Universiteit Leiden
The Netherlands

BsC Computer Science

Fault analysis for a new MsC course

using the ChipWhisperer

Akram Hamdi

Supervisors:
Nele Mentens & Felienne Hermans

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

July 2021

Abstract

Attacks on vulnerable C code by using, for example, the buffer overflow attack, are attacks that are widely known and explored throughout the time. Usually when looking at attacking an embedded system, this one is the first that comes to mind. But what if we could also attack a system via the hardware? More importantly, to attack the encryption that is done by the hardware? It is usually quite hard to try and test these types of attacks however, as it involves attacking the hardware directly and trying to disturb its operations, something which is very hard to do on an average computer due to the defense mechanisms implemented by the kernel. This is where the ChipWhisperer comes in. A cryptographic board, equipped with the parts a regular processor would have as well, made to try and practice these attacks. This thesis gives an insight in how this board could be used in order to assist the teaching of a new hardware security course, planned to commence in the second semester of the masters programme of Computer Science, in 2022. Alongside this thesis, a hands-on tutorial is made about the setup and execution on the attacks of the ChipWhisperer board has been made. Additionally, the necessary background information is given as well.

Acklowledgements

I would like to sincerely thank my supervisors Nele Mentens and Felienne Hermans for supervising this bachelor thesis. I would also like to acknowledge Gijs Burghoorn for his massive help, since his research subject is very similar this this one. I would also like to thank my family and friends for their support, without them I wouldn't be here right now and I would never have been able to get this far. And last, but certainly not least, you.

Contents

1 Introduction

1.1 The situation

The ChipWhisperer, developed by NewAE Technology Inc. (logo in Figure 1), is a cryptographic board, created to expose the weaknesses that exist in the embedded systems that are all around us [15]. With this board, numerous attacks, like *clock glitching* or *side-channel attacks*, could be executed on existing, pre made code for the board specifically, but also self written code using your own algorithms. It should be noted that these "algorithms" tend to be centered around encryption algorithms, like for example AES or RSA, but could also be a simple algorithm that sorts an array. The main point is that these attacks can be used on a wide variety of different programs and algorithms, which is exactly what the board aims to do.

Figure 1: The manufacturer of the ChipWhisperer, NewAE

By using the ChipWhisperer and trying to execute these types of attacks, the weaknesses of those encryption algorithms could be exposed. Since the execution of the code/programs containing these algorithms of encryption rely on the hardware being present and able to properly do the task required, it opens up a new way of attacking. Usually when discussing breaking algorithms like RSA or AES, the discussion almost always curves towards exploiting certain unsafely written code, like the widely known *buffer overflow* attack. These attacks, and solutions however, are already widely known. But attacking the hardware directly, and shifting from attacking the code to attacking the hardware that actually makes the code execute in the first place, is something that a lot less discussions are held about.

For a new course on Hardware Security (or Cryptographic Engineering), the ChipWhisperer is going to be used in the practical part of the course in order to explore the possibilities of executing hardware related attacks on cryptographic algorithms.

1.2 The goal

At Leiden University, not a lot of security courses are being taught in the Bachelors and Masters programs. There is a course on security taught in the second year of the bachelor, although that course mainly goes into software related weaknesses and security aspects, like web security, or application security. And whilst hardware security is briefly mentioned during the course, it's not something that's actually discussion about in depth. The supervisor, Prof. Dr.Ir. Nele Mentens, wants to change this by providing a course on Cryptographic Engineering, starting in the spring semester of 2022, for the masters students of Computer Science. In this course they will learn more about the implementation aspects of cryptographic algorithms, and also about executing attacks that target the weaknesses in these cryptographic algorithms [14].

These attacks are executed during the practical part of the course, using the ChipWhisperer. The ChipWhisperer has been analyzed, used, and tested for this thesis, in order to provide a tutorial for the future masters students, in order to support the makings of a new security course for LIACS. Along with the supervisor and fellow student Gijs Burghoorn, a tutorial has been made for two different types of attacks and exploitation's on these cryptographic algorithms. In this thesis, voltage glitching and clock glitching will be discussed, alongside some power analysis as well. For this thesis, the accessibility of the board, alongside exploring the learning curve for a beginning student, has been researched. With the research, alongside the creation of the tutorial made by students, for students, we aim to open up a new chapter in the security teachings of LIACS.

1.3 Overview

In this thesis, we will mainly focus on the attacks of the earlier mentioned clock glitching, alongside the *voltage glitching* attack. Then, a phenomenon known as *fault analysis* will be discussed as well, as it is very relevant to these types of attacks. Alongside these aspects, some *power analysis* will be discussed as well. All of these subjects will be discussed in section 2. In section 3, we will discuss the methods and also some results of the research. And finally, in section 4, we will conclude our thesis, and answer the research question.

2 Background

In this section I will give the necessary information needed to understand what my thesis is all about. I will go into detail about the the types of attacks, what these attacks aim to exploit, and many more aspects. The research question of this thesis is as follows:

In which ways can we use the Chipwhisperer-board for a hardware security or Cryptographic engineering course in the Computer Science Masters program?

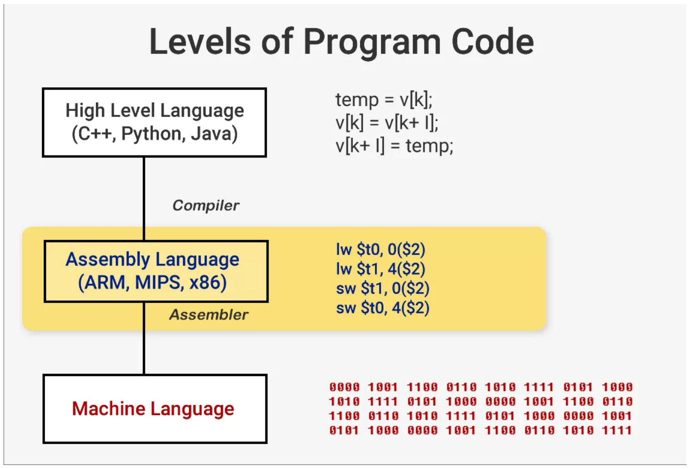
2.1 The CPU

In every hardware device, every smartphone, computer, or even washing machines or the cable box of the TV, there are hardware parts called *central processing units*, or CPU for short. These parts are perhaps the most important part of any embedded device, and could be compared to the brains of the human body [7]. They are also called the processor, and their task includes, but is not limited to, executing instructions of a computer program.

A computer program consists of a sequence of *assembly instructions* that are sequentially executed in order to achieve a certain goal. An instruction could be very simple, like adding two values together, or a little bit more advanced, like for example the comparison between two values, and taking a branched path in the program based on that comparison. These two aspects are usually two separate instructions used in combination with each other. All these instructions together essentially form the base of a program.
When you want to run a program, like a video game, it consists of a long sequence of these instructions.This code is written in a certain higher level language, since writing pure assembly is quite a big chore and is far from ideal. Therefore a higher level language, like Python, C(++), and Java, are commonly used. It's much more convenient to simply call a certain function rather than write the instructions, or machine code (0's and 1's), ourselves. Figure 2 features a clear overview of the levels of program code, and how it looks like.

