator is not verified since it is not in the scope of this project. For these reasons, this security goal is partially verified.

6.3 Verifying P2

The security policy P2 states that "*Validation of a boot stage must only succeed if the boot environment is safe*". It contains the security goal G5.

6.3.1 Verifying G5

The security goal G5 states that "*ROM must use a key dependent on the environment to validate ROM_ext. The validation must fail if the environment has changed since the code was signed*".
Verifying this goal proved to be more difficult than anticipated. Reading through the documentation, it is difficult to discern if this environment check happens at this point in the design (at least for the ROM stage). We know that the Key Manager (cf. section 2.3.3) derives keys also by using system state measurements as input. However, as far as we can tell, the Key Manager is not used until after validation of the ROM_ext manifests is complete. This means that it cannot be the Key Manager that is used during ROM_ext manifest validation. Because of this, along with the fact that we cannot figure out if there are any other cases of keys being dependent on the environment from the documentation, we have chosen not to verify this goal.

6.4 Verifying P3

The security policy P3 states that "*Cryptography key leakage cannot happen*". It includes security goals G7, G8, and G9. Given the limitation of scope (cf. section 3.2), we are only interested in verifying G8 and G9.

6.4.1 Verifying G8

The security goal G8 states that "*Correct keys are only accessible to intended receivers*". Some templates need secret keys to do various actions, like scrambling parts of memory. Passing secret keys to these templates is necessary for them to work properly. Due to the way we have modeled value passing, variables

should only be passed between components that communicate according to the OpenTitan documentation or by passing the data through SRAM. Passing secret keys to other templates presents a security risk. In our UPPAAL model, it is infeasible to write queries for whether a secret key is passed between two templates by mistake. Instead, we try to partially verify this goal by checking whether secret keys are ever stored in SRAM. If a secret key is put into SRAM, then it is publicly available to the whole system, which violates G8.

Queries

To verify G8, we have created one query which can be seen in table 6.4.1.

ID	Name	Query	Passes?
Q14	Never in SRAM	`A[] SRAM.indexIsEmpty(6)`	✓

Table 6.4.1: The query used to verify G8.

Q14: Never in SRAM
`A[] SRAM.indexIsEmpty(6)`

This query is a safety property. It says that the sixth index in the SRAM is always empty. That index is exclusively used for private and secret keys. This query passes, which verifies that secret keys and private keys are never stored in the SRAM. The SRAM template and the *indexIsEmpty* function are described in section 5.3.8.

Results

Knowing that the secret keys are never stored in SRAM is not enough to conclude whether all keys are only accessible to their intended receivers. The SRAM is a way for the templates to share data with every other template. Knowing that the secret keys are never stored in the SRAM only means that the secret keys are never made public to the whole system. It would be interesting to verify that the components that use secret keys do not expose them to the other components that they communicate with. Given that we have been able to verify that secret keys are not stored in SRAM, we conclude that we have partially verified G8.

6.4.2 Verifying G9

The security goal G9 states that "*Secret information stored in memory must be cleared or scrambled after the termination of the respective boot stage*".
For this goal, we look specifically at the SRAM template of the model described in section 5.3.8. We want to ensure that everything stored in SRAM is either scrambled or wiped after the ROM transfers execution to the ROM_ext stage. In this section, we verify that SRAM is scrambled but not cleared after the ROM stage is done.

Queries

To verify G9, we have created three queries which can be seen in table 6.4.2.

ID	Name	Query	Passes?
Q15	Check SRAM Scrambled	`A[] FinalJumpToRomExt.ROMExtRunning imply forall (i : int[0, 7]) (SRAM.scrambled[i] || SRAM.indexIsEmpty(i))`	✓
Q16	Check SRAM Wiped	`A[] FinalJumpToRomExt.ROMExtRunning imply forall (i : int[0,7]) SRAM.indexIsEmpty(i)`	×
Q17	Check Some of SRAM Wiped	`A[] FinalJumpToRomExt.ROMExtRunning imply ((exists (i : int[0,5]) SRAM.indexIsEmpty(i)) || SRAM.indexIsEmpty(7))`	×

Table 6.4.2: Queries associated with G9.

Q15: Check SRAM Scrambled

```
A[] FinalJumpToRomExt.ROMExtRunning imply
forall (i : int[0, 7]) (SRAM.scrambled[i] || SRAM.indexIsEmpty(i))
```

This query is a safety property that uses local variables and functions in the `SRAM` template to check that all fields of `SRAM` are either scrambled or empty when the model is in a state that contains the `ROMExtRunning` location of the `FinalJumpToRomExt` template. The query passes, which is also the expected behavior.

Q16: Check SRAM Wiped

```
A[] FinalJumpToRomExt.ROMExtRunning imply forall (i : int[0,7]) SRAM.indexIsEmpty(i)
```

This query is almost identical to Q14. However, this query only passes if every field of `SRAM` is empty, i.e., wiped. This property does not pass, which means that not all of the data in the `SRAM` is wiped. Q11 tells us that the data that is not wiped is scrambled. An error trace of the property shows that the fields of the `SRAM` template are not wiped. All of the fields in the template have values different from 0, when in the `ROMExtRunning` location in the `FinalJumpToRomExt` template. The only exception to this is the private key field. This property failing is, however, the excepted result.

Q17: Check Some of SRAM wiped

```
A[] FinalJumpToRomExt.ROMExtRunning imply
((exists (i : int[0,5]) SRAM.indexIsEmpty(i)) || SRAM.indexIsEmpty(7))
```

This safety query says that whenever the `FinalJumpToRomExt` is in the `ROMExtRunning` location (i.e., what we consider a successful boot of the ROM stage), then some of the SRAM contents between indices 0 and 5 or index 7 must be empty. Since we know that all of the indices are set during the ROM stage, we know that they must have been cleared if they are empty. The query does not pass. Using a diagnostic

trace, we can see that there exists a trace where index 0-5 and index 7 of the SRAM template is not empty, while the FinalJumpToRomExt template is in the ROMExtRunning location. Since this query fails, we know that none of the contents are cleared. This query excludes index 6 because we know that index 6 is never written to given the result of Q14 and will therefore always appear empty.

Results

For this goal, we have taken memory to only mean SRAM, as this seems logical. We could potentially include flash, but that is not what is meant by this goal. As such, with the three queries seen in table 6.4.2, we can see that nothing in SRAM is wiped after the termination of the ROM stage. But the data stored in SRAM at that time is scrambled. This can be seen in the fact that Q15 passes while Q16 and Q17 fail. As such, we believe that the goal is fulfilled. However, it would be better for the security of the design if SRAM was wiped after a boot stage. That way, an attacker that somehow gets hold of the scrambling key for SRAM would not be able to extract information from SRAM after a boot stage is complete.

6.5 Verifying P4

As mentioned in section 3.2, P4 states that "*All data access rights follow a privilege hierarchy*". This means that the PmpModule template allows reading, writing, and executing a given memory section if and only if a PMP region that allows that operation for that memory section exists. If an invalid request is made, the system deadlocks.

6.5.1 Verifying G10, G11, G12

G10 says only software with write access to some section of memory may write to it. G11 and G12 say the same but are concerned with reading and executing, respectively, rather than writing. The PmpModule template (cf. section 5.3.4) is responsible for enforcing these security goals. The PmpModule template takes requests from the FlashController template to read, write to, or execute certain sections of memory. As mentioned in section 5.3.4, the PmpModule template checks the PmpRegions to compute whether reading, writing, or executing a specific range is allowed. If a read or write request is allowed, then the PmpModule template communicates with the Flash template to read from or write to flash memory. If a request is not allowed, then the PmpModule template synchronizes with the ExceptionHandler template via the RaiseException channel.

Queries

To verify G10, 11, and 12, we have created 13 queries which can be seen in table 6.5.1.

