A customizable side-channel modelling and analysis framework in Julia

Simon F. Schwarz
Robinson College

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for the degree of Master of Philosophy in Advanced Computer Science

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: sfs48@cam.ac.uk

May 31, 2021

Declaration

I, Simon F. Schwarz of Robinson College, being a candidate for the M.Phil in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 14707 TODO

Signed: 

Date: 30. May 2021

Abstract

In hardware security, side-channel attackers can monitor analog signals, like the per-instruction power consumed, during the execution of a cryptographic algorithm. These signals can depend on intermediate values of the cipher, which themselves depend on the secret key. Hence, with this additional side-channel information, reconstructing the key of the cipher may become feasible. For research purposes, such side-channel data is usually recorded with specialized hardware like oscilloscopes. However, those devices are expensive and require advanced knowledge to use. In this project, we create a framework for generating, analyzing and protecting side-channel data without the need for additional hardware.

This framework is written in Julia, a modern language that employs the multiple dispatch paradigm and has a flexible, dynamic type system. At its core, our framework provides custom types that behave like integers. When passing values of these types to a Julia implementation of a cryptographic algorithm, Julia's multiple dispatch mechanism automatically produces an instrumented or transformed version of that algorithm. Usually, this process requires only very few modifications to the algorithm's original implementation. In particular, we focus on two different functionalities that we can integrate via such custom types. Firstly, we will show how to construct types that emit a side-channel trace depending on the processed values. This removes the need for access to analog recording hardware, which is particularly useful when teaching side-channel security concepts in student practicals. Secondly, to explore protection against side-channel attacks, we create integer-like types that implement a range of techniques for splitting register values into multiple shares, which is a popular real-world countermeasure against side-channel attacks. Julia's parametric type system allows us to arbitrarily stack those types on top of each other. For instance, protection types can be stacked on top of logging types. This construction allows us to conveniently collect traces of protected data which can be, for example, used to verify the effectiveness of the protection.

Lastly, this project implements a range of classical and modern side-channel attacks. Those attacks, then, can either be performed against data directly collected with this framework, or against side-channel data that has been sampled from real-world implementations.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Aims . . . 2
- 1.3 Structure . . . 2

2 Background & related work on side-channel attacks **3**
- 2.1 Background . . . 3
- 2.2 Traces . . . 4
  - 2.2.1 Collection . . . 5
- 2.3 Attacks . . . 6
  - 2.3.1 DPA . . . 6
  - 2.3.2 CPA . . . 9
  - 2.3.3 Template attacks . . . 11
  - 2.3.4 Advanced attacks . . . 12
- 2.4 Countermeasures . . . 14
  - 2.4.1 Masking . . . 15

3 Background on Julia **18**
- 3.1 Types . . . 18
  - 3.1.1 Numerical types . . . 19
  - 3.1.2 Bits types . . . 19
  - 3.1.3 Parametric types . . . 20
- 3.2 Multiple dispatch . . . 20
  - 3.2.1 Value types . . . 22
- 3.3 Further reading . . . 22

4 Implementation **23**
- 4.1 Custom types . . . 23
  - 4.1.1 Integer-like types . . . 23
  - 4.1.2 Logging . . . 27
  - 4.1.3 Masking . . . 28
  - 4.1.4 Obtaining side-channel traces . . . 32
- 4.2 Ciphers . . . 33
  - 4.2.1 AES . . . 33
  - 4.2.2 SPECK . . . 34
- 4.3 Attacks . . . 35
  - 4.3.1 DPA . . . 35
  - 4.3.2 CPA . . . 36
  - 4.3.3 Template attacks . . . 37
  - 4.3.4 Key combination . . . 38

4.4 Technical details . . . 39
4.4.1 Installation . . . 39
4.4.2 Development . . . 40

5 Evaluation **41**
5.1 Evaluation of attacks . . . 41
5.1.1 CPA against AES . . . 41
5.1.2 CPA against SPECK . . . 43
5.1.3 Template attacks . . . 45
5.2 Extending foreign cipher algorithms . . . 46
5.2.1 Integrating the `AES.jl` implementation . . . 47
5.2.2 Integrating a RSA implementation . . . 47
5.3 Creation of student exercises . . . 47

6 Summary & future work **48**
6.1 Summary . . . 48
6.2 Future Work . . . 49

A Code changes for AES **54**
A.1 Changes to the file `aes-code.jl` . . . 54
A.2 Changes to the file `aesgf-code.jl` . . . 55

B Code changes for RSA **57**

List of Figures

1.1 Real-world side-channel analysis: An oscilloscope measuring the current consumption of a microchip. . . . 2

2.1 Power consumption during the execution of an AES encryption. The first peak is caused by loading the data, the subsequent ten peaks correspond to the ten rounds of AES. Data source: [1] . . . 5
2.2 A possible schematic wiring for collecting power traces with an oscilloscope. 5
2.3 The SASEBO side-channel evaluation board [2]. . . . 5
2.4 A recorded power trace, and the point in time where the first AES S-Box lookup is performed. . . . 7
2.5 Power consumption of all traces at the first S-Box lookup. . . . 7
2.6 Power consumption for the two partitions at the first S-Box lookup. . . . 8
2.7 DOM against time between the two positions. The spike marks the time of the first S-Box lookup. . . . 8
2.8 DOM against time for two key guesses. The spike for the correct key guess is at the position where the first S-Box output is calculated. . . . 9

3.1 Hierarchy of Julia's numerical types. Abstract types are blue, concrete types are red. . . . 19

4.1 Column-major array storing in Julia. The blue arrow shows the order in which entries are stored in memory. . . . 36
4.2 The memory layout of the trace arrays. . . . 36

5.1 Comparison between the power prediction for the correct key and the actual measured power at the point of the first S-Box computation ($t = 298$). Both are strongly correlated ($\rho = 0.87$). . . . 42
5.2 Correlation for the correct and for an incorrect key against time. The correct key exhibits a spike in correlation at the computation of the first S-Box output ($t = 298$). . . . 42
5.3 Maximal correlation for all key-byte values on real-world AES data. The correct key byte is $0x42$. . . . 42
5.4 Correlation for the correct and for an incorrect key against time on real-world AES data. . . . 43
5.5 Maximal correlation for all key-byte values on real-world AES data. The correct key byte is `0x13`. . . . 43
5.6 Correlation in SPECK for different key bytes (Correct key: `0x12`). . . . 44
5.7 Recorded values and templates for a sample 2-dimensional recording. In this example, the covariance matrix is diagonal and has only one eigenvalue. The value loaded was 3. . . . 45

5.8 Likelihood for each different value. The overall likelihood scales logarithmically with the probability. 45

Chapter 1

Introduction

1.1 Motivation

"Classical" cryptanalysis tries to analyze the security of cipher algorithms based on their mathematical specification. However, this analysis cannot account for security issues arising from the realization of an algorithm. Such issues are very common and can either originate from software or hardware flaws. For example, the NSA used its TEMPEST [3] program as early as in World War II, where they measured electromagnetic emissions to reconstruct secret messages written on a typewriter. More recently, side-channel attacks on popular cryptographic algorithms like AES have been published [4, 5]. Lastly, attacks like SPECTRE [6] or Meltdown [7] have gained popularity by abusing hardware flaws. All those attacks have in common that they utilize additional information gathered via an *unintended side-channel*, like electromagnetic emissions, timing information, or power consumption. *Side-channel analysis (SCA)* is the study of security considering the real-world realization of ciphers. Hence, potential side-channel leakages are explicitly taken into account. Therefore, SCA tries to close the gap between the formal security guarantees of a cipher and its real-world security.

An example setup for SCA could be an oscilloscope connected via a shunt-resistor to the power supply of a processor. The oscilloscope then repeatedly measures the power consumption in small intervals during the execution of a cryptographic algorithm. Such a setup is depicted in Figure 1.1. However, a setup like this is expensive and requires advanced knowledge to use.

Hence, SCA is currently not very accessible, which leads to less awareness about possible attacks. Ultimately, this can then lead to more vulnerable implementations. This project addresses this gap by providing a purely software-side solution for generating and analyzing side-channel information, thus removing the need for expensive hardware and specialized knowledge. In particular, since this project scales effortlessly, results can be used for teaching side-channel security without providing hardware to every student.

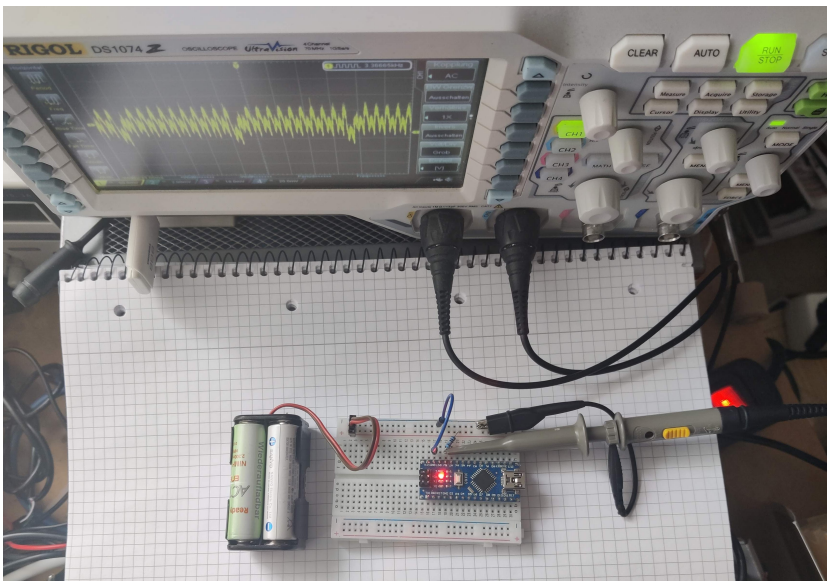

Figure 1.1: Real-world side-channel analysis: An oscilloscope measuring the current consumption of a microchip.

1.2 Aims

This project aims to ease the analysis of side-channel attacks and to eliminate the need for additional hardware. In our framework, a purely software-side solution for trace generation and analysis is implemented.

