University of São Paulo
Institute of Mathematics and Statistics
Bachelor of Computer Science

Gabriel Capella

Reimplementing SMART: a Practical Remote Attestation Example for Embedded Devices

São Paulo
December 2018

Reimplementing SMART: a Practical Remote Attestation Example for Embedded Devices

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: Prof. Dr. Alfredo Goldman vel Lejbman

São Paulo
December 2018

Resumo

Verificação remota é um método de atestar à distância a integridade de um dispositivo. Este trabalho visa estudar esse tipo de monitoramento com foco em dispositivos embarcados. Nele é realizada uma revisão bibliográfica dos artigos mais recentes na área e, entre eles, um artigo específico, denominado *SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust*. Por ser uma referência na área e ter servido de base para diversas publicações, este trabalho refaz todos os seus passos. Além disso, foi criado um protótipo em um FPGA para testar se a implementação teórica sugerida funciona. Para testar o funcionamento desse protótipo em um cenário real, ele foi conectado a Internet. Para responder as requisições desse dispositivo e realizar a verificação remota nela, um servidor específico foi desenvolvido. Além disso, ataques foram simulados no dispositivo e no meio de comunicação. Todos os ataques foram identificados com êxito.

Palavras-chave: segurança em sistemas embarcados, verificação remota, SMART.

Abstract

Remote attestation is a method for verifying the integrity of a remote device. This work studies this concept focused on embedded devices, containing a bibliographical review of the most recent papers in this area and, among them, a specific paper called *SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust*, which referenced remote attestation and served as a basis for several publications. Thus, all procedures in the aforementioned paper have been remade for this work. Furthermore, a prototype using FPGA was developed to verify the outcome of the suggested implementation. The prototype had Internet access in order to test using real scenarios. A specific server was developed to respond to device requests and perform remote attestation. Also, attacks were simulated on the device and in the communication channel. The experiment results show that all attacks were successfully identified.

Keywords: embedded systems security, remote attestation, SMART.

Contents

Chapter 1

Introduction

1.1 Motivation

Nowadays, the term IoT (Internet of Things) is turning into a buzzword. Humans, specifically government agencies and industries have learned that connecting simple devices to the Internet can produce a huge volumes useful data, which can be used to improve production, make houses smarter, and save money and time. Some studies show that by 2020 there will be more than 20 billion IoT devices connected to the Internet [9].

This massive number of devices create new security challenges. The vast number of cyberattacks in the lasts few years shows that the developers responsible for these devices are not taking these challenges seriously. Some of these attacks can be seen in the paper [15]. These kinds of attacks can occur in critical infrastructures, such as nuclear plants, health, and transportation system.

Because these devices are connected to the Internet, their software, network, and hardware layers can be attacked and damaged. Studying techniques to verify if one device was compromised, and test these techniques in real hardware is the main goal object of this work. Attacks will ever occur, but in case it occurs, they need to be quickly identified and defeated.

The process of performing remote verification on these devices receives the name of Remote Attestation. The incorporation of this feature into an embedded device without increasing its cost is another motivation of this work.

1.2 Objective

The objective of this work is to study and implement from scratch the simple and best remote attestation technique for embedded devices. The technique chosen was SMART [6]. This decision was made because it is one of the first systems designed specifically for embedded devices. Also, it tries to implement the remote attestation feature maintaining the low cost of the devices. Other methods, like software remote attestation techniques, TrustLite [11] and Sancus [13] have advantages and disadvantages, a detailed comparison will be provided in section 2.4.

Another goal of this work is to advance SMART research by implementing it into a real device. For this, a prototype was created using the openMSP430 microcontroller unit (MCU) and tested in a field-programmable gate array (FPGA). Unlike the original paper, the main changes were made in the device memory bus and not in the memory backbone. This change was made to simplify the implementation and, as a consequence, to avoid errors.

As a final example, this prototype was used to control a LED remotely, over the Internet. Despite being a simple application, this device can be remotely attested. This feature makes it possible to detect if the source code has been altered or if it is in an unexpected state. For example, the LED status can be remotely attested, making it possible to notice if an intruder modifies its status.

1.3 Structure

Key concepts are introduced in the Chapter 2, where basic concepts of remote attestation and FPGAs are presented. The aforementioned chapter also presents the the most important papers related to remote attestation. Thus, the reader will be able to understand all the necessary requirements to implement and test a device designed by a Hardware Description Language (HDL).

The third chapter contains the premises adopted in the SMART and how to implement a device running SMART without violating any of these premises. Chapter 3 also explains every decision made for this project.

Finally, there are two appendices. The Appendix A shows a detailed explanation of the main source file used for this work. Appendix B describes the rest of the files used during the development of this work.

Chapter 2

Key Concepts

This chapter presents definitions of related technologies, and related works in the field of remote attestation.

2.1 Embedded Devices

An embedded device is a system built for a specific task. The main difference from common integrated circuits is that it has programmable hardware, making it possible to change its functionality by simply changing the program embedded inside it. However, they are not built for a general purpose, and the majority of them do not run full-featured operating systems.

There are embedded systems based on microprocessors and microcontrollers. Both of them have been designed for real-time applications. Microprocessor-based systems usually have a processor and a memory chip in the same printed circuit board (PCB). Because this type of embedded system has multiple components, it is more expensive than the ones that have every part integrated into a single chip. However, they are more customizable in production. For example, it is possible to change the memory size in different models by changing only the memory component. There are some microprocessors capable of running a complete operational system, such as the ones used in home routers.

The microcontroller (or MCU for Microcontroller Unit) is a single chip that provides all computational capabilities. In most cases, the processor, memory and digital I/O ports are in a single integrated circuit (IC). Although their cost and size are lower than microprocessor-based systems, they offer low computational power, limited storage capability and, usually, single-threading. These characteristics enable the use of identical ICs for different situations, only changing its program. Examples of devices with microcontrollers are remote controls, actuators and sensors that have access to the Internet.

This work focuses on low-cost embedded systems that use microcontrollers. Nowadays, these devices are widely adopted by the IoT industry, making it possible to produce low-cost devices capable of exchanging information over the Internet. The usage of these types of devices has increased over the last few years, which caused cyberattacks and invasions to increase as well. Furthermore, several vulnerabilities have been discovered. Unfortunately, there is a lack of in research for security-related topics of this devices. This is partially because of the difficulty to offer security without increasing their cost.

2.2 Trust Anchor and Root of Trust

One of the main themes related to security is the Trust Anchor. For example, Alice wants to communicate with Bob using an encrypted channel, which can be done with either a symmetric or an asymmetric encryption algorithm. If they choose a symmetric algorithm, they will need to share passphrase before start talking. Choosing a symmetric algorithm will require sharing a passphrase (key) before communicating. However, choosing an asymmetric algorithm will require sharing their public key in a secure channel. If they share their public keys in an insecure channel, they will not be able to identify each other. Thus, a reliable channel is required before communicating. This channel is the trust anchor.

Another example of a trust anchor can be seen in the TLS (Transport Layer Security), which requires using previously-saved public keys (certificates) before creating an encrypted connection. These certificates are usually provided during the installation of the Operating System (OS). It is believed that these certificates are not corrupted nor changed during the installation of the OS. This process is the trust anchor of the TLS.

A similar idea of trust anchor can be used for verifying software and hardware and this is called Root of Trust. The main idea is to verify if the internal state of a remote hardware platform is known. In most cases, anything that occurs after a reliable state can be predicted and trusted.

There are two types of Roots of Trusts in the field of software verification [8]: dynamic and static. In the static one, the verification is made only before the software execution. That is, the software is verified before it is loaded. For example, a module that verifies if a file of the operating system has been modified before boot. A problem of the static type is that it does not guarantee anything about the current software state. Any exploit after its initialization will not be noticed.

The dynamic root of trust is similar to the static type, but verifies the software after it has been initialized. However, it usually uses private mechanisms such as keys to perform the attestation. The software does not have access to these keys, and an authenticated channel is required to transmit the key to the device running the software.

2.3 Remote Attestation

Remote attestation refers to the capability of providing attestation across a network [8]. Attestation is the ability to provide clear evidence depending on the field. For example, the state of a device is considered an evidence in security. The state of the device is comprised of what is in the memory at a particular moment. Thus, it can be considered as an image of the device.

Remote attestation techniques have already been developed for high-end devices such as computers. The Trusted Computing Group (TCG[1]), a computer industry consortium, created a standard called the Trusted Platform Module (TPM[14]), to describe architectural elements needed to build a hardware module that provides security capabilities. Among these capabilities is remote attestation.

TPM is in its second version. Figure 2.1 shows all components of this module. It is worth mentioning that this module can be used to encrypt messages. It has a full encryption engine making it and a key stored securely. Furthermore, this module is connected with the processor, making it possible to exchange information.

[1] https://trustedcomputinggroup.org/ (visited on November, 2018)

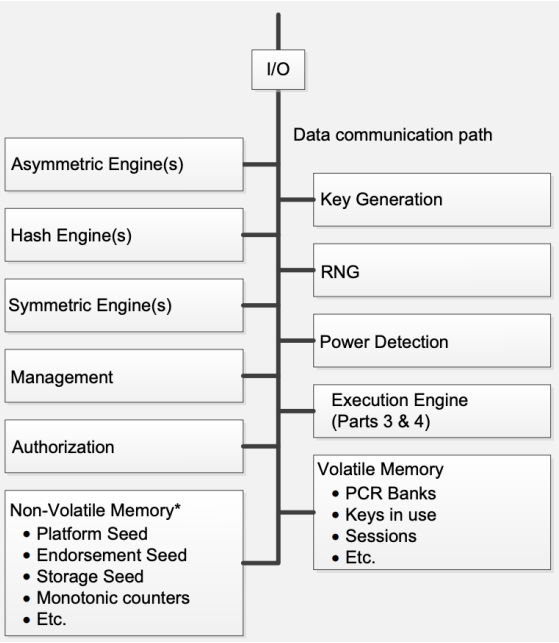

Figure 2.1: *Architectural Overview of TPM. Image from its manual [14].*

Another interesting use is hardware and software verification. Suppose a company does not want to modify the hardware and software of its computers. The TMP can make a hash of the bootloader memory and all modules connected with the processor. Afterwards, it can sign this information and send it to the company. If it suffers any modification, the company will see differences between the expected and received data. Thus, system updates can be stopped or the warranty may be cancelled for example.

TPM is only one a single technical standard of secure cryptoprocessors. Intel and other manufacturers have developed other techniques to optimize security in its core processors. Although, the cost of adding this extra hardware in processor is not huge, when talking about embedded systems it becomes significant. Usually, microcontrollers that drive embedded systems cost few dollars and the cost secure cryptoprocessor has practically the same price of than. In other words, using a cryptoprocessor in an embedded system can approximately double the price. This considerable cost is a major motivation to determine the minimum requirements for a remote attestation.