ID	Name	Query	Passes?
Q18	Exception Raised	`A[] !ExceptionHandler.ExceptionRaised`	✓
Q19	Never negative case - read	`A[] !PmpModule.BadRead`	✓
Q20	Never negative case - write	`A[] !PmpModule.BadWrite`	✓
Q21	Never negative case - execute	`A[] !PmpModule.BadExecute`	✓
Q22	Positive case - read	`A[] PmpModule.GoodRead imply (exists (pmpIndex : int[0,15]) PmpRegions[pmpIndex].startAddress <= PmpModule.start && PmpRegions[pmpIndex].endAddress >= PmpModule.end && PmpRegions[pmpIndex].read)`	✓
Q23	Positive case - write	`A[] PmpModule.GoodWrite imply (exists (pmpIndex : int[0, 15]) PmpRegions[pmpIndex].startAddress <= PmpModule.start && PmpRegions[pmpIndex].endAddress >= PmpModule.end && PmpRegions[pmpIndex].write)`	✓
Q24	Positive case - execute	`A[] PmpModule.GoodExecute imply (exists (pmpIndex : int[0,15]) PmpRegions[pmpIndex].startAddress <= PmpModule.start && PmpRegions[pmpIndex].endAddress >= PmpModule.end && PmpRegions[pmpIndex].execute)`	✓
Q25	Negative case - read	`A[] PmpModule.BadRead imply (!(exists (pmpIndex : int[0,15]) PmpRegions[pmpIndex].startAddress <= PmpModule.start && PmpRegions[pmpIndex].endAddress >= PmpModule.end && PmpRegions[pmpIndex].read))`	✓
Q26	Negative case - write	`A[] PmpModule.BadWrite imply (!(exists (pmpIndex : int[0,15]) PmpRegions[pmpIndex].startAddress <= PmpModule.start && PmpRegions[pmpIndex].endAddress >= PmpModule.end && PmpRegions[pmpIndex].write))`	✓

ID	Name	Query	Passes?
Q27	Negative case - execute	`A[] PmpModule.BadExecute imply (!(exists (pmpIndex : int[0,15]) PmpRegions[pmpIndex].startAddress <= PmpModule.start && PmpRegions[pmpIndex].endAddress >= PmpModule.end && PmpRegions[pmpIndex].execute))`	✓
Q28	Positive case possible - read	`E<> PmpModule.GoodRead`	✓
Q29	Positive case possible - write	`E<> PmpModule.GoodWrite`	×
Q30	Positive case possible - execute	`E<> PmpModule.GoodExecute`	✓

Table 6.5.1: The queries relevant for the G10, G11, and G12 security goals.

Q18: Exception Raised

`A[] !ExceptionHandler.ExceptionRaised`

If the `ExceptionHandler` template reaches the `ExceptionRaised` location, then either an invalid read, write, or execute request has been issued, or the `SRAM` template has been requested to read something that is not in SRAM. We believe that the query is partially relevant for these security goals, which is why it is included here. This query passes as expected, which means that we have verified that no invalid requests are made.

Q19-Q21: Never negative case

`A[] !PmpModule.BadRead`

These queries state that if the `PmpModule` template is ever in one of the "Bad" locations (i.e. `BadRead`, `BadWrite`, or `BadExecute`), then an invalid read request has been issued. These queries pass, which means that it is verified that such requests are never made. This is the expected behavior of the queries.

Q22-Q24: Positive case

```
A[] PmpModule.GoodRead imply (exists (pmpIndex : int[0,15])
PmpRegions[pmpIndex].startAddress <= PmpModule.start &&
PmpRegions[pmpIndex].endAddress >= PmpModule.end &&
PmpRegions[pmpIndex].read)
```

These queries say that whenever the `PmpModule` template reaches any of the "Good" locations (i.e. `GoodRead`, `GoodWrite`, or `GoodExecute`), then there is a PMP region that allows for reading, writing, or executing that section of memory. Q22, Q23, and Q24 all pass. These are also the expected results.

Q25-Q27: Negative case

```
A[] PmpModule.BadRead imply (!(exists (pmpIndex : int[0,15])
PmpRegions[pmpIndex]. startAddress <= PmpModule.start &&
PmpRegions[pmpIndex].endAddress >= PmpModule.end &&
PmpRegions[pmpIndex].read))
```

These queries say that whenever the PmpModule template reaches a "bad" location (i.e. BadRead, BadWrite, or BadExecute), then a PMP region that allows for reading, writing, or executing that section of memory does not exist. E.g., suppose the PmpModule template reaches the BadRead location. In that case, the PmpModule template will reject the request and will not communicate with the Flash template. These queries pass because the left-hand sides of the implication in the queries are always false. Q19-Q21 guarantee this. We have chosen to include them due to the fact that if the model is to be further developed, then these locations (i.e. BadRead, BadWrite, BadExecute) might become reachable. In that case, these properties must be upheld by the model.

Q28-Q30: Positive case possible
`E<> PmpModule.GoodRead`

These queries say that a path from the initial location to a state where the PmpModule template is in a "good" location exists (e.g. GoodRead). They are used to make sure that some of the other queries that imply something based on whether the PmpModule template is in this location do not succeed simply because the location is never reached. We have also verified this for writing and executing. Notably, the query for GoodWrite (Q29) fails, meaning that a valid writing request is never made. We cannot get a diagnostic trace from UPPAAL for this query because it is an E<> property. This means that the proof is that all traces do not uphold the property, making it meaningless to only show one trace. Q28 and Q30 both succeed. The results from the three queries are all what we expect.

Results

Q18 verifies that the ExceptionHandler template never goes to the ExceptionRaised location. Q19-Q21 verify that the PmpModule never reaches any of the "Bad" locations. This verifies that the reason Q18 passes is not due to faulty communication between the PmpModule and ExceptionHandler templates. Q22-Q27 verify the functions *canRead*, *canWrite*, *canExecute*. This means that we have verified that the PmpModule template correctly determines whether a request is valid or invalid. Q28-Q30 verify that the left-hand side of the implications in Q22-Q24 are eventually true, so the queries are not trivially true because the left-hand sides of the implications are always false. Ultimately, we believe that the queries (especially Q22-Q27) prove that the model satisfies the security goals G10, G11, and G12. Because our model only models the ROM stage of the boot process, G10, G11, and G12 are only verified for the ROM stage of the actual system. This means that we have not verified the goals for the subsequent boot stages.

6.6 Summary

Following the verification of the model, we have fully verified security goals G1, G9, G10, G11, and G12, and partially verified security goals G3 and G8. However, since we are only modeling the ROM stage, our results only pertain to the ROM stage of the system. Overall, we are satisfied with the number of fully verified goals, especially given the current state of the OpenTitan system and its documentation. Furthermore, we believe G1 to be one of the most critical security goals for the ROM stage. Therefore, being able to fully verify this goal is a good result on its own. To fully verify G3, we need to verify that the silicon creator signs the ROM_ext manifest, but we cannot guarantee that the Silicon Creator is the entity who signed the signature using the current UPPAAL model. To fully verify G8, we would need more details about the implementation of OpenTitan.

Chapter 7

Introducing Errors in the Model

In this chapter, we introduce errors in the model to show that we can use our queries to detect potential errors in the model or OpenTitan design. This chapter should give an idea of how these errors are caught using the Uppaal tool. In the tables throughout this chapter, we denote a query that passes in Uppaal with a ✓. Conversely, we denote a query that does not pass with a ×. Furthermore, we have included two columns for each query in each table. These are "Results without Error" and "Results with Error". The "Results without Error" column denotes whether the queries pass in the correct model (as it is described in chapter 5). The "Results with Error" column denotes whether the queries pass for the model with the given error in it. The queries are the same as mentioned in chapter 6. A full table of all the queries can be found in appendix D.

7.1 Faulty Hashing

To induce an error into the signature verification, we have tried to make a change to the HASH template shown in section 5.4.1. In this modified version of the HASH template, we have changed the local *SHA* function. The new implementation of the *SHA* function can be seen in listing 7.1.

```
1 int SHA(int msg){
2     return msg;
3     //return msg + 2;
4 }
```

Listing 7.1: The new erroneous implementation of *SHA*.

In this version of this function, we make it so that the *SHA* function does nothing to its input parameter. This means that the function returns its input without modifying it. This symbolizes an error where the hashing does not occur in the system, leading to no possible manifests being considered valid. To catch the error, we run the queries seen in table 7.1.1.

Query ID	Results without Error	Results with Error
Q4	✓	✓
Q9	✓	×
Q10	✓	✓
D1	✓	×
D3	✓	×

Table 7.1.1: Table with results of queries done on the model with and without a faulty hashing algorithm. The IDs of queries correspond to the IDs of the queries in chapter 6 (for a table with all queries see table D.0.1).

7.1.1 Q4: Checking Signature

It is easy to look at Q4 and think that it should fail given the faulty hashing. However, due to the use of "imply" and the fact that the left-hand side of the implication is always false, the property always holds. This is because the model cannot reach the `CheckedSignature` location in the `ROMStage` template with the faulty hashing implemented. To confirm this, we created and processed the following query: `E<>` ↪ `ROMStage.CheckedSignature`. This query does not pass, which means that our assumption is correct.

7.1.2 Q9: Valid Manifest Leads to Running Rom Ext

Q9 does not hold with the implementation of the faulty hashing. Using Uppaal to give a diagnostic trace of why this fails, we can see that there exists a trace where a valid manifest ends in the `BootFailed` location of the `ROMStage` template. This is incorrect behavior as a valid manifest should lead to ROM transferring control over to ROM_ext.

7.1.3 Q10: Invalid Manifest Leads to Failure

Q10 shows that it still holds that invalid manifests lead to the `ROMStage` template either going back to the `StartOfLoop` location or the `BootFailed` location. This means that we at least do not validate incorrect manifests.

7.1.4 D1: Possibly Never Fail

D1 fails, which means that all paths lead to either the `BootFailed` or `RomExtTerminated` location in `ROMStage`. This is in line with what we see by Q9 failing because there should not be any trace that leads to a success state with the faulty hashing implemented. For this query, we cannot get a diagnostic trace from Uppaal. This is because no trace upholds the property. It is difficult to give a single counterexample trace when the counterexample is the collection of all possible traces where no trace always upholds the property.