One goal of this project is the analysis of cipher implementations without major modifications to the original source code. With our framework, only the underlying types of implemented functions must be changed, while all of the other source code can be kept unmodified. This makes the whole process of analyzing a cipher convenient and easily reproducible. Equipped with this method to record leakage emissions from a cryptographic algorithm, our framework provides various methods for analyzing the recorded data with respect to side-channel security.

The project was implemented in the Julia language [8]. Julia is a modern high-performance language based on the *multiple dispatch* paradigm. In combination with Julia's flexible type system, this allows us to implement our framework in a generic way.

1.3 Structure

First, chapter 2 will summarize the background and related work on side-channel analysis, ending with some state-of-the-art attacks. Afterwards, chapter 3 will have a brief look at the background of Julia, especially focusing on Julia's type system and dispatch mechanisms. Next, chapter 4 will discuss this project's implementation in detail. In chapter 5 we will have a look at the results, and analyze two popular encryption algorithms with the help of our framework. Lastly, chapter 6 gives a brief summary of this thesis and highlights potential future work.

Chapter 2

Background & related work on side-channel attacks

In this chapter, we will have a brief look at the motivation and history of side-channel analysis in section 2.1. Afterwards, section 2.2 will define *traces* and look at their relevant properties for side-channel analysis. Using these traces, section 2.3 will focus on some classical and modern side-channel attacks in detail. Lastly, we will explore some countermeasures against different side-channel attacks in section 2.4.

2.1 Background

Cryptanalysis aims to study encryption systems with regard to their security. An important part of cryptanalysis is the "classical" mathematical analysis of an encryption algorithm. This part usually considers only the information directly present in the mathematical specification of the cipher. Such information is usually restricted to the input, the output, and the key. For example, a classical desirable property for an algorithm would be ciphertext indistinguishability, i.e. given a message and two ciphertexts, an adversary is unable to efficiently determine which of the two ciphertexts encrypts the message, even if it has access to an encryption oracle. Modern cryptographic algorithms like the Advanced Encryption Standard (AES) [9] or SPECK [10] combined with suitable modes of operation are generally considered secure in this classical sense.

However, the mathematical analysis (and claims of security made by it) solely rely on the high-level mathematical transformation described by the algorithm. Despite their security in a theoretic scenario, those algorithms are eventually implemented in software and executed on hardware. Both steps can potentially lead to vulnerabilities that cannot be captured by the high-level mathematical specification. Such weaknesses in implementation or hardware are studied in the area of *side-channel analysis*. In general, side-channel attacks exploit additional information about the execution of a cryptographic algorithm.

This information is not part of the cipher's mathematical specification and is gathered via *unintended side-channels*.

For example, one such unintended side-channel is the power consumption of the processor. Usually, a processor internally represents a binary number using wires with high or low voltage. When an operation on the number is performed the voltage level is then changed accordingly. In most hardware processors, pulling the voltage on a line high (or even low) consumes more energy than not changing the level at all. Hence, it stands to reason that, for example, computing $(0000)_2 + (0001)_2$ needs less power than computing $(1010)_2 + (1101)_2$, given that all wires are pre-charged to low voltage. This intuition is later on captured by a power estimation model, which will be discussed in section 2.3.2. Hence, by measuring the power consumption of the processor during cryptographic operations, it can be possible to draw conclusions about the processed values. Ultimately, the goal is to use this knowledge of intermediate values to infer the secret key.

Besides power side-channels, examples of such side-channels include, but are not limited to:

- The wall-clock runtime of the algorithm during an en- or decryption process. For example, Brunley and Boneh showed that it was possible to extract private keys from a remote webserver running OpenSSL ([11]) using a timing side-channel [12].
- The state of the processor cache, which may be modified depending on processed data. Prominent attacks include Meltdown [7] and SPECTRE [6].
- Electromagnetic, acoustic, optical, or other measurable emissions of the underlying device. Historically, NSA's TEMPEST program [3] falls into this category. Recently, Camurati et al. introduced so-called "Screaming Channels" [13], an attack where broadcasted data from mixed-signal chips is analyzed to recover secrets. Such mixed-signal chips are often used for Bluetooth or Wi-Fi and are common in modern mobile devices.

2.2 Traces

In this thesis, we will focus on side-channels that produce a *trace* of emissions during cryptographic operations. For example, if a power side channel is used, the measurement of power against time during a cipher operation forms a *power trace*. Figure 2.1 depicts an example of a plotted power trace during the execution of AES. Usually, only relative power consumption is considered when analyzing power traces. Thus, units are relative and are commonly directly taken from the output of the analog-digital converter used to record the trace.

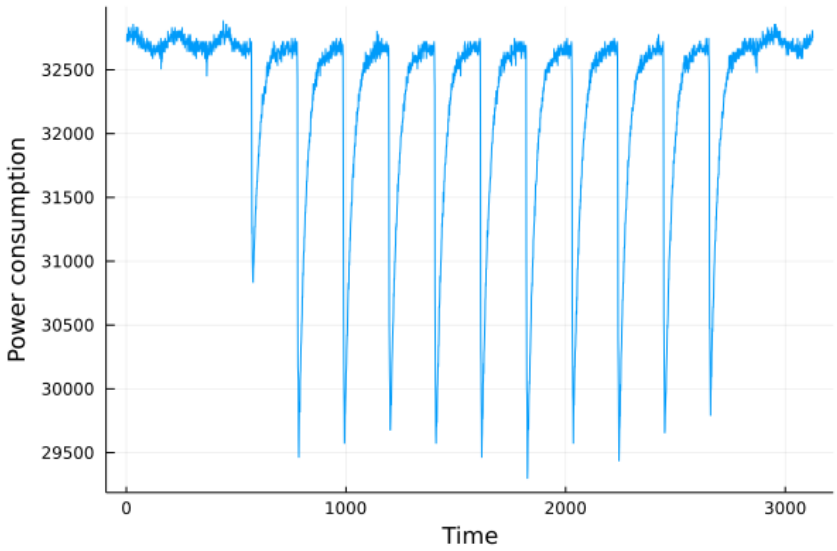

Figure 2.1: Power consumption during the execution of an AES encryption. The first peak is caused by loading the data, the subsequent ten peaks correspond to the ten rounds of AES. Data source: [1]

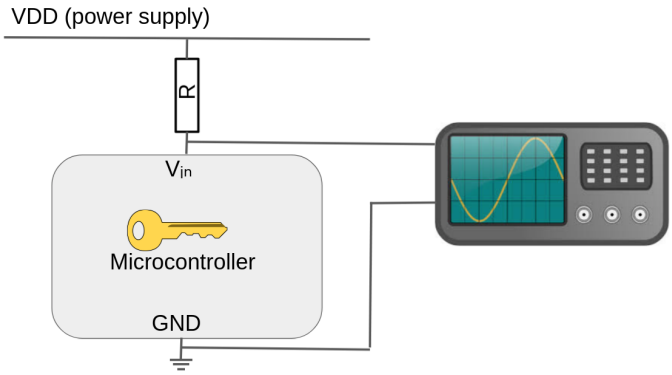

Figure 2.2: A possible schematic wiring for collecting power traces with an oscilloscope.

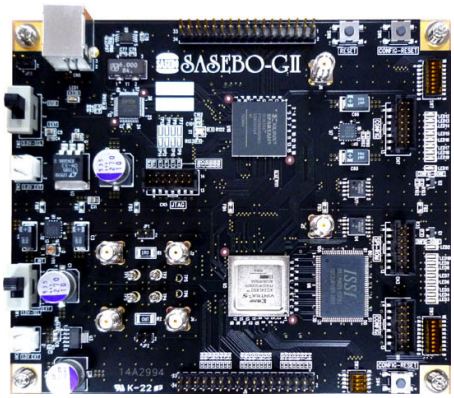

Figure 2.3: The SASEBO side-channel evaluation board [2].

2.2.1 Collection

Side-channel traces are usually collected using appropriate monitoring hardware. For example, power traces are can be collected with an oscilloscope connected via a shunt resistor to the device power or ground line, as shown in figure 2.2. A real-world setup of this schematic can be found in figure 1.1. For research purposes, there also exist various predefined evaluation boards for side-channel attacks, like the SASEBO [2] board family, depicted in figure 2.3. However, side-channel collection is not limited to this approach, as shown recently by the PLATYPUS attack [14]. In this attack, power traces are obtained via the processor's internal power statistics (Intel RAPL) that can be accessed by any user. Hence, this attack can also be carried out remotely without physical access to the device.

For most attacks, multiple traces from the same device and cipher are needed. Usually, those traces should perform the same encryption with the same key, but with *different*

input. A scenario where this attack model is suitable is, for example, a Smart-Card that encrypts values it receives. Now, different inputs can be sent to this chip to obtain multiple traces. The exact number of traces needed for an attack depends on a variety of factors including the recording quality (e.g. signal-to-noise ratio), implemented countermeasures, and the attack itself. Kocher suggests in [4] to use 4000 traces for his differential power analysis, while other authors collect up to 2^{32} traces for specific attacks. However, there also exist approaches that can break cryptographic systems with as little as one trace.

Alignment If multiple traces are collected, most attacks require traces to be *aligned*. This means that at every time point in the trace the same computation was carried out on the targeted chip. There are multiple different approaches to produce aligned traces:

If the monitored device provides a signal like a line pulled high as soon as processing begins, this line can be used as a trigger in the oscilloscope. However, a single trigger cannot account for misalignment due to oscillator tolerances. To solve this problem, a more advanced approach is to use an external oscillator that provides the same clock signal to the microchip and the oscilloscope. This allows for far more fine-grained measurements.

Another possible solution is to re-align traces after collection. Promising results have been achieved by the two approaches, called *phase-only correlation* [15] and *amplitude-only correlation* [16]. Both methods employ results from a Fourier transformation to re-align traces based on similarity.

2.3 Attacks

Historically, the first analysis of side-channel attacks has been introduced by Kocher in 1996 [17], where he conducted timing attacks on asymmetric ciphers. Successively, Kocher introduced Differential Power Analysis (DPA) and Simple Power Analysis (SPA) in 1999 [4]. The introduction of DPA marks a breakthrough in side-channel analysis and is explained in detail in section 2.3.1. Following this breakthrough, DPA has been extended and generalized. One such example is Correlation Power Analysis (CPA) which was published in 2004 [5] by Brier, Clavier, and Olivier. Details on CPA are discussed in section 2.3.2. Another branch of attacks that evolved from DPA are Template Attacks (TA), which were published by Chari, Rao, and Rohatgi in 2002 [18]. A detailed explanation of template attacks can be found in 2.3.3. Recently, more advanced side-channel attacks have been published. A perspective of current work is given in section 2.3.4.

