A Secure Bootloader for Demonstrating Formal Verification of Hardware-Firmware Interactions on SoCs

Elaine Chou
Advisor: Sharad Malik
Second Reader: Aarti Gupta

May 2, 2016

Submitted in partial fulfillment
of the requirements for the degree of
Bachelor of Science in Engineering
Department of Electrical Engineering
Princeton University

I hereby declare that this Independent Work report represents my own work in accordance with University regulations.

Elaine Chou

A Secure Bootloader for Demonstrating Formal Verification of Hardware-Firmware Interactions on SoCs

Elaine Chou

Abstract

The of completeness formal verification that provides either a proof of correctness to determine that a checked property holds, or a counterexample that is in violation, is a valuable asset to security verification, which requires comprehensive testing of all corner cases. While there has been increasing use of model checking as a means of verifying hardware and software separately, unified hardware and software analysis tools are less developed. As integrated circuits become more complex, the prevalence of system-on-chip (SoC) designs with a programmable core that runs firmware to control and interact with hardware accelerators and other peripheral devices presents a growing need for reliable hardware-firmware coverification. In pursuit of the goal of developing a tool for model checking across the hardware-firmware boundary that is scalable and efficient, we build a realistic testing infrastructure that can demonstrate the effectiveness of the models that are in development. This task involves adding hardware accelerators to a 8051 microcontroller to create a substantial hardware-firmware interface, developing a secure bootloader that makes use of interactions between hardware and firmware, and identifying system properties of the secure bootloader that cross the hardware-firmware divide and verifying them with formal methods. A subset of these secure boot properties cannot be expressed with temporal logics, so we explore two proposed languages that address this challenge. To perform verification, we abstract the hardware into software, substituting uninterpreted functions in place of complex operations whose details do not affect the properties being checked.

Acknowledgments

Thanks to Professor Malik and Pramod for their continued guidance, and Professor Aarti Gupta for teaching me the fundamentals of formal verification. This project is based on the research in specification-driven SoC verification for managing hardware-firmware interactions being investigated by Sharad Malik, Pramod Subramanyan, and Yakir Vizel. In addition, it has been supplemented by a COS597B project on hyperproperties in software done with Bo-Yuan Huang. The work on page tables was a product of sophomore independent work by Samuel Miller. This research was supported by the SEAS Friedland Independent Work/Senior Thesis Fund and the Electrical Engineering Kamran Rafieyan '89 Fund for Undergraduate Research.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Advances in design and computation enabling technology to become both smaller and faster have led to the development of the Internet of Things, embedded systems, and a growing variety of smart devices that are integrating technology into everyday life at an astonishing rate. With this proliferation of devices comes a growing potential for vulnerabilities in a system. Especially in hardware, where applying a patch to fix a previous oversight is costly compared to simply applying a software update, the resources devoted to verification can even be greater than for the design process.

One approach to verification is simulation, where test inputs are run on a test bench and the outputs are compared with a reference. However, when the number of possible test cases is extremely large, comprehensive testing of all corner cases becomes infeasible. Unless every possibility is explored, simulation cannot guarantee total correctness; it can only show the existence of bugs in a design, as opposed to confirming their absence.

In contrast, formal methods produce either a proof of correctness to determine that a checked property holds, or a counterexample that is in violation. The guarantees provided by this approach are a valuable asset to verification, especially in areas such as security, where adversaries are actively looking to exploit any vulnerability that has been overlooked. The large threat that security bugs pose are only growing as more data is stored electronically and more controls are relinquished to automated systems. In addition to the security and privacy of consumers that is put at risk by the growing applications of computing, the security of the hardware or software itself against unintended uses is also of concern.

1.1 Background and Context

Formal verification in hardware has been a topic of interest for quite some time, and there are several approaches to formal methods-based verification. Checking behavior of combinational

circuits involves verifying a propositional logic formula, which can be accomplished with SAT solvers, which determine the satisfiability of a boolean function. There are many applications of SAT solvers to hardware design problems in additional to equivalence checking, such as automatic test generation[1] and model checking[2], and there have been advances in SAT solving algorithms, many based on the Davis-Putnam-Logemann-Loveland (DPLL)[3] search algorithm. Approaches to analyzing boolean formulas range from searching for a counterexample or unsatisfiable core[4], a subset of clauses from a Conjunctive Normal Form (CNF) formula that is not satisfiable. A CNF formula is the AND of clauses, where each clause is the OR of boolean variables and variable complements. An alternate to SAT is expressing the Boolean function as a binary decision diagram (BDD) in canonical form[5]. These techniques are also employed in model checking to verify properties of a system, where these properties are expressed as temporal logic formulas on a model describing the states, transition relations, and labels of each state[6].

One formal system developed for software verification is Hoare logic[7], which for a program statement S specifies a precondition P and a postcondition Q that must be true before and after execution respectively, both of which are formulas in first order logic, which is more expressive because it allows propositional formulas to be applied as predicates to describe properties of a variable. However, generally many of the methods applied to formal verification in hardware can also be extended to software. In a program annotated with assertions, clauses to be checked can be generated and piped as input into a solver such as SMT (satisfiability modulo theories)[8], which extends SAT by lifting the constraint of Boolean logic and replacing it with a first order theory.

Modern system-on-chip (SoC) designs are at the core of computing devices that contain many sensitive assets, including private data, photos, and financial information. SoC cores contain both hardware and firmware components, increasing flexibility, decreasing development time, and allowing for in-field updates. With the advent and proliferation of SoCs, where interactions between hardware and firmware are essential to system functionality, it becomes increasingly important to develop methods and tools for reliable hardware-firmware coverification.SoCs are complex systems, and formal verification of the hardware alone poses a huge challenge. The difficulty of verification has only increased by the addition of hardware.

1.2 Problem Statement

Computing systems are not composed of software and hardware working in isolation, especially with contemporary integrated circuits that combine embedded processors, memory, and other computational blocks on the same chip[9]. Verifying the functionality and safety

of these SoCs is challenging because complete formal verification of hardware and firmware together does not scale well, while separate verification misses bugs due to the separation of hardware and software during analysis that introduces discontinuities between the models being checked and the actual implementations. However, there are few open source SoC designs with realistic the hardware, firmware and associated security requirements. To address this problem and contribute towards the goal of combined hardware-firmware analysis, the first step is to build a realistic testing infrastructure that can demonstrate the effectiveness of the models that are in development. Then, taking advantage of the similar approaches used in model checking of hardware and software, we will abstract the hardware of the created system into a software equivalent to do complete analysis. A second challenge in the field of formal methods is that many approaches use temporal logic to encode properties that describe the system state along one possible path; however, to verify security properties, this often is not enough, and new languages have been proposed. By identifying properties of our system that cannot be specified in temporal logic, we will compare and demonstrate the expressiveness of two such specification languages.

The contributions that this thesis makes are the expansion of a previous SoC design consisting of an 8051 microcontroller running firmware and connected to external memory and two cryptographic blocks by adding new Verilog hardware accelerators, in particular an RSA encryption module, along with a memory access module for reading and writing large blocks of data to aid program loading; writing an authenticated boot protocol that demonstrates significant exchange between hardware and firmware; and evaluating the effectiveness of verifying hardware-software properties by hardware abstraction and bounded model checking.

1.3 Overview

The remainder of this thesis is organized as follows: Chapter 2 describes the hardware system, including the RSA accelerator that was implemented, and introduces the interactions between firmware and hardware in the system. Chapter 3 provides details on the authenticated boot protocol program that was developed to ensure that only verified programs can be run. Chapter 4 discusses specific properties of the authenticated boot protocol on this system and the language available to specify these properties. Chapter 5 describes the process of checking these properties using a C abstraction of hardware components and summarizes the results. Chapter 6 presents conclusions and explores limitations and possible directions for future work.

Chapter 2

Design Driver

The system is designed to provide a platform for verifying properties that span the hardware-firmware interface, and accordingly consists of both hardware and firmware components. An 8051 microcontroller running a C program represents the firmware side of interactions. On the hardware side, there are cryptographic accelerators implementing Advanced Encryption Standard (AES)[10] symmetric encryption/decryption, Secure Hash Algorithm 1 (SHA-1)[11], and RSA encryption, as well as external memory (XRAM) and a module to copy data within XRAM. Each of the cryptographic function implementations is put in a wrapper to selectively expose data to the 8051. This reduces the state from the hardware that the firmware sees, enabling more abstraction of the hardware and reducing the size of the problem when checking properties. The 8051 is able to communicate with each of the hardware components through the memory arbiter via memory-mapped I/O, shown in Figure 2.1.

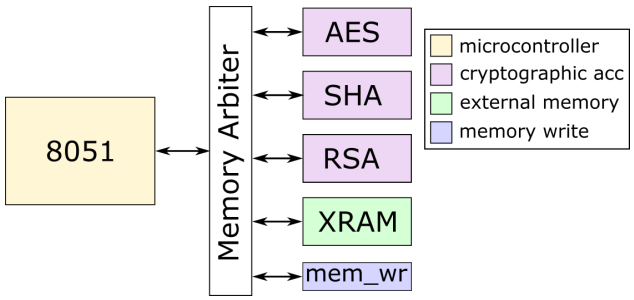

Figure 2.1: Main modules of system

The 8051 communicates with hardware through the memory arbiter. During arbitration, the bus is assigned to the requestor who has the highest priority, and access is not transferred until the transaction completes. The 8051 is the most trusted device in the system and is thus assigned the highest priority so it can make relevant changes before other modules gain

access. Each hardware accelerator takes commands from the 8051 and can read from and write to XRAM, but cannot interact with other hardware components. Table 2.1 summarizes the locations in memory that map to hardware registers, and provides a sample breakdown of the AES module registers.

addr start	addr end	module
0xFA00	0xFD10	RSA
0xFE00	0xFE10	SHA
0xF9F0	0xFA00	mem_wr
0xFF00	0xFF40	AES
0xFF80	0xFFA0	PT_WR
0xFFA0	0xFFC0	PT_RD

(a) memory-mapped locations

variable	address	num bytes
start	0xFF00	1
state	0xFF01	1
addr	0xFF02	2
len	0xFF04	2
keysel	0xFF06	1
ctr	0xFF10	16
key0	0xFF20	16
key1	0xFF10	16

(b) example register layout: AES

Table 2.1: Memory locations mapped to HW

2.1 Hardware

2.1.1 AES

AES used in isolation to directly encrypt a message is not secure enough, so the AES encryption module uses counter (CTR) mode encryption[12] with a 128 bit key. In addition to a key, it also uses a counter that is incremented as a nonce. The use of a counter prevents leaking of information about patterns in the data and strengthens message confidentiality. Encryption splits the input into 128 bit blocks and produces output of the same length. For each block, AES is computed using the key and the current value of the counter as inputs, and this output is XORed with the message block to produce the cipher text. CTR mode is symmetric, so the process for encrypting is identical to the process for decrypting, as shown in Figure 2.2, where E represents AES encryption. We use the AES implementation from OpenCores.org[13].

The AES encryption algorithm is fast, so it is used for encrypting messages, which can be quite long and might have constraints on latency or throughput.

AES is the most minimal in terms of integration with firmware, because it only requires the 8051 to provide basic information about the encryption by initializing registers. For each encryption, the 8051 can set a key, counter state, the length of the the read and write addresses in XRAM, and a start signal.

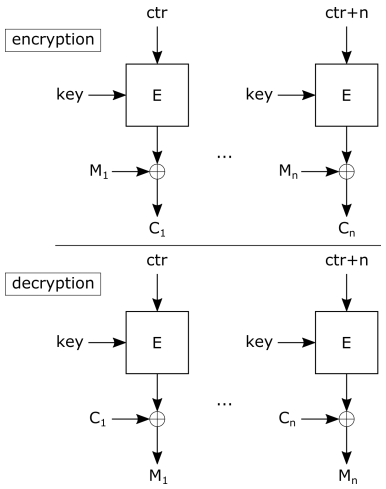

Figure 2.2: Counter (CTR) mode encryption and decryption is symmetric

2.1.2 SHA

In addition to the AES accelerator, we also have a SHA-1 hashing module[14]. Hashing is one-way, so it is useful for proving the integrity of data without revealing the contents. It is also used in RSA for compressing the message length for inputs to encryption and generating pseudo-randomness.