The article [8] shows the necessary elements and functions to produce a secure remote attestation protocol. As defined in the aforementioned articles, Remote Attestation is comprised of the **challenger** and the **prover**. The **challenger** is the device or computer that verifies the internal state of the **prover** across the network. The primary objective of the protocol is to allow an uncompromised **prover** to provide an authentication token to the challenger, expecting if the **prover** is in an expected state. The **challenger** needs to be aware if the **prover** has been modified.

Three functions are required in order for this protocol to work:

- `Setup()`: needs to produce a long key k probabilistically. This function will be called only once. The **challenger** stores that key and the **prover** has access to it, however only in certain circumstances.

- `Attest`(k, s): a deterministic algorithm that, given a state s of the **prover** and a key k, outputs an attestation token α. The **prover** is able to compute this function.

- `Verify`(k, s, α): given a key k, a device state s, and an attestation token α; the

function returns `True` if `Attest`(k, s) is equal α, otherwise, returns `False`. The **challenger** can compute this function.

To verify Remote Attestation, the **challenger** requests an authentication token to the **prover**. The **prover** produces it using the `Attest` function, creating the attestation token (which indirectly is the information of its state). When the **challenger** receives α, it verifies if it can produce the same token, using the `Verify` function.

This protocol requires sharing the key k between the **challenger** and the **prover**. We will suppose that this k will be shared safely during the manufacturing process. Thus, the key will be written in the device before it can be compromised. The company that produces the device is responsible for generating and storing k securely. Afterwards, they will use k to make the attestation.

The attestation token is transmitted following the protocol specification. This transmission occurs over a network and is susceptible to attacks. If this occurs, the **prover** will send a token α and the **challenger** will receive one modified token α'. In this case, the **challenger** will identify the **prover** as a violated device (in an unknown state). Some technologies, such as the TLS, can be used to prevent attacks in this connection. An interesting fact worth mentioning is that an attacker can store an old authentication token for later use. To prevent this type of attack, when the **challenger** asks for the authentication token it will be sent a nonce n in the request. This nonce will modify the state of the **prover**, resulting in a unique state. The nonce is randomly generated for each request of the authentication token. The nonce n is not specified in the above functions, to better explain the basic idea of the protocol, however, it will be discussed in detail afterwards.

Suppose the **prover** is compromised ,the attacker will know the state s of the device. With this information he can produce two types of attacks: simulate the `Attest` function and return a correct α; or return a α that is not the real state of the device. To prevent these attacks, the `Attest` function in the **prover** has to have the following properties:

- **Exclusive Access**: only the `Attest` function can have access to a key k. This makes it impossible to forge the token α without running the `Attest` function.
- **No Leaks**: after the execution of `Attest` function no information of the key k can leak. Otherwise, the key k will become known by the attacker. Cleaning all memory after the function execution can solve this problem.
- **Immutability**: if `Attest` function can be changed, the attacker can take advantage of this flaw to leak k.
- **Uninterruptedly**: if during the computation of `|Attest|`the device is interrupted, information of k can leak.
- **Controlled Invocation**: the `Attest` function can only be called by its beginning. Otherwise, the attacker can skip important parts in the `Attest` function code, such as the part that disables all interruptions.

The way all these properties are provided depends on the device. Some approaches try to provide all these properties using a software implementation. Unfortunately, remote attestation using only software has been proved not to work in the context of the Internet. Subsection 2.4.1 contains more information about this topic and references related papers.

2.4 Works of Remote Attestation for Embedded Devices

This section has the primary objective to show and discuss the mains papers in Remote Attestation for Embedded Devices. Its first subsection contemplates software remote attestation techniques and show why they have been proven to be insecure on the internet. The following subsections focus on hardware implementations.

2.4.1 Software Remote Attestation

Several articles tried to develop a software-only remote attestation algorithm, without hardware changes. One of the main articles in this area is: *Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems* [17]. The basic idea of this article is to send a nonce to the device and wait for the device to produce an attestation token based on the current state of the device. Similar to the premises for remote attestation, the received and expected token are compared to determine if the device is compromised.

However, it is not possible to ensure all the security requirements related to the key storage. As a solution, the computing time is considered to verify the challenger. The device was probably compromised if it took considerable amount of time to calculate the hash. When the Pioneer makes the memory hash, it takes a random look in the memory based on the nonce it receives. If the attacker copies the memory to another place or tries any action to subvert the attestation, this will increase the computation time considerably. The challenger will notice this difference, and it will suspect that the device has suffered an attack.

Unfortunately, the connection between the **prover** and the **challenger** can vary considerably, making the packet transmission time not to be exact and have considerable variance. These techniques proposed in Pioneer work well when this connection has a constant transmission time.

In this implementation, the **challenger** needs to know accurately how much time the device takes to calculate the attestation token. The only way to do this is testing the attestation challenge in a real device. In a real environment with a server verifying a vast number of devices, this process does not scalable.

One interesting fact is that the Pioneer solution was made to work on various device types. The article implements it in an Intel Pentium IV Xeon processor. Pioneer uses the same ideas previously created in a project called *SWATT: SoftWare-based ATTestation for Embedded Devices* [16]. This article shows another remote attestation technique. The idea in those implementations is similar, but the main difference is that SWATT works only in embedded devices with simple CPU architectures, such as micro-controllers.

Some works have shown failures in time based remote attestation systems. The article [5] shows how to forge the SWATT in implementation. Adding simple instructions and reordering the existing ones in the verification source they can forge the authentication token without increasing the computation time significantly. In the conclusion of this article, authors declare that software-based attestation is challenging to design, but not impossible. The main reason for this assertion is the fact that small changes on the verification code can reduce the verification time and cheat the verifier.

The article [12] has tested and simulated attacks to find out the efficiency of the software-based attestation. They showed that depending on the implementation, the attack could be noticed when the **prover** is not connected directly with the **challenger**. They also clarified that the majority of generic attacks against timing-based attestation systems are TOCTOU (time of check to time of use) attacks. This class of attacks involves the race condition between checking one condition and using what this condition provides. However,

in the end, they conclude that the timing-based remote attestation is in its infancy.

As seen, the software remote attestations have several problems. It assumes guarantees that are complicated to exist when involving real embed devices on the internet. To make the remote attestation viable was concluded that hardware change is needed.

2.4.2 SMART

The primary objective of this research is to implement and test the hardware and ideas proposed in *SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust* [6]. This article talks about the minimum hardware requirements for embedded devices to make remote attestation viable.

The basic idea in the article is to change the memory access to get the exclusive access to the key in a ROM memory (read-only memory). Changing it, they made the key only accessible when the device program counter points to the first address of the `Attest`. Other changes have also been made to prevent the `Attest` to be executed from its middle. The memory cleaning, to prevent the leak of the key, is implemented as part of the `Attest` function. The uninterruptedly is guaranteed by a software instruction to disable all interruptions at the beginning of the `Attest` function.

The `Attest` function in the device is made using a software SHA-1 hash implementation. It concatenates the device memory, and the nonce sent by the **challenger** and obtain a hash that represents the device state. Also, the function can look only to one part of the memory, and this makes possible the attestation of single parts of the device in a relatively short time.

Another functionality that has been added to the `Attest` function is the capability to the **challenger** to send a function address to be called at the end of the SMART code execution. This feature makes possible to attest one memory region and, afterward, call its code in a safety way. The article shows the benefits of this in two examples: the first is to attest the reading of measurements and the second is to prove if the device has been successfully reset.

To test the functionality of the idea, they have implemented the SMART modifications in two different micro-controllers units: the Atmel AVR and the Texas Instruments MSP430. According to the article, adding SMART to both chips caused only a 10% increase in their respective surface areas. That is, adding the SMART hardware do not increase the cost of the device significantly.

All the results from the article are obtained from simulations. The authors did not mention that they have tested the SMART code in FPGAs or ASIC devices. Also, they affirm that more experiments using current implementations need to be performed for better overhead evaluation.

The SMART implementation takes several assumptions. Among them is that the attacker has full control of the device program. It can change it or call other functions. Note that the program can be safely saved in read-only memory, but this does not prevent this type of attacks. Some attacks, such as a technique called *Return-Oriented Programming* (ROP) [18], only change the return pointer in the data memory and reuse the program code in a different memory alignment to perform the attack. Therefore, in the article, the authors assumed that the attacker has full access and can modify the code outside the protected memory region of SMART.

Another critical assumption is that the device will not suffer any hardware attacks. This assumption is important because if it is possible, the attacker can, for example, reset the device every time the SMART code is called. Making this successively, extra information of

the key can leak and then the attacker can obtain the key.

Although carefully designed, some flaws were found in the SMART project [7]. In SMART, all parameters sent by the **challenger** can be changed by the attacker. If this happens, the adversary can make the callable function be inside the smart ROM, making an infinite loop, making a very effective denial-of-service attack without actually controlling the device. This problem is not related in the SMART article but can be simply solved by checking if the end function address is inside de smart ROM before going to it.

The second flaw is related to the fact that the SMART authors argued that the interruption disable is optional. However, depending on the device's interrupt handler the attacker can cause the leak of the key. Disabling the device interrupts during the SMART code execution is an option to solve this problem.

2.4.3 TrustLite

TrustLite [11] is another remote attestation project. It uses SMART as inspiration but has several improvements. SMART only supports one protected area in the ROM memory, while in contrast, TrustLite created a Memory Protection Unit (MPU). The MPU is dedicated hardware that contains a table with information about several protected memory regions and the address of functions that can access it. This hardware unit stays between the CPU core and the address space (including memory and the device peripherals).

Some advantage of this implementation is the existence of several trustful applications. These applications are named *trustlets* by the authors. Each *trustlets* is isolated in the sense that no other software on the platform can modify their code. *Trustlets* data can only be read or modified by other *trustlets* according to the policy described in the MPU. One *trustlets* can contain the `Attest` function inside it, making it possible answers attestation requests. That is, the TrustLite can support remote attestation.

To load the memory access policy to the MPU, authors developed a *Secure Loader*. This loader works like a *trustlets*, but it is the first to be loaded, and it is in a known region of the memory. It is responsible for loading the other *trustlets*. Another important thing about this loader is the fact that it cleans memory before loading any other code. This action solves the information leakage on the platform reset, providing the root of trust. The *Secure Loader* can also be remotely updated.

In SMART, when on memory violation occurs the hardware makes a reset. TrustLite constructs a mechanism to handle memory access violation. On any violation, the *trustlets* invalidate the instruction responsible for it. This functionality can prevent an attacker to make the device reset in a critical situation.

TrustLite can also have insecure applications. These applications and the *trustlets* can exchange information, creating an interprocess communication. This information exchanged can be used, for example, to make one untrusted application (such as an input and output control) to be attested.

The TrustLite was tested in an FPGA. The testers used an Intel Siskiyou Peak processor to implement it inside a Xilinx Virtex-6 FPGA. The evaluation of the hardware extension cost was made based on the number of the FPGA registers and LUTs (more information of LUTs and FPGA can be seen in the Section 2.5). According to the article, the cost per module was approximately 8% in the number of registers and 4% in the number of LUTs. However, the base core, without any modifications, uses 552 registers a 14361 LUTs. This total number of LUTS is much higher if compared to the 1810 LUTs used in the openMSP430 in the same FPGA. That is, the 622 LUTs used to implement the TrustLite is approximately one-third of the openMSP430 size. This comparison is not exact, because small changes in

the hardware description language or the FPGA syntax settings can produce a significant change in the total number of LUTs. Despite, it is still possible to note that TrustLite costs more than SMART.