2.3.1 DPA

Kocher's *Differential Power Analysis (DPA)* [4] requires multiple aligned traces. The traces should record a cipher operation on a different, random input to the cipher each time. Note that, however, the cipher algorithm and the key must remain static for this

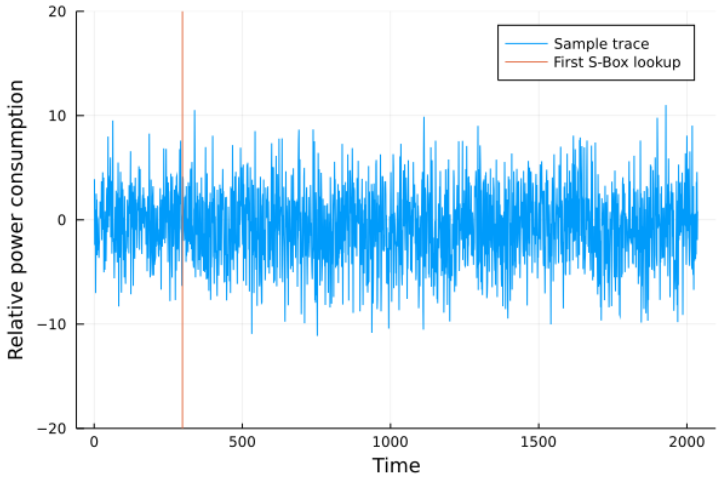

Figure 2.4: A recorded power trace, and the point in time where the first AES S-Box lookup is performed.

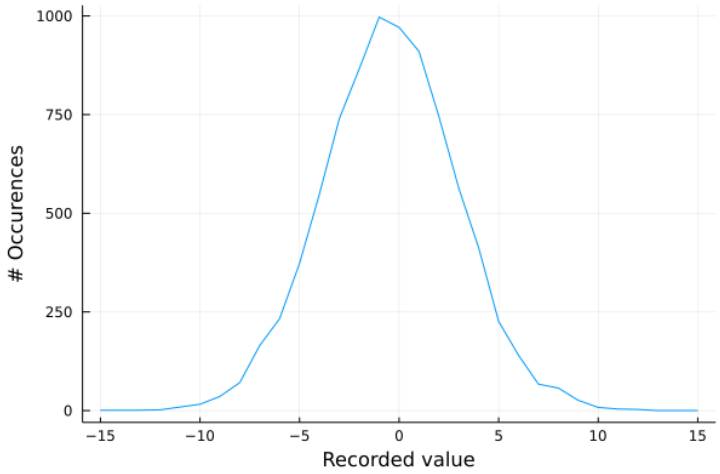

Figure 2.5: Power consumption of all traces at the first S-Box lookup.

attack to work.

Now consider a set of aligned traces recording an en- or decryption. Since all traces are aligned, at a fixed time in each trace, the same operation has been executed while recording. For example, if the recorded device performs an AES encryption, there is a time point where the first S-Box output is computed in all traces. Figure 2.4 shows a sample recorded trace where this specific time point is marked. Considering all traces, we can plot the frequency of a specific recorded value at the time point of the first S-Box lookup. Such a frequency diagram is depicted in figure 2.5.

The fundamental idea of DPA is now to partition the set of all collected traces into two subsets according to a guess of an intermediate value. For example, imagine that we know the secret key used in AES. Then, it is possible to calculate all intermediate values by just following the specification in the AES algorithm. Thus, we can partition the recorded traces from figure 2.5 based on “Is the least significant bit of the first S-Box output 0?”. The frequency of the two induced subsets is plotted in figure 2.6. Notably, the two subsets

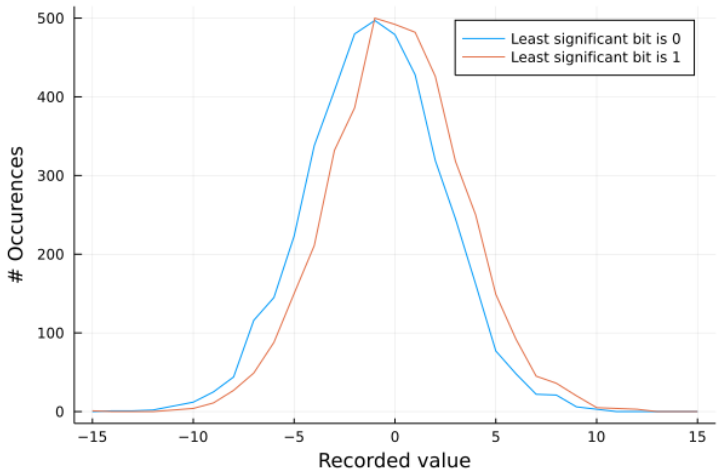

Figure 2.6: Power consumption for the two partitions at the first S-Box lookup.

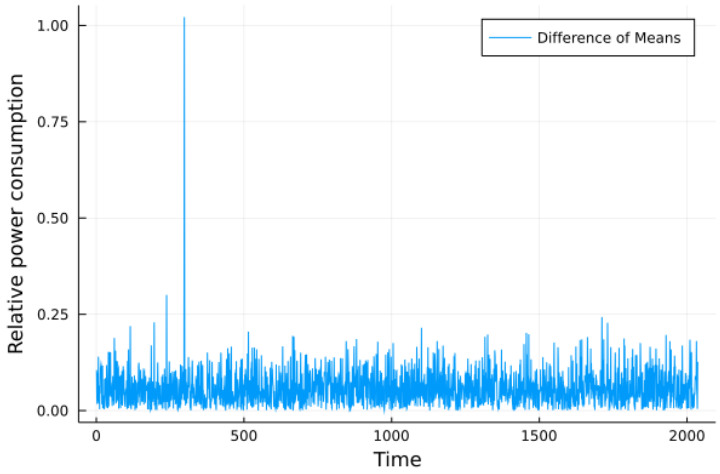

Figure 2.7: DOM against time between the two positions. The spike marks the time of the first S-Box lookup.

are distinguishable and have a different mean.

While analyzing traces, the exact point of the computation of the first S-Box output is not known a priori. Hence, it is necessary to not only compare the difference of means (DOM) at a single position, but rather consider it at all possible positions. Figure 2.7 plots the difference of means against the whole trace length, where the partitioning is performed upon the least significant bit of the first S-Box output. Naturally, the difference graph spikes at the computation of this value and approaches zero elsewhere. This is a good indicator that a value depending on the first S-Box output actually occurs in our trace.

Recall that we assumed to know the key before partitioning the traces based on "Is the least significant bit of the first S-Box output 0?". However, in an attack scenario, we do not directly know the secret key. In AES, the defining equation for the first S-Box output I_i at position i in the first round is:

$$I_i = S[X_i \oplus K_i]$$

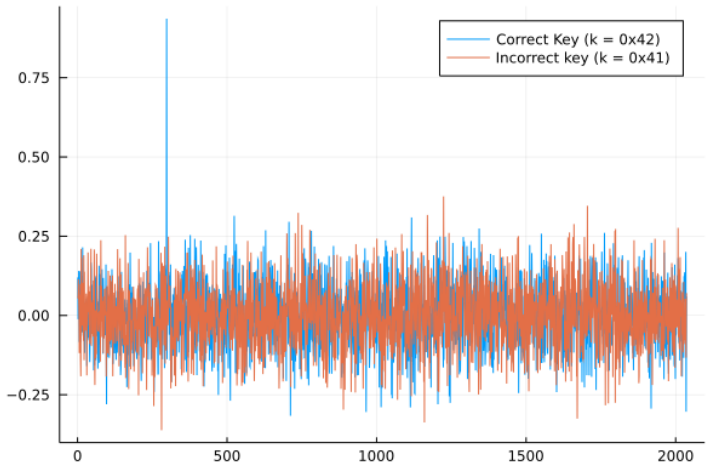

Figure 2.8: DOM against time for two key guesses. The spike for the correct key guess is at the position where the first S-Box output is calculated.

where K_i is the i-th key byte, and X_i is the i-th input byte. Now, note that S is the static and public AES substitution box, and X_i is known for each trace. Thus, the S-Box value only depends on the unknown K_i, which is only a single key byte.

Now, the already established method can be used as an oracle: For each possible key-byte guess $K_i \in \{0, \ldots, 255\}$, calculate I_i for each trace, and partition the traces based on the least significant bit of I_i. If the guess for K_i is correct, then I_i is correct as well. Hence, at the calculation of the first S-Box output, we expect the difference of means between both partitions to be significantly non-zero. Thus, this time point will show up as a "spike" if the DOM is plotted against time.

In contrast, for a wrong guess of K_i, there will be no correlation between I_i and the actual trace values, since I_i is never computed during the recording. Therefore, given enough traces, the difference of means will approach zero at all points, and there will be no "spike" in computation. Figure 2.8 compares typical DOMs for a correct key guess with an incorrect key guess.

The above method effectively provides an oracle that decides if a key byte guess K_i is correct. Given such an oracle, AES can trivially be broken: For each key byte, there are 2^8 different possibilities. For each possibility, the oracle can be queried, resulting in 16×2^8 queries in total. This is a major improvement compared to a naive brute-force attack, which would need up to $2^{8\times16}$ tries.

2.3.2 CPA

Correlation Power Analysis was proposed by Brier, Clavier, and Olivier in 2004 [5]. While DPA takes only a single bit for partitioning into account, CPA tries to use more of the available information. For this purpose, a *leakage model* is established. This model predicts side-channel values during the processing of different values.