The SHA-1 cryptographic hashing function takes any length of input and iteratively hashes it in blocks of 64 bytes. The 20 byte hashing result of each block is combined with the next input block to produce the final 20 byte hash that is written into XRAM. This contrasts with AES, which encrypts each block separately and then concatenates all ciphertext blocks into one long output.

Before the message is ready to be sent to the SHA hardware accelerator, it first must be padded by the 8051. Padding for SHA involves a 1 bit immediately following the original message and ends with the length of the message, expressed with 4 bytes in little endian. Zeros are added in between these two markers to pad the total length to a multiple of 64 bytes. See Appendix A.1 for a sample test program. Because the information about the length is embedded into the message instead of simply stored in a register, from the viewpoint of the 8051 the preparation is more involved compared to AES.

2.1.3 RSA

The last cryptographic hardware module is RSA. RSA can be used to securely transfer the key used in AES, or verify message integrity through a signature signed with a private key and verified with the corresponding public key. It has a 2048 bit key, so it takes more resources to break than AES, but also takes much longer to encrypt, so it is not suited for encrypting

many frequent messages. The main computation of RSA is modular exponentiation, and RSA effectively has two keys: the exponent and the modulus. Among a shared private and public key pair, the modulus is the same, and the private key and public key exponents are related by

$$(m^e \bmod n)^{e'} \bmod n = m, \tag{2.1}$$

and symmetrically,

$$(m^{e'} \bmod n)^e \bmod n = m, \tag{2.2}$$

where n is a 2048 bit modulus, m is a number smaller than n, e is the public key, and e' is the private key. The keys, input, and output are all 2048 bits (256 bytes). More details of RSA are in section 2.3.1.

2.1.4 Other Additions

In order to protect the confidentiality and integrity of data, a page table in front of the memory arbiter can block reads and writes to locked pages. Each page is 256 bytes and can be locked to reading or writing independently. While the 8051 is in supervisor mode, it can set the bits of pt_wren or pt_rden high or low to unlock or lock each page. All pages start locked to both reading and writing, and the pages corresponding to the page table are always unlocked to the 8051 in supervisor mode to prevent becoming locked out of the page table registers, and these registers are always locked in user mode because the user does not have locking privileges. The supervisor mode and page table functionalities were part of a separate project added in parallel to this senior thesis research.

To facilitate copying data from one location in XRAM to another, a memory write module was also added. It operates similarly to the DMA of the cryptographic accelerators, except that it does no computation on the data in between reading and writing. The registers and options for the mem_wr hardware module are explained in section 2.3.2.

2.2 8051 microcontroller

The 8051 microcontroller, used for embedded systems applications, is a system on chip consisting of an 8-bit CPU, 64KBof read-only program memory, 128 bytes of RAM, 64KB of external memory, 4 I/O ports, a serial port, and timers[15]. To simulate we use a Verilog implementation of the 8051 from OpenCores.org[16].

2.2.1 Running firmware on the 8051

The 8051 runs a C program that can direct the behavior of the hardware accelerators. All programs to run on the 8051 are integrated into the hardware system by compiling into Intel Hex Format using Small Device C Compiler (SDCC)[17], and then running a script to parse the hex file into bytes in the program ROM. The 8051 has three types of memory: on chip RAM, on chip ROM, and external RAM, and SDCC has the functionality of specifying where in memory to store variables, illustrated in Figure 2.3. The __code keyword puts a variable into the program ROM. We use this in the secure boot to store the public key hash. The __xdata keyword indicates that a variable is to be stored in external RAM, and the __at(addr) keyword is able to specify which address in memory the variable is placed in. In the hardware modules, registers are used to store information such as state, keys, message length, and message data or addresses. Using __xdata and __at(), we can create variables that correspond to the locations of registers inside the modules and read and write their values. Examples of registers to write include the read/write addresses for accessing memory, length of input, or in the case of encryption, the key being used. Once the register values are initialized, writing 1 to the modules start register will start computation, and the state register is monitored to determine when the module has finished and returned to the idle state.

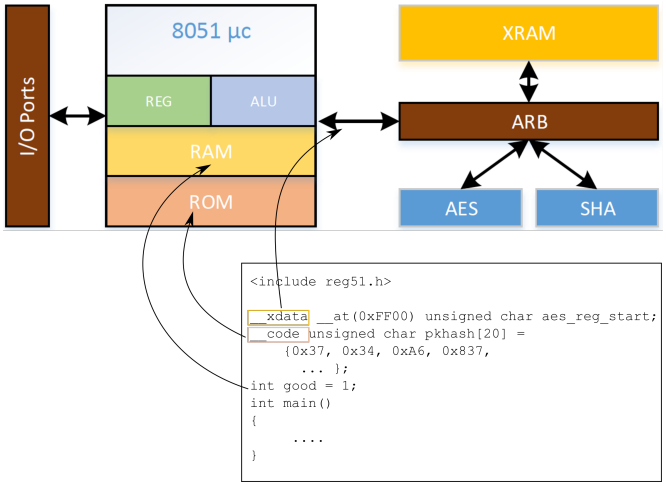

Figure 2.3: Types of Memory in 8051: no prefix puts variables in RAM, __xdata passes it out to the arbiter, and __code puts variables in ROM.

When the 8051 reads or writes the locations mapped to memory-mapped registers, the command is intercepted by the hardware modules. However, to constrain the problem, reads and writes initiated by hardware are directed to XRAM for all addresses, limiting interaction between hardware accelerators. This results in the XRAM memory locations

that are memory-mapped becoming accessible only to the hardware, while the hardware registers are only accessible to the hardware module that owns the register and the 8051.

2.2.2 Preliminary Testing and the HW/FW Boundary

In concert with the effort to create a system spanning the hardware-firmware boundary, testing is performed on a C program running on the 8051 that directs the execution of the hardware modules being tested. To test for accuracy, the testing program also includes a randomly generated input and its corresponding reference output loaded into XRAM, and when computation has completed, the program checks against these values. Before the exit sequence, the P0 line is set high if the output is correct, and low if a mismatch occurred.

The testing procedure consisted of first testing on one block of inputs to test the basic functionality of the encryption and hashing, and then increasing to multiple blocks to test the state transition logic when more than one read, operate, write cycle is needed. Inputs were randomly generated with a python script, encrypted or hashed, and then checked for accuracy. The python libraries used were AES from Crypto.Cipher and hashlib. For AES, the message was encrypted and decrypted, and both sides were checked.

To run the test code, use SDCC to compile, assemble, and link it into a .ihx Intel hex format file[18]. This file is parsed to generate a Verilog file that inserts the program into ROM; changing the file included in ROM will change the program being run. The final Verilog simulation is done using Icarus Verilog.

In the process of developing, testing, and evaluating new specification languages and methods for use in model checking, a suitable testing platform is imperative. Integration of hardware design with firmware in this infrastructure provides such a testing platform for testing and evaluating the performance of models capturing both hardware and software. The test programs illustrate varying degrees and flavors of hardware-firmware boundary crossings. As mentioned previously, in AES, the 8051 only needs to initialize register values, set the start bit high, and wait for a result, but the interaction between modules becomes increasingly complex. SHA requires slight padding modifications before hashing, but it is a simple operation and can still be split completely into software setup and hardware computation. In contrast, RSA encryption involves multiple executions of SHA, with additional software processing in between each call, before the final RSA modular exponentiation call. This means that hardware calls are interleaved throughout the program execution, not just memory access and one final call like in AES. Although the memory write module is very simple, because it is not doing any computation, unlike the cryptographic accelerators, we can execute code that does not need to wait for the result of the copy while it runs, making

it easier to invoke the concurrency of hardware and software, an aspect of the hardware-firmware boundary that is not touched upon with the other hardware modules. The secure boot integrates RSA functionality with memory loading and locks, resulting in the most interactive exchange.

2.2.3 Fixes and Modifications

A slight modification was made to AES in order to make use of its two key registers. This involved adding a key select register, and adding the logic to map it to address 0xFF06 in the AES address space.

As a result of testing, we found a bug in the SHA module: For data longer than one 64 byte block, after reading all blocks, the incomplete hash was written to XRAM and state transitioned to idle while the complete hash was still being computed. To prevent confusing a hash that had not started computing yet for a finished hash, one extra wait cycle was added to separate initializing the SHA core module and waiting for a ready signal. Another minor issue was endianness: while numbers are stored in little endian, messages frequently represent strings and should be stored in big endian.

2.3 Implementations

2.3.1 RSA Module

The RSA hardware module performs just the modular exponentiation step of RSA encryption, while padding is performed in the firmware. While other parts of the system were wrappers enclosing open source core modules, there were no complete and working Verilog implementations of modular exponentiation, so we implemented it from scratch following traditional hardware implementation of modular multiplication and exponentiation[19]. For modular exponentiation, we used the Left-to-Right (LR) Binary Method to compute $c = m^e \bmod n$. The LR binary method scans the bits of the exponent from left to right (most significant to least significant). c starts at 1, and for each bit in e, c is squared mod n, and if $e_i = 1$, this squared value of c is also multiplied by m mod n. Compared to the Right-to-Left (RL) binary method, the LR binary method uses one less 256 byte register, because the RL binary method needs an extra register to store multiples of m.

The way modular multiplication is implemented, all multiplications take the same number of clock cycles, even multiplication by 1, so in order to reduce the number of clock cycles needed for modular exponentiation, no multiplications are performed until the first nonzero exponent bit is found, coinciding with the first instance of multiplicands greater than 1,

because before that $c \times c$ is just 1×1. This significantly reduces the number of clock cycles needed for modular exponentiation, especially because the public key is usually short, and the long private key can be computed offline, when time is not as much of a constraint. The simplified FSM showing relevant signals for modular exponentiation is shown in Figure 2.4a. Note that the square and multiply states represent modular multiplication, which takes more than one cycle and has its own FSM, not shown. During these modular multiplication states, the modular exponentiation module stays in the state until it receives a done signal from the multiplication module indicating that the multiplication has completed.

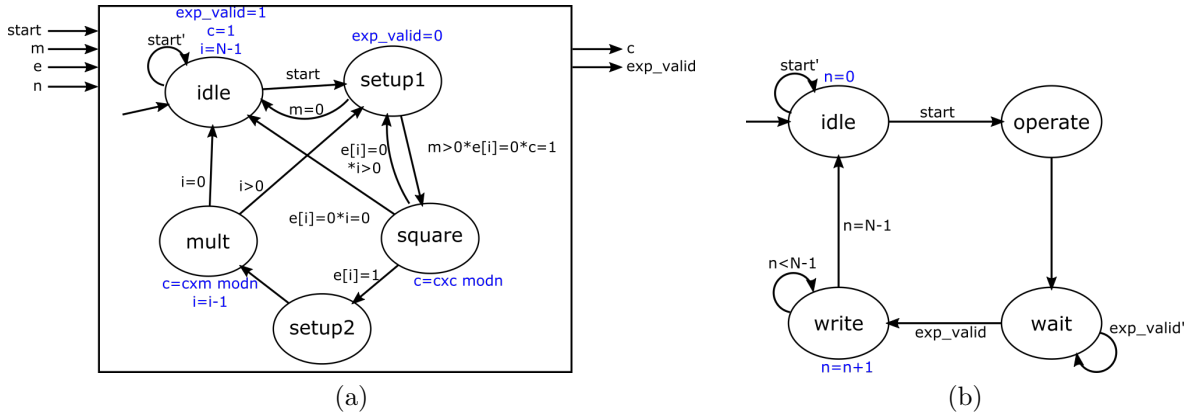

Figure 2.4: FSMs for (a) modular exponentiation and (b) top module of RSA hardware module

Because the input lengths are fixed, the message, exponent, and modulus that RSA uses for modular exponentiation can be stored in registers within the wrapper module for RSA, and it does not need a read cycle to obtain these values from XRAM during the encryption run as AES and SHA do. The FSM states, illustrated in Figure 2.4b, are idle, operate - the start of computation, wait - waiting for modular exponentiation to finish, and write data - copying the result from an internal register not accessible by the 8051 into XRAM at the write address.

