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

Gabriel Capella

Reimplementing SMART

São Paulo
December 2018

Reimplementing SMART

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: Prof. Dr. Alfredo Goldman vel Lejbman

São Paulo
December 2018

Resumo

Elemento obrigatório, constituído de uma sequência de frases concisas e objetivas, em forma de texto. Deve apresentar os objetivos, métodos empregados, resultados e conclusões. O resumo deve ser redigido em parágrafo único, conter no máximo 500 palavras e ser seguido dos termos representativos do conteúdo do trabalho (palavras-chave).

Palavras-chave: palavra-chave1, palavra-chave2, palavra-chave3.

Abstract

Elemento obrigatório, elaborado com as mesmas características do resumo em língua portuguesa.

Keywords: keyword1, keyword2, keyword3.

Contents

1 Introdução **1**

2 Key Concepts **2**

2.1 Embedded Devices . . . 2

2.2 Trust Anchor and Root of Trust . . . 3

2.3 Remote Attestation . . . 3

2.4 Works in Remote Attestation for Embedded Devices . . . 6

2.4.1 Software Remote Attestation . . . 6

2.4.2 SMART . . . 7

2.4.3 TrustLite . . . 8

2.4.4 SANCUS . . . 9

2.5 FPGA . . . 9

2.6 Layers Contemplated . . . 10

3 Implementing SMART **12**

3.1 Premises . . . 12

3.2 Implementation Details . . . 13

3.2.1 MSP430 . . . 13

3.2.2 Development Board and SPARTAN 6 . . . 15

3.2.3 Time Constants . . . 16

3.2.4 SHA256 . . . 17

3.2.5 SMART Memory Access Control . . . 17

3.2.6 Linker Script . . . 19

3.3 Tests . . . 21

3.3.1 Continuous integration . . . 21

3.3.2 Light Remote Control . . . 21

3.4 Results . . . 21

4 Improving SMART **22**

5 Conclusion **23**

A SMART MAC Analysis **24**

B File and Directory Description **28**

Bibliography **29**

Chapter 1

Introdução

falar que atuamente varios sao vulneráveis
pegamos uma solucao, melhoramos, testamos e vimos que eh possivel

Chapter 2

Key Concepts

Nowadays the term IoT (Internet of Things) is turning a buzzword. People, government, and industry have learned that connect simple devices to the internet, can produce a lot of useful data. This data can be used to improve production, make houses more smart e save money and time. Some studies show that by 2020 will be more than 20 billion IoT devices connected to the internet [2].

This massive number of devices create new safety and security challenges. However, the vast number of known attacks in the lasts years show the responsible for this devices is not taking this challenge seriously. Some of this attacks can be in seen in the article X. Some of this attacks occur inside critical infrastructures, like nuclear plants, health, and transportation system.

Software, network, and hardware are layers of this devices that can be attack and damage. Study technics to verify if one device is attacked is the main object of this work. Attacks will ever occur, but if it occurs must have a way to identify than.

In this chapter will be presented definitions about the topic, technologies related to it and the principal works in the field of remote attestation.

2.1 Embedded Devices

One embedded device is a system build to make a specific task. The main difference from normal circuits is the fact that this hardware is programmable, making possible the changes of it functionality only change the program inside it. Normal this devices are low cost, against failures and build work in real time situations. Because they aren't built for general propose, in the most cases they don't have a full operating system running inside it.

Nowadays these devices are changing, they are being connected to the to the internet and making part of the Internet of Things. This makes them vulnerable to attacks and invasions. Make them safe without increasing their cost is one big challenge.

There are embedded system based in processor and microcontrollers. The microcontrollers (or MCUs from microcontroller unit) are a chip with computational capabilities. In most cases, it is a single integrated circuit (IC) that has inside it the processor, memory, and digital I/O ports. This makes possible the use of same IC in different situations only changing its program.

This work the focus will be the low-cost embedded system that uses microcontrollers. Because of its low cost and main objective, this device as limited storage capability and, usually, as a single thread.

2.2 Trust Anchor and Root of Trust

Trust Anchor and Root of Trust

One of the principal themes related to security is the Trust Anchor. Suppose Alice wants to talk with Bob using an encrypted channel. They can use a symmetric or an asymmetric encryption algorithm to do that. If they choose a symmetric algorithm, they will need to share passphrase before start talking. If they choose an asymmetric algorithm, they need to share their public key in a safe channel. If they share their public keys in an insecure channel, they will not be able to prove the identity of the another. That is, there is a need for a reliable channel before they start the conversation. This channel is the trust anchor.

Another example of a trust anchor can be seen in the TLS (Transport Layer Security) use. In this protocol, there is a need to use previous saved public keys (certificates) before the creation of an encrypted connection. These certificates are usually provided during the operating system (OS) installation. There is a belief that the system during the OS installation this certificates are not corrupted and changed. This process is the trust anchor of the TLS.