Leakage models

For example, a leakage model for a power side-channel attack would be a power consumption estimate. Implicitly, such a model was already established for DPA by assuming: "Processing a value with LSB 1 takes more (or less) energy than one with LSB 0". However, this model can be improved by, for example, taking other bits into account. This idea is formalised in a leakage model. Such a model is a function receiving a value R that is processed, and should return a side-channel estimate W_R. published

Hamming weight model: For example, a simple model is the Hamming weight model. Formally, the Hamming weight of a b-bit number $R = \sum_{j=0}^{b-1} d_j 2^j$, $d_j \in \{0, 1\}$ is $H(R) = \sum_{j=0}^{b-1} d_j$. Then, our leakage model is

$$W_R = H(R)$$

This model captures a processor that consumes power for every set bit, for example, by pulling the corresponding wire up.

Hamming distance model: A generalized version of the Hamming weight model is the Hamming distance model. This model captures the idea that power leakage depends on switching bits, i.e. it consumes power to charge or discharge a line or gate. If the state before the computation of R is D, then the Hamming distance model is

$$W_R = H(R \oplus D)$$

Computing correlation

Given an estimate for the side-channel value, the goal of CPA is to check if the measured side-channel values indeed correlate to the prediction. For a reasonable leakage model, we would expect a linear correlation, which is captured by Pearson's correlation coefficient

$$\rho_{W,H} = \frac{\operatorname{cov}(W, H)}{\sigma_W \sigma_H}$$

It holds that $-1 \leq \rho_{W,H} \leq 1$, where $\rho_{W,H}$ is close to ± 1 for a good linear correlation.

The attack

To perform an overall CPA attack, a targeted value in the cipher is needed. For example, in AES this could be the output of the first S-Box. Then, with the input for each trace, a side-channel prediction is computed using the leakage model. Lastly, the correlation between the measurement and prediction is computed. This correlation, as in DPA, is computed for every time point in the trace separately. Now, at the time point where the targeted value is computed, we expect to see a strong correlation for a correct key

guess. Since this time point is not known beforehand, the maximum correlation at any time point is taken into account.

A detailed explanation of a CPA attack against AES can be found in the evaluation in section 5.1.1, an attack against SPECK in section 5.1.2.

2.3.3 Template attacks

Template attacks try to circumvent the restriction that only a limited number of traces may be collected from an attacked device. For this purpose, they are split into two phases: During the *profiling phase*, recordings from an identical experimental device are collected for different operations. Then, *templates* for expected side-channel emissions on certain operations are constructed from this data. In the *attack phase*, side-channel data from the attacked device is collected. This data is then classified for fitness to the collected templates. Likely, the best fitting template model will characterize the operation executed on the attacked device. A central approach here is to take noise into consideration. While all previously explained attacks try to cancel out noise by averaging, template attacks actively characterize noise and use it to extract information.

Technically, template attacks try to construct a template for each of K possible operations. For example, the different operations could be a load instruction of a single key byte. Then, the $K = 256$ different operations would be all possible values for the targeted byte.

In the profiling phase, for each operation $O_m \in \{O_1, \ldots, O_k\}$, a set of n side-channel vectors $x_i^m \in \mathbb{R}^k$ is collected from the experimental device. Next, the mean $\bar{x}_m$ and the covariance matrix $\mathbf{\Sigma}_m$ are computed:

$$\bar{x}_m = \frac{1}{n} \sum_{i=1}^{n} x_m^i$$
$$\mathbf{\Sigma}_m[u, v] = \mathrm{cov}(x_m^u - \bar{x}_m, x_m^v - \bar{x}_m)$$

Note that for large sample sizes, the mean and covariance matrix computed from sample traces will approach the true mean and covariance. The tuple of mean and covariance $(\bar{x}_m, \mathbf{\Sigma}_m)$ will then form our template for the operation O_m.

In practice, $\mathbf{\Sigma}_m$ often captures noise that is independent of the actual operation O_m. This motivates the use for a *pooled* [19] covariance matrix $\mathbf{\Sigma} = \frac{1}{k} \sum_{m=1}^{k} \mathbf{\Sigma}_m$. Since the pooled covariance can be computed from $k \cdot n$ traces, it is a better approximation of the true covariance compared to a single $\mathbf{\Sigma}_m$, which is calculated with only n traces. If a pooled covariance matrix is used, an operation is only identified based on its mean vector. Formally, the template for O_m then consists of $(\bar{x}_m, \mathbf{\Sigma})$.

A common assumption in side-channel analysis is that emissions can be modelled well by multivariate normal distributions. For multivariate normal distributions, mean and

covariance are already *sufficient statistics* and, thus, completely define the underlying probability. Hence, for a template $(\bar{x}_m, \mathbf{\Sigma}_m)$, the probability of observing a leakage vector N is

$$p_m(N) = \frac{1}{\sqrt{(2\pi)^n |\mathbf{\Sigma}|}} \exp\left(-\frac{1}{2}(N - \bar{x}_m)^T \mathbf{\Sigma}_m^{-1}(N - \bar{x}_m)\right)$$

Then, during the *attack phase*, we try to infer which operation $O_{m\star}$ was performed on the attacked device. Let y be a collected leakage vector from the attacked device. Then, the probability for each template $p_m(y)$ for $m \in \{1, \ldots, k\}$ can be computed. The resulting probabilities then denote the likelihood of $m\star = m$. Finally, $m\star$ can be inferred by an exhaustive search in order of descending probability of m.

2.3.4 Advanced attacks

All previously explained attacks use the principle of *divide and conquer*: First, the whole key (e.g. the 16 key bytes from AES) is divided into smaller pieces (e.g. single key bytes), and later combined to form the whole key. This approach is very powerful since almost no knowledge of the exact implementation is used. Moreover, only single values and single points in time are targeted. A clear benefit of this approach is, hence, the simplicity of the attacks. However, this poses the question if more information can be obtained from side-channel traces by considering multiple points or taking the structure of the cipher into account.

Algebraic Attacks

Algebraic attacks are a class of side-channel attacks where *algebraic properties* of ciphers are exploited. Clearly, all intermediate values that occur during the encryption process are algebraically related to each other. This relationship is publicly known by the specification of the algorithm. Attacks from this class then try to combine side-channel information from different time points with their algebraic relationships. For example, assume the Hamming weight of all intermediate values is known. Then, it is possible to obtain a system of equations from the defining equations of the cipher, and from the Hamming weight of all intermediate values. Hence, this system of equations should be overdetermined by the additional constraints on Hamming weight. Those additional constraints can make the resulting system easier to solve, for example, with a SAT- or SMT-Solver.

Some prominent algebraic attacks, in order of publication, are:

- *SPA on AES Key-Expansion*: In [20], Mangard provides a way to reconstruct the AES secret key by knowing the Hamming weights of all intermediate values of the key expansion. By exploiting the additional constraints on Hamming weight combined with properties of the AES key expansion, the time complexity of reconstructing the key is greatly reduced compared to a brute-force search. The resulting attack complexity is feasible in practice.

- *Collision attacks* against DES have been introduced by Schramm, Wollinger, and Paar in [21]. Subsequently, collision attacks have been generalized to cover AES in [22], and improved by Bogdanov, Kizhvatov, and Pyshkin in [23]. The overall idea is to exploit *collisions of intermediate values*. From those collisions, equalities between some intermediate values can be established. Ultimately, these equalities permit conclusions about the used round keys. [23] shows that for each collision, it is possible to infer 8 secret key bits. Following from the birthday paradox, collisions are very likely to occur even for a few (about 20) traces, which is a significant improvement compared to classical attacks.
- *Algebraic attacks* are a generalisation of the attacks outlined above. They were introduced by Renauld and Standaert in [24]. In their paper, they show how to use generic SAT-solving approaches to target all intermediate values, not only the ones occurring in the first or last rounds.

The major benefit of algebraic attacks is the need for vastly fewer traces compared to divide-and-conquer approaches. However, algebraic attacks also exhibit three major weaknesses:

- First, more knowledge of the internal structure of the targeted device is required. For example, it is necessary to know in which order values are computed to establish algebraic relationships. This poses problems especially when the attacked device is a *black box*, i.e. no further information about the software or hardware is known
- Second, algebraic attacks have little *error tolerance*. In the attacks outlined above, the inferred constraints on intermediate values were treated as "hard" constraints. However, in a real-world scenario, some of those values may be wrong. This, most likely, would produce an unsatisfiable system of equations.
- Third, depending on the exact scenario, a worse runtime in comparison to divide and conquer attacks must be anticipated.

Soft analytical side-channel attacks

Soft analytical side-channel attacks (SASCA) were introduced by Veyrat-Charvillon, Gérard, and Standaert in [25]. SASCA tries to address the last two problems of algebraic attacks, namely runtime and error tolerance.

Concretely, SASCA encodes the "soft" probabilistic side-channel information into an algebraic algorithm. In contrast, in classical algebraic attacks, the probability of constraints is usually discarded, and only the most likely option is chosen. Hence, the underlying algebraic algorithm has access to another source of information, namely the probability of the additional constraints. This information is encoded together with the hard algebraic constraints into a graph-like model. Then, the Belief-Propagation algorithm is executed on this instance to draw conclusions about the key. In practice, this approach can reduce

the number of required traces compared to classical template attacks by a factor of 2^3 – 2^4.

2.4 Countermeasures

All presented side-channel attacks exploited the fact that additional side-channel information was present. Furthermore, the emitted side-channel data needed to depend on cryptographic secrets. Hence, countermeasures have two starting points to remediate such attacks:

First, it is possible to reduce emitted side-channel information. Approaches from this category, for example, are

- Shielding: By using appropriate physical enclosures and filters, emissions to the outside can be blocked. This can be especially effective for electromagnetic, thermal, or acoustic emissions.
- Internalizing: By incorporating components that are vulnerable to side-channel attacks directly inside the targeted chip, SCA can become significantly more difficult. For example, an integrated power supply can eliminate power analysis, while an integrated oscillator could prevent over-/underclocking attacks.
- Jamming: It is possible to actively add noise to potential side-channels. For example, random delays in computations can mitigate timing side-channels as well as power side-channels that rely on aligned traces.