The software side of RSA is the padding that must be performed on the raw message before it can be encrypted, for added security. The C code for this is based on a COS 432 assignment implementing RSA in Java. The Optimal Asymmetric Encryption Padding (OAEP)[20] introduces randomness into RSA by appending a random k_2-bit string r that is hashed and XORed with a message padded with k_1 zeros. The resulting string s is then run through another hashing function that is XORed with r to produce t, which is appended to s complete the message. This entire process is shown in Figure 2.5. OAEP increases security because it adds a random r to the otherwise deterministic procedure, and both s and t are needed to decrypt each other.

To compute the hashing functions in OAEP, we use a keyed-hash message authentication

code (HMAC)[21] that takes a key performs two rounds of SHA-1 to produce the final hash. To generate a string of sufficient length to XOR with the message in G, we use a pseudo-random generator (PRG) that generates the next output by hashing its current state with a constant random string. To store the state of the PRG, we add a new array in XRAM, gprg.

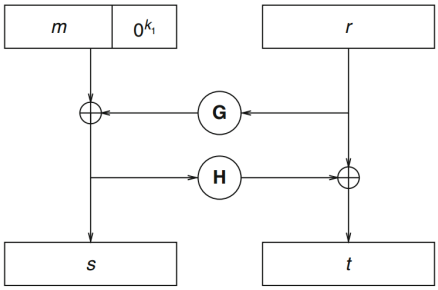

Figure 2.5: Optimal Asymmetric Encryption Padding[22]

In addition to encrypting and decrypting, the RSA code can also do signing, which hashes the input and encrypts it with a private key, and signature verification, which takes a signature, decrypts it with a public key, and compares it to the hash of the original input.

2.3.2 Memory Write Module

The mem_wr module takes a read address, write address, a length, and the mode. The length specifies how many bytes to write The buffer inside the mem_wr module is BUFF_SIZE = 0x2000 bytes, so this is the maximum length. As shown in Figure 2.6, the mem_wr module first reads all min(length, BUFF_SIZE) bytes into the buffer, and then transitions to the write state to write the buffer back into XRAM. Changing mem_wr to cycle through the read and write phases as long as the length has not been reached, as is done in the AES wrapper, is possible, but it would come at the cost of losing the functionality of copying between two memory ranges that overlap when the read address is larger than the write address.

There are two modes of operation for the mem_wr module. The first, when mode = 0, is copy mode, so the read and write operations are both executed. The second, when mode = 1, is write-only mode, so the read step is skipped and the data stored in the buffer is written to the write address, ignoring the read address. The mode bit is stored in the second least significant bit of the start byte, while the least significant bit is the start bit, so setting start to 0x1 will copy data, and setting start to 0x3 will only write data without reading first. Although the 8051 writes the start byte of hardware accelerators like any other hardware register, the start byte is not actually stored; rather, if the state is idle and the address points to the start register location, the hardware module will check the LSB of the input

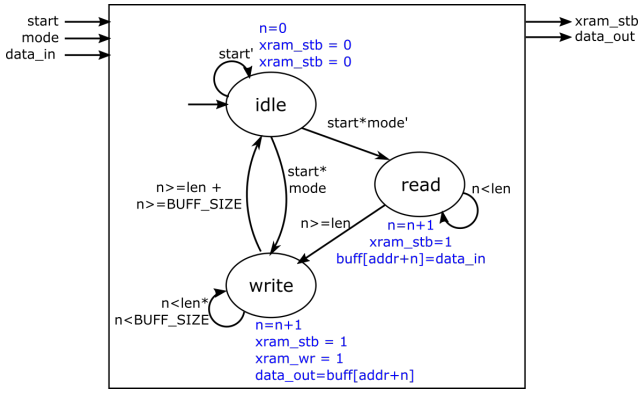

Figure 2.6: simplified mem_wr FSM with main signals

data, and if it is high, it will transition out of the idle state. Because of the absence of a physical register, the skip read bit must be written at the same time as the start bit to take effect; writing the skip read bit first, and then the start bit will still execute in copy mode.

The skip read mode is used in secure boot to write the image from the mem_wr buffer into XRAM. It can also be used to temporarily store data in the buffer until it needs to be written somewhere else, but only if it is guaranteed that another mem_wr operation will not be called. Because the mem_wr module always starts writing to the buffer at index 0, successive calls to mem_wr will overwrite the data from the last call unless the length is 0, which is equivalent to no call.

In order to create a platform where hardware-firmware interaction is prevalent and security properties can be assessed, we connect three cryptographic hardware modules AES, SHA, and RSA to an 8051 microcontroller, in addition to a page table for regulating reads and writes, and a memory write module to facilitate more automated data transfer. The interaction between the 8051 firmware and the hardware is conducted by the memory arbiter sitting at the interface between the two sides. The two main new implementations are an RSA implementation that invokes much interaction between firmware and hardware, and a memory write module that increases the potential for parallel execution.

Chapter 3

Authenticated Boot Protocol

A secure bootloader[23] provides a means of detecting attempts to run malicious code, only loading code that has been verified as safe. Since the program ROM is read only, it is a trusted source; therefore, we place our secure bootloader in ROM, and use it to verify any other outside programs being loaded. An image of the program to be loaded is stored in the mem_wr module's buffer, and at the start of the secure boot execution it is written into XRAM. The format and location of the data relevant to secure boot are shown in Figure 3.1 For the authenticated boot protocol to be successful, it must only accept a program if 1) the program provided is safe, determined by a verification process described below in section 3.2, and 2) if there is no interference during the verification process. If the first condition fails, then the program cannot be trusted and should not be loaded. Even if the first condition passes, if the second condition fails, it is possible that outside manipulation led to an untrusted program to pass verification, or that the program that was loaded does not match the trusted program initially presented for verification. The code for the original secureboot is in Appendix B

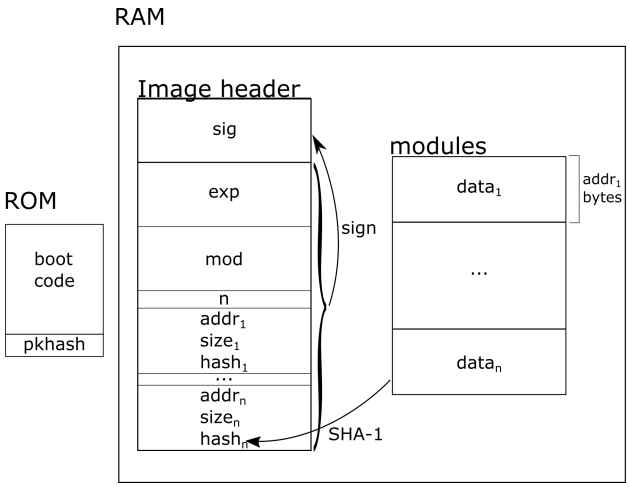

Figure 3.1: Secure boot data layout

3.1 Image Format

The image of the program to load comes in components of three types: a single header block, multiple module blocks, and a signature.

The header includes a public key (RSA exponent and modulus), the total number of modules that the program has been split into, and module data that provides information about each module. Inside ROM, we also insert the public key hash to check the public key provided by the image in the header. For each module, the header specifies at which address to start loading the module, how many bytes it contains, and the hash of the module. The address specification allows spreading out modules in memory; however, the whole program must still fit within the address range allocated to it in secure boot.

The header block is signed with the private key that corresponds to the public key in the header. Because the header size is a function of the number of modules and is not constant, the signature is placed at the start of the image so that the start location of both the signature and the header can be fixed.

The modules following the header contain the code to be loaded. There are no markers in the module blocks indicating boundaries between modules; modules are inserted sequentially, with no spaces in between each other, to be split and arranged as designated by the sizes and addresses in the header.

Secure boot uses five arrays in addition to those required by RSA encryption(Table 3.1). The boot array reserves space in XRAM for the image to be loaded, and the program array reserves space for the modules to be loaded. In order to have more control over the data during encryption and hashing, we reserve memory for sha_in, sha_out, and rsa_out arrays in XRAM and require that the SHA read, SHA write, and RSA write addresses always

correspond to these locations when running. This makes it simpler to check if array bounds have been violated, and it decreases the chances of an attacker to manipulate accelerator inputs or outputs by providing incorrect addresses.

variable	address	num bytes
program	0x0000	0x3000
boot	0x3000	0x2000
sha_in	0xC000	0x2040
sha_out	0xE100	20
rsa_ out	0xE200	256

Table 3.1: Memory locations of secure boot arrays

To provide a source of sample images for testing, we created a python script bootgen.py similar to the parser that converts the .ihx file from sdcc into a Verilog file for the ROM, but extended it to perform hashing, signing, and image formatting. The script takes a .ihx file as input and produces the image with filename prog.hex as output. During testing, we experimented with different combinations of module sizes and load address locations. The current implementation breaks programs into modules of 256 bytes and sets the address of each module such that the program is contiguous when loaded into memory at the end of secure boot execution.

3.2 Verification Process

To verify the firmware, the authenticated boot protocol transitions through 5 main stages after setup: image load, key verification, header verification, module loading, and module verification(Fig. 3.2). During verification, a status variable pass is set to undet, and if a check fails during any step of verification, then pass is set to FAIL and execution terminates. At the end of verification, if nothing has failed, pass is set to PASS.

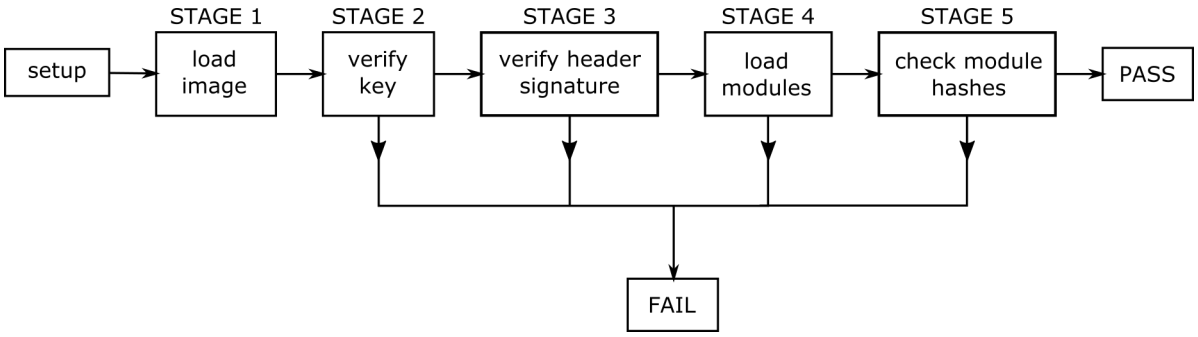

Figure 3.2: Secure boot verification process

We initialize the mem_wr module to hold the contents of the image provided in prog.hex using the Verilog function readmemh. The buffer in mem_wr is BUFF_SIZE = 0x2000 long, so this is the maximum image size that will pass verification; otherwise, the image is truncated, and module verification, possibly even header verification, will fail. The mem_wr buffer is not visible to the 8051 firmware, so during stage 1, the image held in the buffer is loaded into XRAM by invoking mem_wr with the skip read bit high.

Once the image is loaded into memory, the public key provided in the image is verified. First the exponent and modulus provided by the image are stored in the RSA hardware accelerator registers, and then both the exponent and the modulus are hashed together as one 512 byte string using the SHA-1 accelerator. Finally, the resulting hash is compared to the public key hash stored in ROM along with the rest of the secure boot code, ending stage 2.

Next, we use the confirmed key to verify that the header signature is correct; this is stage 3. The size of the header can be computed from the number of modules, which is always at a fixed offset from the start address of the image. If the header is too large to fit within BUFF_SIZE, then verification fails. Otherwise, the header, header signature, and header size are passed to verifySignature() to check for validity.

In stage 4, the bootloader loops through every module to copy its program data to the desired location using mem_wr. Before copying, the address and size are checked to ensure that the module does not extend beyond the memory addresses reserved for the program; if this check fails, verification fails and secure boot stops execution. The maximum program size is MAX_PRG_SIZE = 0x3000. Even though all of the data that is loaded into the program memory is contained in the image, because the program can be spread out with gaps in between modules, it is given a higher size limit than the image, which is compact. The address provided in the header is treated as an offset from the start of the program array where the firmware is being loaded.