7.1.5 D3: Success Possible

D3 also fails, which means that there is no path through the system, which leads to the `FinalJumpToRomExt` template being in the `ROMExtRunning` location. This means that we cannot get to a success state. Similar to the case for D1, we cannot get a single diagnostic trace from Uppaal to show why this property fails. This is because the proof is that the set of all possible traces for the system does not contain a trace that eventually contains the `ROMExtRunning` location.

7.2 Faulty Signature Verification

To investigate how the system acts if the verification of the signature is faultily implemented, we have made a change to the function in the `OTBN` template that is responsible for validating the signature. Based on the change in the *`checkSignature`* function seen in listing 7.2, all signatures will now be validated by the function. This error symbolizes what happens if attackers have found a way to make an invalid signature appear correct.

```
bool checkSignature(){
    return true;
    //return signature == 1 && digest == 3 &&
    //key.modulus == 1 && key.exponent == 1;
}
```

Listing 7.2: The new erroneous implementation of *checkSignature*.

7.2.1 Results

Some of the queries that are affected by the change are shown in table 7.2.1. The results after making the change are as expected. The diagnostic trace backs this for each query.

Variables & Functions	Results without Error	Results with Error
Q4	✓	×
Q8	✓	×
Q10	✓	×

Table 7.2.1: Table with results of queries done on the model with and without a faulty *checkSignature* function. The IDs of queries correspond to the IDs of the queries in chapter 6 (for a table with all queries see table D.0.1).

The change introduced in the model shows that manifests with incorrect signature values will be accepted as correct. These manifests are the ones that are initialized with a signature value that is different from 1. The other fields in the manifests will still have their intended value when in the `CheckedSignature` location since the previous checks (the identifier and public key checks) are not erroneous.

7.2.2 Q4: Checking Signature

With the error introduced in the model, this query now fails. Based on an error trace provided by the UPPAAL verification tool, it shows that Q4 fails because the model can get to the `CheckedSignature` location in the `ROMStage` template, while the current manifest has a value different from 1 in the signature field. This makes sense since if the `ROMStage` is in the `CheckedSignature` location, it is no longer a guarantee that the manifest will contain fields all assigned to 1.

7.2.3 Q8: Valid Signature

Q8 fails instantly. This is expected behavior since a valid signature now has other correct values than the ones listed in Q8. Trying to get a diagnostic trace for this query will result in a trace with only a single state. That is because from the first state, the values in the `OTBN` component will all be zero, yet the *checkSignature* function will return true. Thus Q8 will state that something false equals something true, which is false.

7.2.4 Q10: Invalid Manifest Leads to Failure

Q10 also fails. This is because a manifest that differs from a valid manifest will not necessarily end up in the `BootFailed` or `StartOfLoop` location. This is backed by the verification tool where a trace display that a manifest that differs from the valid manifest, with, e.g., a value of the `image_code` being 2, will end up in the `ROMExtRunning` location in the `ROMStage` template.

7.3 Faulty Calculation of Public Key ID

We have tried to introduce an error to the function that validates the public key found in the `rom_ext_manifest_t` structs. We changed the function *`calculateKeyId`* so that it always sets the ID to 1, regardless of the validity of the key. The new implementation of the function can be seen in listing 7.3. This error symbolizes a bug in the algorithm or implementation for calculating the key ID. We introduce this error to see whether it can be seen in the results of the queries we have used to verify the model so far. It would be interesting to see whether a manifest with an incorrect public key could be validated by the `ROMStage` template with this change. To catch the error, we run the queries seen in table 7.3.1.

```
void calculateKeyId(){
    currentPubKeyId = 1;
    return;

    //if(publicKey.exponent == 1 && publicKey.modulus == 1)
    // currentPubKeyId = 1;
    //else
    // currentPubKeyId = 0;
}
```

Listing 7.3: The new erroneous implementation of *`calculatePubKeyId`*.

Query ID	Results without Error	Results with Error
Q3	✓	×
Q4	✓	✓
Q7	✓	✓
Q11	✓	×

Table 7.3.1: Table with results of queries done on the model with and without a faulty *`calculateKeyId`* function. The IDs of queries correspond to the IDs of the queries in chapter 6 (for a table with all queries see table D.0.1).

7.3.1 Q3: Checking Public Key

Q3 verifies that when the `currentManifest` has passed the first two checks in the `ROMStage` template, then the `currentManifest`'s identifier, public key exponent, and public key modulus are all equal to 1. This query verifies that the checks work correctly. By introducing the error discussed in this section, this query fails, meaning that it is possible to have a "wrong" identifier or public key and still pass the second check. This makes sense since the error allows the `CheckPubKeyValid` template to validate keys that are not valid. The error trace UPPAAL provides shows that the manifest in `Flash` could have an invalid key. That manifest is then validated by the `CheckPublicKey` template. This leads to a state that violates the Q3.

7.3.2 Q4: Checking Signature

Q4 verifies that when the `currentManifest` has passed all the checks, then its identifier, public key exponent, public key modulus, image code, and signature are all equal to 1. Because this query passes, we know that the only situation where the third check is passed is when all the fields are valid, i.e., equal to 1. If the query had failed, we would know that it was possible to pass all three checks with an invalid key provided that the implementation of the key id was faulty. The third check does not pass because it checks whether the signature is correct, and to do that, the signature must be decrypted by the correct public key. Since the key is wrong, the decrypted signature does not correspond to the image code, and signature verification will fail. This means that even though the key ID is correct by mistake, the manifest cannot be validated by the `ROMStage` template without additional errors.

7.3.3 Q7: Valid Key Leads To Valid Key ID

This query verifies that whenever the key in the `CheckPubKeyValid` template is valid (i.e., its modulus and exponent are equal to 1), then the public key ID calculated based on the ID will be equal to 1. This query passes with the error and without. This makes sense since the added error means that the public key ID is always calculated to be equal to 1.

7.3.4 Q11: Invalid key Leads To invalid Key ID

This query verifies that whenever the key is invalid (i.e., either its modulus or exponent are not equal to 1), then the public key ID will be equal to 0. Since this query and Q7 pass without the error in the correct model, we have verified that the value of the `currentManifest`'s public key affects the calculation of the public key ID. However, the query fails after having introduced the error. This means that an invalid public key does not lead to an invalid public key ID, which means that we have found the error. The trace that UPPAAL generates shows that the manifest with an invalid public key can be validated by the `CheckPubKeyValid` template and advance to the next check in the `ROMStage` template. Interestingly, the trace continues to the signature check, which fails (cf. section 7.3.2), and then goes to the `BootFailed` location afterward. The reason the trace is so long is that the states in the trace satisfy the left-hand side of the "leads to" in the query. To disprove the property UPPAAL needs to prove that it can always perform an infinitely repeatable sequence of transitions or reach a deadlock before the right-hand side of the "leads to" is satisfied.

Chapter 8

Reflections

In this chapter, we reflect upon the project and some of the choices we have made.

8.1 Reflections on the UPPAAL Tool

The Uppaal tool is a powerful model checking tool. In this section, we reflect on the use of Uppaal for this project.

8.1.1 Ease of Use

One of the significant strengths of the Uppaal tool is its ease of use. The tool uses a subset of C, which helps new users quickly grasp the syntax. Furthermore, the verification tool in Uppaal allows for easy verification of properties using a subset of TCTL. This makes verification easy, although it might not always be quick. For a query whose failure or success can be proven with one trace, Uppaal can compute and display a trace to prove how and why it fails or succeeds. This is extremely useful in debugging the model or finding errors in the design of the modeled system. This feature is not always given in formal verification tools. For example, Frama-C's WP plugin has little to no hand-holding when tracing errors, which makes it much more challenging to work with.

8.1.2 Expressiveness

The formalism of Uppaal allows for the user to be very expressive in their modeling. Throughout this project, we have taken advantage of the fact that Uppaal allows for the creation of code in a subset of C. We have used this feature to create variables and functions throughout the templates of the model described in chapter 5. The ability to declare structs has been particularly useful as this has allowed us to create types (e.g., for manifests) that make creating and handling the model simpler. Furthermore, the ability to use C code in addition to timed automata means that we have a lot of control regarding the granularity of abstraction that we use throughout the modeling. This allows us to fine-tune the model to both enhance the model itself and adjust the state space. At this point, we believe that this feature has been invaluable to the results of this project.