A similar idea of trust anchor can be used in software and hardware verification. When applied to this area it receives the name of Root of Trust. The main idea is to guarantee that the device or software as o knowledge state. Everything that comes after this stade can be predicted and trusted.

In field software verification are two roots of trust types[10]: 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. Suppose a module that verifies if one operation system files have not been modified before boot, this is an example of dynamic roots of trust. A problem related to static method is the fact that it does not give any guarantees about the current state of the software. If any exploits occur after its initialization, it will not be noticed.

The dynamic root of trust is similar to the static one, but it has a way to perform attestation dynamically after the software load. However, it usually uses secret, like a key, to perform the attestation. The software does not have access to this secreted, and there is a need for an authenticated channel to transmit this secret to the device running the software.

2.3 Remote Attestation

Remote attestation is a term used to design the capability to provide attestation cross a network [10]. Attestation is the ability to serve clear evidence of something. In the area of security, the evidence is the state of the device. The state is formed by what is present in the device memory in a particular moment.

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

TPM is in its second version. Figure 2.1 shows all components that make part of this module. One use of this module can be to encrypt messages. It has a full encrypt engine inside it and a key stored securely. Commonly, this module is connected with the processor, making possible the exchange of information.

[1] https://trustedcomputinggroup.org/

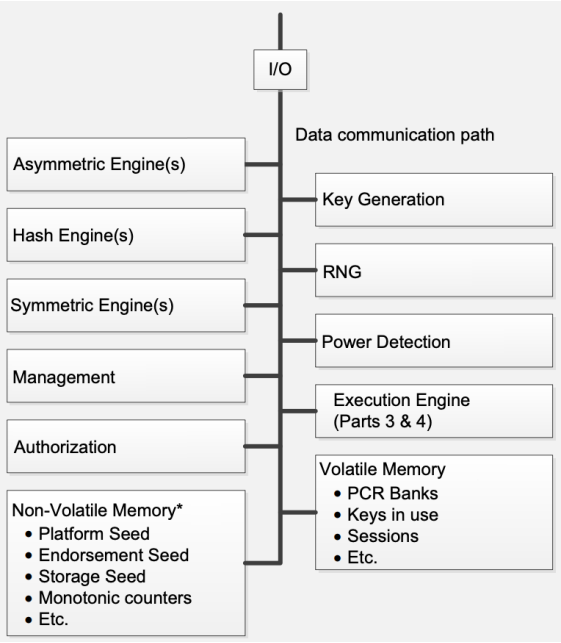

Figure 2.1: *Architectural Overview of TPM. Image from its manual [14].*

Within several uses, another interesting is the hardware and software verification. Assume one company do not want modifications in it computers hardware and software. The TMP can make a hash of the bootloader memory and all modules connected with the processor, after that it can sign this information and send to the company. If it suffers any modification, the company will see divergent in the expected data and the receive information. From this, she can stop providing system update or cancel the warranty, for example.

TPM is only one specification of secure cryptoprocessor. Intel and other brand have developed other technics to optimize the secure in its core processors. However, the cost of this coprocessors is not huge but significant when talking about embed systems. Usually, microcontrollers that driver the embed system costs a few dollars and the secure cryptoprocessor has practically the same price, making the price almost double. This provides an incentive to study and determine the minimum requirements need to provide a remote attestation.

The article [10] shows the essential elements and function to produce a secure remote attestation protocol. Using the language of this article, in a remote attestation, we have the **challenger** and the **prover**. The **challenger** his the device or computer that verifies the internal state of the **prover** across the network. The primary objective of the protocol is to allow a not compromised **prover** to provide an authentication token to the challenger, where it can to find out if the **prover** is in one expected state. In case the **prover** has been modified, the **challenger** needs to notice that.

To this protocol works there is a need for tree function:

- `Setup()`: needs to produce a long key k in a probabilistic way. This function will only be called once. The **challenger** saves that key and the **prover** have it, but it can only access it in certain circumstances.

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

- `Verify`(k, s, α): given a key k, a device state, and an attestation token a. The function returns `True` if `Attest`(k, s) is equal α. Otherwise, returns `False`. The **challenger** computed this function.

In a remote attestation verification, the challenger ask for the authentication token to the **prover**. The **prover** produces it using `Attest` function, creating the attestation token (in an indirect manner the information of its state). When the **challenger** receives α, it verifies if it can produce the same token, using the `Verify` function.

To this protocol works there is a need to share the key k between the **challenger** and the **prover**. We will suppose that this k will be shared in a safe way when the device is manufactory. That is, the key will be write in the device during before it can suffer any attack. The company that produces the device is responsible for generating k and saving it securely. They will use this key after to make the attestation.