After the firmware has been copied from boot to program, the newly written data is hashed with the SHA-1 accelerator and checked against the module hash contained in the header during stage 5, the final stage. This verifies both that the code is trustworthy and that the mem_wr copy succeeded. Because the load stage already checked the bounds of each module and exits on failure, we do not need to check the module address and size again. As long as the verification is successful and there are more modules to load, we continue verifying the next module. Like the previous stage, if verification fails at any time, the secure boot fails and terminates.

If all stages complete without failure, then the secure boot passes. At this time, upon passing secure boot also terminates, but it is possible to add a jump to the verified code to

start execution.

3.3 Locking Memory

The checks performed during the verification process are useful only if we are guaranteed that the data does not change once it has been verified; this is why the page table that locks memory is necessary for security. Because the data being managed is provided in the image and not secret, we do not worry about reads and unlock all pages to reading at the start of secure boot execution.

During secure boot verification, locks essentially follow every load. Once the image is loaded into memory, the whole boot array is locked to writes to prevent tampering by outside sources. After the RSA exponent and modulus are copied into the RSA accelerator registers, these two pages are also locked. When copying the program code from the boot array to the program array, the whole program array is unlocked, all modules are copied, and then the whole program array is locked.

In addition to calling the mem_wr hardware module to write data, secure boot also calls the SHA and RSA accelerators. These calls use a similar setup as in the programs used to test SHA and RSA against reference outputs from section 2.2.2 , but now we are concerned about not just functionality, but security as well, so the setup needs to be bolstered by lock and unlock calls to the page table. To make sure we are hashing to and from the right addresses, the read and write address registers of SHA need to be locked. In order to efficiently check if the addresses have been changed or not, we set the read and write addresses to sha_in and sha_out respectively during the setup phase of secure boot, and lock these registers so they cannot change. Since these address registers are on the same page as the start register, we cant keep them locked the entire time, but we can check after SHA completes to see if the address was changed during the gap between unlocking the page and writing to start, because registers cannot be changed by outside signals while running. Locks to sha_in before SHA executes, and to sha_out after SHA executes are also needed.

For RSA, similarly to SHA, the output address should be locked after it is set to rsa_out, and after a call returns rsa_out should be locked, but since the hardware accelerators do not accept write commands while the state is not idle, we do not have to worry about the RSA message register changing while RSA computes. However, we still need to lock it during RSA message setup when we pause writing to compute with PRG, and gprg that holds the state of the PRG needs to be locked when it is not in use.

The secure boot enables the system to verify firmware that is loaded from an outside source to ensure that the source is trusted. To that end, in addition to the code to execute,

the program image also contains additional information in a header file, including an RSA signature and multiple hashes. The verification process involves checking the RSA public key, the header signature, and the hash of each of these modules. In between loading data and verifying it against a hash or signature, the data is locked to ensure that the verified data is indeed the final data being used, either as a key for encryption, for providing more information about the program modules, or as the loaded program itself.

Chapter 4

System Properties

4.1 Property Types

The main type of security property that will be focused on is propagation properties. Propagation properties require that information flow cannot occur between certain sources and destinations when certain conditions hold, meaning that the data at the source should not affect the data at the destination. Propagation properties satisfy the confidentiality and integrity requirements of security. This is a safety property because a system must never go into a state where information is leaked. An example of a propagation property in hardware is that program memory is locked after loading verified code, as in the case of a system protected by a secure bootloader. The propagation property here states that an untrusted entity is unable to modify the program memory after the program is loaded and the memory is locked. An example that spans the hardware/firmware boundary is that a hardware key cannot be accessed by the firmware while it executes in user mode.

A second type of property is the liveness property of request/response, which specifies that whenever a request is made, it must have a response. This can be applied to hardware functions by specifying that when the request to start a module occurs, it will eventually finish, signified by returning to the idle state. The guarantee that a firmware request for XRAM access will be granted by the memory arbiter is a request/response property that involves both hardware and software.

4.2 Specifying Properties

Model checking has historically been performed to verify properties specified in temporal logics such as CTL[24] and LTL[25], which add temporal modalities over propositional for-

mulas to describe properties of a path through sequential states. This allows us to express behavior over time. The boolean connectives that we use are $\neg$ for NOT, $\wedge$ for AND, $\vee$ for OR, and $p \rightarrow q$ for p implies q, or equivalently $\neg p \vee q$. Some temporal operators in CTL and LTL are Xp, which indicates that p is true in the next state; Fp - p is true eventually; and Gp - p is always true. CTL also has quantifiers Ap - for all paths, p holds; and Ep - there exists a path where p holds; but in LTL all properties implicitly include the A quantifier over a formula that must be quantifier free. For example, the LTL formula

$$G(l \rightarrow \neg w)$$

means that for all paths, in every state l implies NOTw.

In security, it is often important to express information-flow or propagation properties, which cannot be specified in temporal logic[26]. To check propagation properties requires changing the data held in the source and checking if this change produces effects that can be seen in the destination. However, by looking at only one path at a time, as temporal logic does, it is impossible to determine if the destination is being affected by the source. Evaluating information-flow properties inherently involves comparisons between traces, which cannot be done without a method of identifying multiple traces within a property. Instead of properties restricted to a single trace, these hyperproperties are properties over sets of traces[27]. One prime example of a hyperproperty is noninterference, which states that the behavior observed by a low-security user is not affected by the removal of the high-security inputs. Without being able to compare or refer to the two corresponding traces with and without high-security input, it is impossible to evaluate or specify this property.

New extensions of standard temporal logics that are able to specify hyperproperties are hyperCTL* and hyperLTL[28]. By explicitly quantifying over path variables, hyperCTL* and hyperLTL can describe properties that specify behavior over more than one trace. HyperLTL formulas, analagous to LTL formulas, only use the A quantifier, which is required to be applied over the whole quantifier free formula. By using multiple quantifiers over different paths, hyperLTL can label each path and describe the relationship between paths. The $\forall \pi_1$ translates to "for all paths π_1". An example of a hyperLTL property is

$$\forall \pi_1 \forall \pi_2 F(a_{\pi_1} = a_{\pi_2})$$

states that for all paths π_1 and all paths π_2, eventually the value of a in π_1 and the value of a in π_2 are equal.

A specification language tailored to describing the propagation properties introduced in section 4.1 designates a source (src) and a destination (dst), where information cannot

propagate from the source to the destination. The conditions under which this property holds are provided in srcpred and dstpred, which are evaluated when src is being read and when dst is being written, respectively. If srcpred and dstpred both hold, then there can be no information flow between src and dst, so the value being written to dst cannot be influenced by the data held in src. Other predicates that may be useful to describe conditions are globalpred, under which there is no propogation when it is true for the lifetime of the program, and finalpred, which is evaluated upon termination of the program.

4.2.1 Verifying Hyperproperties in Software

There exist methods for model checking of HyperLTL and HyperCTL* for circuits[29], and to complement these, we have developed a method to check properties in software. First the input program in C is compiled by clang, a compiler used as a frontend to LLVM[30], to create a bitcode file. LLVM is a compiler framework that can perform transformations on programs. The majority of the work is done by our LLVM pass that takes an input program and a hyperproperty specification file, and outputs new program bitcode with assertions. This bitcode is analyzed with the SeaHorn[31] software verification tool, which outputs SAT if a counterexample that violates the property is found and unSAT if the program satisfies the property.

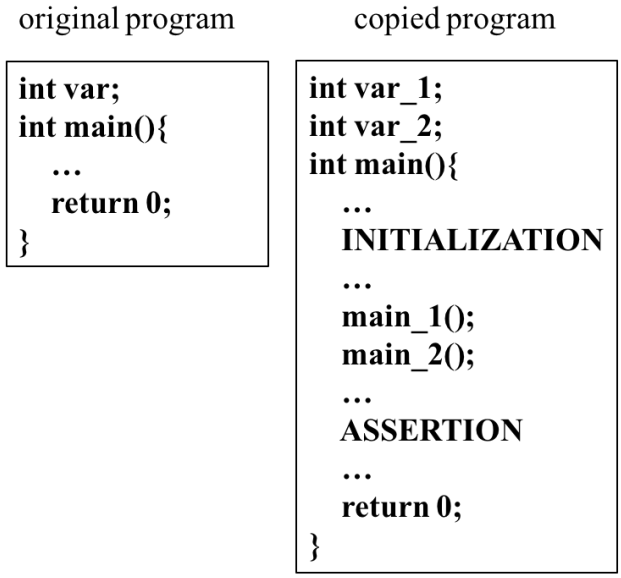

Figure 4.1: Simplified program transformation for hyperproperties

For our analysis, we chose to model the states of the program as the value of variables. The observable states are the initial state and final state, so these are the states that appear

in our assertions. We define classes of variables that can be parsed by the LLVM pass, so it is possible to assign the value of input variables to the initial state and the value of output variables to the final state.

The LLVM Pass works on the entire LLVM program Module, copying global variables and functions to model the multiple paths referred to by the hyperproperty. During program transformation, the program is essentially duplicated, run twice along two paths, and compared for property violation via the inserted assertions, as shown in Figure 4.1 . The assertions are generated by parsing the hyperproperty specification and storing the assertion condition corresponding to that property.

To make the duplication and checking more efficient, before copying functions we check if they call any other functions or access global variables. If neither occurs within a function, it does not need to be copied. Additionally, by inlining functions and creating a product of the two paths, closer to running the two paths in parallel rather than in series, the number of Horn Clauses that SeaHorn needs to check decreases, improving performance. Segments are identified by checking for conditional branches and loops, and interpolating on dominating nodes puts similar segments of code closer together.

4.3 Properties for Secure Boot

The properties of the secure bootloader are roughly divided into five categories: array access locations, memory locking, program flow, hyperproperties, and correctness. The properties on array access locations restrict the addresses in memory that can be written to. Memory locking specifies the behavior of memory addresses that have been locked, and under which circumstances which memory locations should be locked. Program flow properties refer to the execution path through the program. Correctness properties give guarantees on parts of the image for each stage of secure boot. For each category of properties, we provide explanations of each property, the LTL or hyperLTL formula, and if applicable, the src, dst, srcpred, and dstpred for information flow.

4.3.1 Memory Access Locations

The main property on memory access restrictions checks that an array access does not go out of bounds and reveal or corrupt other arrays, or access invalid addresses. Abiding by array bounds is a standard property that any application will usually need, not just secure boot, and model checkers such as CBMC and VARVEL can automatically check for array

bounds violations in a program[32]. The LTL property that describes legal array accesses is

$$G(a \rightarrow (ptr + i < \text{ptrHi}(ptr) \wedge ptr + i \geq \text{ptrLo}(ptr))), \tag{4.1}$$

where a represents an array access or assignment to $ptr[i]$, and the range of addresses associated with ptr is [ptrLo, ptrHi). For each array access to a base memory location ptr plus some index offset i, the final memory address must be within the range of legal values associated with the array that ptr points to. For secure boot, our arrays are program; boot; sha_in; sha_out; rsa_out; RSA key and message registers exp_reg_m, exp_reg_exp, exp_reg_n; and gprg. Because all arrays are statically declared, it is straightforward to determine ptrLo and ptrHi.

Another property that follows property 4.1 is that RSA keys should only change during the public key check and load phase:

$$G((w \wedge addr = \text{RSAkeyrange}) \rightarrow stage = 2), \tag{4.2}$$

where w is the write signal, $addr$ is the write address, RSAkeyrange is the range of memory addresses spanning [exp_reg_n,ptrHi(exp_reg_n)) and [exp_reg_exp,ptrHi(exp_reg_exp)), and stage = 2 means that we are in the range of PC that corresponds to checking the RSA key hash and loading it into the RSA HW accelerator. If the first property (4.1) holds, then this second property (4.2) should always hold, because the only time we try to write to the RSA key registers is when loading the RSA keys, and if it is violated, then this means that a write to another array crossed either ptrHi or ptrLo. It provides a counterexample to the first property and demonstrates why it is important to stay within array bounds. The property only requires that writes to the RSA keys cannot occur outside of stage 2. Because the keys stored in the HW registers are public keys, reading the keys is not a problem, and the property is strong enough as is.