2.2 Clocking

When discussing CPUs, a term that is almost always mentioned is the one on the processor *clock speed*. The clock in a hardware system refers to a signal that regulates the timing and speed of all computer functions. The performance of the CPU can be partially quantified by looking at this statistic, since it is one of the most significant measures of performance of a processor. It is measured in the amount of cycles the CPU executes per second, usually in GHz (gigahertz) [9], although measures in MHz (megahertz) were more prominent in the past. Consider the following figure:



```
temp = v[k];
v[k] = v[k+ l];
v[k+ l] = temp;
```

```
lw $t0, 0($2)
lw $t1, 4($2)
sw $t1, 0($2)
sw $t0, 4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

Figure 2: The levels of Program code, note that the processor can only understand machine code, and not high level code![7]

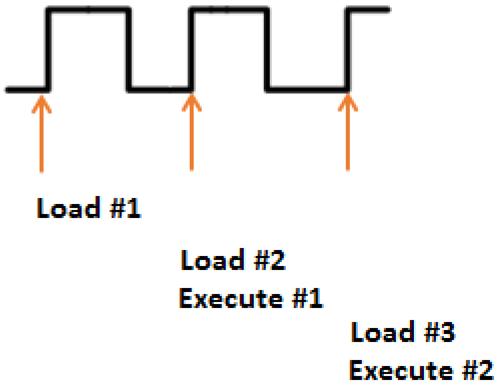


Figure 3: Timing diagram of the clock, note the consistency of the cycles (Source: Taken from the ChipWhisperer tutorial, fault101 Lab1_1 [16])

As we can see in the diagram in Figure 3, every cycle an instruction is issued. The `Load #1` operation loads a new instruction into memory, while the `Execute #1` carries out the instruction. As we can also see, in the second cycle not only is instruction 1 executed, but also a new instruction 2 is loaded in. This is because there are multiple *functional units* that can execute different instructions at the same time. A functional unit is a part of the CPU that performs the operations that a computer program calls for [11]. It would be a waste if every cycle at most one instruction could be executed, since an instruction usually only occupies one functional unit per clock cycle, and tends to move on to another unit during the course of execution. Therefore, the CPU has the ability to execute multiple instructions at once by using as many functional units at the same time as possible. This phenomenon is known as *pipelining*. The goal of pipelining is to increase the *throughput* of the system and, additionally, to increase its performance by lowering the time needed to execute said instructions. While the benefits of pipelining are definitely invaluable in the modern architectures we know today, it also comes with new challenges.

2.2.1 Pipelining challenges

As mentioned before, the effects of pipelining also bring new challenges. Instructions that might be in a certain sequence initially, can be, for example, optimized by the CPU to not be executed in the same sequence anymore, this is known as an *instruction scheduling* problem. Suppose we have 3 instructions, 1, 2 and 3. If instruction 2 takes a lot shorter to be executed, or if other instructions might cause a lot of unneeded stall cycles, it might be tempting to execute that instruction first, then move on to instructions 1 and 3. While that might sound good in theory, it does start to become a problem if certain instructions are dependant of each other. Instruction might want to write to a certain register that instruction 2 then reads from. But if instruction 2 is executed before 1 then it will read the wrong value, or even uninitialized data! This surely is a problem, but fortunately, quite some solutions already exist. A well known one being the algorithm of Tomasulo [19].

2.3 The ChipWhisperer

For this thesis, and its research for the new course, the ChipWhisperer Lite has been explored and researched. The ChipWhisperer, as mentioned before, is a cryptographic chip, manufactured for educational purposes in order to expose the weaknesses of embedded systems that we use in our daily lives. Take a look at the layout of the board in Figure 4:

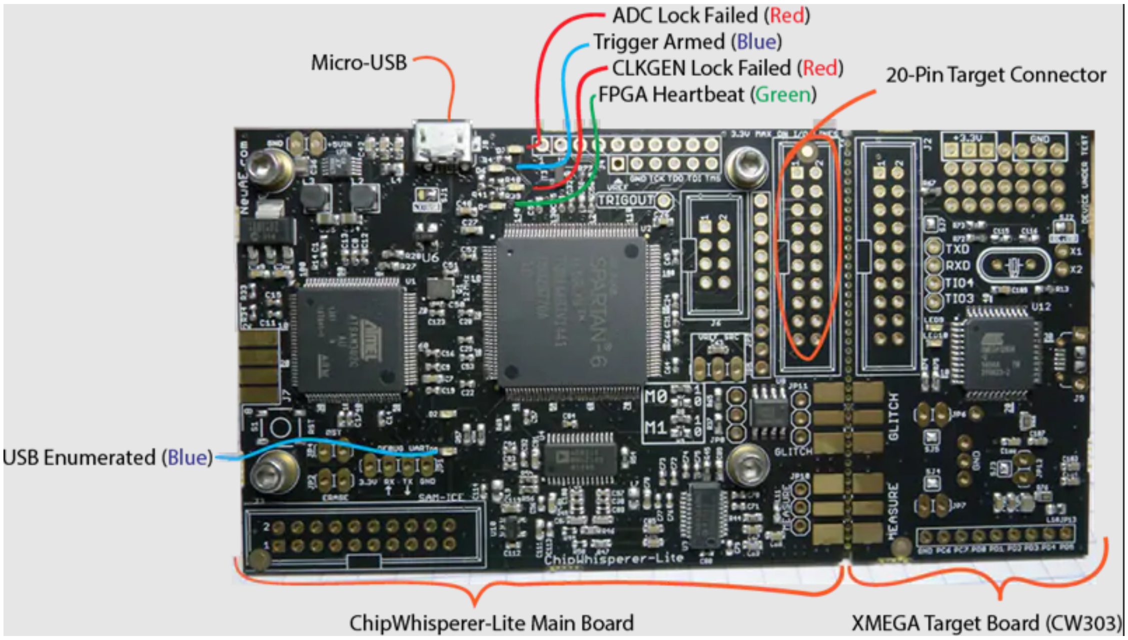


Figure 4: Layout of the ChipWhisperer Lite
Source: newae.com

This board has been created in order to be able to easily, and conveniently, execute attacks on cryptographic algorithms in a number of different ways. Usually when trying to practice attacks on hardware devices, big setups were necessary to actually accommodate such attacks. In the case of, for example, power analysis, big instruments were needed in order to capture the needed power trace, such that it was known where to execute the attack. Additionally, other instruments to do the attacks were also needed. All of these factors combined quickly expose the fact that performing attacks in ways like these are far from convenient. This is where the ChipWhisperer comes in. A small, cryptograhic board, not much bigger than a card in your average card game. It's easy to bring, has all the necessary parts, and even has a simple USB connection so that everyone can simply connect it to his/her computer, and start up. The ChipWhisperer provides both the hardware necessary in order to practice these attacks, as well as the software in order to be able to communicate with the board.

The algorithms that are used on the board, are made from C code, like the AES implementation, which we will be discussing later on. To communicate with the board from your laptop, Python is used. The manufacturers of the ChipWhisperer, NewAE, have provided an API with numerous functions and objects in order to make the communication to the board very convenient. Additional software and functions in order to capture power traces, are also available. All of these factors combined makes the ChipWhisperer an ideal product in order to learn how the encryption algorithms are implemented, as well as explaining how these algorithms can be broken in ways that aren't thought about very often.

2.4 Clock Glitching

As mentioned before, every part of an embedded hardware system relies on some sort of reliable clock. The clock determines the pace of which operations such as issuing an instruction can take place. It could in fact be seen as the beating heart of the system, while the CPU are the brains. But what if we could somehow manage to exploit this reliance on the clock? The hardware probably counts on it that the clock will ”always be right” and trusts on it that when a new cycle begins, it is on the right moment, but this doesn’t always have to be the case! And this is where the phenomenon known as *clock glitching* comes into play. Clock glitching is an example of a bigger phenomenon known as *fault injection*, another example of this is voltage glitching, which will be discussed in a later section. The best way to introduce clock glitching is by giving a visual example, consider Figure 5:

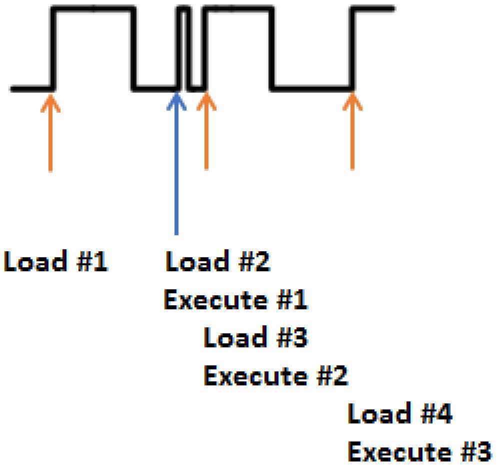


Figure 5: Timing diagram of a fault injected clock, note that the instructions under the blue arrow are never properly executed. (Source: Taken from the ChipWhisperer tutorial, fault101 Lab1_1 [16])

As can be seen here, there’s something different about this figure compared to Figure 3. The clock cycle, where originally the second pulse would trigger, has become much shorter. This is where the injection of the fault comes into play, that clock cycle has been injected by the attacker in order to trigger a new cycle earlier than is supposed to happen. While it does look relatively benign looking at the figure, since the `Load #2` and `Execute #1` instruction are said to happen regardless, following the `Load #3` and `Execute #2` this is actually a massive problem. The instructions that are issued and executed in the faulty clock cycle are actually never executed and effectively skipped completely! The reason for this is because the functional units that do a specific task are tweaked and are built upon the same clock speed the CPU itself provides, meaning it does not have enough time to actually perform the `Load #2` and `Execute #1` instructions. By shortening the clock cycle, we can manage to bypass the consistency that the clock is supposed to give, and therefore cause behavior that’s never intended to happen. We’re going to look at an example of how this can be exploited later on.