During the protocol is transmitted the attestation token. This transmission occurs over a network and is successive to attacks. If this occurs, the **prover** will send a token α and the **challenger** will receives a α'. In this case, the **challenger** will attest the **prover** as a violated device (in not know state). Some technologies, like the Transport Layer Security, can be used to prevent attacks in this connection. One interesting fact to note is that an attacker can save one old authentication token and send this token in a later connection. To prevent this type of attack, when the **challenger** ask for the authentication token, with the request it will send a nonce n. This nonce will modify the state of the **prover** providing a unique state. This nonce is randomly generated in each authentication token request. The n is not specified in the above function, to make clear the basic idea of the protocol, but in the future, it will be used.

Suppose the **prover** suffer an attack, when this happens the state s of the device becomes known by the attacker. With this information he can produce to types of attacks: simulate the `Attest` function a return a correctly α; returns a α that is not the real state of the device. To prevent the attacks mentioned the `Attest` function in the **prover** has to have the following properties:

- **Exclusive Access**: only the `Attest` function can have access to key k. This makes impossible to forge the token α without running the `Attest` function.
- **No Leaks**: after de execution of `Attest` 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 the `Attest` can be changed, the attacker can change it to leak the key k.
- **Uninterruptedly**: if during the computation of `|Attest|`the device suffer one interruption, the key k information can leak.
- **Controlled Invocation**: the `Attest` function can only be called by its beginning. Otherwise, the attacker jumps important parts in the `Attest` code, like the code that disable all interrupts.

The way all these properties will be provided varies according to the device. Some approaches try to provide all these properties using a software implementation. Unfortunately, the remote attestation using only software has been proved that not work in the internet context.

2.4 Works in Remote Attestation for Embedded Devices

This section has the primary objective to show and discuss the mains works in Remote Attestation for Embedded Devices. It fist subsection will contemplate software remote attestation technics and show why they have been proven to be insecure on the internet. The others subsections focus on some hardware implementations.

2.4.1 Software Remote Attestation

Several articles tried to construct a remote attestation algorithm using only software, without hardware changes. One of the main articles in this area is the *Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems* [16]. The basic idea of this article is to send a nonce to the device and for it produces an attestation token based on the device current state. Like in the premises for remote attestation, this token is compared with the expected token to determine if the device is compromised.

However, is not possible to ensure all the guarantee related to the key storage. To work around, the computing time became part of the challenger verification. If the device took too long to calculate the hash, probably the device is attacked. When the Pioneer makes the memory hash, it takes a random look in the memory based in the nonce it receives. If the attacker copies the memory to other place or tried any action to subvert the attestation, this will produce a considerable computation time increase. The challenger will notice this difference, and it will suspect that the device suffered an attack.

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

In this implementation, the challenger needs to know accurately the 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 become scalable.

One interesting fact is that the Pioneer solution was made to work on various device types. For example, the article implements the system in an Intel Pentium IV Xeon processor. Pioneer use same ideas previously created in a project called *SWATT: SoftWare-based ATTestation for Embedded Devices*[15], this is another article that shows a remote attestation technics. The idea in this those implementations is similar, but the main difference is that SWATT works only in embedded devices with simple CPU architectures, like microcontrollers.

Some works have shown failures in the timing based remote attestation systems, like the software one. The [7] 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, they declare that software-based attestation is realy challenging to design, but not impossible. The principal reason for this assertion is the fact that small change on the verification code can reduce the verification time and cheat the verifier.

Another 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. In this class of attacks involving 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 to be 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 thesis is to implement and test the hardware and ideas proposed in *SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust*[8]. 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, send by the challenger and obtain a hash that represents the device state. Also, the function can look only 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 `Attest` function is the capability to the **challenger** send a function address to be called at the end of the SMART code execution. This makes possible attest one memory region and, after, 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 to proof if the device has been successful reset.

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

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 evaluate.

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 not prevent this type of attacks. Some attacks, like a technique called *Return-Oriented Programming* (ROP) [17], only change the return pointer in the data memory and reuse the program code in a different memory align to perform the attack. Therefore, in the article are assumed that the attacker has full access and can modify the code outside the protected memory region of SMART.

Another critical assumption as that the device will not suffer any hardware attacks. This is important because if it is possible, the attacker can, for example, make a reset on every time the SMART code is called. Making this successively residual information of the key can leak and than he can obtain the key.

Although carefully designed, some flaws were found in the SMART project [9]. 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 article is argued that the interruption disable is optional. However, depending on the device's interrupt handler this can cause the leak of the key. Disable the device interrupts during the SMART code execution is a measure to solve this problem.

2.4.3 TrustLite

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

Some advantages of this implementation are the existence of several trust applications. These applications are named *trustlets* by the article. Each *trustlets* are 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). Also, *trustlets* can validate and the local platform state and other *trustlets*. That is, one *trustlets* can contain the `Attest` to provide the remote attestation features.

To load the memory access policy to the MPU was developed a *Secure Loader*. This loader works like a *trustlets*, but it is the first to been load and is in a know 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 remoted updated.