4.3.2 Memory Locks

In this section, locks on memory addresses refer to write locks only; reads are unregulated. If a memory location has been locked by the secure boot, then the values held in those memory locations cannot change, so

$$G(l \rightarrow \neg w), \tag{4.3}$$

where l indicates that memory address M[i] is locked, and w is an event that describes a successful write to M[i]. Locking also means that regardless of what instructions an untrusted attacker module sends, the data inside of M[i] is not affected, so this property can also be

written using our src, dst notation with

$$\begin{aligned} &\text{src: untrusted module} \\ &\text{dst: M[i]} \\ &\text{dstpred: M[i] is locked.} \end{aligned} \tag{4.4}$$

When are which memory locations locked? As described in section 3.3, the input to SHA must be locked while the module is running to ensure that the data being hashed is not changed while SHA is computing:

$$G(state \neq 0 \rightarrow l). \tag{4.5}$$

In this property, *state* refers to the state register of the SHA accelerator, and l is a lock on sha_in to ptrHi(sha_in) Another example lock property to specify that the image must be locked during verification is covered by the formula

$$G((stage > 1 \wedge pass = undet) \rightarrow l), \tag{4.6}$$

where $stage > 1$ means that the image load stage has completed, and l is a lock on boot to ptrHi(boot). $pass = undet$ specifies that while the firmware is undergoing verification, the property holds.

4.3.3 Program Flow

The two properties that fall under program flow are one request/response property on the hardware modules, and one property on the order of program execution. To ensure that the firmware is not stalled waiting for hardware to execute, we add the property

$$G(start_m = 1 \rightarrow XF(state_m = 0)) \tag{4.7}$$

where $start_m$ is a start signal to either the RSA, SHA, or mem_wr modules, $state_m$ is the state of that module, and $state = 0$ indicates that the module idle. This requires that any module that receives a start signal will eventually finish computation and return to the idle state.

The order constraint on program flow checks that all steps of the secure boot are completed, and in the correct order. This is necessary to prevent skipping verification steps, or verifying data with keys that have not been locked or verified yet. This property can be broken down into multiple smaller properties for easier model checking. In the following group

of properties, p indicates that stage=1, q stage=2, r stage=3, s stage=4, and t stage=5. These stages correspond to the stages of secure boot from section 3.2

$$G(p \vee q \vee r \vee s \vee t) \tag{4.8}$$

says that at any point, the program must be in one of the five defined stages.

$$F(pass = F) \vee F(p \rightarrow Xq), \tag{4.9}$$

$$\vdots$$

$$F(pass = F) \vee F(s \rightarrow Xt), \tag{4.10}$$

and

$$G(p \rightarrow X(p \vee q)), \tag{4.11}$$

$$\vdots$$

$$G(r \rightarrow X(s \vee t)) \tag{4.12}$$

$$G(t \rightarrow X(t)) \tag{4.13}$$

together say that each stage can last for multiple states, but it can only transition to the next stage, and it must transition eventually unless verification fails.

4.3.4 Hyperproperties

When secure boot passes, then the function calls and type of variables on the stack should be the same regardless of the contents of the image, as long as the number of modules is the same, so the data inside each image excluding the number of modules should have no influence on the stack pointer unless the image is invalid and verification fails.

$$\forall\pi_1\forall\pi_2(num_{\pi_1} = num_{\pi_2} \wedge F(pass_{\pi_1} \neq F) \wedge F(pass_{\pi_2} \neq F)) \rightarrow G(SP_{\pi_1} = SP_{\pi_2}) \tag{4.14}$$

$$\begin{aligned} &\text{src: entire image} \\ &\text{dst: SP} \\ &\text{srcpred: } num_{\pi_1} = num_{\pi_2} \\ &\text{finalpred: pass=P} \end{aligned} \tag{4.15}$$

A hyperproperty that follows from the array bounds property (4.1) is that there should be no propagation from the image to outside of the whitelist, which includes all hardware registers, the program array, sha_in, sha_out, rsa_out, and gprg. This is because the secure boot should never be writing outside of the image and the whitelist, so regions outside of the whitelist are affected by other processes only. This property only needs src and dst:

$$\begin{aligned} &\text{src: image} \\ &\text{dst: range of addresses not in whitelist} \end{aligned} \tag{4.16}$$

Specifying src and dst along with srcpred and dstpred is an intuitive, user-friendly way to describe properties that might get complicated in LTL or hyperLTL. For example, many of the lock properties can be expressed easily with this notation. For example,

$$\begin{aligned} &\text{src: untrusted module} \\ &\text{dst: boot to boot + MAX\_IM\_SIZE} \\ &\text{srcpred: after the image is loaded into boot} \end{aligned} \tag{4.17}$$

locks the boot array to outside processes after the image has been loaded into it.

4.3.5 Correctness of Authenticated Boot

To guarantee that an incorrect image file will never pass the secure boot, we provide a series of properties that closely follow the verification steps outlined in section 3.2. We check the following properties at the end of secure boot execution:

$$\text{hash(img.key)} \neq hash_{key} \rightarrow XF(pass = F) \tag{4.18}$$

$$\text{!verifySignature(img, img.key)} \rightarrow XF(pass = F) \tag{4.19}$$

For each module in the image, index 1 to img.num,

$$\text{hash(module)} \neq \text{module.hash} \rightarrow XF(pass = F). \tag{4.20}$$

img is the image, img.key is the key contained in the image, and hash() is the hash output computed by the SHA accelerator. $hash_{key}$ is the stored copy of the key hash inside ROM. module is the module data, located after the header in the image, and module.hash is the hash of a module, located in the header of the image. These properties state that if any of the RSA public key, image signature, or modules do not pass their checks, then the secure

boot will fail. Contrapositively, if the secure boot passes, then all of the checks must also pass.

Even though there is the potential for circular reasoning because we sign the image with a key contained in that image, if we ignore hashing collisions, then

$$\text{hash}(\text{img.key}) = \text{hash}(\text{true\_key}) \rightarrow \text{img.key} = \text{true\_key} \tag{4.21}$$

By substitution,

$$(\text{verifySignature}(\text{img, img.key}) \wedge \text{img.key} = \text{true\_key}) \rightarrow \text{verifySignature}(\text{img, true\_key}) \tag{4.22}$$

so this verification process should still be secure. Even without embedding the key in the image, we still need to rely on hashing collisions being infrequent.

In this chapter we give the example of information flow properties to provide motivation for specification languages that extend beyond temporal logic. Then, in the framework of our secure boot, we build up a set of properties, starting from restrictions on memory access and locks on memory, describing the control flow of the program, propagation from the image and untrusted modules, and finally arriving at correctness properties that prove the safety of secure boot. Through these properties, we show the intuitive expressiveness of specifying src, dst, and predicates for information flow properties.

Chapter 5

Verification

5.1 Abstraction

In order to verify properties between hardware and software, we abstract the behavior of the hardware using C. There are three versions of the secure boot: the original program that calls on hardware, the direct C implementation that mimics the hardware behavior, and the CBMC version with two levels that abstracts the hardware further to make verification scalable and includes some extra variables helpful for verification. To create an executable, use gcc with the −DC option; for CBMC, use −DCBMC. Figure 5.1 summarizes the structure of the layers of abstraction.

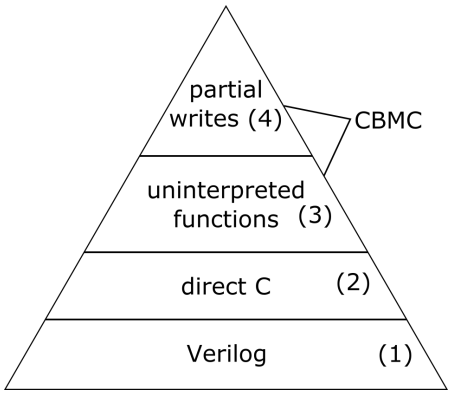

Figure 5.1: Abstraction levels of hardware. (1) behavioral Verilog implementation. (2) replacing all HW accelerators with C. (3) use uninterpreted functions for SHA-1 and modular exponentiation. (4) for array writes, only write to one nondeterministic index

5.1.1 Direct C Transformation

The first version of C code meant to represent hardware has the same functionality as the hardware. In place of the hardware modules, we add function calls to replace the computation performed by hardware. For SHA and RSA modular exponentiation, we use the C crypto library functions, and for mem_wr, we just write the data in software with a loop as we did before the mem_wr module was added to the system.

To model the write permissions of the page table in C, instead of using an array where each bit represents the accessibility of a page and most of the entries were unused, to decrease the amount of memory that the page table data occupies, we use a struct that holds the starting address of a page, the end address, and whether or not the region is locked. This does not mirror the hardware implementation because pages in hardware are 256 bytes with fixed boundaries, while pages in C can be different sizes, but in the hardware we can position each array so that every page has at most one array(Table 2.1 and Table 3.1, and choosing regions in the software that correspond to the ranges that we lock suffices to keep the behavior consistent. For example, Figure 5.2 shows that the RSA accelerator module registers can be split up into three independently locking sections: the message, the exponent and modulus, and the remaining registers. To add an array to the page table, first add it to the memory of the system with mem_add(), passing the size of the array as a parameter. mem_add() will return a pointer that is passed with the size again into pt_add(), which returns the page number that the array was added to. For each write, first the page that the address resides in is located, and if the page is unlocked, the new value is recorded; otherwise, the write fails. In order to minimize the looping required to locate an addresss page for every byte written, we add a pt_find() function, which we can call once on an array and pass to multiple subsequent writes to that array. By decreasing the amount of unrolling necessary, this gives CBMC a smaller program to perform model checking on. The different implementations of page table locking and unlocking are shown in Appendix B.1.

All arrays used in the hardware are also used in software with the exception of the page table registers, because the page table is implemented differently. However, as specified in property 4.16, there are large gaps in XRAM that secure boot never touches, so to avoid a 64kB memory array, we pack the individual arrays more closely. In hardware we were constrained in how closely packed arrays could be because locking resolution was one page. For example, the sha_out array at address 0xE100 is only 20 bytes, but we can't put anything else in the remaining 236 bytes of page E1 because otherwise they would be locked and unlocked together, which is undesirable. However, our software page table has one byte granularity, so side by side arrays is not a problem.

This C implementation of the hardware still retains the full functionality of the crypto-

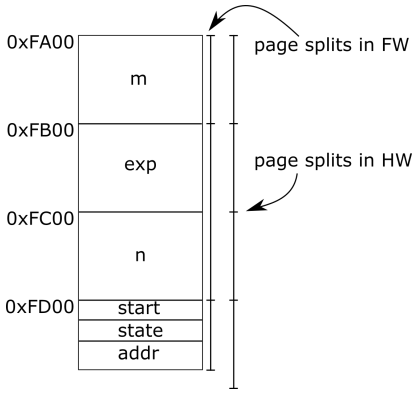

Figure 5.2: Page splitting in HW vs. FW

graphic hashing and encryption, so as we find bugs and make further changes to the secure boot, we are able to use the results of execution to check that the behavior is still consistent between C and Verilog. Because the C executable takes significantly less time to run than the full Verilog simulation, less than one second versus almost an hour, if it is a faithful approximation of the hardware behavior, then it is extremely useful as a debugging tool. In addition to speed, it also offers the possibility of debugging using gdb, which is easier than viewing waveforms, and printing out informative error messages.

5.1.2 Levels of Abstraction in CBMC

For the CBMC[33] abstraction, there are two levels of abstraction depending on the property being checked. At the basic level of abstraction, we abstract away SHA and RSA only. Because SHA and modular exponentiation are used in cryptography and thus necessarily involve complicated computations, we abstract these away as uninterpreted functions. An uninterpreted function returns a nondeterministic value, but return consistent values if called on the same input more than once. CBMC has the capability of defining uninterpreted functions with the __CPROVER_uninterpreted_ prefix, but this does not work very well with our array inputs and outputs, so we make our own. uninterp_sha() can take an input and an expected output to store as a pair, so that if it is called with the same input again, it will output the previously provided output. Use of uninterpreted functions corresponds to level 3 of abstraction in Figure 5.1.

For another small abstraction, note that when creating the image, it is unnecessary to initialize all MAX_IM_SIZE bytes because we are not actually performing verification with real hashing or encryption on the data, so we only need sizes and addresses. When verifying properties that involve hashes, we can partially initialize the hashes and data being hashed, enough so that they are distinguishable, i.e. not all zeros.