Second, side-channel attacks can be prevented by *uncorrelating* the emitted data from cryptographic secrets. Here, multiple approaches exist:

- Constant-time cryptography: The number of instructions, or even the exact sequence of instructions executed does not depend on secrets, but is always the same. This effectively prevents timing side-channels, but either can be rather hard to implement or can significantly slow down the implementation.
- Blinding: In some cryptographic algorithms, user input can be blinded before en-/decryption. Here, blinding means applying an attacker-unknown permutation to the input. With blinding, it is harder for an attacker to know which values were actually passed to the cryptographic primitive. This technique is mainly used for asymmetric cryptography like RSA and elliptic curve cryptography, as algebraic properties of these ciphers directly provide opportunities for blinding.
- Masking is described in the next chapter.

2.4.1 Masking

The *masking* technique was first described by Goubin and Patarin in [26]. It tries to eliminate side-channel dependencies on cryptographic secrets by randomizing all processed values. This is achieved by splitting values into two or more *shares*. Only when all shares are combined, the original value can be reconstructed. If more than two shares are used, this technique is also called *higher-order masking*.

Now, during the execution of a cryptographic algorithm, operations are always performed on all shares separately. Hence, the original intermediate values never occur in memory. Now, if multiple traces are collected, the random split into shares should be different for every trace. Hence, no direct correlation between the secret, the input, and the measured data should be present.

Boolean Masking

One method of splitting a value into shares is *Boolean masking*. Here, a value V is split into two shares A_V and M_V such that

$$V = A_V \oplus M_V$$

Now, each operation involving any intermediate values V and V' should only operate on A_V and M_V separately. In other words, the value $V = A_V \oplus M_V$ should never be computed except at the very end of our algorithm. Clearly, some operations can easily be split to operate on both shares. For example, the XOR operation $V = X \oplus Y$ can be written as follows:

$$\underbrace{X \oplus Y}_{=V} = \underbrace{A_X \oplus A_Y}_{=A_V} \oplus \underbrace{M_X \oplus M_Y}_{=M_V}$$

Hence, A_V and M_V can be calculated without computing X or Y. Similar equations for other bitwise operations can be found in section 4.1.3.

Besides bitwise operations, another important operator for many cryptographic algorithms are array lookups. For example, the S-Box lookup of AES is usually implemented with a static S-Box array. The paper [27] suggests that for a table lookup $Y = T[X]$, with $X = A_X \oplus M_X$ a table T' with the following specification is pre-computed:

$$T'[X] = T[X \oplus M_X] \oplus M'_X$$

Now, a masked table lookup can be performed on A_X by

$$T'[A_X] = T[\underbrace{A_X \oplus M_X}_{X}] \oplus M'_X$$

Thus, the resulting value itself is then masked with M'_X, which can be either a fresh ran-

dom value, or $M'_X = M_X$ if no re-randomisation is desired. However, note that calculating T requires $\mathcal{O}(|T|)$ steps. Hence, it can be desirable to fix M_X once at the beginning of the algorithm, and to not re-randomise afterwards. Then, T' can be precomputed once and stored in memory.

Arithmetic Masking

Boolean masking provides a convenient way for masking bitwise operations and array lookups. However, many symmetric encryption algorithms, like Simon and SPECK [10] or Twofish [28], also use arithmetic operations like additions over $\mathbb{Z}/2^k\mathbb{Z}$. Those operations cannot be directly computed on Boolean shares. However, *arithmetic masking* is another masking representation that can solve this problem. Here, a k-bit value V is stored as

$$V = (A_V + M_V) \bmod 2^k$$

In this representation, arithmetic operations can be split up to operate only on the shares. For example, the addition of two values becomes

$$\underbrace{X + Y}_{=V} = \underbrace{A_X + A_Y}_{=A_V} + \underbrace{M_X + M_Y}_{=M_V}$$

where both A_V and M_V are calculated without having X or Y as an intermediate value. In section 4.1.3, this construction on arithmetic shares is extended to more arithmetic operators.

Goubin's algorithm

Clearly, for ciphers that mix bitwise and arithmetic operators, a method for converting between both masking types has to be found. The first steps in this direction have been taken by Messerges in [27]. However, his conversion algorithm was again vulnerable to differential power attacks. Subsequently, Goubin proposed in [29] an improved algorithm to convert from arithmetic to Boolean masking and vice-versa. In his algorithm, no intermediate values correlate with the data that is masked. Hence, this algorithm is not vulnerable to first-order power analysis.

Surprisingly, converting between the two types of masking is fairly efficient. The conversion from arithmetic to Boolean masking on k-bit numbers needs only $5k + 5$ elementary operations. The other direction is possible in constant time with 7 elementary operations. In 2015, an algorithm with logarithmic runtime in k for conversion between arithmetic and Boolean masking has been published in [30]. However, since k is already very small in most real-world appliances, this new algorithm only provides a minor wall-clock time improvement.

Higher-Order Masking

If masking is correctly used, the leakage at any single point in time does not correlate with the secret key. Thus, attacking only a single time point, as in classical DPA, CPA, or template attacks, cannot provide information about secret intermediate values. However, it still is possible to collect side-channel values at *multiple different* time points $t_0, \ldots, t_n$. Then, the system may be broken by analyzing the correlation between a predicted intermediate value I (without masking) and a combination of the leakage values $L(t_0), \ldots, L(t_n)$. For example, Messerges shows in [31] that it can be successful to measure leakage at two time points and then maximizing the correlation between I and $|L(t_0) - L(t_1)|$.

This motivates *higher-order masking*. In n-th order masking, all values are split into n shares (in comparison to 2 shares). Hence, our masking equations become

$$V = M_V^1 \oplus M_V^2 \oplus \cdots \oplus M_V^n \qquad \text{for Boolean masking}$$
$$V = M_V^1 + M_V^2 + \cdots + M_V^n \qquad \text{for arithmetic masking}$$

Higher-order masking is currently less used in practice. However, [32] shows how to construct an efficient way to fully mask the AES algorithm for any order. Theoretically, any n-th order masking can be broken by a $n+1$th-order attack. In practice, the authors of [32] have found out that breaking nth-order masking requires exponentially many collected traces in n. Thus, higher-order attacks are less likely in practice, especially if trace collection is limited or throttled.

Chapter 3

Background on Julia

Julia [8] is a modern, high-level programming language first published in 2012. Currently, Julia's focus lies mostly on scientific computing, but combines it with a high-performance low-level approach. Hence, Julia code can be nearly as performant as in statically typed low-level languages like C. Thus, Julia is a competitive choice for efficient applications that need high-level language features.

3.1 Types

Julia is a *dynamically* typed language. Hence, types of objects may not be known at the time of writing or compiling the code. In comparison to *statically* typed languages, this allows for greater flexibility. Since code can be reused with different types, this enables programmers to write generic code with less redundancy. However, dynamically typed languages usually exhibit less performance than comparable static languages. Julia tries to compensate for this drawback by distinguishing between abstract functions and concrete methods. Every time a function is called, the types of all arguments must be known. For those known types, Julia then dispatches either an already compiled, concrete method, or compiles and calls an optimised method for those types just in time. Julia determines suitable methods based on the *multiple dispatch* paradigm, which is explained in detail in section 3.2.

Internally, Julia's type system allows for *subtyping*. For this feature, Julia distinguishes *abstract* and *concrete* datatypes. An abstract type is only a type declaration without any fields and can never be instantiated. Thus, an abstract type's main purpose is grouping its subtypes together, and providing default implementations for all concrete subtypes underneath it. A special abstract datatype is the `Any` type, which is a supertype of all other types. In contrast, concrete types are always instantiable, final (i.e. they cannot have subtypes), and are either composite or primitive. Primitive types represent values that consist of "plain bits", while composite types contain a collection of named fields. In

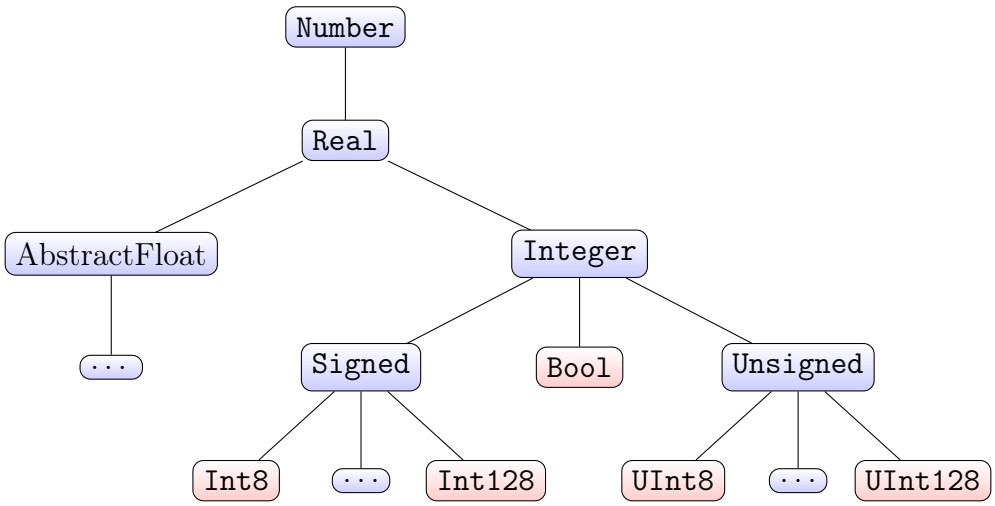

Figure 3.1: Hierarchy of Julia's numerical types. Abstract types are blue, concrete types are red.

this thesis, we will focus on declaring custom composite types. An example of the subtypes provided by Julia's base is the numerical type hierarchy, which is shown in figure 3.1.

3.1.1 Numerical types

Numbers, and particularly integers, form an important part of cryptography. Hence, understanding Julia's handling of numbers in detail is an important prerequisite for our project. Internally, Julia's numerical types form the hierarchy in figure 3.1. Note that per default, there is an alias `Int = Int64` or `Int = Int32` depending on the processor's architecture. Hence, by default, Julia will operate on those values.

However, the type definitions only specify a very limited structure of the types. In Julia, types are better characterized in the context of functions that operate on them. For numerical types, the most significant functions are arithmetic and bitwise operations, comparison operators, array indexing, and randomness. Hence, for types to "behave like integers", those functions must behave similarly as on Julia's built-in integer types.