In SMART, when on memory violation occurs the hardware make 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 insecurity applications. This applications and the *trustlets* can exchange information, creating an interprocess communication. This information exchange can be used, for example, to make one untrusted application (like an input and output control) to be attested.

The TrustLite was tested in an FPGA. The group that built it used an Intel Siskiyou Peak processor to implement it inside a Xilinx Virtex-6 FPGA. The measure of the hardware extension cost was made based on the number od 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 use 552 registers a 14361 LUTs. This total number of LUTS is much higher 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 exactly, 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, is still possible to note that TrustLite is more costly than SMART.

In this thesis, it was chosen to be made the SMART implementation 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. Will be only necessary to attest the code responsible for loading these apps, because if this code is trustworthy every code load by it will also be reliable. Note that will be not possible a dynamic root of trust in this case.

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, like software module isolation and remote attestation, in a network of interconnected devices.

Like the other projects, the SANCUS do not use asymmetric cryptography, because of the high cost of implementing it. However, to provide security the 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. With application was with own area in memory. Each application was a single key, but this key is no accessible directly, it only can be used by the processor when particular functions are called, like SMART. In SANCUS, every application needs to have a special hardware related to it. This hardware is responsible for saving the application key, the address of de application code and data. Also, it has Memory Access Logic (MAL, as the author), the hardware 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, the 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 microcontroller. The maximum frequency decreases with 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 the key size is larger, caused 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 is this thesis.

2.5 FPGA

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

The major benefits are that you can test new hardware and circuits without the need to produce a new integrated circuit. One of the tasks in this work is to change the memory

backbone of on microcontroller and test the results. Using the FPGA the assignment to test the modification became easy because changing it is like reprogram.

There are two principal HDL languages used in the market: Verilog and VHDL. This language is totally differences from normal programming languages. The principal difference from normal programming languages is the fact that everything is happening in parallel. In this languages, variables are replaced by registers and connected with wires. There are some attempts to convert sequential programming languages, like C, in hardware description languages, like the LegUp High-Level Synthesis project [4]. But transforming sequential code to hardware normally require big circuit.

An approach used in 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 designer is the possibility to connect an hardware in the processor to make specifics tasks. Some companies that produce FPGA's are build processor optimized 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). Basically, a LUT is a hardware truth table. That is, depending on its input there is a specific logic output. One important FPGA characteristic is it 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 was flip-flops) are responsible for this property.

2.6 Layers Contemplated

This work will build a device and program it, involves a different abstraction layers. In figure 2.2 is possible to see all the steps and layer 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 image 2.2 in a grey box is possible to see the hardware description language. The primary objective is to transform the HDL source into a file that and be understood and load to 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 a NGC files. That file contains both logical design data and constraints, that is, is the hardware description using only primary elements, like logical port. This file is also known as netlist file.

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

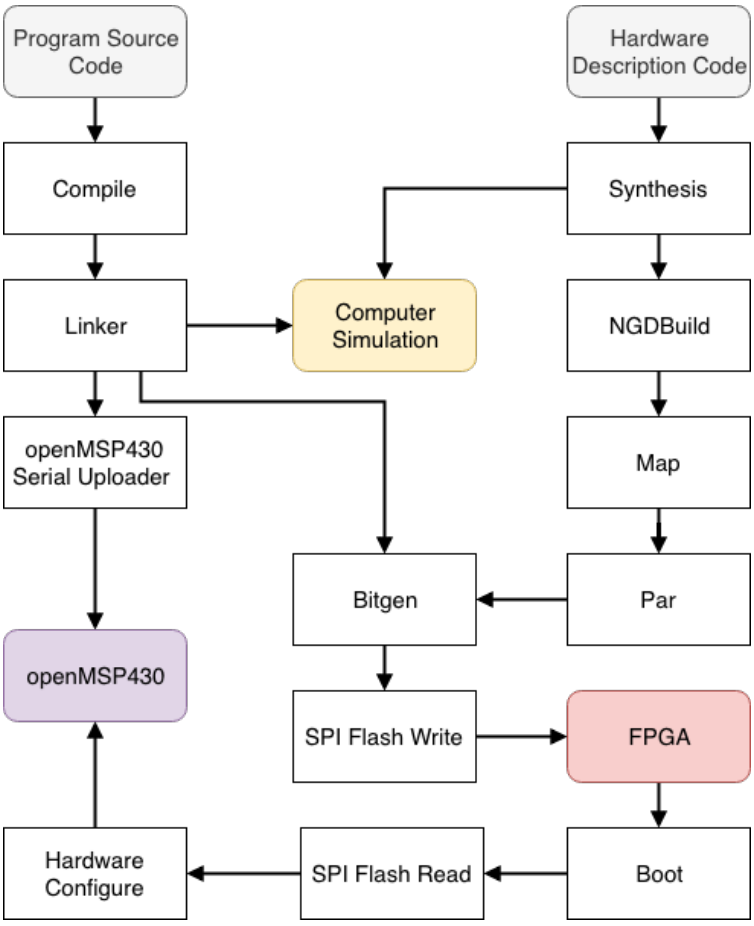

Figure 2.2: *Building and loading steps.*

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 afpg boot it will read its flash memory, configure it internal gates (Hardware Configuration) and at the end will be a microcontroller. In this thesis case, the openMSP430.