In this research, it was chosen to implement the SMART because it contains all necessary mechanisms to make the remote attestation viable. Besides that, suppose we want to have multiple applications running in a device, so it will be only necessary to attest the code responsible for loading these apps. If this code is trustworthy, then every code loaded by it will also be reliable. Note that this will be not possible in a dynamic root of trust context.

2.4.4 SANCUS

Unlike the two projects mentioned above, SANCUS 2.0 (*A Security Architecture for Low-Cost Networked Embedded Devices*) [13] project was created to attend a network of devices. The main idea of this project is to provide some capabilities, such as software module isolation and remote attestation, in a network of interconnected devices.

Like the other projects, the SANCUS does not use asymmetric cryptography, because of the high cost of implementing it. However, to provide security capabilities, the microprocessor was changed internally to add new special instructions. This kind of changes significantly impacts on the cost of the final device. For example, the remote attestation features were implemented by extending the processor with two more instructions.

Like TrustLite, in SANCUS is possible to load multiple security application inside the device considering each application with its area in memory. Each application has a single key, but this key is not accessible directly, it only can be used by the processor when particular functions are called, the same way as in SMART. In SANCUS, every application needs to have special hardware related to it. This hardware is responsible for saving the application key, the address of the application code, and the data. Also, it has Memory Access Logic (MAL), the hardware that is used to enforce the memory access rules for a single application and to produce a violation signal if something unexpected happens.

Differently, from the other projects, the area needed to implement SANCUS is given based on the maximum number of the security application that it can support. The hardware design is evaluated using two different types of logic gates and three different key sizes. However, in short, we have for SANCUS, with only one application, an average increase in hardware size of 90%.

Since the article made changes to the CPU core, it also evaluates the maximum frequency supported by the new micro-controller. The maximum frequency needed to be decreased when increasing the number of security applications. According to the authors, this happens due to the large multiplexer needed to get the module key out of the MAL circuits. Additionally, the maximum frequency decreases much faster when using a larger key size, because of the same factor quoted.

All experiments conducted by the authors are made using an XC6SLX25 Spartan-6 FPGA with a speed grade of 2, synthesized using Xilinx ISE Design Suite. This family of FPGAs is the same used in the tests in this research.

2.5 FPGA

FPGA is an abbreviation for "field-programmable gate array". This hardware is a particular type of integrated circuit created in the 80s. The main difference to others integrated circuits is the possibility to reprogram the circuit using HDL (hardware description language).

The major benefits are that it makes possible to test new hardware and circuits without the need to produce a new integrated circuit. One of the main tasks in this work is to change the memory backbone of one micro-controller and test the results. Using the FPGA facilitated the assignment to test the modifications due to its reprogramming interface.

There are two main HDL languages used in the market: Verilog and VHDL. These languages are differences from conventional programming languages. The most noticeable difference is the fact that everything is happening in parallel. In these languages, variables are replaced by registers and connected with wires. There are some attempts to convert following programming languages, like C, in hardware description languages, like the LegUp High-Level Synthesis project [3]. However, transforming a sequential program to run in one FPGA, typically, is not efficient and requires more resources of the FPGA that what is needed.

An approach used in most FPGA projects is to program a small processor inside it. This processor can be programmed using a sequential programming language. The good thing about this design is the possibility to connect dedicated hardware in the processor to make specifics tasks. Some companies that produce FPGAs have built optimized processors for their products. One of the most famous projects is MicroBlaze, a 32-bit RISC microprocessor, designed by Xilinx. A Linux kernel implementation has been made for this processor, making possible to run a Linux inside an FPGA.

Another frequent topic related to FPGAs is the LookUp Tables (LUTs). A LUT is a hardware truth table. That is, depending on its input there is a specific logic output. One important FPGA characteristic is its number of LUTs. The LUTs can be classified as the number of accepted inputs. For example, there is 4-inputs LUTs and 6-inputs LUTs. The k-input LUT means that it has k configuration pins, making possible a truth table with 2^k inputs.

The capability of saving temporary information is essential for FPGAs. The registers (commonly implemented as flip-flops) are responsible for this property.

2.6 FPGA Testing and Programming Steps

In this work, we will build a microcontroller from its hardware description and program it. This process involves different abstraction layers, understanding all its steps is crucial to comprehend this work. Figure 2.2 presents all the steps and layers in the process of testing and building a microcontroller inside an FPGA.

Initially, we have the hardware description language (HDL), responsible for describing all parts of the microcontroller. In the right part of the Figure 2.2, inside a grey box, it is shown the hardware description language. The primary objective is to transform the HDL source to be loaded in the FPGA. Like a computer program, this language syntax needs to be verified. The synthesis process will check code syntax and analyze the hierarchy of the source design (usually multiple files and modules compose a device).

After the synthesis check, we build NGC files that contain both logical design data and constraints. NGC files represent the hardware description using only primary elements, such as logical ports. This file is also known as netlist file. NGC is only a file format created by Xilinx and has no specific meaning, the official description of NGC in the Xilinx developer resources[2] is "Netlist file with constraint information".

The Translate process merges all of the input netlists and design constraints. This process outputs a Xilinx native generic database (NGD) file, which describes the logical design

[2]Xilinx Command Line Tools User Guide https://goo.gl/crxWL6

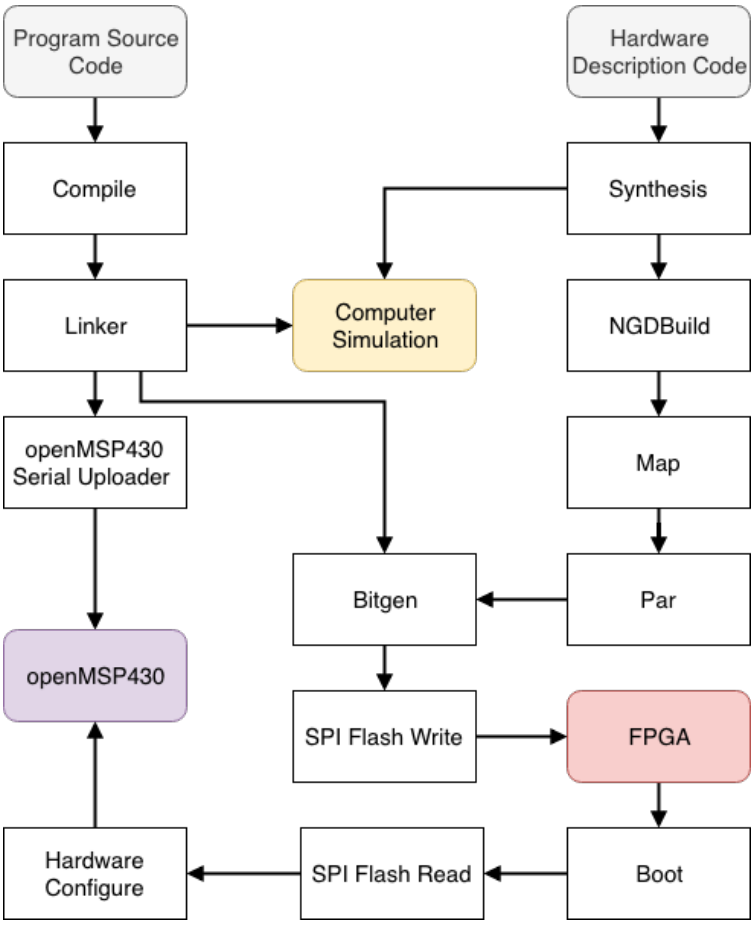

Figure 2.2: *This figure shows the building and loading steps needed to simulate and run the hardware description in one FPGA and the source C on microcontroller formed by it. The gray boxes represent a program source. The yellow one represents the simulation programs. The red one is the hardware FPGA, and the purple is the microcontroller running in an FPGA. All the with boxes are steps taken by scripts during the development process.*

reduced to Xilinx primitives. The Map process maps the logic defined by an NGD file into FPGA elements. The output design is a native circuit description (NCD) file that physically represents the design mapped to the components in the FPGA.

The Place and Route (PAR) process takes a mapped NCD file, places and routes the design, and produces an NCD file that is used as input for bitstream generation.

The Bitgen process produces a bitstream for the FPGA device configuration (bin file). This file can be sent to de device flash memory. This update is done via a USB serial cable and is in the diagram as SPI Flash write. After that, when the FPGA boots it will read its flash memory, configure its internal gates (Hardware Configuration) and at the end it will be a programmed micro-controller. In this research case, the openMSP430.

On the left side of the Figure 2.2 there is the responsible component for managing the source code in C. Initially there is a C code, and we want to transform it into an executable file. For this, there was a need to compile the code, place the assembly generated in specific locations and send it to the device via a serial connection. One interesting thing is the fact that the executable code can be merged in the FPGA bitstream. In this case, the when the FPGA loads the bitstream it will also initialize its memory blocks.

The last unexplained block is the Computer Simulation. This block is responsible for simulating de hardware and testing its results. These tests usually take a long time, because it sequentially simulates all the hardware parallelism.

The layer and this diagram can change depending on the tools used in the process. In Figure 2.2, we have the specific process for the Spartan 6 FPGA family using it manufacture (Xilinx) tools. All FPGA building steps description are based in the Xilinx documentation[3].

[3] https://www.xilinx.com/support.html#documentation

Chapter 3

Implementing SMART

As has been said, the SMART implementation is one of the main purposes of this undergraduate thesis. In this chapter, more details of the SMART project will be presented and all information about how it was implemented.

A Github repository[1] contains all the source code used during the implementation. Additionally, there is instruction for how to build and test the project.

3.1 Premises

In the main article [6], a few assertions are made to allow the proof of the SMART security. These assertions were used to drive the construction of the hardware that implements SMART. Because of that lets present the assertions and discuss some of that:

- A1: is impossible to forge the hash value. One significant difference from the main article is the fact that the hash is computed using special hardware and not software code. This hardware is designed to receive a data chunk and digest it to make the hash value. There is a guarantee on a hardware level to make impossible to recover the data sent to the hash computation.
- A2: the device with SMART will not suffer any hardware attack, only software.
- A3: only the SMART code can access the SMART key.
- A4: the smart code is immutable. Nobody can change it.
- A5: SMART code can only be called from the first memory address.
- A6: if the SMART code moves the instruction pointer from outside its region, it is impossible to return to the previous code. This assertion is a little different from the original but provides the same guarantees.
- A7: all interrupts are turned off during the SMART code execution.
- A8: after the SMART code execution the SMART key cannot be recovered.
- A9: the SMART code compute the hash correctly after receiving a challenge.
- A10: after a reset, erase all memory data segment. This procedure prevents the leak of any information remaining in the memory.

[1] https://github.com/capellaresumo/MAC0499

- A11: none information from the SMART key can be a leak.