3.1.2 Bits types

Bits types capture "plain data" types. Specifically, bits types must be immutable, have a fixed size, and contain only references to other bits types. For example, all primitive types and, thus, all concrete numerical types are bits types. Furthermore, numerical composite types like `Complex` numbers are bits types as well.

Note that *closures* that only capture global variables are bits types, even if the global variable itself is not a bits type. This can be, for example, used for referencing mutable objects like arrays:

```
julia> a = Vector{Int}(1)
```

```
julia> typeof(a)
Vector{Int64}
julia> isbitstype(typeof(a))
false
julia> closure = () -> a
julia> isbitstype(typeof(closure))
true
```

3.1.3 Parametric types

Another class of Julia's types are *parametric types*. Those types can be parametrized over any other type, or over any value of a bits type. Hence, declaring a single parametric type adds a whole family of new types. For example, `Array{T, N}` is a parametric datatype. It takes two type parameters, the underlying type of the array `T` and the dimensionality `N`. Note that neither `T` nor `N` is constrained to any types in the definition. However, it makes sense for `T` to be a type, and for `N` to be an integer, and thus a value of a bits-type. Hence, a 2-dimensional array of `Int64` is declared with `Array{Int64, 2}`. Syntactically, free type parameters in method signatures are written after a `where` clause. For example, the following function takes a three-dimensional array with any underlying type `T` as an argument:

```
function f(x::Array{T, 3}) where T
```

3.2 Multiple dispatch

Julia distinguishes between *functions* and *methods*. A function is an abstract object mapping a tuple of arguments to a return value. However, this abstract function may behave differently for different arguments.

For example, the `show` function prints objects to the stream `io`. Based on the passed object, different actions are dispatched:

```
show(io::IO, ::Nothing) = print(io, "nothing")
show(io::IO, b::Bool) = print(io, b ? "true" : "false")
show(io::IO, n::Unsigned) = print(io, string(n))
```

A concrete implementation of a function for specific types is called a *method*. As shown above, Julia motivates *polymorphism*, i.e. the use of multiple different methods for a single function. Internally, for the execution of a program, each call to a function must be mapped to an actual method. From all different methods for a called function, the

one with the best suitable type signature is dispatched in an actual call. This method selection depends on the types of *all* arguments, which stands in contrast to classical object-oriented languages like `C++` or `Java`, where method selection is only based on a single object. This distinctive paradigm of the Julia language is called *multiple dispatch*.

In this paradigm, dispatching methods can only happen if the types of all arguments are known. However, due to the dynamic nature of Julia, the types of the arguments may not be known before running a program. In such cases, the actual dispatch can only be performed during runtime. Thus, with static analysis during compilation, it is not clear which methods must be compiled into a binary. Julia solves this problem with just-in-time (JIT) compilation combined with pre-compiled methods. Thus, for a function call, Julia first checks if the suitable method is already compiled. Otherwise, the method is compiled during runtime.

There may be multiple suitable methods for a function call. In this case, the *most specific* method is invoked. For example, consider the following two methods for function `f`:

```
f(x::Integer, y::Integer) = x + y + 1
f(x::Any, y::Any) = 42
```

Then, the call `f(1, 2)` would evaluate to 4. Even though both method declarations would be suitable (recall that `Integer` is a subtype of `Any`), the first method will be dispatched, since it is more specific for two integer arguments. In contrast, the call `f(1, "Hello")` evaluates to 42, since only the latter definition matches the type signature.

However, there may not always be a most specific method. For example, consider the following declaration for a function `f`:

```
f(x::Integer, y::Any) = 1
f(x::Any, y::Integer) = 2
```

Now, a call that passes two integer arguments (for example, `f(1, 2)`) cannot be dispatched, since there is a *method ambiguity*. In this case, Julia terminates with an error. A possible fix for such situations is to define an additional method with the problematic signature:

```
f(x::Integer, y::Integer) = 3
```

This problem will play a central role in section 4.1.1.

3.2.1 Value types

In some cases, it may be desired to dispatch different methods based on the *value* of a function argument rather than on its type. However, this is not natively possible in Julia. Instead, note that dispatching also works based on arguments of parametric types. This motivates the use of *value types*, which are essentially wrapper types around any bits value.

Value types are defined in the following way:

```
struct Val{x} end
Val(x) = Val{x}()
```

Those types allow, for example, dispatching based on a Boolean value. More importantly, for every different value, a different version of the function is compiled. This can yield performance benefits in some cases where only a few different values can ever be passed on to a function.

For example, consider a cipher with a `rounds` parameter, for which only a small set of values are permitted (e.g. AES only allows $10 \leq$ `rounds` ≤ 14). If the `rounds` parameter is passed as a value type, for all different number of round an extra function is compiled. During those compilations, loop unrolling or other optimizations may be applied. Conversely, if `rounds` is not a value type, only one single function without optimizations based on the actual value can be compiled. However, note that using value types can lead to extremely inefficient code, for example, if a function has to be repeatedly recompiled in a loop. Hence, using value types should be done with great caution.

3.3 Further reading

More resources about Julia can be found in the Julia documentation. In particular, the chapters on types and methods contain additional details and syntax for the topics discussed in the last sections.

Chapter 4

Implementation

In this chapter, the central implemented concepts for our framework will be described. First, section 4.1 describes the *custom types* used in our framework. Next, section 4.2 discusses our implementation of the AES and SPECK cipher. Afterwards, in section 4.3, we analyze the implemented side-channel attacks. Lastly, section 4.4 contains some technical details about the implementation of this project.

In this report, the focus lies on the high-level description of concepts and their realization. For low-level code descriptions with additional examples refer to the framework documentation at `https://parablack.github.io/CryptoSideChannel.jl/dev/`.

4.1 Custom types

This section first describes in 4.1.1 how a custom type that *behaves like an integer* can be added to the Julia language, focusing on useful properties for cryptography. Afterwards, section 4.1.2 constructs a logging datatype. Lastly, section 4.1.3 considers custom types that implement masking.

4.1.1 Integer-like types

An important goal of this library is to provide new types that *behave similarly* to integers but include custom functionality. Those types are constructed using a duck typing approach:

> "If it walks like a duck and it quacks like a duck, then it must be a duck."

Thus, our newly created types will not be a subtype of `Integer`, but instead only *behave* like integers in certain contexts. A detailed discussion on this choice can be found in the paragraph "Subtypes of `Integer`".

Technically, a definition of a very simple custom integer type could look as follows:

```
struct MyInt
    value::Int
end
```

Now, using Julia's powerful multiple dispatch functionality, we can start defining methods for our custom integer. For example, the addition method could be defined in the following way:

```
function Base.:(+)(a::MyInt, b::MyInt)
    # custom code (if desired) here
    MyInt(a.value + b.value)
end
```

Since we want our type to be compatible with normal integers, it is also necessary to define the following two methods for addition:

```
function Base.:(+)(a::MyInt, b::Integer)
    # custom code (if desired) here
    MyInt(a.value + b)
end
function Base.:(+)(a::Integer, b::MyInt)
    # custom code (if desired) here
    MyInt(a + b.value)
end
```

However, note that the above construction results in code duplicates. These can be avoided by the use of metaprogramming to generate those functions automatically. Consider the following, equivalent method definitions for addition instead:

```
extractValue(a::MyInt) = a.value
extractValue(a::Integer) = a

function Base.:(+)(a::MyInt, b::MyInt)
    MyInt(extractValue(a) + extractValue(b))
end
function Base.:(+)(a::MyInt, b::Integer)
    MyInt(extractValue(a) + extractValue(b))
end
function Base.:(+)(a::Integer, b::MyInt)
    MyInt(extractValue(a) + extractValue(b))
end
```

Now, all three procedures have the exact same body. Hence, we can subsume all three procedures with a loop in metaprogramming:

```
for type = ((:MyInt, :MyInt), (:MyInt, :Integer), (:Integer, :MyInt))
    eval(quote
        function Base.:(+)(a::$(type[1]), b::$(type[2]))
            # custom code here
            MyInt(extractValue(a) + extractValue(b))
        end
    end)
end
```

Here, the outer `for`-loop is evaluated at compile time. Thus, three new methods are registered by evaluating the generated function body. Of course, we like to extend this construction to other operators. We can achieve this by another metaprogramming loop as well. This loop iterates over all (binary) operators that we want to define:

```
for op = (:+, :*, :-, :div, :mod)
    for type = ((:MyInt, :MyInt), (:MyInt, :Integer), (:Integer, :MyInt))
        eval(quote
            function Base.$op(a::$(type[1]), b::$(type[2]))
                MyInt(Base.$op(extractValue(a), extractValue(b)))
            end
        end)
    end
end
```

Similarly, we can implement all essential integer functionality described in the Julia documentation. In our case, this includes the methods relevant to cryptographic operations, including

- Basic Arithmetic: `+`, `-`, `*`, `/`, `^`, `div`, `rem`, `fld`, `cld`, `gcd`, `lcm`
- Bitwise Operators: `~`, `<<`, `>>`, `>>>`, `xor`, `bitrotate`
- Comparison: `==`, `!=`, `<`, `<=`, `>`, `>=`, `isequal`
- Mathematical predicates: `isfinite`, `isnan`, `isinf`, `iseven`, `isodd`
- Type construction: `one`, `zero`, `typemin`, `typemax`
- Array accesses: `getindex`, `setindex`
- Randomness: `rand`

Furthermore, we can define an automatic conversion rule from built-in integer types with `convert(::Type{MyInt}, x) = MyInt(x)`. This makes even more methods from the standard library compatible with our custom type. For example, the colon operator (constructing a range with `x:y`) is automatically defined using conversion, comparison, and addition.