While mentioned before that these instructions are not executed at all, there might however be an attempt on trying to actually execute the instruction anyways. This has to do with another phenomenon called the *propagation delay*. Consider Figure 6:

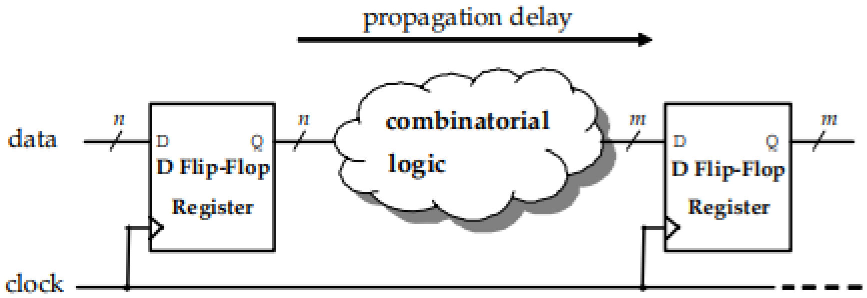


Figure 6: Digital integrated circuit [1]

As we can see here, we have a integrated circuit with a data stream, and the clock stream which is connected to the *D flip-flop registers*. These are registers that capture the input of the data stream at a specific defined portion of the clock cycle. This is mostly on the rising edge of the cycle. This captured value is then the output (m in the figure) [23]. The propagation delay is then defined as the time it takes to propagate the output of one flip-flop, through the combinatorial logic (functional units), and into the next register, essentially moving the processed data from one register to another. In we want to ensure that our circuits operates correctly, the clock period has to be greater than the propagation delay, combined with the time it takes to setup the register. [1] If we want to exploit this delay in order to produce a faulty clock cycle and therefore cause glitching, we have to make sure that our flip-flops accept a state which it is not supposed to accept. Because if the state is not accepted, the cycle is simply not recognized, and incorrect values get stored. Finding the sweet spot and the correct glitch width are essential parts of finding out how and when to insert our glitches, since the moment in time also matters a lot. Why the moment of inserting the glitch is so essential, will be discussed in section 2.5.

2.5 Voltage Glitching

A voltage glitch attack works a bit differently as opposed to the clock glitching attack, although their goals are the same. With a voltage glitch attack, a short electrical pulse is injected into the circuit of a hardware device [10]. By injecting the pulse at the correct timing, the attacker might be able to push the device out of its normal security routine, causing it to skip certain key points in authenticating processes. Additionally, data read from an external device might get corrupted as well, and even instructions might get decoded in the wrong way, causing them to be malformed, or perhaps even skipped entirely [10]!

Practical uses of this kind of attacks are included, not limited to, pay TV boxes, which could be used to gain access to entertainment services without actually paying for subscription. They are usually sold for a high cost (also called IPTV boxes) [10]. Another usage of this attacking method is with modern game consoles such as the PlayStation 3 and the Xbox 360. The hacker, named GliGli, tried to bypass a certain validation step in order to write custom firmware to the console, allowing it to run self made code, copied games, or even Linux [5]!

Voltage glitches could also be used for more malicious purposes though. These purposes include, but are not limited to, corrupting data read from external sources, and induce the system to incorrectly decode instructions. It might also jolt the device out of it's normal routine, being able to skip, for example, authorization processes. However, fortunately protections against these attacks already exist. It is possible to protect against these glitching attacks within the silicon chips, they come provided with glitch detection on both the hardware level, and the software level. The key is knowing when a glitch is attempted, a specific part, called the voltage glitch detector, can be used. It is provided as IP (intellectual property) by INVIA [10].

2.6 Hazards

The before mentioned Tomasulo algorithm has the purpose to prevent hazards in the instruction scheduling by renaming registers in order to prevent other instructions from using the wrong value. However, what Tomasulo didn't account for is for the instructions to not be executed at all, which is what the attacks of voltage and clock glitching aim to do. In section 2.1 we have seen that a computer program consists of a large sequence of instructions. These instructions are then translated into machine code which can be read by the CPU and its functional units to execute said instructions. When we want to target an encryption algorithm to try and exploit it, we first need to know how our algorithm actually works, before we want try and attack it. The example that will be discussed in the thesis, is known as the *Advanced Encryption Standard*, or AES for short. AES is a form of encryption that works in a number of rounds, each performing the same operations, and a final round. Due to the round-like nature of the algorithm, and it doing the same loop, breaking out of it has disastrous consequences to the encryption.

2.6.1 AES briefly explained

AES is a form of encryption established by the National Institute of Standards and Technology of the US in 2003 [20]. It is a block cipher algorithm aimed at encrypting certain plaintext into ciphertext, making it an *encrypted* piece of text which contents are unknown without the *key*. The algorithm works in different ways depending on the size of the keys. The smallest size being 128 bits, and the biggest being 256. The higher the size of the key, the safer the algorithm but also the slower, which is an important consideration in the choosing of the size of the key [3]. To showcase the workings of AES, consider Figure 7:

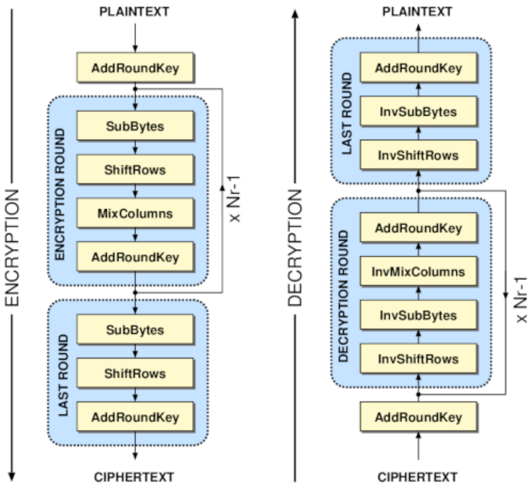


Figure 7: The AES Structure [8]

The `N-1` arrow in the encryption side of the figure represents the different rounds that the algorithm does. Because the operations in the *encryption round* are performed multiple times. For 128-bit keys, there are 10 rounds. For 192-bit keys 12, and for 256-bit sized keys there are 14 rounds total. Each round has a fixed set of operations performed on the plaintext, which has the ciphertext as output. The fact that there are multiple rounds, and that there is a check whether or not the last round has already been reached, is the reason why this algorithm is vulnerable against glitching attacks. It should be noted however that exploitations like these only tend to work on branching type algorithms. When a certain program is compiled, the higher level code (syntax as we write it in C, Python, etc.), is translated into lower level assembly code. This assembly code consists of instructions that represent the same operations that are written in the high level code (see Figure 2). The assembly *instruction set architecture* or ISA for short, has an instruction that's called a *branching instruction*. For example, the `bne` instruction checks whether or not the equal flag, set by another type of instruction, is set. If so, the branch will be taken, and the program jumps to another piece of code that the usual control flow (if the branch is not taken), would not go to. The instruction that sets the flag is usually a *compare* instruction, `cmp` in assembly.

2.6.2 Exploiting

Now we'll take a look at an actual example of how AES tends to be exploited by looking at a real implementation of AES. In this instance, we'll take a look at a code snippet used in a tutorial on breaking AES, provided with the ChipWhisperer firmware [17]:

```
static void Cipher(void)
{
  volatile uint8_t round = 0; // Needs to be volatile for attack

  // Add the First round key to the state before starting the rounds.
  AddRoundKey(0);

  // There will be Nr rounds.
  // The first Nr-1 rounds are identical.
  // These Nr-1 rounds are executed in the loop below.

  for(round = 1; round < Nr; ++round)
  {
    SubBytes();
    ShiftRows();
    MixColumns();
    AddRoundKey(round);
  }

  // The last round is given below.
  // The MixColumns function is not here in the last round.

  SubBytes();
  ShiftRows();
  AddRoundKey(Nr);
}
```

As we can see in this program, firstly the first `AddRoundKey` operation is performed, followed by a for loop lasting for the amount of rounds necessary. `Nr` is a constant variable corresponding to the size of the key. Here, `Nr` will be 10 since we are using a 128-bit sized key. When we look at the assembly, given in section 5, we can see that if we skip the `cmp r3, #9` instruction, the compare flag is never set, and therefore the branch will not be taken, skipping all the encryption rounds completely and heavily crippling the *confidentiality* of sent text! When using the ChipWhisperer, the tutorial gives an overview of all the available tools, objects, and variables in order to connect to the board, and inject the glitched clock cycles into the system. However, since it's not clear from the beginning when the code reaches the critical point, it requires a lot of trial and error in order to find the ”sweet spot” to insert our glitch.

2.7 Fault analysis and prevention of clock glitching

The implementation of the AES algorithm in the previous section was however not a true real life example. Usually when code like this is implemented it is not always done in the same way, this code has been specifically built and modified for the tutorial in order to teach how to properly execute the attack [17]. When looking at the tutorial specifically, we can see that when looking at the assembly code of the unmodified AES algorithm one round of AES cannot be skipped. However, when changing the round variable that keeps track of how many rounds have taken place to a *volatile* variable, all rounds of AES can effectively be skipped! This heavily implies that within the code that's written modifications to ensure that the encryption cannot be skipped can be made.

However, this does not fundamentally solve our issue. Even though it is possible to prevent the entire encryption to be skipped, having a skip of even one encryption round, or step, can already be a massive issue. This is due to the fact that if we know at what specific step and loop iteration the glitch was inserted, it will not result in the same ciphertext as when all the rounds were completed, leading to a faulty ciphertext. Problems start to arise when this faulty text is analyzed and compared to the text that is produced when all iterations move normally. This is where the phenomenon known as *fault analysis* comes into play. Fault analysis is a cryptographic attack based on injecting faults into cryptographic implementations to expose internal states [22]. Let's take Figure 8 for example which showcases the last two rounds of AES. If a fault is induced in, for example, the last round, but just before it commences, we obtain a faulty ciphertext as output. This ciphertext is the result of inducing a fault on one bit of a certain byte of the temporary ciphertext (temporary since the encryption isn't complete yet). If we then let the encryption algorithm continue normally, and then try it with different samples to verify if our key round key guess was correct. Once this can be verified, we can then apply the inverse of the Key Scheduling in order to obtain our AES key K. By using 3 faulty ciphertext, the odds of obtaining the byte successfully are around 97% [6]. The paper of [6] goes into much more detail about this attack on AES, I highly recommend reading this paper as it perfectly explains the attack and how the attack is structured, along with elaboration on how these odds and bytes are deduced.

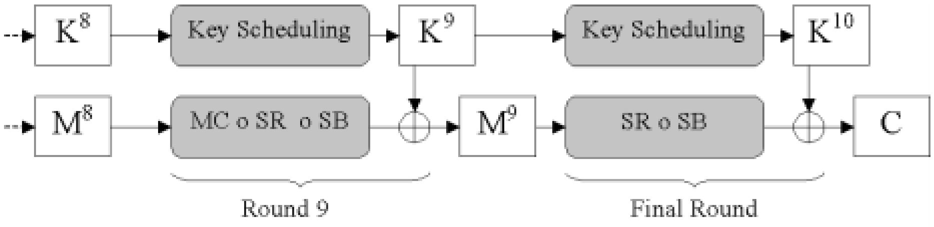


Figure 8: The last two rounds of AES [6]

Lastly, a solution might not only be limited to how the code is written. As we have seen before with the voltage glitching attack, it is also possible to take a more hardware focused approach. While the hardware solution of the voltage glitching attack does not protect against faulty clock cycles, it could be a great starting point to build further upon. A multi purpose anti-glitch chip could be produced to ensure the integrity of the clock signals and electrical pulses, although this might also bring the issue that these parts themselves are also dependant on these same signals and pulses. More thoughts on this will be present in section 4. It should be noted however that there are many more factors going into fixing the hazard in this way. Since it requires manufacturers to implement specific new hardware for an attack that is already not as widely used as other prevalent attacks such as the buffer overflow. Even though the prevalence of these glitching attacks is on the rise [18], choosing to secure it in this way may not be as lucrative at the moment, because if fundamental changes to the micro architecture of the CPU are necessary to ensure that the security measures are performing adequately, it might be very costly.

2.8 Teaching

For this thesis, the main goal was not only to explore the ChipWhisperer. Another very important part of the research was to provide a hands-on tutorial for future masters students in a new course for the spring semester, starting next year. The course, Cryptographic Engineering, will focus on creating a more deep understanding of cryptography. It will be focused on both the hardware and the software side of the implementation of cryptographic systems [14]. For this course, a tutorial has been made for performing a clock glitching attack, a voltage glitching attack, and also, another tutorial has been made for performing side channel attacks [2].

When researching on how to teach something entirely new to students (note that this course will be the first course at Leiden University that will have dedicated practicals to hardware related security), it is important to first understand how to actually bring over new knowledge. One framework for the structure of explanation, which has been used in this course, is known as the *semantic wave* framework, seen in Figure 9:

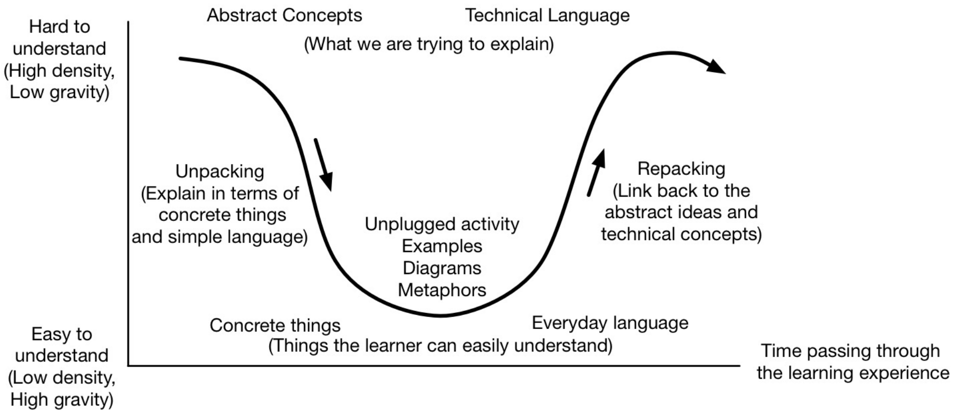

Hard to understand (High density, Low gravity)
Abstract Concepts
Technical Language
(What we are trying to explain)
Unpacking (Explain in terms of concrete things and simple language)
Unplugged activity
Examples
Diagrams
Metaphors
Repacking (Link back to the abstract ideas and technical concepts)
Easy to understand (Low density, High gravity)
Concrete things
(Things the learner can easily understand)
Everyday language
Time passing through the learning experience

Figure 9: Diagram showcasing an example of a semantic wave [4]

When looking at the figure, we see some sort of parabolic graph, starting high but then "dropping" to a local minimum. This drop is also known as *unpacking*. During the unpacking, the explanation changes from using more technical terms to the more simple language that is a lot more simple to understand. With unpacking the explanation becomes more concrete, which helps new students to better understand what they're dealing with. The *sematic density* drops, while the *semantic gravity* rises.

Semantic gravity (SG) is defined as as the degree to which the meaning of something is related to the context of that meaning [13]. What that means is that the stronger the SG is, the more dependant the meaning of a given concept is on it context. Which means that the weaker the SG, the less dependant it is on context. An example: the meaning of a plants name in Biology has a higher semantic gravity than the species of said plant [13]. The name "Rose" is a good example of this. Since it could be a flower, but also, for example, a name. As we can see, context matters a lot here, therefore "Rose" embodies a higher semantic gravity.

Semantic density (SD) however, is referring to the degree of condensation of meanings [13]. In less dense terms, the degree of the usage of technical terms in explanations, and also of terms themselves. The semantic density of a certain term depends on the context in which the explanation is done. A research paper explaining terms of a branch tends to have a stronger density than explanations done in regular textbooks. Also terms that are in everyday use, for example: "gold" commonly refers to a shiny, yellow metal used in coinage and jewellery. However, within chemistry, the term may also have more meanings to it, like the atom number [13]. It should also be noted that strengthening and weakening SD is more so a relative process than an absolute process. Even concepts that get unpacked (SD lowered) might still be difficult to grasp, therefore it should be looked at from a more relative standpoint.

When we want to learn students something new, it is important that the SD and SG are being held into account and that the gravity lowers after introducing students to the new knowledge. It is of course much easier to understand concepts using everyday language instead of first having to figure out what all the dense, technical terms actually mean. And then to also use those same technical terms, that the students are unfamiliar with, to try and gain new knowledge, is a challenge that may be very inconvenient, but also unnecessary. Usually, when starting a course and when introducing new concepts, what is started with is a very abstract definition of said concept. From that abstraction, we unpack into a more everyday type of situation, therefore the student can become more affiliated with the concept, and can more easily understand what is actually going on. Afterwards, we *repack*, and try to relate back from our unpacked understanding to the abstract definition that was started with. This process of unpacking and repacking, is the key of the framework, the semantic wave.