8.1.3 UPPAAL Verification Time

As described in chapter 6 the time it took us to verify our `A[]` queries was around 13 minutes each. While this might seem like a long time for a single query, it is worth noting that we are not just running tests or doing bug hunting. By running such a query through Uppaal, we are getting mathematical guarantees on the properties of the model. These properties are guaranteed to hold (or not, depending on the result) for the model as a whole. This is much more powerful than testing, which cannot give such guarantees. With this in mind, 13 minutes should not seem overwhelming. Furthermore, we would argue that even if the verification time increases by a large margin, as long as the verification succeeds, then the resulting guarantees should make up for the time spent verifying. However, if we want to change the model and

run the query several times, there are a few ways to mitigate the amount of time spent on a single query. We can run the query on better hardware since a faster processor helps with a shorter verification time. Another way is to combine several `A[]` queries into one. The reason behind the amount of time it takes to run `A[]` queries (if they pass) is that UPPAAL has to calculate the entire state space to verify them each time a query is verified. However, if we combine `A[]` queries UPPAAL only has to calculate the state space once for all of the combined properties. E.g., combining two `A[]` queries in the current model takes the verification time from around 26 minutes down to 13 minutes. The downside of this is that if a combined property fails, then it is harder to figure out which part of the query failed immediately. However, at that point, we can either get a counterexample trace from the verification tool or split the query at the cost of longer verification time.
UPPAAL does support a feature for reusing the state space when verifying multiple queries. However, this feature does not seem to work in UPPAAL 4.1.24[1]. It would be nice if UPPAAL itself reused the calculated state space between verifying queries. That way, the state space would only have to be calculated for one query, and then every subsequent query could be checked against the already computed state space instead of re-calculating it.

8.2 Validity of the Model

The following sections describe reflections that we have regarding the model shown in chapter 5. We reflect upon OpenTitan as the chosen case for this project and the level of abstraction found in the model.

8.2.1 OpenTitan as a Case

For this project, we chose to use OpenTitan as a case. However, this has not been the easiest case to work with. This is because OpenTitan is still under heavy development, which means that much of the design is not yet complete or is subject to change. Furthermore, the design being under constant change means that the documentation leaves a lot to be desired. While we have worked to try and replicate the OpenTitan design as best as we can, there have been times where we have had to deviate from the design or make a decision on something because it is not yet documented precisely how it is supposed to work. As an example of how the lacking documentation has affected the project, we had a hard time understanding how the HMAC, KMAC, and Key Manager components fit into the ROM stage. It was only late into the project period that we figured out that they are not used during manifest validation (at least at the time of writing). Furthermore, we had a hard time understanding exactly how public key IDs are stored. This was, however, alleviated by contacting the OpenTitan developers. These uncertainties and deviations mean that the modeling gap (cf. chapter 4) between our model and OpenTitan is noticeable in some places. Furthermore, as the OpenTitan design changes going forward, this gap will only become more significant. While this is undesirable in general, we believe that this project is still valuable to show just how UPPAAL can be applied to this type of verification. Even with these discrepancies between the OpenTitan design and the model, the fundamental process of using model checking as described in this report does not change. Furthermore, we still model very close to how OpenTitan functions at this point wherever possible. To minimize the modeling gap, we have tried to keep each of the components in the model to the responsibilities described in their respective documentation pages. When the documentation was lackluster, we have used the OpenTitan Github repository to ask our questions to the OpenTitan team and tried to incorporate the answers in the model design. This means that the model is still useful to show that the overall design of the ROM stage is correct. Another thing to note is that our current model can be used to document how the parts

[1] There seems to be some reuse happening in certain cases, but we do not know exactly when this happens.

of the OpenTitan design interact. Furthermore, if this model was to be continuously developed alongside OpenTitan, we might detect if errors are introduced in the design in the future.

If we had used another case that was not under development, we might have had a more concrete model, but this does not detract from the results of this project. The best-case scenario for a project such as this would be to develop the model along with the system from the inside rather than as a spectator of the system design process. This would mean that we could test the design step by step through a developing model. Results from this model could then influence the design as well.

8.2.2 Abstraction in the Model

For the model of the OpenTitan ROM stage, we have abstracted away from a lot of detail. This is done because we wanted to model the flow between the software and the hardware components. If we had tried to model the system so broadly without a high level of abstraction, we would undoubtedly have run into the state space explosion problem. The current level of abstraction means that we sometimes are not as near to the actual implementation as could be desired. However, if it is necessary to remove some of the abstraction we have made, it should be done only in the parts of the model that need it and not the whole model. We might be able to reduce abstraction in parts of the model and still be able to avoid the state space explosion problem. E.g., if we wanted to verify something regarding the implementation of the hashing algorithm, we could try to model this more closely to an actual implementation without changing other parts of the model.
With the current level of abstraction, we show how components in the system interact instead of verifying that their implementations are correct. For the results from our model to be correct, it is assumed that the individual hardware components and algorithms are correctly implemented. If we wanted to test that an algorithm is correct, we could create a separate model for this algorithm to verify it without increasing the state space of our current model.

8.2.3 Scalability of the Model

Currently it takes roughly 13 minutes to verify a single `A[]` property as described in chapter 6. This is using a high level of abstraction and running one flash bank (cf. chapter 6) as opposed to the two that are currently in place in the OpenTitan design. If we add more complexity to the model, the verification time increases. As an example, at one point during this project, we had another value to the manifest IDs (cf. section 5.1) so that we had three possible values (0 - missing, 1 - valid, 2 - invalid). Having only two values decreased the verification time of `A[]` queries by roughly 3 minutes, meaning that the verification time went from around 16 minutes to almost 13 minutes. Running the command line version of Uppaal with verifyta to check the state space decrease from this change, we can see that the size of the state space decreases from 35,386 states (when we have three values for IDs) to 26,474 states (two values).
The size of the state space should increase exponentially as we add more possible values to the manifests. We also tried to run the model with two flash banks. This made a single `A[]` query run for about 16 hours, at which point we stopped verification. These examples show that the state space grows quickly. It is important for the scalability of the model that we maintain control of the state space to ensure that we do not end up with an unusable model.

8.2.4 Modeling the Pipeline Architecture

During this project, we have not modeled any CPU architecture, e.g., pipelines and caches. We could do this following the METAMOC approach [56]. However, we believe that we do not need to model this level of detail for this project. We are not interested in worst-case execution time, as is the case with METAMOC. We are interested in how the parts of OpenTitan (software and hardware) interact. Furthermore, if we try to model the CPU to that level of detail, we would probably encounter the state space explosion problem. The case of OpenTitan does not lend itself to modeling specific architecture either. OpenTitan is still very much a work in progress, meaning that we know only little about the specifics of the architecture, although we know that it is a RISC-V chip. Furthermore, we do not even have finished OpenTitan boot code.

8.2.5 Parallelism in the Model

The use of synchronizations has heavily restricted the number of possible interleavings in the model. We have made it so that a function or hardware template that initiates another template waits for a return from that template before continuing. Since we are modeling hardware and low-level code, this might not be how the system works in reality. The amount of parallelism that can occur in the OpenTitan hardware might be more than what we allow. However, we believe that the ROM code description that OpenTitan has is designed to limit the amount of parallelism that can occur. This limits the potential for race conditions in the final system, making it easier to ensure that the system works correctly. Therefore, we believe that our choice to limit the number of possible interleavings in the model is well-founded.

8.3 Boot Policy Failure Functions

In the model, the boot policies contain fields that represent addresses of failure functions. These functions are supposed to be executed if no valid ROM_ext manifests are found or if ROM_ext fails during execution. This functionality is part of the OpenTitan design (cf. [38]), but they are not implemented in the model as of yet. However, an interesting thing to note about these functions is that we cannot find any documentation stating that validation is done for these functions. This means that an attacker could potentially run arbitrary and malicious code by changing where these addresses in the boot policy point to. This seems like a significant oversight in the current design. We expect that this is something that the developers of OpenTitan know and that validation of the boot policy is just not yet implemented. Once the design is adjusted to also include validation of the boot policies and these addresses, it could be interesting to incorporate it into the model. Depending on how it is implemented, it should not be difficult to add to the model.

8.4 ROM Controller

In the later part of the project period, the OpenTitan documentation added a webpage [57] describing the ROM Controller component. The ROM Controller component would have been an interesting component to implement in our modeled system based on the ROM Controller's functionality. The ROM Controller is responsible for validating the contents of ROM just after the boot is initiated. The validation is done with the use of a cryptographic hash function. The benefit of validating the contents of ROM is to determine whether the ROM has been tampered with while the system was turned off. Implementing the component into the model could give valuable insights into physical attacks on the system. Since validation of the ROM with the ROM Controller can prevent the subverting of the ROM by tampering with it, which is a way of attacking the system [57]. The ROM Controller and its functionality are within the scope of the

system. We did not include this component in the model due to the limited time left on the project after the documentation was added.

8.5 UPPAAL vs. CBMC

During this project, we have worked side-by-side with SV107f21 that performed formal verification on the same case and source code as us using the CBMC tool [58]. In this section, we briefly reflect on the CBMC tool compared to UPPAAL and the other group's results compared to ours. This should give an idea of some of the strengths and weaknesses of each tool.
CBMC stands for C Bounded Model Checker and is a tool that works directly on annotated C code to perform formal verification [59]. This means that, compared to UPPAAL, CBMC does not suffer as much from the modeling gap (as a potential UPPAAL model does) since it works on the actual code that needs to be verified. However, it is still important to annotate the program correctly so that the tool gives the expected results. This is similar to how we need to be careful when formulating our queries for the UPPAAL model.