On the left side of the image is the part responsible 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 load 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. This 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 the image, we have the process specific for the Spartan 6 FPGA family using it manufacture (Xilinx) tools. All FPGA building steps description are based in the Xilinx documentation[2].

[2] 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 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] 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 a few assertions are made to informal proof 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 as 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) can are part of in this model, but will not be part of this thesis 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 code was 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 behaviour in the code. In the adversary model, this behaviour 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 [8].

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 thesis, 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 has the same set of processor instructions[2]. What makes them different is their memory size,

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

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

The MSP430 use 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 a 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.

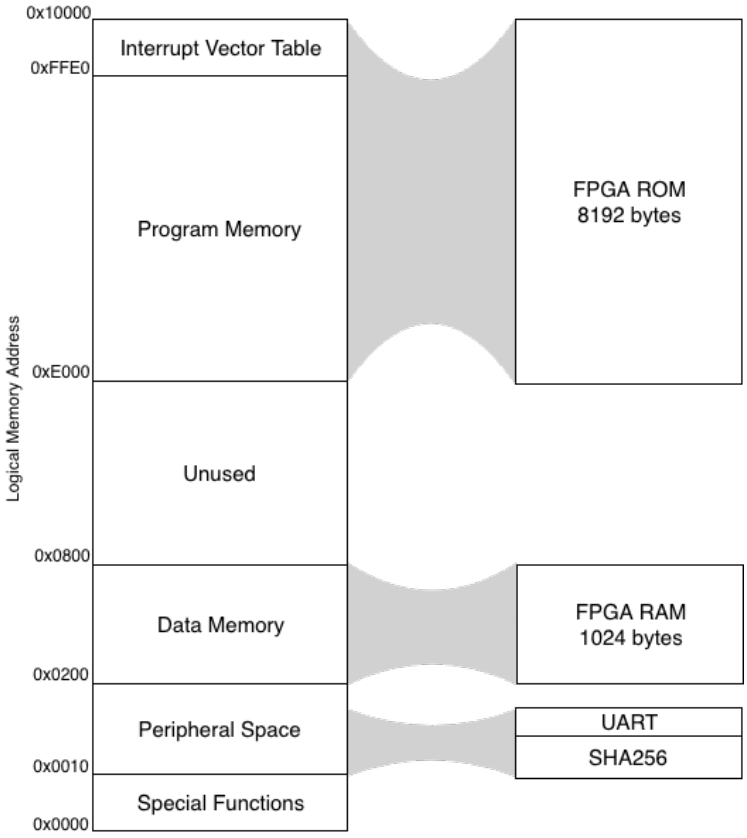

Figure 3.1: *Memory map using a ROM with 8192 bytes and RAM with 1024 bytes. Also, there are two peripheral, one responsible for controlling the serial communication (UART, universal asynchronous receiver/transmitter) and other to calculate SHA256 hashes.*

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 in this theses sometimes 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 are 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 thesis will use the 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 a 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 start. This code has 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 its. 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.

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

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

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 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 memory. 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 thesis 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 dispose 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 the time unit. And the time precision is used by the simulator to know how they can round the summation values. In the tests are 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 $100MHz$. Is chosen 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, the bits are sent one by 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 [5] 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 has using 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 use 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 [18], to show how an attacker can forge two distinct PDF documents with the same SHA-1 hash.

It is important to notice that exist a new version of the secure hash algorithm, the SHA3. This version made some improvements and NIST advise to use it. There was the attempt 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 segure, and it fit inside the FPGA.

A SHA3 implementation written by Joachim Strombergson [3] has used. A peripheral adaptor was built to communicate with the openMSP430 core. This adaptor is responsible for interfacing the SHA3 implementation with the peripherals pins in the microcontroller. The file `SMART/rtl/verilog/sha256/sha256per.v` contains this interface.

Another change is the use of a hardware hash function and not a software one. 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 of 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 allows the access to a specific memory region. We will call this area as the protected memory region. If there is any trial 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 occur 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 $= X' + 1$ 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 one byte to the physical memory address of the SMART function because it is outside the protected memory region.