3 Methods

In this section, some more discussion will be held about the research performed for the thesis, alongside more elaboration on the actual goals and actions related to the subject.

3.1 The research

When getting the ChipWhisperer, the first task at hand was to go through all the relevant tutorials provided by NewAE. In these tutorials, as mentioned before, these tutorials are meant to be an assistance in helping to understand how to start using the board yourself. There are tutorials for getting to know Jupyter Notebook, one that's meant to test the connection between your PC and the chip and also tutorials on multiple subjects, not only clock glitching. For example, side channel attacks, or tutorials on power analysis and CPA attacks [21] are also available. It's clear that it can be hard for a student to easily find the relevant tutorials on the subject they're supposed to learn about, since there's a lot of different ways to use the ChipWhisperer board, and this is exactly what makes it so interesting. But nonetheless, the task at hand at first was going through all the tutorials, and understanding what the tutorials are trying to learn to the user.

After that has been done, the next job was to look what parts of the tutorial might have been missed, it's unnecessary to reinvent the wheel, but it might be a good idea to make it easier for the future students to have all the necessary information in one place. This is what the tutorial aims to solve, by using only this tutorial, a student should be able to set up the board by getting all the necessary prerequisites, and properly be able to use the guide in the tutorial to properly mount a clock glitching and/or voltage glitching attack.

Lastly, another task was to bring up something new. The tutorials merely give ways to attack, but the long term goal is of course to keep going with the attack maneuvers learned from these guides and apply them to newer situations. There are multiple ways of achieving this, which is the goal of the assignment related to the practical part of which the tutorial is about. Two examples will be discussed. The first one is to use an algorithm not discussed in the guide, and trying to break it using clock glitching or voltage glitching, and making a report on how this has been achieved, alongside the used scripts. There are many, many more encryption algorithms out there, AES and RSA are not the only ones that have the potential to be broken by these attacks. It's up to the student to decide which one to use. With the SimpleSerial protocol (more on that later), other algorithms can be compiled and sent to the board. Another variant is to implement a defense mechanism in order to actually detect said clock glitching attacks and preventing them from taking place, making sure that the integrity of said encryption is ensured and is completed accordingly. Clock glitches, for example, may not always work, and this trait can be used and exploited in order to prevent it from taking place. However, this might significantly influence performance! This should also be taken into account when implementing said mechanism, since the most obvious choice for a mechanism like this is simply to run the code multiple times, and compare the results. If the cipher texts for the same plain texts are not the same, we know something went wrong, and actions can be taken accordingly.

3.2 Setting up the board

When unboxing the ChipWhisperer and opening it up (see Figure 10), it comes provided with a USB Cable, the board itself, a handout providing some information about the testing of the board (to make sure it's operating correctly when coming out of the box), other information like the serial number are also provided on this handout. Another important handout that is also provided is the quick start guide, this guide is supposed to give some background information about the board, and about setting up as well, but there is one issue with this specific handout, the wiki linked to by the guide leads to a webpage that no longer exists! When copying the url from the quick start guide (https://wiki.newae.com/CW1173_ChipWhisperer-Lite), it leads to a page saying that it has been moved. However, when clicking the link provided by this page, it leads to a 404 page! This leads to a lot of confusion since now it's up to the student to find out for themselves how to set up the board without a guide that was provided directly by the package.

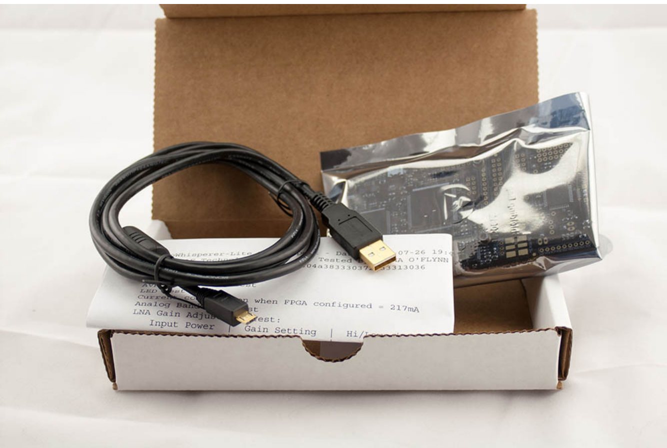


Figure 10: The contents of the ChipWhisperer Lite Box (quick start guide missing)
Source: `newae.com`

The solution to this has been provided as a goal and product for this thesis. A tutorial on how to execute certain attacks on cryptographic algorithms has been provided. However, another very essential part of the tutorial is a clear and concise guide on how to actually set up our board. This tutorial contains a platform dependant guide in order to properly gain the necessary programs in order to start programming with the board. Even a student without any experience with the board should be able to use this tutorial in order to easily start using it. The board typically makes use of Python using Jupyter Notebooks in order to communicate with the board and set it up accordingly. Jupyter Notebook is an environment which allows you to edit and run *notebook documents* containing computer code (e.g Python or Bash for compiling algorithms), and also regular text elements [12]. Using these notebooks it's possible to easily run Python/Bash code on demand in order to learn step-wise how every part of the notebook works, and also on how to use it in a more practical manner for the glitching tutorial. The ChipWhisperer comes provided with such a tutorial on how to use Jupyter Notebook, which is automatically installed on your system when following the guide given in the tutorial. A fragment of this introductory tutorial can be viewed in the appendix.

3.3 Mounting an attack

All of the provided tutorials from NewAE come in the form of these before mentioned Jupyter Notebooks. With these notebooks an attack can be executed in a number of steps, depending on the attack we choose to do. NewAE provides a python API full of functions that can be used to communicate with the board and perform certain operations. For example, it's possible to send messages to the board in plain-text, and the board can then perform a certain encryption algorithm and send the encrypted text back. However, in order to actually make use of these algorithms these first need to be compiled, using the installed compilers from the tutorial, and then also needs to be sent to the board in order to use it. The protocol which allows us to properly communicate with the board is called the SimpleSerial protocol.

In general, an attack using the board goes as follows:

1. Firstly, define the variables needed for the compilation of the source code. ChipWhisperer has multiple environmental variables for the different kinds of boards, and also different variables for the SimpleSerial version and the scope type (differs between the voltage glitching and clock glitching attacks). But for all intents and purposes, setting these variables goes in the same way as setting environmental variables in bash, and need to be the first code cell to be run in every attacking script.

   We will now shortly discuss how an attack using the ChipWhisperer looks like, we're trying to attack the TINYAES128 implementation mentioned before:

   ```
   SCOPETYPE = 'OPENADC'          # Scope type for clock glitching
   PLATFORM = 'CWLITEARM'         # Depends on the version of the board
   CRYPTO_TARGET = 'TINYAES128C'  # None if not trying to break encryption
   SS_VER='SS_VER_2_0'            # Ver 1.1 also available
   ```

2. As the second step, the setup script from NewAE has to be run in order to find the USB device (note that the chipwhisperer is connected to the PC using a USB connection). Additionally, the necessary variables to flash the source code to the target of the board (so it will run encryption algorithms) are also declared here for later use in step 4.

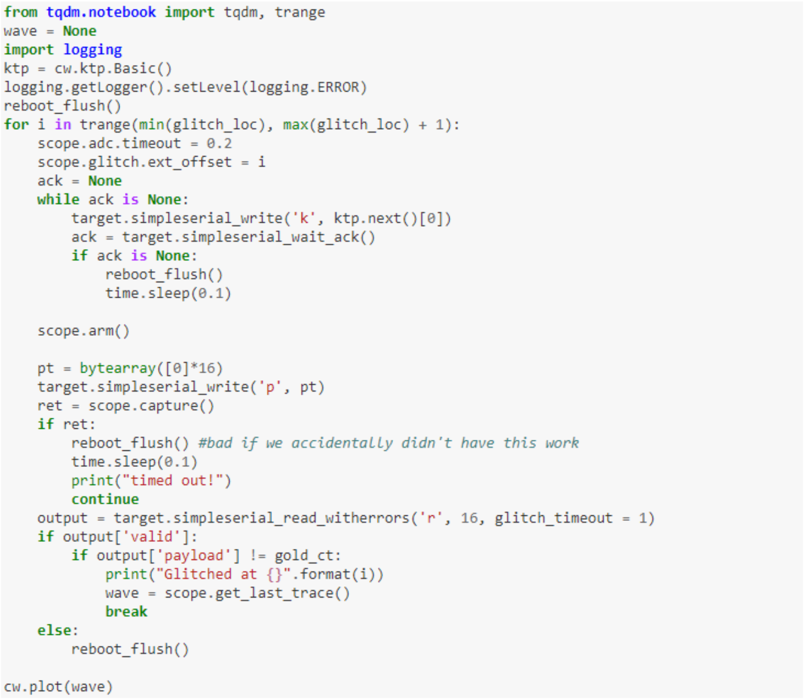
3. In this step, the source code written in C is compiled using the ARM-GCC compiler (since we, of course, have an ARM target on the ChipWhisperer). Via this compilation, a load file for the flashing of the firmware for step 4 is created **NOTE**: steps 2 and 3 are interchangeable, but both have to be completed before step 4!.

4. The compiled code is flashed to the target on the ChipWhisperer board. ChipWhisperer defines the function `program_target` which is used in order to achieve this. The other variables going into the function have been defined in step 2 using the setup script, so the order of steps is very important!

5. In this step some more necessary functions and variables are defined. The board has several variables that are necessary to tweak in order to successfully execute said attack. Also, a function to reset the cryptographic target, `reboot_flush`, is defined here. Clock glitching may cause the target that's running the code to crash, resetting it prevents it from stopping to respond altogether and allows multiple glitch injections. Reboot flush is defined as follows:

```
def reboot_flush():
scope.io.nrst = False
time.sleep(0.05)
scope.io.nrst = "high_z"
time.sleep(0.05)
#Flush garbage too
target.flush()
```

6. After all the preparation steps have been done, it's time to try and break the encryption! For this specific example of AES, we're trying to prevent the loop from taking place by glitching in just the right spot. We want this to happen on the last bit of the `AddRoundKey(0)` operation, so that the for loop is never entered at all. This can be done by using a glitch loop and trying to inject glitches until we find one. Consider the following figure for an example of the loop:



```
from tqdm.notebook import tqdm, trange
wave = None
import logging
ktp = cw.ktp.Basic()
logging.getLogger().setLevel(logging.ERROR)
reboot_flush()
for i in trange(min(glitch_loc), max(glitch_loc) + 1):
    scope.adc.timeout = 0.2
    scope.glitch.ext_offset = i
    ack = None
    while ack is None:
        target.simpleserial_write('k', ktp.next()[0])
        ack = target.simpleserial_wait_ack()
        if ack is None:
            reboot_flush()
            time.sleep(0.1)

    scope.arm()

    pt = bytearray([0]*16)
    target.simpleserial_write('p', pt)
    ret = scope.capture()
    if ret:
        reboot_flush() #bad if we accidentally didn't have this work
        time.sleep(0.1)
        print("timed out!")
        continue
    output = target.simpleserial_read_witherrors('r', 16, glitch_timeout = 1)
    if output['valid']:
        if output['payload'] != gold_ct:
            print("Glitched at {}".format(i))
            wave = scope.get_last_trace()
            break
    else:
        reboot_flush()