8.5.1 Result Comparison

Group SV107f21 has been working with CBMC and OpenTitan to verify a subset of the same security policies as us, with their goals being very similar to ours (cf. section 3.2). By dividing their goals into 11 properties, they have been able to verify four goals successfully [58]. These correspond to our G1, G10, G11, and G12. In comparison, we have fully verified these same goals along with G9. Furthermore, we have partially verified two additional goals (G3 and G8). However, they have been able to verify their goals with less abstraction than we have. For example, some of the properties they have regarding signature verification checks that the RSA-3072 is exactly 3072 bits long. This is not something we can easily do with UPPAAL. Therefore, while we work on similar goals, the way that we achieve our outcome with the tools is slightly different. However, it is interesting to note that our results are similar to the other group's results in their outcome. The fact that our verified goals are the same can be seen as strengthening the results.
To do their verification in CBMC group SV107f21 has needed to incorporate hardware as well. When working with CBMC, the model is built by the code and annotations. Therefore their modeled hardware is represented by function calls in the code. The performance they experience when verifying properties is superior to what we experience when verifying our queries in UPPAAL [58]. They can verify all of their properties in around 16 minutes, whereas we can run a single `A[]` query in roughly 13 minutes. In addition, the guarantees generated by CBMC might also be stronger than our guarantees from UPPAAL. This is because CBMC works directly on the code and does not need as much abstraction. However, it is not easy to directly compare their code to our model as the levels of abstraction vary a lot.
What we consider one of UPPAAL's greatest strengths is the flexibility it provides in terms of modeling. As a tool, it is easy to switch between levels of abstraction due to both having the possibility of writing code and building graphical models. Because UPPAAL uses a subset of C, its users can model many different abstraction levels, which leads to a rich formalism. The modeling formalisms supported in UPPAAL are also easily applicable to building both hardware and software models. The flexibility of UPPAAL is what we believe, in a case like this, can make it advantageous compared to a tool like CBMC.

Chapter 9

Conclusion

To conclude on the work made in this project, we first look at the security goals we have tried to verify using the Uppaal model checking tool. The goals are G1, G3, G5, G8, G9, G10, G11, and G12. The remaining goals have been deemed out of the scope of this project. The description for each goal is found in section 3.2. As already mentioned in section 6.6 the goals have the following statuses. The fully verified goals are G1, G9, G10, G11, and G12. These are considered fully verified based on the results of the queries presented in chapter 6. Their combined results show that the model behaves in accordance with these goals. The remaining goals, G3 and G8, are only considered partially verified. These goals are only partially verified because we do not think that we can show enough through our model to fully verify them (cf. section 6.6). We have not been able to verify G5. This is because the documentation does not specifically state how the environment is validated. Furthermore, the Key Manager is the only component we could find that derives keys based on the environment. However, the Key Manager is not used for ROM_ext manifest validation as far as we can tell. We consider it a success that we have fully verified G1, G9, G10, G11, and G12. These goals constitute essential security aspects of the OpenTitan design. However, the guarantees given by the verification are only applicable to a system that corresponds to the model that we have built. Based on the deviations and assumptions we have had to make (as described in section 5.6) and OpenTitan not being fully designed, there is a modeling gap between our model and the OpenTitan design. We believe that if the model is not maintained, the modeling gap will eventually become so large that the results of this project will become obsolete as the OpenTitan project advances. However, the results documented in this report indicate that the OpenTitan project is heading in the right direction, at least in terms of its documented design.

We have previous experience with Uppaal. However, not on a case as comprehensive as this. The Uppaal model checking tool has been an excellent choice for this project. Modeling software and hardware together and the communication between those components have been a central and straightforward task. The only challenge we have encountered with the tool has been to limit the complexity of the model to avoid state space explosion. By using abstraction and careful manipulation of variable ranges, we have been able to reduce the state space to a point where we can still do meaningful verification without encountering the state space explosion problem. Furthermore, the tool has provided all the needed functionality in terms of modeling and verification. Both relying on timed automata to build a graphical model and express functionality in the subset of C often provides multiple satisfactory solutions when faced with a modeling problem. As the model became more and more complex, the error trace functionality has become more valuable. It drastically eases finding bugs in the model and the design, thus improving the validity of what we have modeled.

It has been challenging to work with the case chosen for this project. The documentation is often lackluster and difficult to understand while also being in a constant state of change. Getting an overview of how the components interact is a daunting task. The components that have had the most description overall have been the hardware components. The software does not yet have a corresponding degree of documentation.

Lack of documentation has caused trouble understanding both storage of public keys and the inclusion of HMAC, KMAC, and Key Manager components. Furthermore, understanding the derivation of keys and seeds and how this influences the ROM stage has been particularly difficult. Once faced with some issues, we have consulted the Github repository. Luckily we got some answers, but they more or less reflected the documentation; namely that there is still a lot to be decided and implemented in the system.

Since we have been working on the same case as SV107f21, we have compared some of our work. They have to a large extent, been able to verify the same properties as us. We consider it a good sign that two groups using different tools on the same case can end up with similar results. It aids the trust we have in our results that we, independently of each other, can draw the same conclusions about OpenTitan.

Finally, we will conclude on the problem statement that we have presented in chapter 1. This project has shown how the Uppaal model checking tool can be used to model a selected part of the booting process of the OpenTitan system. The project has also demonstrated that the tool is a suitable tool for modeling a boot process through a combination of hardware and software. Furthermore, we have used different modeling patterns and techniques to represent the OpenTitan system, e.g., using templates as functions and synchronizations as function calls. The selected parts of the security verified with Uppaal are related to the first part of the booting process. For this, we have been able to prove that the OpenTitan design upholds the properties described in G1, G9, G10, G11, and G12 of this report (cf. section 3.2). We have not experienced any times where Uppaal has come short in terms of the provided functionality or lack of expressivity. If more detailed C code was available for the OpenTitan design, it could be interesting to see if Uppaal could scale to incorporate this level of detail. If not, then we could use a tool such as CBMC in addition to Uppaal to have a model that is closer to the implementation level.

Chapter 10

Future Work

Based on chapter 9 and the current state of the OpenTitan project, an obvious angle on future work would be to continue work on the model as the OpenTitan project and documentation advances. If the model is continually developed alongside OpenTitan, we could use it to make sure that errors are not introduced in the design further down the line.

10.1 Continued Development of OpenTitan

It would be interesting to see if we can find new security flaws as the development advances. Once OpenTitan reaches a state where the software is no longer just pseudocode, our boot code and the corresponding template will need a rework. This could benefit the model by reducing the modeling gap since the ROM stage could then be modeled closer to the actual implementation.

Future work could also include the incorporation of the next boot stage, the ROM_ext stage. The work that has been done in this project has revolved around the ROM stage. Since the ROM_ext stage has several responsibilities and relies on other components than the ROM stage, there might be flaws related to security in the ROM_ext stage. Following the ROM_ext is the boot loader stage for which security interests also apply.

10.2 Different Component Versions

The Uppaal tool has a very rich modeling formalism that allows its users to model components in any level of abstraction they prefer. We could take advantage of this and make multiple versions of every template. E.g., one that is detailed and one that is abstract. This means if we wanted to run some queries that center around a specific subset of templates, we could use the detailed version of those templates, and the rest can be abstract. This could decrease verification time since the model will have a smaller state space. It also makes the model more modular because we can focus on many different aspects and disregard others. Having detailed versions of templates that are currently very abstract expands the model so that new properties that may be relevant to later boot stages can also be queried. With so many component versions, it makes sense to validate all the models that consist of relevant combinations of component versions.

Bibliography

[1] Christian Wienberg. *Maersk Says June Cyberattack Will Cost It up to $ 300 Million*. URL: https://www.bloomberg.com/news/articles/2017-08-16/maersk-misses-estimates-as-cyberattack-set-to-hurt-third-quarter (visited on 05/09/2021).

[2] Scott Ikeda. *Half a Million Zoom Accounts Compromised by Credential Stuffing, Sold on Dark Web*. URL: https://www.cpomagazine.com/cyber-security/half-a-million-zoom-accounts-compromised-by-credential-stuffing-sold-on-dark-web/ (visited on 05/09/2021).

[3] SecureTeam. *Boothole vulnerability explained*. URL: https://secureteam.co.uk/news/vulnerabilities/boothole-vulnerability-explained/ (visited on 05/09/2021).

[4] Microsoft. *Microsoft Guidance for Addressing Security Feature Bypass in GRUB*. URL: https://msrc.microsoft.com/update-guide/en-US/vulnerability/ADV200011 (visited on 05/09/2021).

[5] Edsger W. Dijkstra. *Notes On Structured Programming*. URL: https://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF (visited on 05/09/2021).

[6] LowRISC contributors. *OpenTitan Documentation*. URL: https://docs.opentitan.org/README/ (visited on 02/19/2021).

[7] OpenTitan. *OpenTitan*. URL: https://opentitan.org/ (visited on 05/26/2021).