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

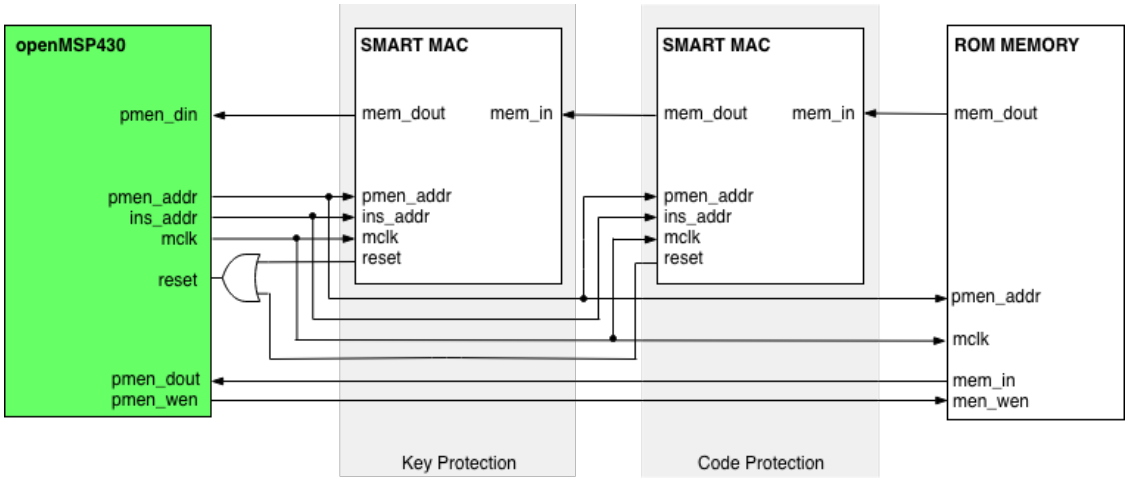

Figure 3.2: *The interface between the ROM memory and the openMSP430. In the middle, is SMART memory access control module.*

Using two SMART MAC is possible to grant the three remaining assertions. In figure 3.2 is possible to 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 B 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. Linker script (or linker command file) is a particular file that describes to the compiler this attributes.

[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.

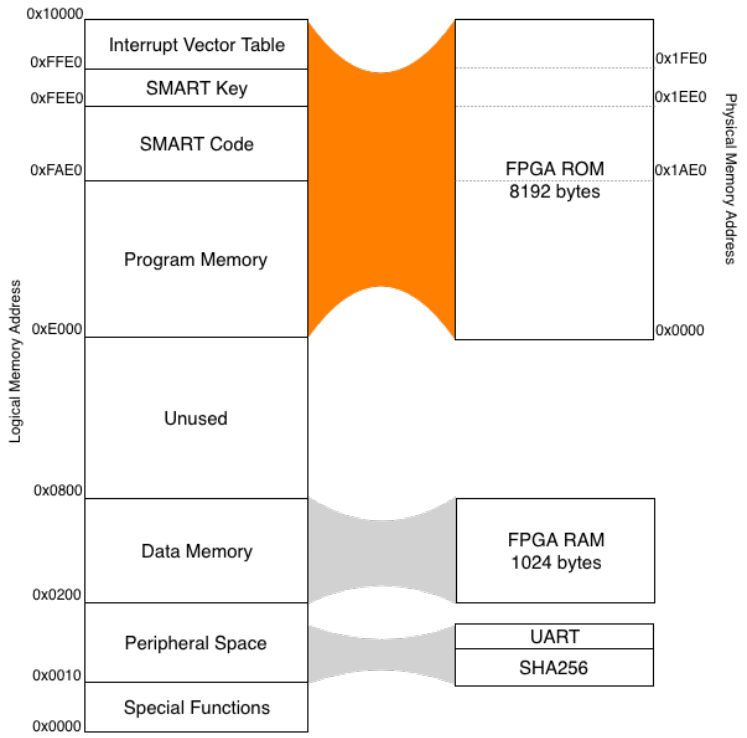

Figure 3.3: *The orange link is the program memory backbone where the smart modules are introduced. Differently, from figure X, this image includes de physical and logical addresses used by the SMART code and the SMART key.*

In figure X 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 was 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 is the 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 it 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 real application using SMART.

3.3.1 Continuous integration

A continuous integration system is used to prevent and 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 behaviour of the hardware: the Icarus Verilog[7] (an open-source project) and the Xilinx ISE (a property software). Every test is executed using this two simulation softwares. This decision was made because during the development are notice that some errors only occur in one of them and not in the other or vice-versa.

3.3.2 Light Remote Control

subvert

3.4 Results

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

Chapter 4

Improving SMART

The opemMSP430 memory backbone is changed, allowing writing in the program memory. This change makes possible remote device updates, but also introduces several security breaches. However, the SMART implementation can identify if any breach occurs.

A system is built to control, verify and make a remote update using the SMART as the final example.

The article *ASSURED: Architecture for Secure Software Update of Realistic Embedded Devices*[6] has the base of this system.

Cuidas das excecoes geradas, nao resetar, somente pular a instrucao....

Chapter 5

Conclusion

Texto texto[1].

[1] Exemplo de referência para página Web: www.vision.ime.usp.br/~jmena/stuff/tese-exemplo

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 behaviour.