Before CBMC can perform any model checking, it unrolls all loops to create a program with only if statements directing the program control flow. In secure boot, the largest loops result from copying long arrays. If the unwinding bound is too short, then CBMC may not find bugs resulting from the end of an array copy overflowing, but a large unwinding bound takes time to unroll and creates much more code to run the model checker on. To counter this problem, we add one more layer of abstraction on top of uninterpreted functions by also abstracting memory writes for large ranges to nondeterministically write to only one index in the entire range. For many properties, we are not concerned with what values are being written to arrays, but rather which locations are being written to. Like other model checkers, CBMC takes assignments to nondeterministic values. During verification, when encountering a nondeterministic variable assignment, it will check that assertions hold for any assignment of the variable. By taking advantage of nondeterminism, we can reduce writes to arrays, which originally require a loop of writes, to just one write, where the index of the array write is nondeterministic. If any of the possible array write locations causes a property violation, CBMC will find this counterexample, so there is no need to explicitly write to every index. Cutting out this loop saves a significant amount of unrolling, because when the upper bound for a loop is a variable, CBMC does not effectively detect this bound well, so it will unwind the loop up to the maximum unwind bound provided, even for write calls that only need to unwind a few times.

All of these abstractions reduce complexity as intended, making the properties easier for CBMC to verify, although even with these abstractions, for some properties array sizes had to be decreased from the full size used in the hardware (Table 5.1).

value	full size	reduced size
N	256	16
H	20	4
K1, K2	16	4
MAX_PRG_SIZE	0x3000	150
MAX_IM_SIZE	0x2000	100

Table 5.1: CBMC reduced sizes. N: number of bytes for RSA. H: output length of SHA. K1, K2: padding sizes in RSA OAEP. MAX_PRG_SIZE: maximum program size. MAX_IM_SIZE: maximum image size.

5.2 Proving Properties with CBMC

Some properties such as property 4.1, which restricts the range of memory writes, were able to be verified with a single run of CBMC; however, in many cases, the size of the state space

was too large, so it was necessary to run CBMC multiple times with different assertions and different abstraction levels. In this section, we explain the verification process for several properties. CBMC runs a sat solver MiniSat[34] on a program with assertions. If the solver outputs unSAT, the assertions hold, and if it outputs SAT they do not.

5.2.1 Effectiveness of Locks

The secure boot verification process is heavily reliant on a working memory lock (property 4.3), so this was the first property checked. The first test, lock, proved that when a page is locked, no writes can change any values in the page. As a sanity check, we also check in nolock that when the page is not locked, the same assertion fails. Next, we introduce a valid bit for each of the pages that is 1 when the page is reset and becomes 0 if any writes not initiated by the secure boot succeed on that page, and we check in lockval that if the data on a page is corrupted by an untrusted write, it is recorded in the page's valid bit. The valid bit never changes while the lock is on, so we know that no untrusted writes succeed while pages are locked. Finally, we introduce a monitor that keeps track of whether pages are unlocked. This allows us to prove property 4.6 (the image does not change once it is loaded and locked) by resetting the image page during stage 1 and checking that the unlock monitor is still low at the end of execution. In Table 5.2, locks is the run that proves property 4.3, and bootlock proves property 4.6. Every CBMC run except for bootlock was run on a small sample program locking.c in Appendix C.1 because locking works in isolation with secure boot.

log file	steps	VCCs	vars	clauses	time(s)	sat/unsat
lock	346	1	3276	11655	0.045	unsat
nolock	435	2	4998	17723	0.055	sat
lockval	461	2	5036	17725	0.058	unsat
lockover	379	2	3708	11573	0.04	sat
locks	484	1	5913	21232	0.078	unsat
bootlock	18694	1	1758195	8617144	8.467	unsat

Table 5.2: CBMC run data for locking properties

5.2.2 Checking Array Bounds

Although the page table was originally implemented in order to minimize memory consumption, it is also useful for checking array bounds. For arrays, the start and end addresses of each page table correspond to ptrLo and ptrHi of that array. The writec() function takes four

parameters: the page number, write address, write value, and source. The source parameter indicates whether the source was initiated by the secure boot or by an untrusted module. By checking that the write address is within the addresses of the page provided, we can verify that array bounds are not violated. Although the hardware does not perform this check, by proving that the page check never fails for trusted writes, we guarantee that the secure boot program will never write out of bounds, even if incorrect sizes or addresses are provided to it by an untrusted module.

Because this property does not depend on the values of the data being written to arrays, but only the write locations, we can check this property with abstraction level 4, the highest level. Although checking this property with full sizes for the whole program was too large and caused an out of memory exception, it is possible to break down the program into smaller sections, for example, by stage, and check that all writes in each section of the program are legal. In Table 5.3, loadwrite proves that during the load stage, all of the writes are to legal addresses, and loadfail shows that if we don't check the address or size constraints on modules, the same property fails. For checking that the image does not change during signature verification, we had to limit unrolling during the decryption of the G hash due to memory constraints.

log file	steps	VCCs	vars	clauses	time(s)	sat/unsat
loadimwr	902	4	656096	1864562	736.041	unsat
keywrite	4684	12	295446	1751872	1229.22	unsat
sigwrite	13895	85	1333084	6477606	37.348	unsat
loadwrite	1695	10	5249856	19165871	2158.96	unsat
loadfail	1565	10	5250916	19170435	2187.06	sat
shawr	3434	12	12664027	59698246	1984.2	unsat

Table 5.3: CBMC run data for memory access properties

5.2.3 Correctness of mem_wr

In order to take advantage of running hardware and software in parallel, we created the mem_wr module to perform memory operations while the 8051 continues execution of instructions that do not need to access XRAM. However, the mem_wr module is not trusted, so we need to check that it faithfully copies data. In order to check this property, we need to carry out every write, so level 3 abstraction is used. For this property, we write every value in the array, but we use nondeterminism to check that every written value is correct by checking one nondeterministic address in the write range. The two main uses of the mem_wr module are for loading the image, and for copying the modules from the image to the fi-

nal program location. Recall from section 5.1.2 that we use a different image initialization method for CBMC, so we focus on module loads only.

First, we verified that directly after each module load, the module data inside the program array matches the module data provided in the image. In Table 5.4, load is the CBMC run that shows loading just the first module works correctly, and load2 shows that for two modules, both load correctly. Although the program load stage involves a loop over the number of modules, which has the potential to be very large, it suffices to limit CBMC to two loop iterations. This is because the difference in behavior between loop iterations are the module sizes and pointer offsets. We use CBMC to verify that there are two ways for a load to fail: either the address and size combination puts pointers out of bounds, or two modules overlap. We check this in loadlast and overlap in Table 5.4. But any load address or size of an arbitrary module that would have cause the load to fail due to overstepping bounds would also cause the load to fail if the same data was in module 1, so this will be caught by CBMC. Similarly, any two overlapping modules can be moved to module 1 and module 2. Therefore, proving the property for 2 modules is the same as proving it for any number of modules.

log file	steps	VCCs	vars	clauses	time(s)	sat/unsat
load	11877	9	2658348	14579473	733.26	unsat
load2	17258	11	3972133	22385404	7038.55	unsat
loadlast	17384	9	3961077	22290097	25819.5	unsat
overlap	17368	9	3961070	22290078	292.309	sat

Table 5.4: CBMC run data for checking mem_wr

5.2.4 Hash Comparison

We used our uninterpreted version of SHA with level 3 abstraction to verify that the key hashes and module hashes match their respective reference hashes as long as nothing interferes with the copying process. The uninterpreted function essentially checks for equality of the input and previously seen arguments. If there is a match, it returns the output associated with that input. Because these outputs were randomly generated, there is no extra value in checking the output in addition to the input, because we know that the outputs match if and only if the inputs match. It is possible for the outputs to mismatch even if the inputs match if an attacker writes to the output after SHA writes the hash to XRAM. Thus, to check the validity of the hash without directly comparing the outputs, we need to check if the inputs match and that the output has not been changed since it was written by SHA,

indicated by the valid bit of the sha_out page. Once we have verified that the valid bits of sha_in, sha_out, boot,

Applying this approach to verify the RSA key for property 4.18, first we verified in shapass of Table 5.5 that if the key in the image matches the hash in ROM and the image data is not modified, the computed hash matches the stored hash, then we isolated the cases of failure and showed in shafail that the hashes do not match if the provided key is incorrect, the image data is changed, or either of sha_in or sha_out are written to by an outside module. We check the hashes of the modules for property 4.20, the same way, with the addition that an additional valid bit, addr_val must also hold. addr_val is evaluated during the module load phase discussed in section 5.2.3, and if addr_val is 0, there is overlap in the modules, so a later module will corrupt the one loaded before. These checks are not for proving properties, but rather to emulate the correctness checks performed by the hardware while abstracting complicated functions like hashing and exponentiation.

log file	steps	VCCs	vars	clauses	time(s)	sat/unsat
shaok	12897	23	1507616	961983	6.085	unsat
shapass	13127	19	2850534	20875046	89.158	unsat
shafail	13648	22	3388785	23176121	207.28	unsat

Table 5.5: CBMC run data for checking hashes

5.3 Results

The properties that we are able to prove most cleanly are those relating to restricted memory access and memory locking, the properties covered in sections 4.3.1 and 4.3.2. By constructing data structures and extra variables that held auxiliary information about the state of the program, we were able to prove that locking is effective and create valid bits on different aspects of the secure boot verification that reflected when the protection of our memory locks was held consistently. We were also able to identify the possible sources of error that caused verification to fail, recognizing that they are overapproximations.

Errors in the secure boot that were fixed were changing the key load and verify order, because the key needs to be loaded in to RSA first, locked, and then checked to guarantee that the key that passes verification is the same as the key that is loaded; and adding more checks to ensure that verification only passes if the read and write addresses of SHA and RSA do not change.

Most of the time was spent fixing the abstraction and refining it to conform to CBMC, rather than fixing bugs in the secure boot. One large structural change was making all arrays

part of one large the mem array representing all of memory, created to combat dereferencing invalid pointers that caused locking properties to always pass regardless of whether the lock was activated. This verification process demonstrates the value in an abstraction that can be generated and checked automatically or with minimal user input.

The size of the problem given to the model checker varied in size depending on the scale of the property. For smaller properties, it was possible to expand verification to the full array sizes. In Appendix C, the property being checked in each log file is explained briefly, and Table C.1 gives an overview of different CBMC runs and how they compare in size of the program execution, number of variables and verification conditions, and time to solve.

Chapter 6

Concluding Remarks

In order to address challenges posed by systems with heavy integration of hardware and firmware, we build a sample infrastructure consisting of a 8051 microcontroller running firmware that directs the execution of several hardware blocks, making use of available open source resources. Each hardware module added shows in increase in the amount of interaction between hardware and firmware and touches upon different aspects of the interaction. We develop a secure bootloader application and identify properties of the secure boot that cross the hardware-firmware boundary, walking through the steps required when abstracting hardware to prove properties about both hardware and firmware in the same analysis. The process of verification demonstrates the utility of an automatically generated and verifiable abstraction method to aid coverification.

6.1 Limitations and Future Directions

There exist possible extensions on both secure bootloader functionality and verification. Currently, while the secure boot performs verification on whether or not a presented image file is trusted and loads the program into the addresses specified in the image header, after verification completes, regardless of success or failure, the program terminates. Extending the secure boot to jump to the first instruction after verification succeeds is necessary to make it a practical utility. Another change that would make the secure boot more realistic while introducing more interactions would be to load the image from an outside source, through the 8051's other modes of communication, rather than it being stored inside the mem_wr module.

Using CBMC as our model checker also limited the number of properties that were provable. With just CBMC, we are unable to prove information flow hyperproperties that are less direct than write access, so integrating the ability to check hyperproperties into the model

checker is desirable. Additionally, regarding the program flow properties of section 4.3.3, in C the program does not have information about PC or SP, so it is difficult to verify these properties. There is no way to manipulate PC with this abstraction, so the order property should always hold.

As the current abstraction framework stands, the direct C implementation can be run and monitored to check that behavior is consistent with the hardware, but there is no automated way to check that the two implementations agree. Consistency between the code that CBMC checks and the hardware is even less clear. Especially in cases where the implementation is fundamentally different, for example the page table locking, a proof in CBMC may not translate to guarantees about the hardware properties.