[8] William D. Casper and Stephen M. Papa. "Root of Trust". In: *Encyclopedia of Cryptography and Security*. Ed. by Henk C. A. van Tilborg and Sushil Jajodia. Boston, MA: Springer US, 2011, pp. 1057–1060. ISBN: 978-1-4419-5906-5. DOI: 10.1007/978-1-4419-5906-5_789. URL: https://doi.org/10.1007/978-1-4419-5906-5_789.

[9] lowRISC. *lowRISC*. URL: https://lowrisc.org/ (visited on 06/01/2021).

[10] Cook B. et al. *Model Checking Boot Code from AWS Data Centers. In: Chockler H., Weissenbacher G. (eds) Computer Aided Verification. CAV 2018. Lecture Notes in Computer Science, vol 10982. Springer, Cham.* Apr. 15, 2020. URL: https://doi.org/10.1007/978-3-319-96142-2_28 (visited on 02/24/2021).

[11] OpenTitan. *Logical Security Model*. URL: https://docs.opentitan.org/doc/security/logical_security_model/ (visited on 02/09/2021).

[12] Bjarke Hilmer Møller et al. *Evaluation of Tools for Formal Verification of OpenTitan Boot Code*. URL: https://projekter.aau.dk/projekter/files/402371507/P9__13_.pdf (visited on 08/02/2021).

[13] moidx. *[doc] Validation of public keys in ROM_EXT manifests*. URL: https://github.com/lowRISC/opentitan/issues/5671 (visited on 03/19/2021).

[14] OpenTitan. *Secure Boot*. URL: https://docs.opentitan.org/doc/security/specs/secure_boot/ (visited on 03/23/2021).

[15] OpenTitan. *OpenTitan Use Cases*. URL: https://docs.opentitan.org/doc/security/use_cases/ (visited on 03/01/2021).

[16] Michael Cobb. *RSA algorithm (Rivest-Shamir-Adleman)*. URL: https://searchsecurity.techtarget.com/definition/RSA (visited on 03/02/2021).

[17] Tutorialspoint. *Understanding RSA Algorithm*. URL: https://www.tutorialspoint.com/cryptography_with_python/cryptography_with_python_understanding_rsa_algorithm.htm (visited on 03/02/2021).

[18] DI Management. *RSA Algorithm*. URL: https://www.di-mgt.com.au/rsa_alg.html (visited on 03/02/2021).

[19] Microsoft. *HMACSHA256 Class*. URL: https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.hmacsha256?view=net-5.0 (visited on 03/01/2021).

[20] geeksforgeeks. *HMAC algorithm*. URL: https://www.geeksforgeeks.org/what-is-hmachash-based-message-authentication-code/ (visited on 03/02/2021).

[21] OpenTitan. *HMAC HWIP Technical Specification*. URL: https://docs.opentitan.org/hw/ip/hmac/doc/ (visited on 04/06/2021).

[22] Hannes Gross et al. *Higher-Order Side-Channel Protected Implementations of KECCAK*. URL: https://eprint.iacr.org/2017/395.pdf (visited on 04/07/2021).

[23] OpenTitan. *KMAC HWIP Technical Specification*. URL: https://docs.opentitan.org/hw/ip/kmac/doc/ (visited on 04/06/2021).

[24] OpenTitan. *Key Manager HWIP Technical Specification*. URL: https://docs.opentitan.org/hw/ip/keymgr/doc/ (visited on 04/06/2021).

[25] OpenTitan. *Identities and Root Keys*. URL: https://docs.opentitan.org/doc/security/specs/identities_and_root_keys/ (visited on 05/10/2021).

[26] OpenTitan. *Life Cycle Controller Technical Specification*. URL: https://docs.opentitan.org/hw/ip/lc_ctrl/doc/ (visited on 04/07/2021).

[27] OpenTitan. *OpenTitan Big Number Accelerator (OTBN) Technical Specification*. URL: https://docs.opentitan.org/hw/ip/otbn/doc/ (visited on 04/07/2021).

[28] OpenTitan. *SRAM Controller Technical Specification*. URL: https://docs.opentitan.org/hw/ip/sram_ctrl/doc/ (visited on 04/09/2021).

[29] OpenTitan. *Flash Controller HWIP Technical Specification*. URL: https://docs.opentitan.org/hw/ip/flash_ctrl/doc/ (visited on 04/07/2021).

[30] OpenTitan. *OTP Controller Technical Specification*. URL: https://docs.opentitan.org/hw/ip/otp_ctrl/doc/ (visited on 05/05/2021).

[31] OpenTitan. *ENTROPY_SRC HWIP Technical Specification*. URL: https://docs.opentitan.org/hw/ip/entropy%5C_src/doc/ (visited on 05/05/2021).

[32] OpenTitan. *CSRNG HWIP Technical Specification*. URL: https://docs.opentitan.org/hw/ip/csrng/doc/ (visited on 05/05/2021).

[33] OpenTitan. *EDN HWIP Technical Specification*. URL: https://docs.opentitan.org/hw/ip/edn/doc/#overview (visited on 05/05/2021).

[34] OpenTitan. *AON Timer Technical Specification*. URL: https://docs.opentitan.org/hw/ip/aon_timer/doc/ (visited on 04/07/2021).

[35] OpenTitan. *Power Manager HWIP Technical Specification*. URL: https://docs.opentitan.org/hw/ip/pwrmgr/doc/ (visited on 04/07/2021).

[36] OpenTitan. *otp_ctrl.sv*. URL: https://github.com/lowRISC/opentitan/blob/master/hw/ip/otp_ctrl/rtl/otp_ctrl.sv (visited on 05/08/2021).

[37] OpenTitan. *Mask ROM*. URL: https://docs.opentitan.org/sw/device/silicon_creator/mask_rom/docs/ (visited on 06/01/2021).

[38] OpenTitan. *Reference Mask ROM: Secure Boot Description*. URL: https://docs.opentitan.org/sw/device/mask_rom/docs/ (visited on 03/18/2021).

[39] OpenTitan. *EDN HWIP Technical Specification*. URL: https://docs.opentitan.org/hw/ip/edn/doc/ (visited on 04/08/2021).

[40] OpenTitan. *CSRNG HWIP Technical Specification*. URL: https://docs.opentitan.org/hw/ip/csrng/doc/ (visited on 04/08/2021).

[41] OpenTitan. *ENTROPY_SRC HWIP Technical Specification*. URL: https://docs.opentitan.org/hw/ip/entropy_src/doc/ (visited on 04/08/2021).

[42] OpenTitan. *AES HWIP Technical Specification*. URL: https://docs.opentitan.org/hw/ip/aes/doc/ (visited on 04/12/2021).

[43] OpenTitan. *Alert Handler Technical Specification*. URL: https://docs.opentitan.org/hw/ip/alert_handler/doc/ (visited on 04/12/2021).

[44] OpenTitan. *NMI Generator Technical Specification*. URL: https://docs.opentitan.org/hw/ip/nmi_gen/doc/ (visited on 04/12/2021).

[45] OpenTitan. *Interrupt Controller Technical Specification*. URL: https://docs.opentitan.org/hw/ip/rv_plic/doc/ (visited on 04/12/2021).

[46] OpenTitan. *Timer HWIP Technical Specification*. URL: https://docs.opentitan.org/hw/ip/rv_timer/doc/ (visited on 04/12/2021).

[47] Andrew Waterman et al. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*. URL: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMFDQC-and-Priv-v1.11/riscv-privileged-20190608.pdf (visited on 02/18/2021).

[48] RISC-V. *About Risc-V*. URL: https://riscv.org/about/ (visited on 02/18/2021).

[49] Edmund M. Clarke et al., eds. *Handbook of Model Checking*. Springer, 2018. ISBN: 978-3-319-10574-1. DOI: 10.1007/978-3-319-10575-8. URL: https://doi.org/10.1007/978-3-319-10575-8.

[50] UPPAAL. *UPPAAL Homepage*. URL: https://uppaal.org/ (visited on 04/13/2021).

[51] PRISM. *PRISM Homepage*. URL: https://www.prismmodelchecker.org/ (visited on 04/13/2021).

[52] Storm. *Storm Homepage*. URL: https://www.stormchecker.org/ (visited on 04/13/2021).

[53] Luca Aceto et al. *Reactive Systems: Modelling, Specification and Verification*. English. Cambridge University Press, 2007. ISBN: 9780521875462.

[54] Alexandre David and Kim Larsen. "A Tutorial on Uppaal 4.0". In: (Jan. 2006).

[55] Sam Elliot et al. *Reference Mask ROM: Secure Boot Description*. URL: https://docs.opentitan.org/sw/device/rom_exts/docs/manifest/ (visited on 03/01/2021).

[56] Andreas Engelbredt Dalsgaard et al. "METAMOC; Modular Execution Time Analysis using Model Checking". English. In: *Proceedings of the 10th International Workshop on Worst-Case Execution-Time Analysis (WCET2010)*. 2010.