Although the small size of the module, it has a reasonable complexity. The microcontroller clock synchronize 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 behaviour happens because when a 16 bits bus changes its value, the bits are not changed together. This time variation is very small, but if the module does not use a sycronize mechanism, an incorrect reset signal can be produced.

```
1  // OUTPUTs
   //=============================================================================
3  output                      in_safe_area;
   output                      reset;
5  output              [15:0]  mem_dout;

7  // INPUTs
   //=============================================================================
9  input  [SIZE_MEM_ADDR:0]    mem_addr;
   input                       mclk;
11 input               [15:0]  mem_din;
   input               [15:0]  ins_addr;
13 input                       disable_debug;

15 // LOGIC
   //=============================================================================
17 reg    inside_code = 0;
   reg    to_be_reset = 0;
19
   wire addr_in_safe = (mem_addr <= HIGH_SAFE) & (mem_addr >= LOW_SAFE);
21 wire pc_in_code = (ins_addr <= HIGH_CODE) & ( (ins_addr+1) > LOW_CODE);

23 assign safe_reset = addr_in_safe & ~inside_code;

25 assign reset = to_be_reset & ~disable_debug;

27 assign mem_dout = reset ? 16'b0 : mem_din;
   assign in_safe_area = inside_code;
```

```
29
always @ (posedge mclk) begin
31    if (ins_addr == LOW_CODE) begin
          inside_code <= 1'b1;
33    end
      else begin
35        if (~pc_in_code) inside_code <= 1'b0;
      end
37    to_be_reset <= safe_reset;
end
```

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

Line 30 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 synchronize area (start at line 30 and ends at 38) 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 save 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 during the course of 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.

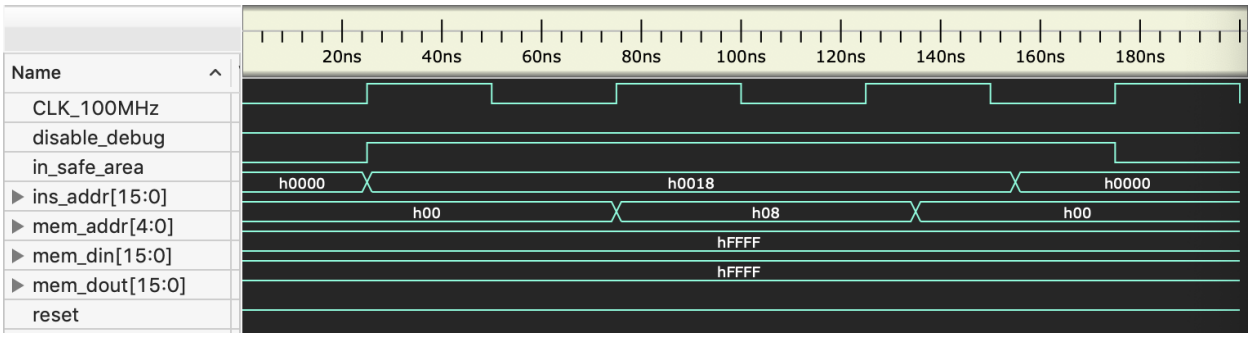

Figure A.1: *Simulation of success access to the protected memory region.*

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.

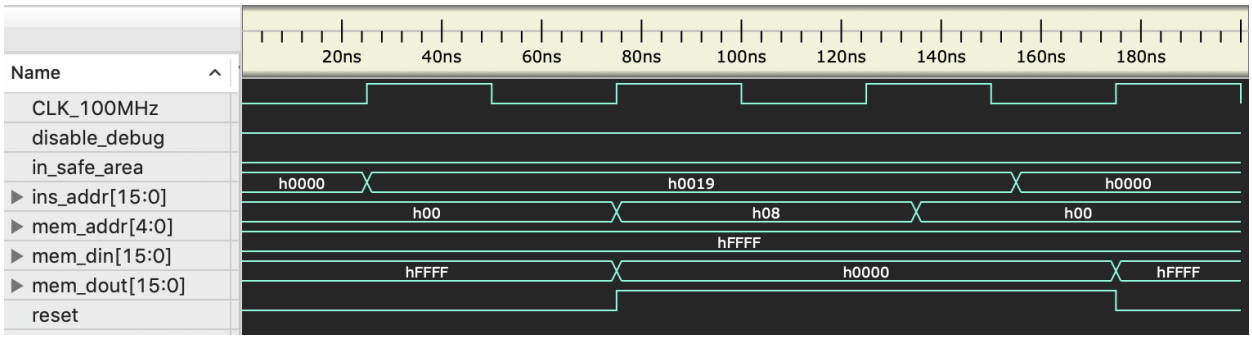

Figure A.2: *Simulation of what happens when the instruction pointer goes to the second code address and access to the protected memory region.*

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.