The use of the hardware implementation of the SHA256 algorithm guarantees the A1 assertion, more information at subsection 3.2.4.

The assertion A2 is part of the adversary model. As said before, we will assume that the adversary has two main capabilities. The first, he can manipulate all the software in the microcontroller. That is, he can deploy a malicious software into the module. The second, we assume attackers can control the communication network that is used by the **challenger** and the **prover** to communicate with each other. In this model, the attacker does not perform any hardware attacks on the **prover** and any attack in the **challenger**. Note that denial-of-service attack (DoS attack) are part of in this model, but will not be part of this work because there is specific bibliography for this.

For assertions A3, A5 and A6 a specific module, called SMART Memory Access Control (SMART MAC), was designed. More information about this module and how it ensures these items are specified in subsection 3.2.5.

ROM memory saves all the SMART code and the key used by it, ensuring that their content is immutable (A4).

The microcontroller used in the implementation has a particular function (`DINT`) responsible for disabling all interrupts (A7). A specification of the microcontroller and a link to the supported assembly function is in the subsection 3.2.1.

The way the SMART has programmed guarantee the assertions A8 and A11. All register and memory space used in the SMART execution are cleaned (all e values are set to zero) after it uses. Besides that, the hash computing hardware was not designed to permit any recovery of data sent to it.

We suppose the hash algorithm is correct and it correctly computes the hash (A9). The hash code source has several tests to prove this assumption.

The assertion A10 is guaranteed by including the internal structure of the microcontroller. When it initializes, all mutable variables are copied from the ROM to the RAN, removing any remaining information. When the compiler builds the assembly from the source code, the compiler inserts this behavior in the code. In the adversary model, this behavior can be inhibited, not guaranteeing the cleaning of the memory. This problem can be bypassed with a safety reset, as described in the SMART article [6].

3.2 Implementation Details

One of the main objectives of this work is to implement the SMART as described in the article. The authors made two implementations of SMART, one in an Atmel AVR microcontroller and other in a Texas Instruments MSP430. In this work, we will only implement SMART in the MSP430 microcontroller. However the implementation focus on changing the memory bus, making it easily portable to other microcontrollers.

This section will describe the devices, softwares and hardware parts used to make the SMART implementation.

3.2.1 MSP430

The MSP430 is a family of microcontrollers created by the Texas Instruments. All of them have the same set of processor instructions[2]. What makes them different is their memory

[2]MSP430 Family Instruction Set Summary: https://www.ti.com/sc/docs/products/micro/msp430/userguid/as_5.pdf

size, number of GPIO pins and the integrated peripherals in the chip.

The MSP430 uses a 16-bit CPU, unlike most low-cost microcontrollers that use only an 8-bit CPU. However, curiously, instruction simple arithmetic operations, like multiplication, are not present in the MSP430 arithmetic logic unit (ALU). As a solution to this, a peripheral with this function is usually connected to the chip. Another advantage of having a 16-bit CPU is the largest memory address range. It can map the memory in 2^{16} distinct bytes.

One of the most significant functions of the MSP430 is the way it uses this memory map. The program, data, and peripherals memory are mapped in a single range of addresses. The figure 3.1 shows how the memory works.

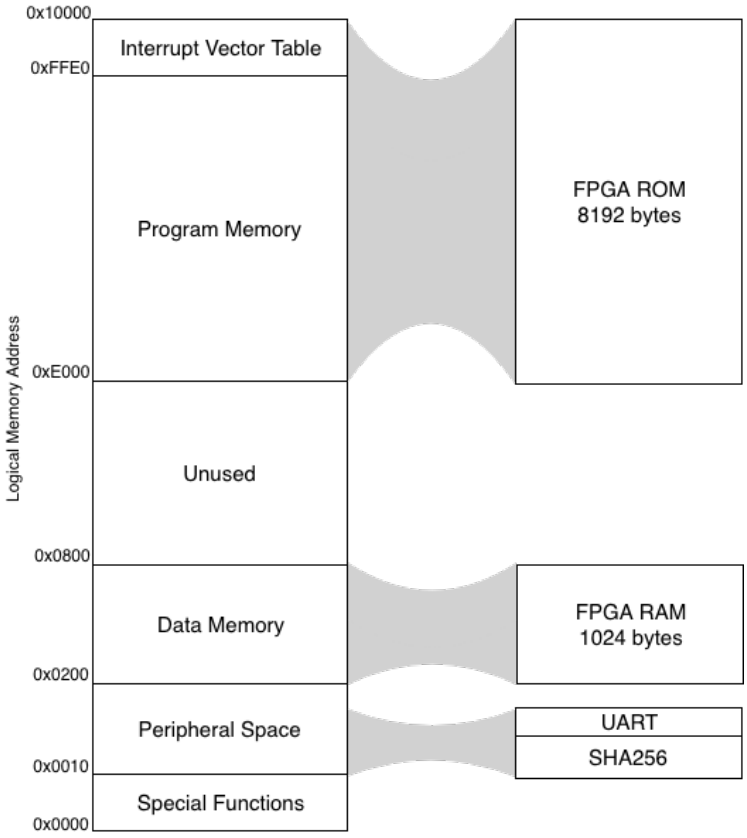

Figure 3.1: *Memory map using a ROM with 8192 bytes and RAM with 1024 bytes. Also, there are two peripheral, one responsible for controlling the serial communication (UART, universal asynchronous receiver/transmitter) and other to calculate SHA256 hashes.*

The scheme showed in the picture are almost the basic MSP430 memory map. The unique difference is the addition of two peripherals to made particular tasks. Using the peripheral directed attached to the memory make easy write and read values from it. Peripherals can be physically added to the microcontroller or can be built into the same integrated circuit.

When we talk about memory, we have two types of addresses: logical and the physical address. The logical is used by the program to refer to the memory area that it wants to access. The physical one is the address of the device memory. The distinct of this two types is important because depending on the situation will be a reference to one type or other.

Make a clear difference between this types, let's give an example. Suppose a program wants to write an integer value (2-bytes) in the address physical address $0x0200$. When the processor runs this instruction, it will calculate the physical address and output this value in the data memory address bus to communicate with the memory. In this example, the value of the physical address will be 0x000 (LOGICAL_ADDRESS $= 2 \times$ PHYSICAL_ADDRESS $+$

0x200). The number 0x200. The number 0x200 is a default value used by the MSP430 family.

How the MSP430 and its functions works are well known, but how these things are implemented on a hardware level is an industry secret. However, Olivier Girard created an open implementation of the MSP430 in Verilog and made it public under the GNU Lesser General Public License. This project was named openMSP430. This work will use this open version to make changes and implement SMART.

Already has been said that the main difference of models in the MSP430 family is the RAM and ROM memory size. Using the openMSP430 this two value are fully configurable, that is, using the same code is possible to simulate multiple devices. To make things simple, all tests will use a ROM with 8192 bytes and RAM with 1024 bytes, like shown in figure 3.1. These two sizes are chosen intrinsically, but they are enough to save and execute programs with reasonable complexity.

To build the binary code to run in the openMSP430 Texas Instruments use a derived version of GCC[3]. This compiler accepts flags to describe the target device. Also, it supports a memory linker scripts to build the executable to devices that use the MSP430 instruction, but do not correspond to any device of the family.

This compiler additionally came with the MSP430 header files. Some crucial tasks are the responsibility of these files. For example, when we declare a string (array of chars), this data will be saved in the ROM memory (the only memory that will be preserved between device resets). If the microcontroller program has an instruction to change one byte of this string, this cannot be done in the ROM. To solve this problem, the compiler inserts a code to be run when the device starts. This code is responsible for copying all initialized variables to the RAM. Also, the compiler changes the reference of these variables to point it to the RAM.

There are many other specifications of this microcontroller. The main ones were mentioned here or will have a subsection discussing it. Other specifications are outside the scope of this work.

3.2.2 Development Board and SPARTAN 6

A Mimas V2 Spartan 6 FPGA development board is used to test and develop the SMART implementation. This development board has several components to make possible its use in different scenarios. Its manufacture website[4] has a list of all the board components and specification. Here we will discuss some of them.

The SPI flash memory of 16 Mbytes (M25P16) is responsible for saving the Bitgen file, as explained in the diagram. Unfortunately, the programming of this memory is made using a PIC microcontroller that interfaces it with the USB. Because this type of connection and data conversion, there is a significant delay in writing the Bitgen to this memory.

This PIC microcontroller also serves as a USB-UART interface, which helps to establish the communication between the code in the FPGA and any application running on the PC. Data can be sent and received from the FPGA by using a serial terminal at the fixed of baud rate 19200. This interface will be connected to the UART openMSP430 peripheral, making possible to exchange information between the computer and the microcontroller.

Because the PIC can have two different functions, there is a switch to select the required function.

Some LEDs, buttons, and switches are present in the development board. The switches are used in the hardware tests to help to turn on/off some modules and peripherals. The

[3] http://www.ti.com/tool/msp430-gcc-opensource

[4] Numato Website: http://www.ti.com/tool/msp430-gcc-opensource

LEDs are used to show the internal status of the microcontroller. Moreover, one button is used to be the openMSP430 reset pin.

Despite all the components, the principal one is the Xilinx Spartan 6 XC6SLX9 in a CSG324 package with a speed grade of 2. This component as pins connected to all the other devices. Inside it, also has some pre-built components (these components need to be connected using a specific HDL code), like a RAM. This model of FGA has 576 Kbytes of RAM. This memory will be used in the implementation to supply the ROM, and RAM needs by the openMSP430. Note that we will use a RAM as a ROM, this not implies in any security fault, because the memory openMSP430 memory backbone prevents any the ROM to be writable.

A considerable time amount of this work is spent adapting the openMSP430 implementation to run in the Spartan 6 and this development board. Several times all computer simulating tests in the run without any errors, but when tried in using a real FPGA the tests stopped to work. Unfortunately, to discovery and debug these problems are a difficult task because it involves real hardware. It was necessary to use an oscilloscope several times.

3.2.3 Time Constants

One important thing when working with FPGA is the time constants. These values are essential for testing, and the allocation of the connections inside the FPGA.

For Verilog simulations, the time constants are set using a reserved macro named `timescale`. This macro receives two values: the time unit and time precision. The simulations delays and any time value use this time unit. So the time precision is used by the simulator to know how they can round the summation values. In the tests we used $1ns$ for time unit and $100ps$ for precision.

```
1 initial
  begin
3 CLK_100MHz = 1'b0;
  forever #5 CLK_100MHz <= ~CLK_100MHz; // 100 MHz
5 end
```

Listing 3.1: *Simulation clock signal generator.*

Code 3.1 is used in simulation to generate the clock signal. It changes the value of wire `CLK_100MHz` in intervals of $5ns$ (5 units of time), making an output signal of $100MHz$. We choose the frequency of $100MHz$ because the testing FPGA has a clock of this frequency. These values are calculated using the frequency formula $F = \frac{1}{T}$, where $F = 100MHz$ is the desired frequency and T is the period. After solving this equation $T = 10^{-8}s$, this is equal to 10 units of time (10ns). The FPGA clock uses a duty cycle of 50%, that is that the clock will need to be 50% of their time active and the other part inactive (the signal will change to high to low or vice-versa in an interval of 5 units of time).