To bridge the gap between hardware and software, an Instruction Level Abstraction (ILA)[35] to comprehensively model the hardware as software instructions is being completed. The ILA models the hardware by capturing all updates to states accessible by the firmware and is synthesized from a template specification. To check consistency between the hardware implementation and the generated ILA, a golden model is also generated from the ILA, and a refinement loop continues to modify the ILA until the golden model matches the hardware. The ILA goes beyond previous work on compositional SoC verification that proposes using a bridge module to create a formal model of the system[36] because it provides an easy way to construct and verify the bridge specifications, and it also addresses the challenge of ensuring that the software model and RTL model are consistent, which is a problem left open by previous approaches of hardware/software coverification using symbolic execution of software that models the hardware, plus the original software that the hardware interacts with[37]. Using the ILA to verify secure boot will strengthen the support for its security properties, and we can use the C abstraction as a baseline to compare with the ILA.

Appendix A

Test Programs

A.1 SHA-1

code testing SHA-1 module showing padding before calling hardware

```
#include <reg51.h>

void quit() {
  P0 = P1 = P2 = P3 = 0xDE;
  P0 = P1 = P2 = P3 = 0xAD;
  P0 = P1 = P2 = P3 = 0x00;
  while(1);
}

__xdata __at(0xFE00) unsigned char sha_reg_start;
__xdata __at(0xFE01) unsigned char sha_reg_state;
__xdata __at(0xFE02) unsigned int sha_reg_rd_addr;
__xdata __at(0xFE04) unsigned int sha_reg_wr_addr;
__xdata __at(0xFE06) unsigned int sha_reg_len;
__xdata __at(0xE000) unsigned char d1[64];
__xdata __at(0xE100) unsigned char d2[128];
__xdata __at(0xE200) unsigned char hash[20];

/*-------------------------------------------------------------------------*/

void main() {

  unsigned char pyhash[20];
  int i;
  int good=1;
  int N = 128;

  pyhash[0] = 0xc6; pyhash[1] = 0x13; pyhash[2] = 0x8d; pyhash[3] = 0x51;
  pyhash[4] = 0x4f; pyhash[5] = 0xfa; pyhash[6] = 0x21; pyhash[7] = 0x35;
  pyhash[8] = 0xbf; pyhash[9] = 0xce; pyhash[10] = 0xe; pyhash[11] = 0xd0;
  pyhash[12] = 0xb8; pyhash[13] = 0xfa; pyhash[14] = 0xc6; pyhash[15] = 0x56;
  pyhash[16] = 0x69; pyhash[17] = 0x91; pyhash[18] = 0x7e; pyhash[19] = 0xc7;

  // reset the entire block.
  for(i=0; i<N; i++) { d2[i] = 0; }
  // initialize bytes 0-63
  for(i=0; i < 64; i++) { d2[i] = i; }
  // add binary string of the form 10* after this (only need the 1).
  d2[64] = 0x80;
  // put message size (in bits) in last 8 bytes of the block
  d2[126] = 0x02;
  d2[127] = 0x00;

  sha_reg_rd_addr = (unsigned int) &d2;
  sha_reg_wr_addr = (unsigned int) &hash;
  sha_reg_len = N;

  // now start encryption.
  sha_reg_start = 1;
  // now wait for encryption to complete.
  while(sha_reg_state != 0);

  // read encrypted data and dump it to P0.
  for(i=0; i < 20; i++) {
    P1 = 2;
```

```
    P0 = hash[i];
    if(hash[i] != pyhash[i])
      {
        good = 0;
        break;
      }
  }

  P0 = good;

  // finish.
  quit();
}
```

A.2 RSA

Listing A.1: load function that calls mem_wr module

```
// set up data transfer
void load(unsigned char* data, unsigned int length, unsigned int startaddr,
          unsigned char skipread)
{
  memwr_reg_rd_addr = (unsigned int)data;
  memwr_reg_wr_addr = startaddr;
  memwr_reg_len = length;
  memwr_reg_start = (unsigned char)(skipread << 1 | 1);

  // wait for load to finish
  while(memwr_reg_state != 0);
}
```

Listing A.2: HMAC using SHA-1 hardware module

```
void HMAC(unsigned char *key, unsigned int klen, unsigned char *message,
          unsigned int mlen)
{
  unsigned int i;

  // inner hash
  for(i=0; i<klen; i++)
    data[i] = key[i] ^ 0x36;
  for(i=klen; i<64;i++)
    data[i] = 0x36;
  for(i=0; i<mlen; i++)
    data[i+64] = message[i];
  sha1(data, 64+mlen);

  // outer hash
  for(i=0; i<klen; i++)
    data[i] = key[i] ^ 0x5c;
  for(i=klen; i<64;i++)
    data[i] = 0x5c;
  for(i=0; i<20; i++)
    data[i+64] = hash[i];
  sha1(data, 84);
}
```

Listing A.3: PRG using HMAC, needs gprg state array

```
__xdata __at(0xFD30) unsigned char gprg[20];

// generate random number, put in hash
void PRG(unsigned char* state)
{
  unsigned int i;
  unsigned char next[20];

  HMAC(state, 20, one, 32);
  for(i=0; i<20; i++)
    next[i] = hash[i];

  HMAC(state, 20, zero, 32);
  for(i=0; i<20; i++)
    state[i] = next[i];
}
```

Listing A.4: OAEP uses PRG for G and HMAC for H

```
void OAEP()
{
  unsigned int i,j;

  // K1 0s
  for(i=0; i<K1; i++)
    exp_reg_m.zeros[i] = 0;

  // do G to compute X
  PRG(rprg); // generate new r and write it to hash
  for(i=0; i<K2; i++)
    exp_reg_m.r[i] = hash[i];
  PRGinit(hash, K2, gprg);
  PRG(gprg);
  i=0; j=0;
  while(i < N-K2-1){
    if(j == 20){
      PRG(gprg);
      j = 0;
    }
    exp_reg_m.m[i] = exp_reg_m.m[i] ^ hash[j];
    i++;
    j++;
  }

  // do H to compute Y
  HMAC(Hseed, 20, exp_reg_m.m, N-K2-1);
  for(i=0; i<K2; i++)
    exp_reg_m.r[i] = exp_reg_m.r[i] ^ hash[i];

  exp_reg_m.padbyte = 1; // marker byte
}
```

Appendix B

Secure Boot

Listing B.1: secureboot.c, HW version

```
#include "rsa.h"

__xdata __at(0x0000) unsigned char program[0x3000];
__xdata __at(0x3000) unsigned char boot[0x2000];
__xdata __at(0xC000) unsigned char sha_in[0x2000 +0x40];
__xdata __at(0xE100) unsigned char sha_out[20];
__xdata __at(0xE200) unsigned char rsa_out[256];

struct modules{
  unsigned int addr;
  unsigned int size;
  unsigned char hash[H];
};

struct image{
  unsigned char sig[N];  // signature of header
  unsigned char exp[N];
  unsigned char mod[N];  // n in modexp
  unsigned int num;      // total number of blocks
  struct modules module[1];
};

enum status{
  UNDET,
  FAIL,
  PASS
};

//public key hash
CODE unsigned char pkhash[20] = {0x37, 0x34, 0xA6, 0x83,
                                 0x5F, 0xFC, 0xE0, 0x2B,
                                 0xC6, 0xEE, 0xCB, 0x81,
                                 0x6C, 0x92, 0x6C, 0x7C,
                                 0xBA, 0x79, 0xCB, 0x8F};

void quit() {
  P0 = P1 = P2 = P3 = 0xDE;
  P0 = P1 = P2 = P3 = 0xAD;
  P0 = P1 = P2 = P3 = 0x00;
  while(1);
}

void main() {
  unsigned int i, j;
  unsigned int num;   // total number of blocks
  struct image* im;
  struct modules* block; // current block
  unsigned int size;
  unsigned char* moddata;
  unsigned int ldaddr = 0;
  enum status pass = UNDET;

  // STAGE 0: set up
  // set SHA read and write addresses
  unlock_wr((unsigned char*)&sha_regs.rd_addr, (unsigned char*)(&sha_regs.wr_addr+1));
  sha_regs.rd_addr = sha_in;
  sha_regs.wr_addr = sha_out;
  lock_wr((unsigned char*)&sha_regs.rd_addr, (unsigned char*)(&sha_regs.wr_addr+1));
  // unlock memwr registers
  unlock_wr((unsigned char*)&memwr_regs.start, (unsigned char*)(&memwr_regs+1));
```

```
// set up RSA
unlock_wr((unsigned char*)&rsa_regs.opaddr, (unsigned char*)(&rsa_regs.opaddr+1));
rsa_regs.opaddr = rsa_out;  // set up address to write to
lock_wr((unsigned char*)&rsa_regs.opaddr, (unsigned char*)(&rsa_regs.opaddr+1));

if(!RSAinit(rsa_out, sha_in, sha_out)){
  pass = FAIL;
  quit();
}

// STAGE 1: read image into RAM
unlock_wr(boot+MAX_IM_SIZE);
load(0, MAX_IM_SIZE, boot, 1);

// image is loaded.
// now we need to lock boot to boot + MAX_IM_SIZE
lock_wr(boot, boot+MAX_IM_SIZE);

// STAGE 2: check that key matches hash
im  = (struct image*) boot;
// set signature key
unlock_wr(rsa_regs.exp, rsa_regs.exp+N);
writecarr(rsa_regs.exp, im->exp, N);
lock_wr(rsa_regs.exp, rsa_regs.exp+N);
// set signature modulus
unlock_wr(rsa_regs.n, rsa_regs.n+N);
writecarr(RSA_KEYS, rsa_regs.n, im->mod, N);
lock_wr(rsa_regs.n, rsa_regs.n+N);

// check the hashes
sha1(rsa_regs.exp, 2*N);
for(i=0; i<H; i++){
  if(sha_out[i] != pkhash[i]){
    pass = FAIL;  // FAIL: key hash mismatch
    quit();
  }
}

// STAGE 3: verify signature in boot
num = im->num & 0xFFFF;  // number of modules

// sizeof image struct includes extra signature and first module
size = sizeof(struct image) - (im->exp -(unsigned char*)im)
  + sizeof(struct modules)(num-1);
if(size > MAX_IM_SIZE){
  pass = FAIL; // FAIL: image too large
  quit();
}

if(!verifySignature(im->exp, size, im->sig)) {
  pass = FAIL;  // FAIL: signature mismatch
  quit();
}

// STAGE 4: load blocks
if(num == 0){  // no blocks to load, done
  pass = PASS;
  return;
}
// unlock memory space for program
unlock_wr(program, program + MAX_PRG_SIZE);
block = im->module;  // block data in header
moddata = (unsigned char*)(block + num); // program data of this module
size = 0;

for(i=0; i<num; i++){
  // check that size and address are valid
  size = block->size & 0xFFFF;      // size of current module
  ldaddr = block->addr;   // address to load this module into

  // the data does not fit inside the image
  if(moddata + size > boot + MAX_IM_SIZE ||
     moddata + size < moddata){
      pass = FAIL;
      quit();
  }
  // the data does not fit in memory range allocated for it
  if(size + ldaddr > MAX_PRG_SIZE ||
     ldaddr + size < ldaddr){
      pass = FAIL;
      quit();
    }

  // load data
  if(size != 0)
    load(moddata, size, program+ldaddr, 0);

  // update to next module
  moddata += size;
  block++;
}
```

```
  // lock newly loaded data
  lock_wr(program, program + MAX_PRG_SIZE);

  block = im->module;  // go back to first module
  moddata = (unsigned char*)(block + num); // program data of this module

  for(i=0; i<num; i++){
    unsigned int ldaddr;
    size = block->size & 0xFFFF;      // size of current module
    ldaddr = block->addr;    // address to load this module into

    // check module hash
    sha1(program+ldaddr, size);
    for(j=0; j<H; j++){
      if(sha_out[j] != block->hash[j]){
        pass = FAIL;
        quit();
      }
    }

    // update to next module
    moddata += size;
    block++;
  }

  // PASS or FAIL
  if(pass != FAIL)
    pass = PASS;

  P0 = pass;
  quit();
}
```