Subtypes of `Integer`

An interesting question to ask is whether the new type `MyInt` should be a subtype of `Integer`. At first glance, this would seem like the right choice since it would model their natural relationship. Moreover, code that restricts argument or return types to `Integer` would automatically be compatible with our framework. However, establishing this subtype relationship poses the following two issues:

Firstly, multiple dispatch only works when no ambiguities in method dispatch are present. However, if multiple arguments are passed to a function, subtyping in more than one argument may introduce ambiguities. Consider the following piece of code which runs without an error:

```
struct MyInt
    value::Int
end
Base.getindex(v::AbstractArray, i::MyInt) = v[i.value]

v = [1, 2, 3]; i = MyInt(2)
v[i]
```

Now, consider the next block of code, where the only difference to above is the subtyping declaration `MyInt <: Integer`:

```
struct MyInt <: Integer
    value::Int
end
Base.getindex(v::AbstractArray, i::MyInt) = v[i.value]

v = [1, 2, 3]; i = MyInt(2)
v[i]
```

Note that the second block produces an error on execution:

```
ERROR: MethodError: getindex(::Vector{Int64}, ::MyInt)
    is ambiguous. Candidates:
getindex(v::AbstractArray, i::MyInt) in Main at [...]
```

```
getindex(A::Array, i1::Integer, I::Integer...) in Base at [...]

Possible fix, define
getindex(::Array, ::MyInt)
```

To fix this issue, a more concrete method signature has to be defined. In the example above, we must define another method `getindex(::Array, ::MyInt)`. However, defining this single method is not sufficient. In order to be compatible with a concrete subtype `T <: AbstractArray`, a corresponding method `getindex(::T, ::MyInt)` must be defined. But this process cannot be completed at compile-time, since new subtypes may be added dynamically. For example, the `StaticArrays` defines a type `StaticArray{...} <: AbstractArray{...}`. However, this package may be added after the pre-compilation of our package. Hence, an additional method not known at compile-time is required.

Another argument against subtyping the `Integer` is the variety of integer types. Recall that a benefit of subtyping is compatibility to code that restricts arguments to `Integer` types. In such code, type annotations do not need to be changed to work with this framework. However, many projects restrict argument types to, for example, `Unsigned` or `Int64`. Such type annotations then have to be manually exchanged or removed again.

Following the two arguments outlined above, we will not use subtypes of `Integer` throughout this project. Instead, they will be "duck types" of `Integer` without any specific supertype.

4.1.2 Logging

The `Logging` module allows for recording traces of program executions. At its core, this module provides the type `GenericLog` which behaves like an integer type. Additionally, this type appends a *reduced* value to an array every time an operation is performed on it. The type is defined in the following way:

```
struct GenericLog{U,S,T}
    val::T
end
```

Notably, this type contains three type arguments:

- `T` is the underlying type that should be mimicked. `T` may be a primitive integer type like `UInt8` or `Int`, or any other type that behaves like an integer, for example, another `GenericLog` or `Masked` (see section 4.1.3) type.
- `U` should be a container holding a *reduction function*. The purpose of this reduction function is to preprocess values of type `T` for logging. Most commonly, not the whole

value but only a footprint should be logged. For example, such a footprint could be the Hamming weight or the least significant bit.

- `S` is a *closure* returning the array where values should be logged to. Recall that this array cannot be passed directly as a type argument, since it is no bits-type. Hence, the logging array is captured by a closure.

Usually, to construct this type, the following method is used:

```
GenericLog(val, logging_array, reduction_function)
```

Note that `T` is inferred from `typeof(val)`, `U` is constructed from `reduction_function`, and `S` is obtained from `logging_array`.

Apart from the type construction method, this framework defines all methods required for an integer type, as described in 4.1.1. Basically, all methods include the following (simplified) code that logs a value reduced with `U` to the array captured by `S`:

```
function Base.$op(a::GenericLog{U,S,T}, b) where {U,S}
    res = Base.$op(extractValue(a), extractValue(b))
    push!(S(), U(res))
    GenericLog{U,S,typeof(res)}(res)
end
```

4.1.3 Masking

Recall from chapter 2.4.1 that masking can be performed in two different ways: A number can be split into either *arithmetic* or *Boolean* shares. In Julia, we will model the masking method with a type parameter. Remember from section 3.1 that for each different type signature, a different method will be compiled. Thus, different methods will be compiled for Boolean and arithmetic masking. Hence, optimisations depending on the masking method are possible. While this can result in a performance gain, it also introduces a problem discussed in the paragraph "Problems with high-level masking".

With this type parameter, the definition of our masking types looks as follows:

```
@enum MaskType Boolean=1 Arithmetic=2
struct Masked{M, T1, T2}
    val::T1
    mask::T2
end
```

Here, our masking type stores a value in two shares called `val` and `mask`. The three type parameters denote the following:

- `M` is the masking method that is used. It always is of type `MaskType` and, thus, can either be `Boolean` or `Arithmetic`.
- `T1` is the type of `val`, and, thus, the type of the first share. This type should behave like an integer.
- `T2` is the `mask`'s type. It should behave like an integer as well, but additionally should not be another `Masked`-type.

With this construction, it is possible to simulate higher-order masking if `T1` is chosen as another `Masked`-datatype. For example, a value is split into three arithmetic shares if it is of the type `Masked{Arithmetic, Masked{Arithmetic, T, T}, T}`.

Boolean Masking

If Boolean masking is used, recall that a value V is split into

$$V = A_V \oplus M_V$$
$$\texttt{unmask(v)} = \texttt{xor(v.val, v.mask)}$$

With this definition, it is possible to define operators on masked values. While computing results for those, plain values of all operands should never be observable in memory. A comprehensive list of operators on Boolean masked values can be found in the following table, where $X = A_X \oplus M_X$ and $Y = A_Y \oplus M_Y$ are two masked values, and C is a constant:

Operation	Resulting A_Z	Resulting M_Z
$Z = \sim X$	A_X	$\sim M_X$
$Z = X \oplus Y$	$A_X \oplus A_Y$	$M_X \oplus M_Y$
$Z = X \oplus C$	A_X	$M_X \oplus C$
$Z = X \ll C$	$A_X \ll C$	$M_X \ll C$
$Z = X \gg C$	$A_X \gg C$	$M_X \gg C$
$Z = X \,\&\, C$	$A_X \,\&\, C$	$M_X \,\&\, C$
$Z = X \mid C$	$A_X \mid C$	$(M_X \mid C) \oplus C$
$Z = X \,\&\, Y$	$(A_X \,\&\, A_Y) \oplus (A_X \,\&\, M_Y) \oplus (M_X \,\&\, A_Y)$	$M_X \,\&\, M_Y$
$Z = X \mid Y$	$A_{(\sim X)\&(\sim Y)}$	$\sim M_{(\sim X)\&(\sim Y)}$

In addition to the operators mentioned in the table, array lookups are implemented on Boolean masked shares as well. They are realized using a pre-computed masked array as described in section 2.4.1. The actual Julia implementation of the operators can be directly inferred from the specification in the table. For example, bitwise negation is realized in the following way:

```
function Base.:(~)(a::Masked{Boolean})::Masked{Boolean}
    Masked{Boolean,typeof(a.val),typeof(a.mask)}(a.val, ~a.mask)
end
```

However, note that this method can only be dispatched on Boolean masked types. To also dispatch on arithmetic types, it is necessary to define the following *wrapper method*:

```
Base.:(~)(a::Masked{Arithmetic}) = ~arithmeticToBoolean(a)
```

In a similar way, wrapper methods must be defined for all other functions that operate on Boolean masked values.

Arithmetic Masking

In arithmetic masking, a value V is split into

$$V = A_V + M_V$$
$$\texttt{unmask(v)} = \texttt{v.val} + \texttt{v.mask}$$

For arithmetic masking, the following operations are defined in our framework, where $X = A_X + M_X$ and $Y = A_Y + M_Y$ are masked values, and C is a constant.

Operation	Resulting A_Z	Resulting M_Z
$Z = -X$	$-A_X$	$-M_X$
$Z = X + C$	A_X	$M_X + C$
$Z = X + Y$	$A_X + A_Y$	$M_X + M_Y$
$Z = X - C$	A_X	$M_X - C$
$Z = X - Y$	$A_X - A_Y$	$M_X - M_Y$
$Z = X \cdot C$	$A_X \cdot C$	$M_X \cdot C$
$Z = X \cdot Y$	$A_X \cdot A_Y + A_X \cdot M_Y + M_X \cdot A_Y$	$M_X \cdot M_Y$

The implementation in Julia is similar to the implementation of Boolean masked methods. Note that wrapper methods that dispatch on `Masked{Boolean}` types are required for arithmetic operations. Those methods, again, simply perform conversion to arithmetic masking before calling the corresponding operator.

Conversion

For ciphers that mix arithmetic and bitwise operations, it is necessary to define a conversion between both types. For this conversion, Goubin's algorithm is used (cf. section 2.4.1). Both directions of Goubin's algorithm can be accessed with the following functions:

```
function booleanToArithmetic(a::Masked{Boolean})::Masked{Arithmetic}
function arithmeticToBoolean(a::Masked{Arithmetic})::Masked{Boolean}
```

However, only defining those two functions is not sufficient, since they may introduce the need for manual conversion. Consider the following example:

```
function plusone(x::T)::T where T
    x + 1
end
plusone(BooleanMask(10))
```

At method dispatch, it holds that `T = Masked{Boolean, ...}`. However, the return value is automatically converted to the arithmetic type `Masked{Arithmetic, ...}`, since `+` is an arithmetic method. Thus, this example code errors, as no value of type `T` is returned. Such cases can be resolved by Julia's automatic *type conversion*. Internally, a type `T` can be converted to a type `U` if a method `convert(::Type{U}, x::T)` is defined. Hence, our desired type conversion can be achieved with the following two methods:

```
convert(::Type{Masked{Boolean, T1, T2}}, a::Masked{Arithmetic}) where {T1, T2}
    = arithmeticToBoolean(a)
convert(::Type{Masked{Arithmetic, T1, T2}}, a::Masked{Boolean}) where {T1, T2}
    = booleanToArithmetic(a)
```

Note that converting types only happens in some situations, for example when values are returned or assigned to an array. Then, the respective value is converted to the desired type, or an error is thrown if impossible. An exhaustive list of situations where `convert` is automatically called can be found in the Julia documentation. However, for method dispatching, *no automatic conversion* happens. Thus, the wrapper methods defined above cannot be omitted.

Problems with high-level masking

We defined the `Masked` datatype as a construct in the high-level Julia language. This poses some problems with respect to compiler optimisations. Essentially, all masking security guarantees rely on the fact that some intermediate values will never appear in memory. However, the masking approach introduces new, "unnecessary" computation instructions. Depending on compiler optimisations, some of those instructions may not appear in the final executable.