The communication with the simulator in testing or the computer in a real FPGA is made using the RS-232 standard. This standard is a serial protocol. In there the bits are sent one at a time. A direct consequence is that the protocol use time constraints to send the information. In all test is used the 19200 bit/s baud rate, making each bit transmission time be $52100ns$.

To reduce the timing conflicts and problems inside the openMP430 [4] the microcontroller will receive a clock input of $20000MHz$. To change from $100000MHz$ to the desired input speed, a Digital Clock Manager (DCM) [1] will be used. The DCM is a dedicated hardware inside the FPGA for clock frequency conversion. To this module works it needs several

configuration parameters, to simplify it a module named `clock.v` was created with all configurations inside it.

The microcontroller uses one hardware peripheral to interface with the RS-232 serial port. This hardware need to know the bit transmission time, in all software that uses serial has the `UART_BAUD = BAUD;` line. This line is responsible for setting a unique memory address to the correct bit transmission time.

3.2.4 SHA256

One improvement in this implementation compared to the original article was the use of a hardware SHA2 (Secure Hash Algorithm 2) to hash the memory. The original implementation uses a software SHA1. Different from SHA1, SHA2 is a family of hash function, in implementation we used the SHA256 function.

The SHA256 has some improvements compared to the SHA1. The first one has the input size, SHA1 needs to receive only 160 bits to produce a hash, the SHA256 needs 256 bits. SHA1 was deprecated by NIST (National Institute of Standards and Technology) in 2011. There also some articles, like [19], to show how an attacker can forge two distinct PDF documents with the same SHA-1 hash.

It is important to notice there exists a new version of the secure hash algorithm, the SHA3. This version made some improvements and NIST advise to use it. We tried to implement the SHA3 in the FPGA, but it exceeds the number of available LUTs in the FPGA, making the implementation impossible with the openMSP430 core. However, SHA256 is still considered secure, and it fits inside the FPGA.

A SHA256 implementation written by Joachim Strombergson [2] has used. A peripheral adaptor was built to communicate with the openMSP430 core. This adaptor is responsible for interfacing the SHA256 implementation with the peripherals pins in the microcontroller. The file `SMART/rtl/verilog/sha256/sha256per.v` contains this interface.

This change makes the hash timing faster and saves several bytes in ROM. However, it uses approximately 2000 FPGA LUTs.

3.2.5 SMART Memory Access Control

A specific module was developed to build one hardware that guarantees the A3, A5, and A6 assertions. We give the name of SMART Memory Access Control (SMART MAC) to this module. It usually needs to be between the microcontroller and the memory that we want to protect.

To allow the full comprehension of this module and its capabilities, table 3.1 show all hardware pins it has and the table 3.2 shows all parameters.

This basic module idea is to monitor the current instruction pointer address and depending on its position allow the access to a specific memory region. We will call this area as the protected memory region. In case of any attempt to access this region when the instruction pointer is in an invalid position, the module will send a reset signal to the microcontroller.

Most specifically, the protected memory region will only be readable when an internal module register is activated. This activation happens when the instruction pointer points to the first instruction of a specific function (the LOW_CODE and the HIGH_CODE parameters describe the place of this function). After it is activated, the deactivation only occurs if the instruction pointer is outside the function addresses area. The protected memory region is described by the LOW_SAFE and the HIGH_SAFE parameters.

Port Name	Direction	Width	Description
in_safe_area	Output	1 bit	This pin becomes HIGH between the moment the ins_addr point to the HIGH_SAFE address to the moment it exits from the safe code area.
reset	Output	1 bit	This pin becomes HIGH when some violation occurs and a microcontroller a reset is needed.
mem_dout	Output	16 bits	If no violation occurs and all states are correct this value will be equal mem_din. Otherwise will be 0x000.
mem_addr	Input	variable	Memory physical address.
mclk	Input	1 bit	Microcontroller clock.
mem_din	Input	16 bits	Memory data input.
ins_addr	Input	16 bits	Instruction pointer logical address.
disable_debug	Input	1 bit	When this pi is HIGH, the module will be disabled. This feature is used for debug propose.

Table 3.1: *This table describe all the input and output pins in the SMART Memory Access Control (SMART MAC). When using this module with a real FPGA, only the pin in_ safe_ area can be not connected to the circuit. To make the module work uninterrupted the pin disable_ debug can be continuously set to zero.*

Parameter Name	Description
SIZE_MEM_ADDR	width of mem_addr
LOW_SAFE	The lower physical address in the region's memory to be protected.
HIGH_SAFE	The higher physical address in the region's memory to be protected.
LOW_CODE	Where the function that will access the protected region in memory begins. Virtual address.
HIGH_CODE	Where the function that will access the protected region in memory ends. Virtual address.

Table 3.2: *When building a module using Verilog is possible to insert some parameters. This table is the list of all possible parameters and it description of the SMART Memory Access Control (SMART MAC) module.*

A single SMART MAC can be used in the memory bus to make the A3 assertion valid. In this case, the memory protected area will be the key and the allowed function to read it will be the smart function.

To assertion A5 be valid a SMART MAC is used differently. Suppose the smart function started it code in X and ends in the Y logical memory address. A module will be built with: LOW_SAFE $= X' + 2$ byte, HIGH_SAFE $= Y'$, LOW_CODE $= X$ and HIGH_CODE $= X$. X' and Y' are the physical memory address derived from the logical addresses. We add two bytes to the physical memory address of the SMART function because it is outside the protected memory region (the first instruction is not protected).

Also, assertion A6 is guaranteed by this second module, because if the instruction pointer goes outside the function memory region, the function can only be callable from its beginning. If someone tries to call this function by its middle, a reset signal will be produced.

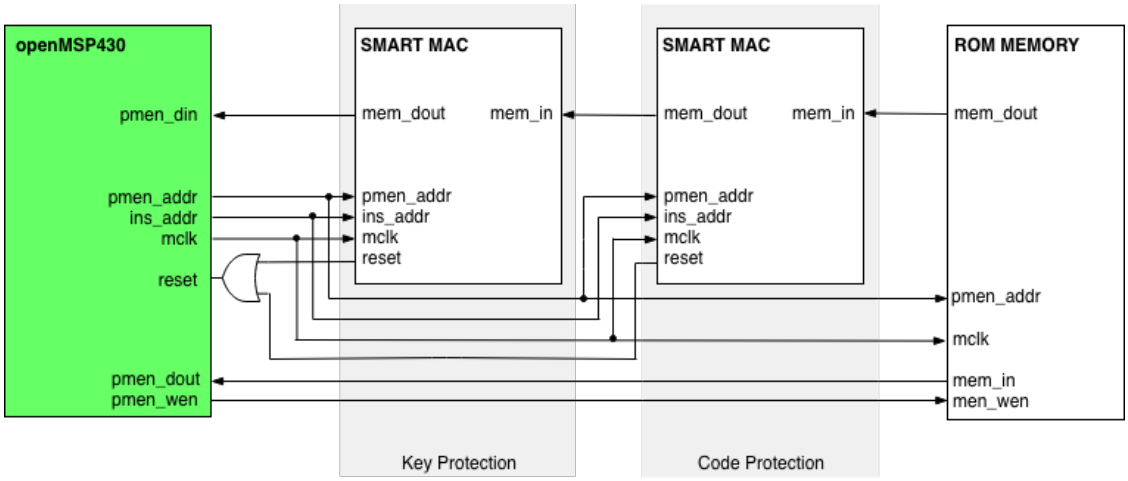

Figure 3.2: *The interface between the ROM memory and the openMSP430. In the middle, is SMART memory access control module.*

By using two SMART MAC it is possible to grant the three remaining assertions. In figure 3.2 is possible to see how these modules are placed between the microcontroller and the memory. Note that the left module is responsible for protecting the key and the right one to protect the code.

The original SMART article changes the openMSP430 memory backbone to validate the A3, A5, and A6 assertions. With the creation of SMART MAC, there was a much less invasive modification the microcontroller and produce the same guarantees. Besides that, the use of a module that does not make changes in the core elements of microcontroller make the solution more portable to other platforms.

Appendix A make a deeper analysis of SMART MAC. Also, it contains the module core code[5].

3.2.6 Linker Script

When building the binary source to a microcontroller the compiler needs to know where to place the code, data and other pieces of information in the memory. The linker script (or linker command file) is the file that describes to the compiler where to place this information.

[5]The SMART MAC source file is in the *SMART/rtl/verilog/smart_mac.v* directory of this thesis repository. The implementation of the connections described in figure 3.2 is present in the *SMART/rtl/verilog/openMSP430_fpga.v* directory.

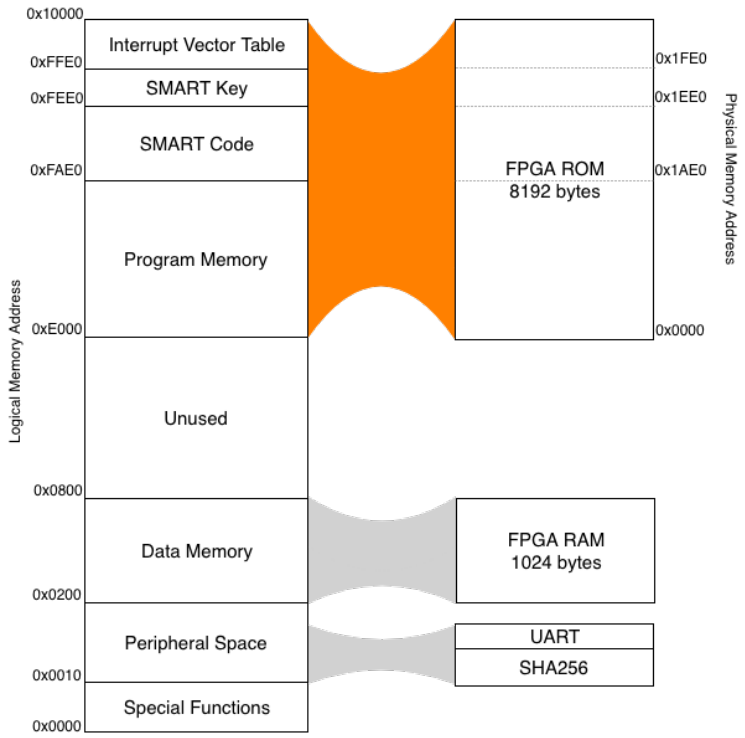

Figure 3.3: *The orange link is the program memory backbone where the smart modules are introduced. Differently, from figure X, this image includes de physical and logical addresses used by the SMART code and the SMART key.*

In figure 3.1 it is possible to see where the SMART MAC was implemented. Also, this image contains information about the new memory sections created for salving the SMART key and code. It was not quoted before, but the image has a special section named *Interrupt Vector Table*. This memory region has 16 processor addresses (the equivalent of 32 bytes) and in there is functions addresses to be callable when one interruption occurs. The first 16-bit word in this section (address 0xFFFE) is the instruction that the processor will jump and a reset is performed. There is an importance to do not block this memory sections because it provides primordial functionalities to the microcontroller.