B.1 Abstractions

Listing B.2: meminit for minimally initializing image

```
void meminit(struct image* image)
{
    int i;
    image->sig[nondet_uint()%N] = nondet_uchar();
    image->exp[nondet_uint()%N] = nondet_uchar();
    image->mod[nondet_uint()%N] = nondet_uchar();
    image->num = nondet_uint();
    for(i=0; i<2; i++){
      image->module[0].addr = nondet_uint();
      image->module[0].size = nondet_uint();
      image->module[0].hash[nondet_uint()%H] = nondet_uchar();
    }
}
```

Listing B.3: writecarr abstractions for level 4 abstraction

```
// write len bytes from data to addr
int writecarr(int page, unsigned char* addr, unsigned char* data, unsigned int len)
{
#ifndef CBMC
  unsigned int i;
  for(i=0; i<len; i++){
    writec(page, addr+i, data[i], 1);
  }
  return page;
#else
  unsigned int offset = nondet_uint()%len;
  int wr_success = writec(page, addr+offset, data[offset], 1);
  //assert(wr_success);
  return wr_success;
#endif
}
```

Listing B.4: lock in HW vs. lock in C/CBMC

```
// HW lock
int lock_wr(unsigned char* startaddr, unsigned char* endaddr)
{
  // index of pt_wren
  unsigned int curr = (unsigned int)startaddr >> 11;
  unsigned int end = (unsigned int)endaddr-1 >> 11;
  // range of pages in pt_wren[i]
  unsigned int lowpage = (unsigned int)startaddr >> 8 & 7;
  unsigned int highpage = (unsigned int)endaddr-1 >> 8 & 7;
```

```
  // no memory in range
  if(startaddr > endaddr)
    return 0;

  // all pages are in the same index of pt_wren
  if(curr == end)
    pt_wren[curr] &= (1 << highpage+1) - (1 << lowpage) ^ 0xFF;
  else{
    // don't unlock pages below lowpage
    pt_wren[curr] &= (unsigned char)(0xFF << lowpage & 0xFF);
    for(;curr < end; curr++)
      pt_wren[curr] = 0x00;
    // don't unlock pages above highpage
    pt_wren[end] &= (1 << highpage+1) - 1 ^ 0xFF;
  }
  return 1;
}

// C lock
int lock(int page, unsigned char* startaddr, unsigned char* endaddr)
{
  // addresses are not in this page
  if(page < 0 || page >= PAGES ||
     startaddr < pt.start[page] || endaddr > pt.end[page])
    {
      //assert(0);
      return 0;
    }

  pt.locked[page] = 1;
  return 1;
}
```

Appendix C

CBMC

C.1 Locking properties

Listing C.1: code used for checking locking properties isolated from secure boot

```
#include "rsa.h"

unsigned char* nondet_ptr();
unsigned int nondet_uint();
unsigned char nondet_uchar();
int nondet_int();

unsigned char mem[4];
XDATA_ARR(0x5000, 2, unsigned char, boot);

void main() {
  unsigned char before, after;
  const unsigned int compind = nondet_uint()%2;
  // put new arrays into pt
  int page;
  boot = mem_add(2);
  page = pt_add(boot, 2);

  // initialize memory
  unlock(page, boot, boot+2);
  pt_reset(page);
  writec(page, boot+compind, nondet_uchar(), 1);
  before = boot[compind];

  // something might break the image here
  if(nondet_uint())
    writec(nondet_uint(), nondet_ptr(), nondet_uchar(), 0);

  // image is loaded.
  // now we need to lock boot
  lock(page, boot, boot+2);
  //before = boot[compind];
  //before = pt_valid(page);

  if(nondet_uchar())
    unlock(nondet_int(), nondet_ptr(), nondet_ptr());

  if(nondet_uchar())
    writec(nondet_int(), nondet_ptr(), nondet_uchar(), 0);
  if(nondet_uchar())
    writec(nondet_int(), nondet_ptr(), nondet_uchar(), 1);
  //after = pt_valid(page);
  after = boot[compind];
  //assert(after==before);
  assert(pt_lockchange(page) || !pt_valid(page) || before==after);
  // assert(before != after || pt_valid(page) && pt_lockchange(page));

}
```

C.2 Model Checker statistics

Description and notes for runs:

lock - if lock is on, writes do not change the locked memory
nolock - if lock is off, writes can change the locked memory
lockval - if an untrusted write succeeds, the valid bit of that page will record it, and the valid bit does not change while the page is locked.
locks - if the lock is never unlocked after initialization and the page is not corrupted before the lock, then the memory value is the same as at initialization
bootlock - once it is loaded, the image does not change during secure boot
loadimwr - all writes are within array bounds during image load
keywrite - all writes are within array bounds during key verification. Not enough memory to perform full size verification
sigwrite - all writes are within array bounds during signature verification
sigwrfull - even with only one iteration of loop OAEP.1, not enough memory
loadwrite - all writes are within array bounds during module loading
shawr - all writes are within array bounds during module hash checking
loadfail - write fails if we do not check address/size constraints
load - loading one module succeeds
load2 - loading two modules succeeds, checking immediately after each load
loadlast - if modules do not overlap, all modules load correctly, check after all loads complete
overlap - load fails if we do not check for overlapping modules. A counterexample is found much more quickly than it takes to prove the property (loadlast)
shaok - if image is valid and no outside writes, key check always passes
shapass - if key check passes, then exp in RSA acc and image match
shafail - if key check fails, then one of SHA registers, sha_out, sha_in, RSA keys, or the image was corrupted
keylock - all locks up to key check pass

log file	lvl	full	steps	VCCs	vars	clauses	time(s)	unsat
lock	4	-	346	1	3276	11655	0.045	y
nolock	4	-	435	2	4998	17723	0.055	n
lockval	4	-	461	2	5036	17725	0.058	y
lockover	4	-	379	2	3708	11573	0.04	n
locks	4	-	484	1	5913	21232	0.078	y
bootlock	4	n	18694	1	1758195	8617144	8.467	y
sigwrite	4	n	85	13895	1333084	6477606	37.348	y
sigwrfull	4	y	8507	58	-	-	-	-
loadimwr	4	y	902	4	656096	1864562	736	y
keywrite	4	y	4684	12	295446	1751872	1229	y
shawr	4	y	3434	12	12664027	59698246	1984	y
loadwrite	4	y	1695	10	5249856	19165871	2158	y
loadfail	4	y	1565	10	5250916	19170435	2187	n
load	3	n	11877	9	2658348	14579473	733	y
load2	3	n	17258	11	3972133	22385404	7038	y
loadlast	3	n	17384	9	3961077	22290097	25819	y
overlap	3	n	17368	9	3961070	22290078	292	n
shaok	3	n	12897	23	1507616	961983	6.085	y
shapass	3	n	13127	19	2850534	20875046	89	y
shafail	3	n	13648	22	3388785	23176121	207	y
keylock	4	y	4868	8	11929853	53280801	1271	y

Table C.1: CBMC run data, showing level of abstraction, full or reduced array sizes, number of steps in the program expression, number of verification conditions, number of variables and clauses for MiniSAT, runtime decision procedure, and if the solver returned SAT or unSAT

Bibliography

[1] P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Combinational test generation using satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, pp. 1167–1176, Sep 1996.

[2] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, (London, UK, UK), pp. 193–207, Springer-Verlag, 1999.

[3] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, pp. 394–397, July 1962.

[4] L. Zhang and S. Malik, "Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications," in *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*, p. 10880, IEEE Computer Society, 2003.

[5] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. C-35, pp. 677–691, Aug 1986.

[6] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Lang. Syst.*, vol. 8, pp. 244–263, Apr. 1986.

[7] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, pp. 576–580, Oct. 1969.

[8] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability* (A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, eds.), vol. 185 of *Frontiers in Artificial Intelligence and Applications*, ch. 26, pp. 825–885, IOS Press, Feb. 2009.

[9] R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P. P. Pande, C. Grecu, and A. Ivanov, "System-on-chip: Reuse and integration," *Proceedings of the IEEE*, vol. 94, pp. 1050–1069, June 2006.

[10] "NIST FIPS. 197: Announcing the advanced encryption standard (AES)," tech. rep., National Institute of Standards and Technology, 2001.

[11] "NIST FIPS. 180-2: Secure Hash Standard (SHS)," tech. rep., National Institute of Standards and Technology, 2001.

[12] H. Lipmaa, P. Rogaway, and D. Wagner, "CTR-mode encryption," in *First NIST Workshop on Modes of Operation*, 2000.

[13] H. Hsing. http://opencores.org/project,tiny_aes, 2014.

[14] J. Strombergson. https://github.com/secworks/sha1, 2014.

[15] M. Mazidi, R. McKinlay, and J. Mazidi, *The 8051 Microcontroller: A Systems Approach*. Pearson Education, 2012.
[16] S. Teran and J. Simsic. http://opencores.org/project,8051, 2013.
[17] "SDCC compiler user guide." http://sdcc.sourceforge.net/doc/sdccman.pdf.
[18] "Hexadecimal object file format specification," January 1988.
[19] Ç. K. Koç, "RSA hardware implementation," *RSA laboratories*, 1995.
[20] M. Bellare and P. Rogaway, *Advances in Cryptology — EUROCRYPT'94: Workshop on the Theory and Application of Cryptographic Techniques Perugia, Italy, May 9–12, 1994 Proceedings*, ch. Optimal asymmetric encryption, pp. 92–111. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995.
[21] H. Krawczyk, M. Bellare, and R. Canetti, "Hmac: Keyed-hashing for message authentication," 1997.
[22] D. Pointcheval, *Encyclopedia of Cryptography and Security*, ch. OAEP: Optimal Asymmetric Encryption Padding, pp. 882–884. Boston, MA: Springer US, 2011.
[23] "Atmel application note: Safe and secure bootload implementation." http://www.atmel.com/Images/doc6282.pdf.
[24] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.
[25] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*. New York, NY, USA: Springer-Verlag New York, Inc., 1992.
[26] J. McLean, "A general theory of composition for trace sets closed under selective interleaving functions," in *Research in Security and Privacy, 1994. Proceedings., 1994 IEEE Computer Society Symposium on*, pp. 79–93, May 1994.
[27] M. Clarkson and F. Schneider, "Hyperproperties," in *Computer Security Foundations Symposium, 2008. CSF '08. IEEE 21st*, pp. 51–65, June 2008.
[28] M. R. Clarkson, B. Finkbeiner, M. Koleini, K. K. Micinski, M. N. Rabe, and C. Sánchez, "Temporal logics for hyperproperties," *CoRR*, vol. abs/1401.4492, 2014.
[29] B. Finkbeiner, M. Rabe, and C. Sánchez, "Algorithms for model checking HyperLTL and HyperCTL*," in *Proceedings of the 27th International Conference on Computer Aided Verification (CAV)*, 2015.
[30] C. Lattner and V. Adve, "Llvm: a compilation framework for lifelong program analysis transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pp. 75–86, March 2004.
[31] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas, "The seahorn verification framework," in *Computer Aided Verification*, pp. 343–361, Springer, 2015.
[32] G. Balakrishnan, A. Gupta, S. Sankaranarayanan, N. Maeda, T. Imoto, R. Pothengil, M. Hussain, *et al.*, "Scalable and scope-bounded software verification in Varvel," *Automated Software Engineering*, vol. 22, no. 4, pp. 517–559, 2015.
[33] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ansi-c programs," in *In Tools and Algorithms for the Construction and Analysis of Systems*, pp. 168–176, Springer, 2004.
[34] N. Sorensson and N. Een, "Minisat v1. 13-a sat solver with conflict-clause minimization," *SAT*, vol. 2005, p. 53, 2005.
[35] P. Subramanyan, Y. Vizel, S. Ray, and S. Malik, "Template-based synthesis of instruction-level abstractions for soc verification," in *Proceedings of the 15th Conference*

on Formal Methods in Computer-Aided Design, FMCAD '15, (Austin, TX), pp. 160–167, FMCAD Inc, 2015.

[36] F. Xie, X. Song, H. Chung, and R. Nandi, "Translation-based co-verification," in *Proceedings of the 2Nd ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, MEMOCODE '05, (Washington, DC, USA), pp. 111–120, IEEE Computer Society, 2005.

[37] A. Horn, M. Tautschnig, C. Val, L. Liang, T. Melham, J. Grundy, and D. Kroening, "Formal co-validation of low-level hardware/software interfaces," in *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pp. 121–128, Oct 2013.