[57] OpenTitan. *ROM Controller Technical Specification*. URL: https://docs.opentitan.org/hw/ip/rom_ctrl/doc/ (visited on 05/11/2021).

[58] Jacob Gosch Søndergaard and Kristoffer Skagbæk Jensen. "Formally Verifying the Correctness and Safety of OpenTitan Boot Code using CBMC". To Appear. MA thesis. AAU, 2021.

[59] CBMC. *The CBMC Homepage*. URL: `https://www.cprover.org/cbmc/` (visited on 05/20/2021).

[60] OpenTitan. *Manifest Format*. URL: `https://docs.opentitan.org/sw/device/silicon_creator/rom_exts/docs/manifest/` (visited on 06/02/2021).

Appendix A

OpenTitan ROM_ext Manifest Description

The ROM_ext manifests in OpenTitan are blocks of data containing all necessary elements to validate and boot a ROM_ext. These are the manifests that the ROM stage needs to validate before a ROM_ext can take over. This section is used to describe the structure of these manifests in OpenTitan. For this, we refer to [60] which details the format of manifests in OpenTitan. The layout of a manifest can be seen in table A.0.1.

Field Name
manifest_identifier
image_signature
image_length
image_version
image_timestamp
signature_key_public_exponent
usage_constraints
peripheral_lockdown_info
signature_key_modulus
extension0_offset
extension0_checksum
extension1_offset
extension1_checksum
extension2_offset
extension2_checksum
extension3_offset
extension3_checksum
code image

Table A.0.1: The layout of an OpenTitan ROM_ext manifest as seen in [60]. We have excluded reserved places.

Furthermore, the following list gives a brief description of each of the fields seen in table A.0.1.

- **manifest_identifier:** This is used by the ROM stage to check that the manifest is of the format described in this section. It is a 32-bit enumeration value.
- **image_signature:** The signature takes as input the digest of all the following fields in the manifest. With this digest and the silicon creator private key, the image_signature is calculated with the RSA-3072 algorithm.

- **image_length:** The image length is used to give the offset of the image code end so that its length can be deduced.
- **image_version:** This is a 32-bit numeric value that denotes the ROM_ext image version number.
- **image_timestamp:** This is the Unix timestamp for when the ROM_ext image was created.
- **signature_key_public_exponent:** This is the RSA-3072 public exponent that is part of the public key used for signature verification.
- **usage_constraints:** This is used to mark a ROM_ext as only being able to validate properly on one or more devices.
- **peripheral_lockdown_info:** ROM uses the peripheral_lockdown_info to configure certain hardware components so that later stages cannot modify them.
- **signature_key_modulus:** This is the second part of the RSA-3072 public key used for signature verification.
- **extension0-3:** These extensions are made so that it is possible to extend the ROM_ext manifest without completely changing the format.
- **code image:** This is the ROM_ext code and data.

Appendix B

Our Boot Code

Listing B.1 shows the boot code that we have written in collaboration with group SV107f21.

```
/*
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;
    rom_ext_manifest_t rom_exts_mfs[];
```

```
} rom_exts_manifests_t;


//Represents boot policy
typedef struct boot_policy_t{
    int identifier;

    //which rom_ext_slot to boot
    int rom_ext_slot;

    //what to do if all ROM Ext are invalid
    void (*fail) ();

    //what to do if the ROM Ext unexpectedly returns
    void (*fail_rom_ext_terminated) (rom_ext_manifest_t);

} boot_policy_t;



typedef void(rom_ext_boot_func)(void); // Function type used to define function pointer
    ↪ to the entry of the ROM_EXT stage.


extern int* READ_FLASH(int start, int end);

boot_policy_t read_boot_policy()
{
    int* data = READ_FLASH(0, sizeof(boot_policy_t));

    boot_policy_t boot_policy;

    memcpy(&boot_policy.identifier, data, sizeof(boot_policy.identifier));
    memcpy(&boot_policy.rom_ext_slot, data + 1, sizeof(boot_policy.rom_ext_slot));
    memcpy(&boot_policy.fail, data + 2, sizeof(boot_policy.fail));

    return boot_policy;
}

rom_exts_manifests_t rom_ext_manifests_to_try(boot_policy_t boot_policy) {}

pub_key_t read_pub_key(rom_ext_manifest_t current_rom_ext_manifest) {
    return current_rom_ext_manifest.pub_signature_key;
}

int check_pub_key_valid(pub_key_t rom_ext_pub_key); // returns a boolean value

extern char* HASH(char* message);

extern int RSA_VERIFY(pub_key_t pub_key, char* message, int32_t* signature);

int verify_rom_ext_signature(pub_key_t rom_ext_pub_key, rom_ext_manifest_t manifest) {
    return RSA_VERIFY(rom_ext_pub_key, HASH(manifest.image_code), manifest.signature);
    ↪ //0 or 1
}

extern void WRITE_PMP_REGION(uint8_t reg, uint8_t r, uint8_t w, uint8_t e, uint8_t l);

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     //Maaske 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 B.1: Boot code developed together with group SV107f21. The code is created to mimic how we understand the OpenTitan ROM stage.

Appendix C

Structs

The declarations for the structs used in the model can be seen in listing C.1. The **boot_policy_t** contains information about an identifier, the rom_ext_slot, an address of a fail function, which should be called upon boot failure. Also an address of another fail function, the `rom_ext_failure` function, which is the one that should be called if the execution has been passed to the ROM_ext stage. An example of the `boot_failed_rom_ext_terminated` function can be seen in line 35 in listing 3.1.
The **pub_key_t** struct contains an exponent and modulus. Both these values are limited to a range of $[0, 2]$ this gives us the possibility of having each being empty, correct, or incorrect, represented by 0, 1, 2, respectively.
The **rom_ext_manifest_t** struct is explained in section 5.1. The **rom_exts_manifests_t** struct is only used to contain the instances of **rom_ext_manifest_t**, since there can be multiple of these based on the number of flash banks.
The **PmpRegion_t** struct is used to divide the memory into PMP regions, with different read, write, execution, and locked values. These fields correspond to fields described in section 2.6

```
1  typedef struct{
2      int identifier;
3      int rom_ext_slot[NumberOfFlashBanks];
4      int fail_function_address;
5      int rom_ext_failure_function_address;
6  } boot_policy_t;
7
8  typedef struct {
9      int[0,2] exponent;
10     int[0,2] modulus;
11 } pub_key_t;
12
13 typedef struct {
14     int[0,2] identifier;
15     int entryPoint;
16     pub_key_t publicKey;
17     int[0,2] signature;
18     int[0,2] image_code;
19 } rom_ext_manifest_t;
20
21 typedef struct {
22     int size;
23     rom_ext_manifest_t rom_exts_mfs[NumberOfFlashBanks];
24 } rom_exts_manifests_t;
25
```