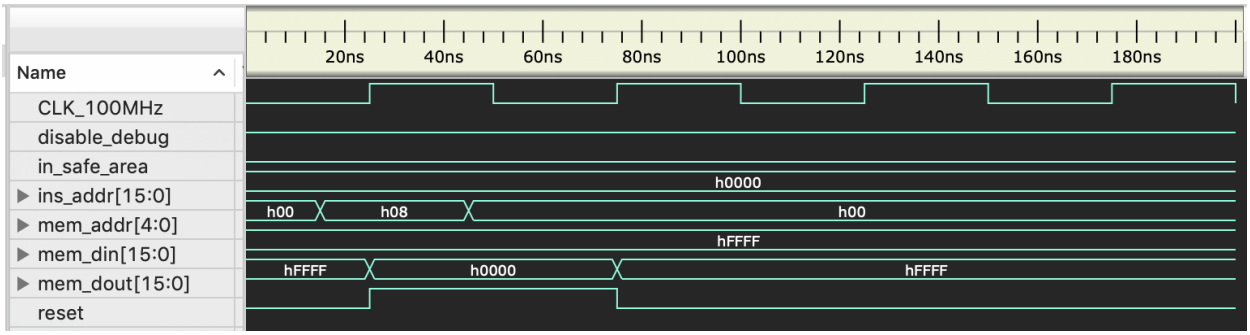

Figure A.3: *Simulation of one not authorized access to the protected memory region.*

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

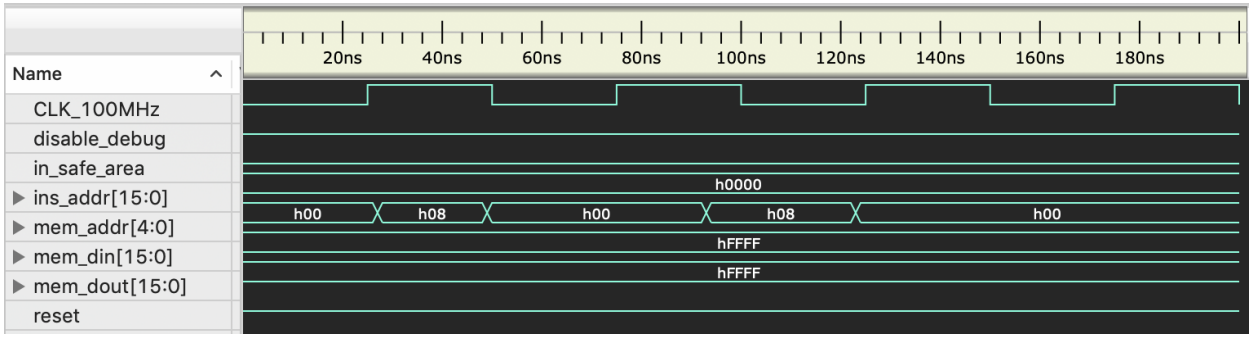

Figure A.4: *Simulation of the behaviour when men_ addr pin inconsistency states happens.*

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

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

Verilog vvp runtime engine.

Appendix B

File and Directory Description

Bibliography

[1] *Spartan-6 FPGA Clocking Resources*, 2018. https://www.xilinx.com/support/documentation/user_guides/ug382.pdf [Accessed: September 2018]. 16

[2] 2018. https://www.gartner.com/en/newsroom/press-releases/2017-02-07-gartner-says-8-billion-connected-things-will-be-in-use-in-2017-up-31-percent-from-2016, [Accessed: August 2018]. 2

[3] *Hardware implementation of the SHA-256 cryptographic hash function*, 2018. https://github.com/secworks/sha256 [Accessed: September 2018]. 17

[4] *LegUp High-Level Synthesis*, 2018. http://legup.eecg.utoronto.ca/ [Accessed: September 2018]. 10

[5] *openMSP430*, 2018. https://opencores.org/project/openmsp430 [Accessed: August 2018]. 16

[6] N. Asokan, T. Nyman, N. Rattanavipanon, A.-R. Sadeghi, and G. Tsudik. Assured: Architecture for secure software update of realistic embedded devices. 07 2018. 22

[7] 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. 6

[8] K. Eldefrawy, D. Perito, and G. Tsudik. Smart: Secure and minimal architecture for (establishing a dynamic) root of trust. 01 2012. 7, 13

[9] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik. Systematic treatment of remote attestation. *IACR Cryptology ePrint Archive*, 2012:713, 2012. 8

[10] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik. A minimalist approach to remote attestation. pages 1–6, 01 2014. 3, 4

[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. 8

[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. 6

[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. 9

[14] T. Published. Trusted platform module library - family 2.0 - revision 01.38, 09 2018. 3, 4

[15] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. Swatt: Software-based attestation for embedded devices. 04 2004. 6

[16] 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. 6

[17] 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. 7

[18] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov. The first collision for full sha-1. pages 570–596, 07 2017. 17