Shortly after this section of memory we introduce a memory protect region to save the SMART key. The linker script described this section, that way the compiler will not introduce any code in there. A particular attribute was used in the C source code to force the key to be kept in that region. The SMART key memory region starts at byte 0xFEE0 and has a length of 0x0100 bytes.

A region for SMART code is also introduced. It starts at byte 0xFAE0 and has a length of 0x0400 bytes. All the smart code will be saved in this memory section. Because of that, this section can contain multiple functions. Depending on the order the header of this functions are declared the compiler can locate one function after other. As a consequence, the main SMART code function needs to be the first declared. Otherwise, if the SMART main function is not at the first address, its call will not unlock its executable source and will provoke an unexpected reset. That is, in the SMART code, the order of the function header matters.

The linker script can be found in `SMART/software/libs/linker.msp430-elf.x` file. This script was based the original linked script provided by Texas Instruments.

3.3 Tests

This section explains how SMART implementation was tested. The last subsection will show a real application using SMART.

3.3.1 Continuous integration

A continuous integration system is used to prevent and to identify any code error or change that affected the correct work of the microcontroller. A GitHub project is set to track the code changes and versioning the code. On every push, a server receives a notification, run all test in the project and build the FPGA files.

Jenkins[6] is used to build the continues integration system. Other systems, like Travis or Gitlab CI, are not used because they have restrictions on the maximum size of their builds. To test and mount the code the ISE WebPACK Design Software is used, which has 6 GBytes. Jenkins makes possible install this software on the tester machine only one time and uses it when needed.

Two simulation softwares are used to reproduce the behavior of the hardware: the Icarus Verilog[7] (an open-source project) and the Xilinx ISE (a proprietary software). Every test is executed using this two simulation softwares. We made this decision because there are errors that only occur in one of testing softwares.

3.3.2 Light Remote Control

To test and study SMART a full application is built using it. The basic idea was to build a device able to receive remote commands to turn off and on this led with all the security guarantees. As said before, this device was built in an FPGA, that run the openMSP430 project with two SMART MAC modules in the memory bus. As the most IoT devices, it runs as a client on the internet. Because of that, a simple python server was written to answer the requests, save the device key and to calculate the expected hash values.

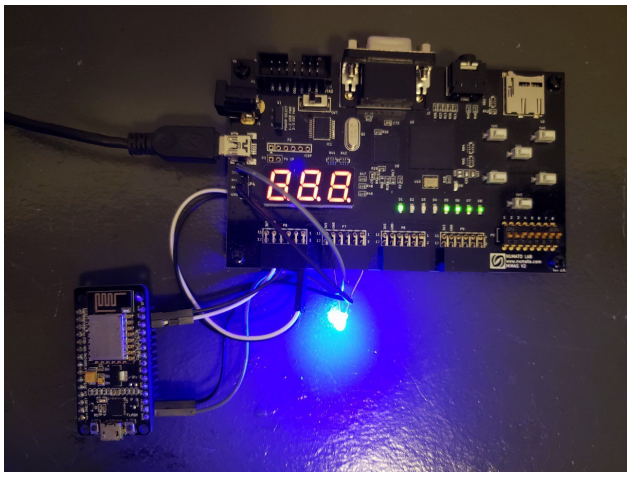

Figure 3.4: *The Numato Mimas V2 (in the right) connected with an Amica Node MCU (in the left) board with the ESP8266 module. A wall charger supplies the device by the USB port.*

In figure 3.4 is possible to see the final hardware prototype. An ESP8266 module was used to make the FPGA able to connect to the internet. A small software was written to this

[6] https://jenkins.io/
[7] http://iverilog.icarus.com/

module control the connection to the internet and among this two devices. It creates a TCP connection between it and the server. All data received by internet is streamed to the FPGA serial connection and, if the FPGA send any byte, the module gets the byte and sent to the TCP connection. The ESP866 module was programmed using the Arduino environment with the Arduino core for ESP8266 WiFi chip[8].

In the implementation this module was not attested. That is, if it suffers any attack it will not be detected. However, it is possible to connect some pins of this module to allow the FPGA to read it internal memory. Making it is possible to create an attestation algorithm to validate this module memory. Note that, this algorithm will be a program in the openMSP430, this program can be attested using the SMART functions.

Because the connection between the two devices is made using the serial protocol, some problems have arisen. There is no attempt to change to other protocol because this protocol is simple to use and we already had an HDL implementation of a peripheral with this protocol. The main problem is that some bytes are getting wrongly when the internet module send it the FPGA. To work around this problem a particular communication protocol was written.

On the other side of the communication channel, there is a Python server responding all the requests. The *socketserver* library was used to implement the server. This library helps the creation of a new process to handle each received connection. This library enables multiple connections at the same time. Unfortunately, the server was hardcoded the device key and the path to its source code. In a real application, each device must have a unique key and a database stores all this data.

The device was responsible for creating the TCP connection, but when performed, only the server can send commands to it, and the device only becomes responsible for responding to the requests. This protocol has only five commands:

- `TAKE_HASH`: used to server request a hash from a memory area. The smart code is responsible for calculating this hash. A 256 bits hash is expected as a response to this command.
- `SET_LED_ON`: turn on the led.
- `SET_LED_OFF`: turn off the led.
- `SET_RESET`: forces a reset in the device.
- `GET_RESET_HASH`: if a reset is successfully made, it will produce a hash from the previous command. This command is responsible for requesting this hash from the device.

As said, a protocol is designed to send this commands. On every server request, 263 bytes are sent to the device. The first byte is the command byte. Each command has a byte associated with it. The device will receive it and call the correct functions after that. The next two bytes (16 bits) are a memory address number that will be used to start the hash computation. Next, there are more two bytes, that represent how many bytes will be part of the hash. The next 258 bytes are part of the attest nonce. These bytes can be divided in three section of 86 bytes. Each section is identical to with each other. This decision was made because if any byte gets wrongly by the device, it can restore the original one.

The device uses a nonce of 256 bytes o compute the hash. Because the serial connection error, an error-correcting algorithm was used. Algorithms like the Hamming code can be

[8]urlhttps://github.com/esp8266/Arduino

able to correct errors with a small overhead in the connection. However, these algorithms are complicated to implement and uses a lot of the memory of the microcontroller (only 5120 bytes are available is the program memory). As a solution, the nonce size has reduced and sent three times. The reduction is necessary the save the data memory. Using this method, became easy to correct any communication error.

Also, to reduce the communication error a low baud rate of 4800 bit/s was used. Another method to reduce this errors would be soldering the internet module in the FPGA, but this would damage the test devices.

For all server commands, the device needs to send a response. However, for only the commands `TAKE_HASH` and `GET_RESET_HASH` this response need to be a valid 256-bit hash. With this hash, the server became able to attest the device remotely. In the device response to the server are not any error prevention measures.

To test the protocol and the SMART functionality a simple communication mock has built. First of all, the server asks for a hash from the address 0x0000 with 0 bytes. Although this command does not appear useful, it cam detected if the correct device is connected with the server. Id other device connected, it will not be able to produce the expected hash. After that a `SET_LED_ON` command is sent. Nest, the server asks for a hash from the led status variable address with 1 byte to verify if this command takes effect. Again, if the value was not the expected one, the hash will be incorrect. In the mock, this verification is made ten times with a small time interval between them. The same procedure is repeated turning off the led.

After that, is requested a hash of the full program memory. The response of this request is critical to see if the device code has not changed. To verify if this function is correctly working some untrusted code was injected into the device and was successfully identified. To make is identification more precisely, the server several requests different memory blocks, following a binary search algorithm. As a result, we obtain the first changed byte.

Subsequent the code verification, a reset signal is sent. The hash computes from the information send by this command is preserve in the hash peripheral, becoming available after the reset. After 5 seconds, the server sends the `GET_RESET_HASH` command to retrieve this hash and verify if a successful reset was made.

In the server and the device are set timeouts if some errors occur. These timeouts are essential because if any the internet connection or the cable between the devices fail, the server and the device will know how to handle this problem.

Two attacks are made to test this implementation. The first one is a man-in-the-middle attack. In this attack every time the command `SET_LED_ON` was sent, the attacker changes the command to `SET_LED_OFF` and vice-versa. As a result, all the verification of led variable value produces an unexpected hash value, proving that the device or the connections have been changed. The attack is made making an ARP spoofing, changing the packets change in a specific port and redirecting the data.

Note that during this procedure, the attacker successfully changes the led status. This problem happens because the device cannot identify the server. One way to mitigate this attack is the device generate a random number and make a hash of it. After that, the device must send this random value and ask for the server the expected hash value. If the server returns the correct value, the device can confirm that is the correct server. Unfortunately, the openMSP430 did not come with a random number generator to make this feature viable. Generate a random number in this microcontroller unit an be part of future work.

The second attack was made by changing the code of the device. When this happens, the function responsible for calculating the hash from the program memory return an incorrect value.

	Number of Slice Registers	Number of Slice LUTs
SMART without SHA256 peripheral	1025	2492
SMART with SHA256 peripheral	2702	4764
MCU with SHA256 and without SMART	2694	4710

Table 3.3: *Use of the FPGA LUTs and registers depending activation of SMART and the SHA256 peripheral.*

Other information about this test and the source code are in the code repository.

3.4 Results

As a result of the tests are obtained table 3.3. This table shows that the SMART implementation in the memory bus uses a small number of LUTs and registers. However, using a hardware implementation of the hash function almost double the necessary hardware. Unfortunately, this is not a good thing, because one of the main objectives is to maintain a lower device cost.

One way to solve this problem is to expand the device program memory size and use a software SHA256 hash implementation. This changes also increase the cost of the device. By the Xilinx Spartan 6 manual, each slice can store 8 bits of information, that is, one byte. The SHA256 software uses approximate 1300 bytes of memory [9]. That is, the cost of using a hardware version will be a little more significant, not 180% as shown in table 3.3.

Another significant result obtained was the hash computation time in the network, that is the device response time after a command to compute a hash. The theoretical computation time from the hash peripheral can be obtained from the hash source code. This value is not so interesting, because it is count in the number of the clock cycles and does not involve any variation. 0.72 seconds was the average time response after a `TAKE_HASH` command in the Light Remote Control experiment. This result was obtained after 1000 measures, using one byte to compute the hash. The experiment was run in a local network, with the computer as a server and the device, both connected by wireless with a 2,4 GHz Wi-Fi (manufacturer Askey and model RTF3505VW-N1).

Another similar test was run. However, instead of asking the hash of one byte, was requested the hash of 8192 bytes. 0.73 seconds is the average obtained. This little change in this scenarios shows that most of the time is used by the communication and the preprocessing computation and not for the hash calculus. Probably this time has significant because of the serial protocol between the FPGA and the wireless module.