cw.plot(wave)
```

Figure 11: Example of glitch loop, it doesn't have to be the exact same as this one in order to work!

What essentially happens in this loop, is that for every value in between the self defined ranges of where you want to glitch, the loop sends a simple plain text containing only zeroes to the board. This is then written to, and then the output (meaning the encrypted text), is read back by the program. If this output and the originally defined ciphertext `gold_ct` don't match, we know a succesful glitch has been peformed, and the loop stops early.

7. The last line in the loop, `cw.plot(wave)`, makes a plot of the power trace of the executed algorithm. When a certain algorithm is ran on the ChipWhisperer, it gives a certain power trace. By looking at this power trace and comparing it to the original AES power trace we can try and narrow down where the glitch location actually is. A power trace of a glitched AES execution will look very different to the normal AES execution, since the 10 encryption rounds never actually happen! For reference, Figure 12 shows a normal AES power trace looks like:

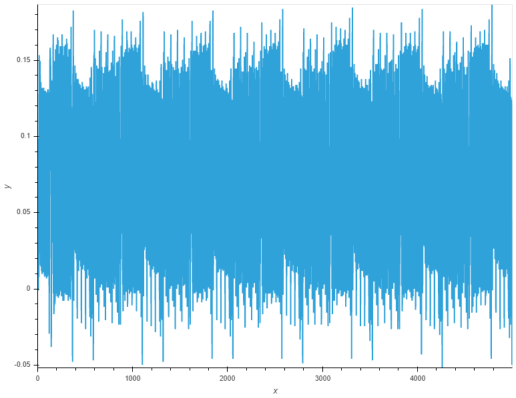
   

   Figure 12: Normal AES power trace

   Figure 13 shows a glitched power trace:

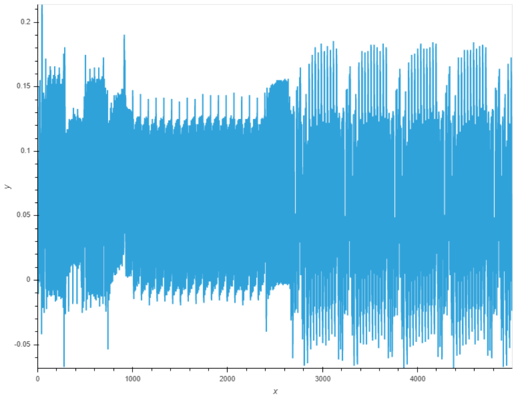
   

   Figure 13: Glitched AES power trace

8. Then, this loop is executed twice more with different plain-texts (recall that the odds of getting the correct key with three glitches is 97%! [6])

9. Using these cipher-texts and the scheduled keys, a key guess is made by doing the inverse of the AES operations.

4 Conclusions and Further Research

In this section, we'll conclude this thesis. We'll look back at the research question, and also look ahead into the possibilities of further researches and potential.

4.1 Concluding

To recall, the research question was as follows:

In which ways can we use the Chipwhisperer-board for a hardware security or Cryptographic engineering course in the Computer Science Masters program?

As we have seen in this thesis, there is a lot to unpack for trying to understand the concepts of the ChipWhisperer and the methods of attacking the algorithms. From doing the setup of the board to understanding the low level concepts of CPU and clock signals, to knowing how to read the assembly of said algorithms and trying to figure out how to break them and if possible, at what points. There are many aspects of Computer Science that are being combined into this project. From software related problems, like the implementation of algorithms in C, and knowing when they can be broken, to understanding how it works on the hardware level.

A good answer to the research question could perhaps be another question: how can't it be used? Not only can the board be invaluable in knowing how to break encryption in a unique way, also the encryption itself can be learned by using this board, as it provides all the resources to also test out the encryption itself in normal situations, simply by sending messages of plain text to the board and receiving the cipher text back. But to properly conclude and to answer our research question: The ChipWhisperer can be used in order to learn about multiple types of cryptographic attacks, practice them, learning how the algorithms that are attacked are implemented, and also to learn how to carry on on your own with the board and explore the many possibilities of attacks and variations. Both in attack types and algorithms that are being attacked these variations can exist.

4.2 Discussion / Further research

However, as mentioned before, the potential of this board has not been fully extracted in this thesis. The focus with this research subject was mostly on the subjects of using voltage glitching and clock glitching to exploit encryption algorithms like AES, but the board has many more possibilities. The goal of the assignment is also to make the student try to explore these other possibilities and let them come up with something not mentioned before. This can be in the form of using a new algorithm and trying to break it with these attacks. Or perhaps implementing defense mechanism, or maybe even combining forms of attacks together. Alternatively, what also could be worth looking into is the influence of other factors when trying to perform said attacks. Factors like temperature might play a role as well. And lastly, in the tutorial of the clock glitching attack on AES there was a specific example of the TINYAES implementation without the `round` variable being volatile. By specifically changing this variable to have this property already made the algorithm that much more vulnerable, so more research into writing code that resists these types of attacks is certainly also one worth performing.

References

[1] Michel Agoyan et al. "When clocks fail: On critical paths and clock faults". In: *International conference on smart card research and advanced applications*. Springer. 2010, pp. 182–193.

[2] Gijs Burghoorn a.k.a CoastalWhite. *Power analysis Introductory Walkthrough*. 2021. URL: https://coastalwhite.github.io/intro-power-analysis.

[3] Michael Cobb. *Advanced Encryption Standard (AES)*. URL: https://searchsecurity.techtarget.com/definition/Advanced-Encryption-Standard.

[4] Paul Curzon. *SEMANTIC WAVES*. June 29, 2019. URL: https://teachinglondoncomputing.org/2019/06/29/semantic-waves/.

[5] Eric DeBusschere and Mike McCambridge. "Modern game console exploitation". In: *CSc 566 2012: Research Presentations* (2012), pp. 466–566.

[6] Christophe Giraud. "DFA on AES". In: May 2004, pp. 27–41. ISBN: 978-3-540-26557-3. DOI: 10.1007/11506447_4.

[7] William Glyde. *How CPU's are designed and built*. July 9, 2020. URL: https://www.techspot.com/amp/article/1821-how-cpus-are-designed-and-built/.

[8] Frank Kagan Gürkaynak. *Advanced Encryption Standard (AES)*. July 11, 2021. URL: https://iis-people.ee.ethz.ch/~kgf/acacia/c3.html#tth_sEc3.2.

[9] Intel. *What is clock speed?* July 10, 2021. URL: https://www.intel.com/content/www/us/en/gaming/resources/cpu-clock-speed.html.

[10] Invia. *How a voltage glitch attack could cripple your SoC or MCU - and how to securely protect it*. July 11, 2021. URL: https://www.design-reuse.com/articles/48553/how-a-voltage-glitch-attack-could-cripple-your-soc-or-mcu.html.

[11] JavaTpoint. *Functional Units of Digital System*. July 10, 2021. URL: https://www.javatpoint.com/functional-units-of-digital-system.

[12] Jupyter. *Jupyter Notebook Quick Start Guide*. July 16, 2021. URL: https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/what_is_jupyter.html.

[13] Karl Maton. "Making semantic waves: A key to cumulative knowledge-building". In: *Linguistics and Education* 24 (Apr. 2013), pp. 8–22. DOI: 10.1016/j.linged.2012.11.005.

[14] Prof.dr.ir. N. Mentens. *Studiegids Computer Science master 2021-2022 – Cryptographic Engineering*. 2021-2022.

[15] NewAE. July 8, 2021. URL: https://www.newae.com/chipwhisperer.

[16] NewAE. *Fault1_1 Introduction to clock glitching*. 2021.

[17] NewAE. *Lab1_1B, Loop skip fault attack in practice, sca201*. July 11, 2021.

[18] Bart Stevens. *Fault Injection Attacks: A Growing Plague*. Mar. 14, 2019. URL: https://www.eeweb.com/fault-injection-attacks-a-growing-plague/.

[19] Wikipedia. July 9, 2021. URL: https://en.wikipedia.org/wiki/Tomasulo_algorithm.

[20] Wikipedia. *Advanced Encryption Standard*. July 11, 2021. URL: https://en.wikipedia.org/wiki/Advanced_Encryption_Standard.

[21] Wikipedia. *Chosen-plaintext attack*. July 18, 2021. URL: https://en.wikipedia.org/wiki/Chosen-plaintext_attack.

[22] Wikipedia. *Differential fault analysis*. July 11, 2021. URL: https://en.wikipedia.org/wiki/Differential_fault_analysis.

[23] Wikipedia. *Flip-flop (electronics)*. July 11, 2021. URL: https://en.wikipedia.org/wiki/Flip-flop_(electronics)#D_flip-flop.

5 Appendix

5.1 TINYAES128 C and assembly, from section 2.6.2

C Implementation:

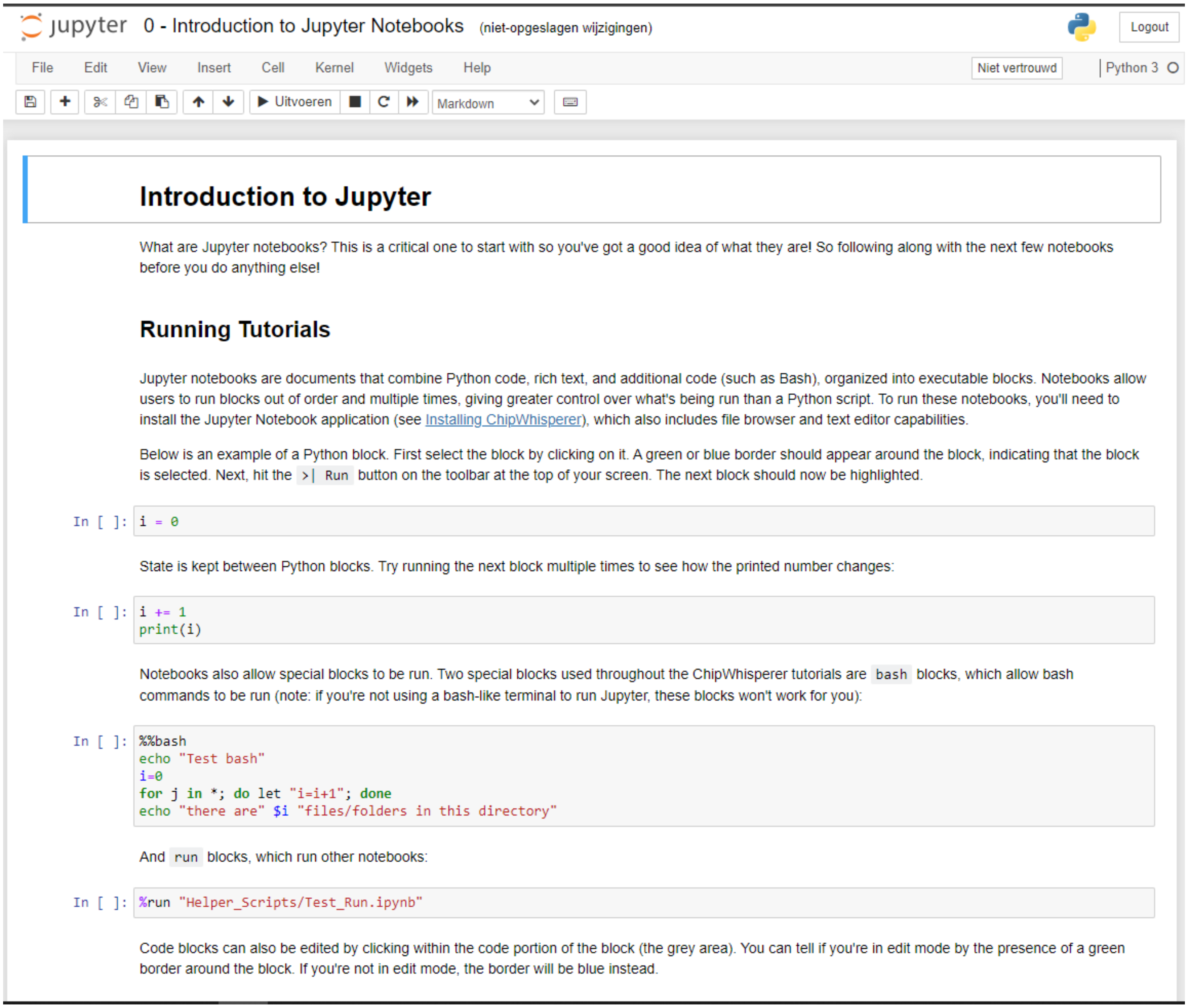
```
static void Cipher(void)
{
  volatile uint8_t round = 0;