```
26 typedef struct{
27     int startAddress;
28     int endAddress;
29     bool read;
30     bool write;
31     bool execute;
32     bool locked;
33 } PmpRegion_t;
```

Listing C.1: The implementation of our structs.

Appendix D

UPPAAL Queries

Table D.0.1 contains all the queries we use to validate the model and verify the security goals in chapter 6. The queries with a Q identifier are used to verify the security goals, and the queries with a D identifier are used to validate the model. In the tables throughout this appendix, we denote a query that passes in UPPAAL with a ✓. Conversely, we denote a query that does not pass with a ×.

ID	Name	Query	Passes?
Q1	Checking Manifest Identifier	`A[] ROMStage.IdentifierChecked imply ROMStage.currentManifest.identifier == 1`	✓
Q2	Checking Valid Manifest Identifier	`A[] (CheckRomExtManifest.manifest.identifier == 1) == CheckRomExtManifest.checkRomExtManifest()`	✓
Q3	Checking Public Key	`A[] ROMStage.CheckingSignature imply (ROMStage.currentManifest.identifier == 1 && ROMStage.currentManifest.publicKey.modulus == 1 && ROMStage.currentManifest.publicKey.exponent == 1)`	✓
Q4	Checking Signature	`A[] ROMStage.CheckedSignature imply (ROMStage.currentManifest.identifier == 1 && ROMStage.currentManifest.publicKey.modulus == 1 && ROMStage.currentManifest.publicKey.exponent == 1 && ROMStage.currentManifest.signature == 1)`	✓
Q5	PMP Execute	`A[] ROMStage.ReadyToRunROMExt imply (PmpRegions[0].execute && PmpRegions[0].startAddress <= ROMStage.currentManifest.entryPoint && PmpRegions[0].endAddress >= ROMStage.currentManifest.entryPoint + 4)`	✓

Id	Name	Query	Passes?
Q6	Valid Key ID	A[] (CheckPubKeyValid.currentPubKeyId == 1) == CheckPubKeyValid.checkPublicKey()	✓
Q7	Valid Key Leads To Valid Key ID	(CheckPubKeyValid.publicKey.exponent == 1 && CheckPubKeyValid.publicKey.modulus == 1) --> CheckPubKeyValid.currentPubKeyId == 1	✓
Q8	Valid Signature	A[] (OTBN.signature == 1 && OTBN.digest == 3 && OTBN.key.modulus == 1 && OTBN.key.exponent == 1) == OTBN.checkSignature()	✓
Q9	Valid Manifest leads to running Rom Ext	EqualManifestContents(validManifest, ROMStage.currentManifest) --> (FinalJumpToRomExt.ROMExtRunning && EqualManifestContents(validManifest, ROMStage.currentManifest))	✓
Q10	Invalid Manifest Leads to Failure	!EqualManifestContents(validManifest, ROMStage.currentManifest) --> (ROMStage.StartOfLoop || ROMStage.BootFailed)	✓
Q11	Invalid Key Leads To Invalid Key ID	(CheckPubKeyValid.publicKey.exponent != 1 || CheckPubKeyValid.publicKey.modulus != 1) --> CheckPubKeyValid.currentPubKeyId == 0	✓
Q12	Never Failure	A[] !ManifestObserver.Failure	✓
Q13	Verify Observer	A[] FinalJumpToRomExt.ROMExtRunning imply ManifestObserver.Success	✓
Q14	Never in SRAM	A[] SRAM.indexIsEmpty(6)	✓
Q15	Check SRAM Scrambled	A[] FinalJumpToRomExt.ROMExtRunning imply **forall** (i : int[0, 7]) (SRAM.scrambled[i] || SRAM.indexIsEmpty(i))	✓
Q16	Check SRAM Wiped	A[] FinalJumpToRomExt.ROMExtRunning imply **forall** (i : int[0,7]) SRAM.indexIsEmpty(i)	×
Q17	Check Some of SRAM Wiped	A[] FinalJumpToRomExt.ROMExtRunning imply ((**exists** (i : int[0,5]) SRAM.indexIsEmpty(i)) || SRAM.indexIsEmpty(7))	×

Id	Name	Query	Passes?
Q18	Exception Raised	`A[] !ExceptionHandler.ExceptionRaised`	✓
Q19	Never negative case - read	`A[] !PmpModule.BadRead`	✓
Q20	Never negative case - write	`A[] !PmpModule.BadWrite`	✓
Q21	Never negative case - execute	`A[] !PmpModule.BadExecute`	✓
Q22	Positive case - read	`A[] PmpModule.GoodRead imply (exists (pmpIndex : int[0,15]) PmpRegions[pmpIndex].startAddress <= PmpModule.start && PmpRegions[pmpIndex].endAddress >= PmpModule.end && PmpRegions[pmpIndex].read)`	✓
Q23	Positive case - write	`A[] PmpModule.GoodWrite imply (exists (pmpIndex : int[0, 15]) PmpRegions[pmpIndex].startAddress <= PmpModule.start && PmpRegions[pmpIndex].endAddress >= PmpModule.end && PmpRegions[pmpIndex].write)`	✓
Q24	Positive case - execute	`A[] PmpModule.GoodExecute imply (exists (pmpIndex : int[0,15]) PmpRegions[pmpIndex].startAddress <= PmpModule.start && PmpRegions[pmpIndex].endAddress >= PmpModule.end && PmpRegions[pmpIndex].execute)`	✓
Q25	Negative case - read	`A[] PmpModule.BadRead imply (!(exists (pmpIndex : int[0,15]) PmpRegions[pmpIndex].startAddress <= PmpModule.start && PmpRegions[pmpIndex].endAddress >= PmpModule.end && PmpRegions[pmpIndex].read))`	✓
Q26	Negative case - write	`A[] PmpModule.BadWrite imply (!(exists (pmpIndex : int[0,15]) PmpRegions[pmpIndex].startAddress <= PmpModule.start && PmpRegions[pmpIndex].endAddress >= PmpModule.end && PmpRegions[pmpIndex].write))`	✓

Id	Name	Query	Passes?

Q27	Negative case
	- execute
		`A[] PmpModule.BadExecute imply`
		`(!(exists (pmpIndex : int[0,15])`
		`PmpRegions[pmpIndex].startAddress <=`
		`PmpModule.start &&`
		`PmpRegions[pmpIndex].endAddress >=`
		`PmpModule.end &&`
		`PmpRegions[pmpIndex].execute))`
			✓

Q28	Positive case possible - read	`E<> PmpModule.GoodRead`	✓
Q29	Positive case possible - write	`E<> PmpModule.GoodWrite`	×
Q30	Positive case possible - execute	`E<> PmpModule.GoodExecute`	✓
D1	Possibly Never Fail	`E[] not( ROMStage.BootFailed` `|| ROMStage.RomExtTerminated)`	✓
D2	Guaranteed Never Fail	`A[] not( ROMStage.BootFailed` `|| ROMStage.RomExtTerminated)`	×
D3	Success Possible	`E<>` `FinalJumpToRomExt.ROMExtRunning`	✓
D4	Success Guaranteed	`A<>` `FinalJumpToRomExt.ROMExtRunning`	×
D5	Exception Never Raised	`A[] !ExceptionHandler.ExceptionRaised`	✓

D6	Memory
	Unscrable
	Urgent
		`A[]`
		`FinalJumpToRomExt.ROMExtRunning`
		`imply GlobalMemoryIsScrambled == false`
			✓

D7	Flash Immediately Unscrambled	`A[] FlashController.ScrambleCheck` `imply !GlobalMemoryIsScrambled`	✓
D8	SRAM Immediately Unscrambled	`A[] SRAMController.ScrambleCheck` `imply !GlobalMemoryIsScrambled`	✓

D9	Exactly
	One Choice
		`A[]MemoryHandling.ChooseType imply`
		`(((GlobalBootPolicy != GlobalEmptyBootPolicy) +`
		`(GlobalImageCode != 0) +`
		`(GlobalDigest != 0) +`
		`(GlobalPublicKey != GlobalEmptyPublicKey) +`
		`(GlobalManifest != GlobalEmptyManifest) +`
		`(GlobalSignature != 0) +`
		`(GlobalPrivateKey != GlobalEmptyPublicKey)) == 1)`
			✓

D10	Writing
	Manifest
	is Disallowed
		`A[]`
		`!PmpModule.canWrite`
		`(30 + 35 * ROMStage.loopIndex,`
		`41 + 35 * ROMStage.loopIndex)`
			✓

Table D.0.1: All the queries used in verification and validation of the model described in chapter 5.

Appendix E

Query Times

Table E.0.1 contains all queries that are not A[] queries that pass. Furthermore, we have included the time it takes to verify each query running a breadth-first strategy on a Ryzen 3700x processor.

ID	Name	Query	Time	Passes?
Q7	Valid Key Leads to Valid Key ID	`(CheckPubKeyValid.publicKey.exponent == 1 && CheckPubKeyValid.publicKey.modulus == 1) -- > CheckPubKeyValid.currentPubKeyId == 1`	12:57	✓
Q9	Valid Manifest Leads to Running Rom Ext	`EqualManifestContents(validManifest, ROMStage.currentManifest) -- > (FinalJumpToRomExt.ROMExtRunning && EqualManifestContents(validManifest, ROMStage.currentManifest))`	13:10	✓
Q10	Invalid Manifest Leads to Failure	`!EqualManifestContents(validManifest, ROMStage.currentManifest) -- > (ROMStage.StartOfLoop || ROMStage.BootFailed)`	35:31	✓
Q11	Invalid Key Leads to Invalid Key ID	`(CheckPubKeyValid.publicKey.exponent != 1 || CheckPubKeyValid.publicKey.modulus != 1) -- > CheckPubKeyValid.currentPubKeyId == 0`	12:19	✓
Q16	Check SRAM Wiped	`A[] FinalJumpToRomExt.ROMExtRunning imply forall (i : int[0,7]) SRAM.indexIsEmpty(i)`	12:37	×
Q17	Check Some of SRAM Wiped	`A[] FinalJumpToRomExt.ROMExtRunning imply forall (i : int[0,5]) SRAM.indexIsEmpty(i) || SRAM.indexIsEmpty(7)`	12:36	×
Q28	Positive Case Possible - Read	`E<> PmpModule.GoodRead`	01:25	✓
Q29	Positive Case Possible - Write	`E<> PmpModule.GoodWrite`	12:36	×
Q30	Positive Case Possible - Execute	`E<> PmpModule.GoodExecute`	12:17	✓

Id	Name	Query	Time	Passes?
D1	Possibly Never Fail	`E[] not( ROMStage.BootFailed || ROMStage.RomExtTerminated)`	23:22	✓
D2	Guaranteed Never Fail	`A[] not( ROMStage.BootFailed || ROMStage.RomExtTerminated)`	00:03	×
D3	Success Possible	`E<> FinalJumpToRomExt.ROMExtRunning`	12:29	✓
D4	Success Guaranteed	`A<> FinalJumpToRomExt.ROMExtRunning`	00:05	×

Table E.0.1: Queries and the time it takes to verify them (denoted in minutes and seconds). Passing `A[]` queries are excluded since these all take the same time.