[9] This value is obtained running the tool msp430-elf-size in the with and without a software SHA256 implementation. The source for the hash code was obtained in the RFC4634.

Chapter 4

Conclusion

Any work that talks about security needs to be carefully made because if it provides a solution with any flaw, one attacker can study it and use this breach for a malicious propose. This thesis works with a simple solution to provide the remote attestation ability for one embedded device. As said, this involves multiple programming layers and change hardware at a low level. This work made small changes in the hardware layer. However, all of them were carefully made to prevent any error. Tests were written for most of the device changes and codes for the same motive, guarantee the absence of flaws.

During the development, several errors occur. Debugging this errors sometimes is not trivial, because at principle the layers it happens is not known and find where it is a hard task. Also, some tests run correctly in the computer simulations, but when executed in a real FPGA they mysterious fails.

This situation happens two times during the thesis development, consuming a lot of work time. The first one was related to the reset pin. In the computer tests, the buttons pins were set to the LOW state when unpressed. However, the real FPGA set the value to HIGH when unpressed. This small difference produced sequential resets in the device and was discovered only using an oscilloscope.

Another similar error occurs in the tests of SMART code. When it was called, the device simple restart itself. After many tests and hours debugging, was discovered that the connection of the instruction pointer was being made in way correctly by the Verilog language, but not valid to the Xilinx HDL text processor. Because of this stupid error, the software responsible for building the connection of the FPGA had trimmed out the SMART module.

Furthermore, simulation tests that involve the full microcontroller behavior take significant time to be executed. This delay happens because of this kind of tests simulated the full asynchrony serial communication. Also, depending on the size of the microcontroller code or if it was several loops, the instructions decode end execution takes too long to end. This situation happens, for example, when a test with a software SHA256 implementation was run. The test took more than one hour.

Although all the difficulties, the SMART was successfully reimplemented. Everything was made to guarantee SMART premises to be valid and unviolated. Also, the primary objective of this work, to run SMART in a real FPGA, was accomplished, proving that SMART is a simple and an efficient solution to implement the remote attestation capability on embedded devices.

This work has some improvements to be made as a part of future work. Implementing the SMART in others microcontrollers units, like the AVR, is not done. Doing it in future work could prove that the generality of the decision of only changing the memory bus.

Also, as the original article, this thesis not formally verified the SMART implementation

and premises. Some recent work has been done in this area [10]. In this article, a formally verified SMART implementation is suggested.

Another interesting topic of future work is extending the device hardware changes to add more security feature and study how they can be used with SMART. For example, adding a secure random number generator peripheral can add the capability for the device to verify the identity of the remote server. It generates a random sequence of bytes, produces a hash from it and requests the server to send the expected hash of the random sequence. If the server produces the correct hash, the device can confirm that the server has its key.

Hardware improvements can also be made. For example, changing the wireless module can improve the connection speed and reduce the number of communication error. Study the SMART changes necessary to make SMART enable to handle and manage remote updates is also an exciting topic.

Concluding, it was showed that SMART is implemented with ease and its make remote attestation viable for embedded devices.

Appendix A

SMART MAC Analysis

The purpose of this appendix is discussing how the SMART Memory Access Control (SMART MAC) was implemented. Also, some tests will be showed and commented. Below, at code box A.1, is possible to see the core code of the of the module. The module header and parameters initializations are not showed because they do not influence in the module behavior.

Although the small size of the module, it has a reasonable complexity. The microcontroller clock synchronizes all actions taken by the module. This mechanism is important because sometimes, during a fetch, decode or execution of an instruction in the microcontroller, the value of some pins, like the instruction pointer or the memory address, can become a random number. This strange behavior happens because when a 16 bits bus changes its value, the bits are not changed together. This time variation is minimal, but if the module does not use a synchronizing mechanism, an incorrect reset signal can be produced.

```
1  module  smart_mac (
       // OUTPUTs
3      output                     in_safe_area,       // On on safe area
       output                     reset,              // High to reset device
5      output           [15:0]    mem_dout,           // Memory data output

7      // INPUTs
       input  [SIZE_MEM_ADDR:0]   mem_addr,           // Memory adress
9      input                      mclk,               // Memory clock
       input            [15:0]    mem_din,            // Memory data input
11     input            [15:0]    ins_addr,           // Instruction pointer adress
       input                      disable_debug       // Disable protection on HIGH
13 );

15 // PARAMETERs
   //=============================================================================
17 parameter SIZE_MEM_ADDR = 15;                       // size of mem_addr

19 parameter LOW_SAFE      = 200;                     // Low address safe code
   parameter HIGH_SAFE     = 200;                     // High address safe code
21
   parameter LOW_CODE      = 200;                     // Low adress code
23 parameter HIGH_CODE     = 200;                     // High address code

25 // LOGIC
   //=============================================================================
27 reg    inside_code = 1'b0;
   reg    to_be_reset = 1'b0;
```

```
29
wire addr_in_safe = (mem_addr <= HIGH_SAFE) & (mem_addr >= LOW_SAFE);
31 wire pc_in_code = (ins_addr <= HIGH_CODE) & (ins_addr >= LOW_CODE);

33 assign safe_reset = addr_in_safe & ~inside_code;

35 assign reset = to_be_reset & ~disable_debug;

37 assign mem_dout = reset ? 16'b0 : mem_din;
assign in_safe_area = inside_code;
39
always @ (posedge mclk) begin
41     if (ins_addr == LOW_CODE)    inside_code <= 1'b1;
       else if (~pc_in_code)        inside_code <= 1'b0;
43     to_be_reset <= safe_reset;
end
45
endmodule
```

Listing A.1: *Core code of the SMART MAC*

Line 40 is responsible for the synchronizer mechanism. This line tells the hardware to take one action only when the clock signal change from low to high. By the openMSP430 microcontroller documentation, it clock cycle start on the rising edge and ends before the next rising edge. This guarantees that the device state is consistent during the `posedge mclk` signal.

The module implementation uses two registers. The first one is the `to_be_reset`. This register is responsible for synchronizing the reset signal. The output of the reset pin will be the value of this register. Inside the synchronizing area (start at line 40 and ends at 44) is the only place where this register changes its value. Has said before, this mechanism is used to prevent any reset caused by an inconsistent state of the pins connected with the module.

The other register is the `inside_code`. It is used as a memory to save if the instruction pointer was pointed to the first code instruction. Also, it saves if the instruction pointer goes outside the code region.

Figures A.1, A.2, A.3 and A.4 shows some simulations. In the left bar of this images is a list of variables. All these variables are described in table K, with the exception of `CLK_100MHz` that replaces the SMART MAC `mclk` pin. The parameters of the modules in all simulations are: SIZE_MEM_ADDR = 0x04, LOW_SAFE = 0x08, HIGH_SAFE = 0x10, LOW_CODE = 0x18 and HIGH_CODE = 0x20.

The right part of the images is a graph with the value pins value over time. All values changes in the input pins are part of the test. That is, the test consists of changing the inputs and see how the outputs react.

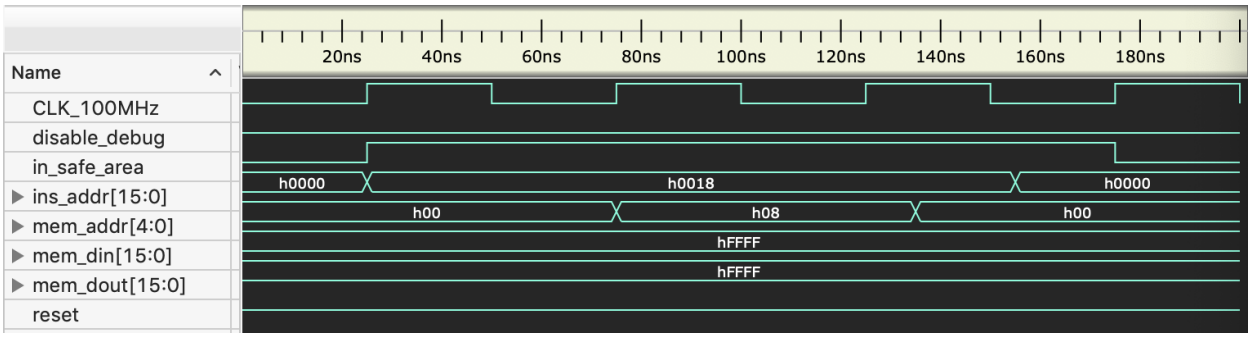

Figure A.1: *Simulation of success access to the protected memory region.*

Figure A.1 shows a simulation that reflects the following steps: the instruction address go to LOW_CODE value; memory address go to a value inside protected region memory, in this case, LOW_SAFE value; memory address go to 0x00; instruction address go to 0x0000. The module work as expected. In the first step, it correctly changes the `in_safe_area` value to high, because it runs the first instruction of the safe code. As there was no memory access violation, no reset signal is triggered.

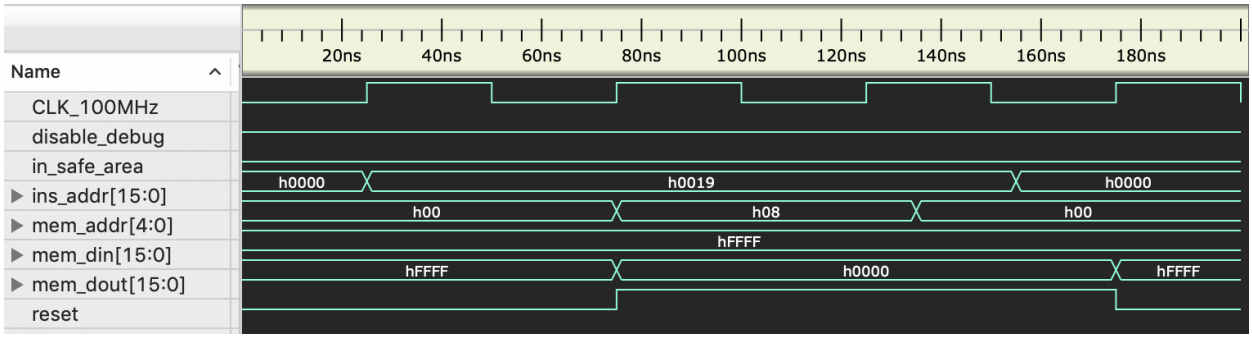

Figure A.2: *Simulation of what happens when the instruction pointer goes to the second code address and access to the protected memory region.*

The only difference from figure figure A.1 and figure A.2 is that in the first step the instruction address go to LOW_CODE+1. Since this value is not the first instruction in safe code area, `in_safe_area` stay with a low signal. In the second step, when there is an attempt to read a protected memory region, a reset signal is triggered. Another important aspect of this simulation is that the `mem_din` pin is a constant value, but `mem_dout` change to 0x0000 when the violation occurs.