  // Add the First round key to the state before starting the rounds.
  AddRoundKey(0);

  // There will be Nr rounds.
  // The first Nr-1 rounds are identical.
  // These Nr-1 rounds are executed in the loop below.
  for(round = 1; round < Nr; ++round)
  {
    SubBytes();
    ShiftRows();
    MixColumns();
    AddRoundKey(round);
  }

  // The last round is given below.
  // The MixColumns function is not here in the last round.
  SubBytes();
  ShiftRows();
  AddRoundKey(Nr);
}
```

Assembly code:

```
8001378:    e92d 4ff7     stmdb   sp!, {r0, r1, r2, r4, r5, r6, r7, r8, r9, sl, fp, lr}
800137c:    2000          movs    r0, #0
800137e:    f88d 0007     strb.w    r0, [sp, #7]
8001382:    4f29          ldr    r7, [pc, #164]    ; (8001428 <Cipher+0xb0>)

 // Add the First round key to the state before starting the rounds.
8001384:    f7ff ff9e     bl    80012c4 <AddRoundKey>

 // There will be Nr rounds.
 // The first Nr-1 rounds are identical.
 // These Nr-1 rounds are executed in the loop below.
8001388:    2301          movs    r3, #1
800138a:    f88d 3007     strb.w    r3, [sp, #7]
800138e:    f89d 3007     ldrb.w    r3, [sp, #7]
8001392:    2b09          cmp    r3, #9                  ; <-- round < Nr check
8001394:    d909          bls.n    80013aa <Cipher+0x32> ; <--

 // The last round is given below.
 // The MixColumns function is not here in the last round.
8001396:    f7ff ffaf     bl    80012f8 <SubBytes>
800139a:    f7ff ffc5     bl    8001328 <ShiftRows>
800139e:    200a          movs    r0, #10

80013a0:    b003          add    sp, #12
80013a2:    e8bd 4ff0     ldmia.w    sp!, {r4, r5, r6, r7, r8, r9, sl, fp, lr}
80013a6:    f7ff bf8d     b.w    80012c4 <AddRoundKey>
80013aa:    f7ff ffa5     bl    80012f8 <SubBytes>
80013ae:    f7ff ffbb     bl    8001328 <ShiftRows>
80013b2:    f8d7 10b4     ldr.w    r1, [r7, #180]    ; 0xb4
80013b6:    f101 0a10     add.w    sl, r1, #16
80013ba:    f891 9000     ldrb.w    r9, [r1]
80013be:    784d          ldrb    r5, [r1, #1]
80013c0:    788c          ldrb    r4, [r1, #2]
80013c2:    f891 8003     ldrb.w    r8, [r1, #3]
80013c6:    ea89 0005     eor.w    r0, r9, r5
80013ca:    ea84 0b08     eor.w    fp, r4, r8
80013ce:    ea8b 0600     eor.w    r6, fp, r0
80013d2:    f7ff ffc7     bl    8001364 <xtime>
80013d6:    ea89 0000     eor.w    r0, r9, r0
80013da:    4070          eors    r0, r6
80013dc:    7008          strb    r0, [r1, #0]
80013de:    ea85 0004     eor.w    r0, r5, r4
```

```
80013e2:    f7ff ffbf     bl    8001364 <xtime>
80013e6:    4045          eors    r5, r0
80013e8:    4075          eors    r5, r6
80013ea:    704d          strb    r5, [r1, #1]
80013ec:    4658          mov    r0, fp
80013ee:    f7ff ffb9     bl    8001364 <xtime>
80013f2:    4044          eors    r4, r0
80013f4:    4074          eors    r4, r6
80013f6:    708c          strb    r4, [r1, #2]
80013f8:    ea89 0008     eor.w    r0, r9, r8
80013fc:    f7ff ffb2     bl    8001364 <xtime>
8001400:    ea88 0800     eor.w    r8, r8, r0
8001404:    ea86 0608     eor.w    r6, r6, r8
8001408:    70ce          strb    r6, [r1, #3]
800140a:    3104          adds    r1, #4
800140c:    4551          cmp    r1, sl
800140e:    d1d4          bne.n    80013ba <Cipher+0x42>
8001410:    f89d 0007     ldrb.w    r0, [sp, #7]
8001414:    f7ff ff56     bl    80012c4 <AddRoundKey>
8001418:    f89d 3007     ldrb.w    r3, [sp, #7]
800141c:    3301          adds    r3, #1
800141e:    b2db          uxtb    r3, r3
8001420:    f88d 3007     strb.w    r3, [sp, #7]
8001424:    e7b3          b.n    800138e <Cipher+0x16>
8001426:    bf00          nop
8001428:    200006dc     .word    0x200006dc
```

5.2 Jupyter Notebooks



Introduction to Jupyter

What are Jupyter notebooks? This is a critical one to start with so you've got a good idea of what they are! So following along with the next few notebooks before you do anything else!

Running Tutorials

Jupyter notebooks are documents that combine Python code, rich text, and additional code (such as Bash), organized into executable blocks. Notebooks allow users to run blocks out of order and multiple times, giving greater control over what's being run than a Python script. To run these notebooks, you'll need to install the Jupyter Notebook application (see Installing ChipWhisperer), which also includes file browser and text editor capabilities.

Below is an example of a Python block. First select the block by clicking on it. A green or blue border should appear around the block, indicating that the block is selected. Next, hit the `>| Run` button on the toolbar at the top of your screen. The next block should now be highlighted.

```
In [ ]: i = 0
```

State is kept between Python blocks. Try running the next block multiple times to see how the printed number changes:

```
In [ ]: i += 1
        print(i)
```

Notebooks also allow special blocks to be run. Two special blocks used throughout the ChipWhisperer tutorials are `bash` blocks, which allow bash commands to be run (note: if you're not using a bash-like terminal to run Jupyter, these blocks won't work for you):

```
In [ ]: %%bash
        echo "Test bash"
        i=0
        for j in *; do let "i=i+1"; done
        echo "there are" $i "files/folders in this directory"
```

And `run` blocks, which run other notebooks:

```
In [ ]: %run "Helper_Scripts/Test_Run.ipynb"
```

Code blocks can also be edited by clicking within the code portion of the block (the grey area). You can tell if you're in edit mode by the presence of a green border around the block. If you're not in edit mode, the border will be blue instead.

Figure 14: Fragment of the introductory Jupyter Notebook guide, taken from NewAE's provided notebooks

As we can see here, pieces of Python code alongside bash commands are written in these code cells. When going through the notebook you can easily run the same cell multiple times in a row, this could come in use in for example testing or looking if a loop behaves properly when the code is ran for multiple iterations. The benefit of using these notebooks over whole Python scripts is that it's much easier to run specific portions of the code multiple times for testing purposes, all surrounding code is remembered, so it's still one whole script, but certain parts can be ran multiple times. The ChipWhisperer tutorials provide some glitching tutorials of themselves in these notebook forms as well, although the usage of regular python scripts is still possible.