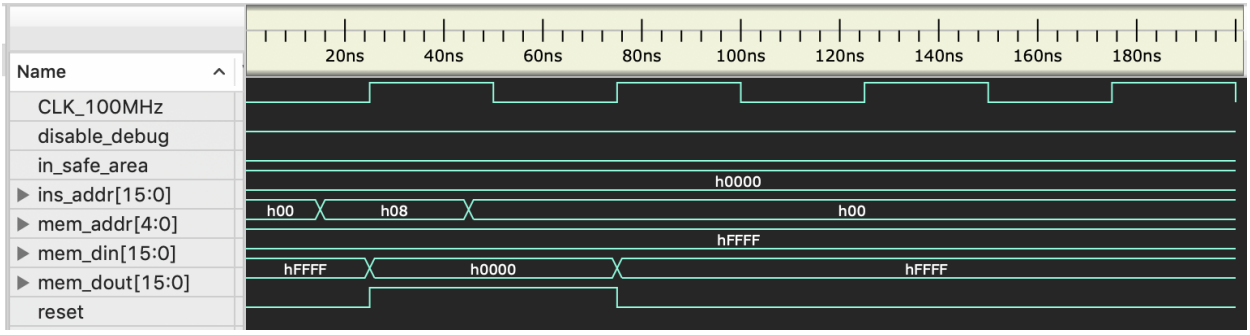

Figure A.3: *Simulation of one not authorized access to the protected memory region.*

Figure A.3 is very similar to previous simulation. It shows a memory access violation.

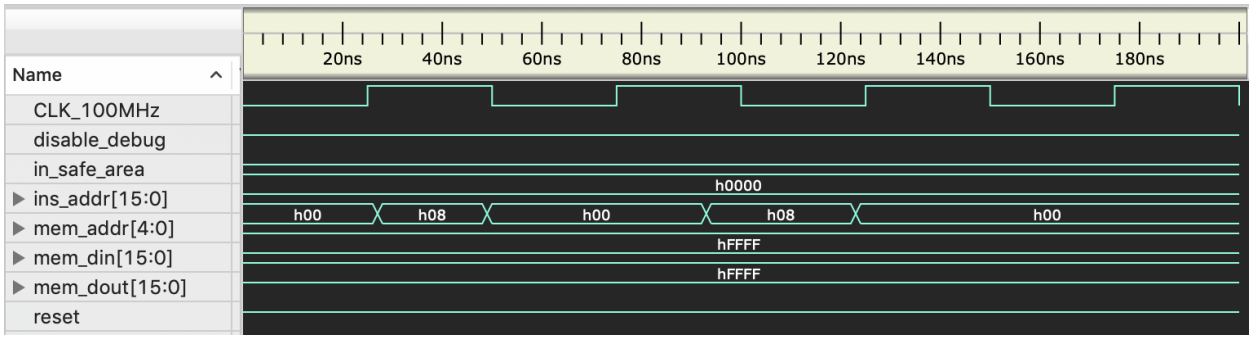

Figure A.4: *Simulation of the behaviour when men_ addr pin inconsistency states happens.*

The figure A.3 aims to show that inconsistent states do not provoke accidental resets. In the image occur two inconsistent states, the first one is the change of `mem_addr` between a half clock cycle (start at 22ns and ends at 48ns). The second one happens in the same pin, and start ate 93 ns and ended at 123ns.

All images from the tests are generated using the Scansion[1] application and the Icarus Verilog vvp runtime engine.

[1] http://www.logicpoet.com/scansion/

Appendix B

File and Directory Description

This thesis involves a lot of files and directories. To makes an easy visualization of all its contents a directory tree was built below[1]. To simplify the integration with other project and to make it easier to understand, the directory structure is based on the OpenCores[2] recommendations, the same used in the openMSP430 microcontroller unit (MCU) project.

FULL_APP	contains files related to light remote control application
PythonServer	
server.py	python script to TCP server to control the application
WiFiClient	
WiFiClient.ino	Arduino (C) code of the ESP8266 module
README.md	Information about how to run and build other files
SMART	directory of MCU project
bench	contains the test bench files
verilog	
glbl.v	control MCU initialization
msp_debug.v	connect the processor and decode its internal state during the tests
registers.v	make the MCU registers available for the tests
tb_openMSP430_fpga.v	connect the MCU to the test settings and initialize it
timescale.v	set the timescale used in the tests
rtl	describe real hardware
verilog	
clock.v	module for transforming a 100MHz to a 20Mhz clock signal
coregen	Information about the Xilinx modules
coregen.cgp	Information about the FPGA
spartan6_dmem.xco	describe the data memory
spartan6_pmem.xco	describe the program memory

[1] A Github repository with all the source code used during the implementation and tests. https://github.com/capellaresumo/MAC0499

[2] http://cdn.opencores.org/downloads/opencores_coding_guidelines.pdf

omsp_uart.v	describe de peripheral responsible for serial communication
openMSP430_fpga.v	connect the memory, SMART modules, and peripherals to the MCU
openmsp430	openMSP430 files. This is a directory and the files are not listed.
sha256	
sha256_core.v	main SHA256 module
sha256_k_constants.v	constants used by SHA256
sha256_w_mem.v	manage the dynamic memory used by the module
sha256per.v	get the SHA256 module and make it a peripheral
smart_mac.v	SMART MAC module
sim	contains simulation files
rtl_sim	
bin	tests helpers
asm2ihex.sh	convert one asm file to ihex
helper.sh	helper fiction used by more helpers
ihex2mif.tcl	convert one asm file to mif
msp430sim	coordinates the tests execution
omsp_config.sh	contains values dynamic inject in c code
rtlsim.sh	control a test after compiled
template.x	linker script for assembly tests
template_defs.asm	definitions for assembly tests
run	scripts to start test execution
cores_build	build the Xilinx cores
run	example of how to run a test
run_clean	clean all files generate by a test
test	run all tests scripts in tests file
tests	tests scripts
acess_key_reset	test the SMART key violation and correct access
hw_uart_isim	test the serial peripheral using ISim
hw_uart_iverilog	test the serial peripheral using Icarus Verilog
led_blink	verify if IO peripheral works
sha256	verify if SHA256 peripheral works
smart_code_block	verify if f a SMART perform a successful code access and after an invalid one
smart_v1	test a the MCU with SMART calling the hash from the serial
src	hardware setup for the tests in tests. If the test do not have an software, will be used the assembly file provide here.
acess_key.s43	
acess_key.v	
hw_uart.v	
led_blink.v	
main.s43	

main.v	
sha256.v	
smart_code_block.s43	
smart_code_block.v	
smart_v1.v	
submit.f	Describe the module for ISim
submit.prj	Describe the module for Icarus Verilog
write_flash.v	
software	contains the softwares (c programs) used during the test and deployment
hw_uart	serial communication software
main.c	
makefile	
led_blink	software to make a led blink for simulation
Makefile	
main.c	
led_blink_real	software to make a led blink for in the real device
Makefile	
main.c	
libs	common files used by softwares
Makefile	
cprintf	control the input and output of the serial peripheral
cprintf.c	
cprintf.h	
hardware.h	constants and definitions of peripherals
linker.msp430-elf.x	linker script
omsp_system.h	constants and definitions of MCU
smart	
smart.c	
smart.h	
test.py	script to generate the expected hash
sha256	software to compute a hash
main.c	
makefile	
smart_app	light remote control source
Makefile	
main.c	
makekey.py	script to generate the expected hash
smart_v1	smart used for tests
Makefile	
main.c	
makekey.py	script to generate the expected hash
synthesis	
xilinx	script to automate the process of build the FPGA
0_create_bitstream.sh	

			1_initialize_pmem.sh		
			2_generate_prom_file.sh		
			3_usb_upload.py		
			bitstreams		Directory that contains the generated bitstreams
			scripts		
				ihex2mem.tcl	
				memory.bmm	
				openMSP430_fpga.prj	Contains information of modules necessary to build the project.
				openMSP430_fpga.ucf	User Constraints File
				xst_verilog.opt	options used during the synthesis
SMART_MAC_TEST					Directory responsible for graphical tests in SMART MAC module
	run_tests.sh				Script to run the tests. Only works in Mac OS.
	test.v				HDL code for configuring the test.

The dark gray files represent that it has copied from the openMSP430 project or the SHA256 open implementation. The light gray is file based in pre-existing one. The cells with a white background were written from scratch.

Bibliography

[1] *Spartan-6 FPGA Clocking Resources*, 2018. https://www.xilinx.com/support/documentation/user_guides/ug382.pdf [Accessed: September 2018]. 18

[2] *Hardware implementation of the SHA-256 cryptographic hash function*, 2018. https://github.com/secworks/sha256 [Accessed: September 2018]. 19

[3] *LegUp High-Level Synthesis*, 2018. http://legup.eecg.utoronto.ca/ [Accessed: September 2018]. 11

[4] *openMSP430*, 2018. https://opencores.org/project/openmsp430 [Accessed: August 2018]. 18

[5] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente. On the difficulty of software-based attestation of embedded devices. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 400–409, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-894-0. doi: 10.1145/1653662.1653711. URL http://doi.acm.org/10.1145/1653662.1653711. 7

[6] K. Eldefrawy, D. Perito, and G. Tsudik. Smart: Secure and minimal architecture for (establishing a dynamic) root of trust. 01 2012. 1, 8, 14, 15

[7] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik. Systematic treatment of remote attestation. *IACR Cryptology ePrint Archive*, 2012:713, 2012. 9

[8] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik. A minimalist approach to remote attestation. pages 1–6, 01 2014. 4, 5

[9] Garther. Gartner Says 8.4 Billion Connected. 1

[10] N. R. M. S. G. T. Karim Eldefrawy, Ivan O. Nunes. Formally verified hardware/software co-design for remote attestation. 01 2018. 28

[11] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 10:1–10:14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2704-6. doi: 10.1145/2592798.2592824. URL http://doi.acm.org/10.1145/2592798.2592824. 1, 9

[12] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth. New results for timing-based attestation. In *2012 IEEE Symposium on Security and Privacy*, pages 239–253, May 2012. doi: 10.1109/SP.2012.45. 7

[13] J. Noorman, J. V. Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling. Sancus 2.0: A low-cost security architecture for iot devices. *ACM Trans. Priv. Secur.*, 20(3):7:1–7:33, July 2017. ISSN 2471-2566. doi: 10.1145/3079763. URL http://doi.acm.org/10.1145/3079763. 1, 10

[14] T. Published. Trusted platform module library - family 2.0 - revision 01.38, 09 2018. 4, 5

[15] A.-R. Sadeghi, C. Wachsmann, and M. Waidner. Security and privacy challenges in industrial internet of things. 2015, 07 2015. doi: 10.1145/2744769.2747942. 1

[16] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. Swatt: Software-based attestation for embedded devices. 04 2004. 7

[17] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. *SIGOPS Oper. Syst. Rev.*, 39(5):1–16, Oct. 2005. ISSN 0163-5980. doi: 10.1145/1095809.1095812. URL http://doi.acm.org/10.1145/1095809.1095812. 7

[18] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-703-2. doi: 10.1145/1315245.1315313. URL http://doi.acm.org/10.1145/1315245.1315313. 8

[19] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov. The first collision for full sha-1. pages 570–596, 07 2017. 19

Saturday 2nd February, 2019

17:12