For example, consider the masked array lookup. Recall that for a table lookup $Y = T[X]$, with $X = A_X \oplus M_X$, a table T' with $T'[K] = T[K \oplus M_X] \oplus M'_X$ is computed. Later

on, the table is only accessed at index A_X. Hence, a compiler may notice that all other fields of the table are never accessed, and may optimize the code in the final program to only compute $T'[A_X] = T[A_X \oplus M_X] \oplus M'_X$. However, note that for computing the latter, $A_X \oplus M_X$ appears as an intermediate result. Thus, this can allow an attacker to draw conclusions about the unmasked value of X.

This issue implies that this project shall only be used for academic, testing, and educational purposes. The `Masked` datatype should never be used as a protection measure in a real-world system. Mitigating this issue by exploring ways to preserve masking through compiler optimisations in Julia could be realized in future work.

4.1.4 Obtaining side-channel traces

One of the main features of this project is the ability to obtain traces of an arbitrary program that works with integers. For example, this feature can be used to obtain traces from custom cryptographic algorithms which then can be analyzed for vulnerabilities. Furthermore, trace generation can also be used for generating data for student exercises. A possible task is to release a set of a few thousand traces with the goal to reconstruct the secret key used.

Unmasked traces

To generate unmasked traces, the following must be provided:

- A trace collection array. All collected values will be appended to this array. Recall that the logging array must be a global variable.
- The reduction function. If a value `x` is processed, the value `reduction_function(x)` is appended to the trace collection array. In the following example, we choose `reduction_function = x -> Base.count_ones(x) + rand(d)`, which computes the Hamming weight with uniform noise.

Equipped with those two values, the trace collection code is:

```
trace = Float64[]

function encrypt_log_trace(plaintext)
    global trace
    trace = Float64[]
    clos = () -> trace

    d = Distributions.Normal(0, 3)
    reduction_function = x -> Base.count_ones(x) + rand(d)
```

```
    key_log = GenericLog.(SECRET_KEY, clos, reduction_function)
    pt_log  = GenericLog.(plaintext, clos, reduction_function)

    # cryptographic code, for exampl
    # AES_encrypt(pt_log, key_log)
    # SPECK_encrypt(pt_log, key_log)

    return copy(trace)
end
```

Masked traces

Collecting masked traces is a very similar process. The only difference is that a `Masked` datatype encapsulates the key and plaintext logging types:

```
key_log = GenericLog.(SECRET_KEY, clos, reduction_function)
pt_log  = GenericLog.(plaintext, clos, reduction_function)

key_masked = BooleanMask.(key_log)
pt_masked = BooleanMask.(pt_log)
```

4.2 Ciphers

In this project, both the AES and the SPECK cipher are implemented. The implementation of both ciphers is presented in the following two sections. Additional code examples can be found in the "Ciphers" chapter of the documentation.

4.2.1 AES

The *Advanced Encryption Algorithm (AES)* [9] has been standardized in 2000. Currently, AES is widely used throughout different applications. AES is also a popular algorithm for hardware applications like smart cards. For example, the very popular "MIFARE DESFire" series of access control cards supports AES. Thus, AES plays a significant role in hardware security analysis.

In this project, the AES algorithm is implemented in the file `src/cipher/AES.jl`. It provides the module `AES` with the following functions to interact with:

En-/decryption

```
AES_encrypt(plaintext::MVector{16,T}, key::Vector{T})::MVector{16,T} where T
AES_decrypt(ciphertext::MVector{16,T}, key::Vector{T})::MVector{16,T} where T
```

The methods en-/decrypt a block of 16 bytes using the provided `key`. The `key` vector must be either of length 16, 24, or 32. Depending on its length, the different variants of AES are dispatched:

- Length 16: AES-128
- Length 24: AES-196
- Length 32: AES-256

Both methods are parametrized over a type `T` which must behave similarly (cf. section 4.1.1) to the type `UInt8`. Hence, `T` can be instantiated with logging, masking, or stacked types to obtain traces of the AES en- or decryption process.

For testing purposes, the following two helper methods are provided as well:

```
AES_encrypt_hex(plaintext::String, key::String)::String
AES_decrypt_hex(ciphertext::String, key::String)::String
```

Key schedule

Internally, AES makes use of a *key schedule*. Thus, different *round keys* are derived from the original key. This computation happens in the following two methods:

```
key_expand(k::Vector{T})::Vector{T}
inv_key_expand(k::Vector{T})::Vector{T}
```

While `key_expand` is an integral component of the AES algorithm, note that the function `inv_key_expand` is only needed for attacking purposes. For example, it is a common scenario that the decryption process is monitored. Since the last round of encryption is the first round of the decryption process, usually the last round key of AES is attacked and recovered. Fortunately, the key expansion of AES is reversible. This process is implemented in `inv_key_expand`, which provides the whole key schedule given only the *last* round key of AES. Note that the original key can be trivially obtained from the full key schedule, as it consists of the first one (AES-128) or two (AES-256) round keys.

4.2.2 SPECK

SPECK [10] is a lightweight block cipher proposed by the NSA in 2013. Internally, SPECK is an *add-rotate-xor* cipher. Thus, SPECK mixes arithmetic operations (addition) with bitwise operations (xor, bitwise rotation). In this project, the `SPECK` module is implemented in `src/cipher/SPECK.jl`.

En-/decryption

SPECK, like AES, offers a high-level interface for encryption and decryption:

```
SPECK_encrypt(plaintext::Tuple{T, T}, key::Tuple{T, T}; rounds = 32)::Tuple{T,T}
↪  where T
SPECK_decrypt(ciphertext::Tuple{T, T}, key::Tuple{T, T}; rounds =
↪  32)::Tuple{T,T} where T
```

Here, `T` must be a type that behaves like `UInt64`. Internally, SPECK operates on two blocks of 64 bit, which are passed as a `Tuple{T, T}`. Another parameter of SPECK is the number of rounds to execute. The original paper [10] proposes to use 32 rounds. Both methods return a `Tuple{T, T}` containing the 128-bit en- or decrypted data in two shares of 64 bit.

Key schedule

As the AES, the SPECK cipher uses a key schedule for its rounds. The full key schedule from the original key can be computed with:

```
SPECK_key_expand(key::Tuple{T, T}, rounds)::Vector{T} where T
```

The result is a vector of length `rounds` that contains each round key. Note that, in contrast to AES, the original key cannot be inferred from a single round key, since round keys consist only of 64 bits while the original key has 128 bits. Thus, recovering the original key requires multiple round keys. A key recovery process from two successive round keys is described as part of the CPA attack against SPECK in section 5.1.2.

4.3 Attacks

4.3.1 DPA

Differential power analysis attacks are implemented in the module `DPA` at `src/attacks/DPA.jl`. Currently, DPA attacks are only implemented against AES.

At its core, the DPA module provides the following two functions:

```
function DPA_AES_analyze_traces(plaintexts::Vector, traces::Matrix)
function DPA_AES_analyze(sample_function; N = 2^10)
```

Both functions perform a DPA attack against AES and return the recovered key. However, they differ in the way they are receiving the side-channel data. The first function receives

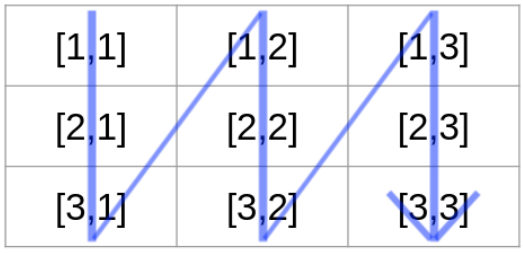

Figure 4.1: Column-major array storing in Julia. The blue arrow shows the order in which entries are stored in memory.

	N		
	trace[:,1]	trace[:,2]	trace[:,3]
M	trace[1,1]	trace[1,2]	trace[1,3]
	trace[2,1]	trace[2,2]	trace[2,3]
	trace[3,1]	trace[2,3]	trace[3,3]

Figure 4.2: The memory layout of the trace arrays.

the following two arguments:

- `plaintexts` is a vector of size N, where N is the number of power traces sampled.
- `traces` is a matrix of size $M \times N$, where M is the number of samples per trace. Power traces are stored in column-major order, i.e. it is expected that `traces[:,i]` refers to the power trace for input `plaintexts[i]` (see figure 4.2). Storing traces in this order can give a performance benefit, since Julia stores two-dimensional arrays in column-major order, as shown in figure 4.1. Usually, single points in a trace are accessed consecutively. With this memory layout, caching of memory blocks then can speed up the process of reading values.

In contrast, the second function receives a `sample_function` which acts as a trace provider. Thus, this function should receive an AES input as a single argument, and output the corresponding trace as a vector. The parameter `N` specifies how many traces should be sampled for the attack. For example, set `sample_function` to `encrypt_log_trace` from section 4.1.4 to directly check if the used cipher is vulnerable to a classical DPA attack.

4.3.2 CPA

Similarly to DPA, the `CPA` module (`src/attacks/CPA.jl`) provides functions for attacking a cipher with correlation power analysis. Correlation power analysis attacks are pre-implemented against AES and SPECK.

Most notably, the following methods for attacks are provided by the CPA module:

```
CPA_AES_analyze(sample_function, leakage_model; N = 2^12)
CPA_AES_analyze_traces(plaintexts::Vector, traces::Matrix, leakage_model)

CPA_SPECK_analyze(sample_function, leakage_model; N = 2^12)
CPA_SPECK_analyze_traces(plaintexts::Vector, traces::Matrix, leakage_model)
```

Note the similarities to the DPA attack functions described above. Additionally, for a CPA attack, it is necessary to provide a *leakage model* (cf. section 2.3.2). Recall that a leakage

model returns a side-channel estimate W_R for a processed value R. Thus, in Julia, our leakage model must be a function taking a single numerical argument R and returning a numerical side-channel estimate W_R. For example, to perform an attack targeting the Hamming weight of an intermediate value, we can set `leakage_model = Base.count_ones`, while for an attack on the least significant bit we choose `leakage_model = x -> x & 1`.

A detailed evaluation of the attacks against both ciphers can be found in section 5.1.1.

4.3.3 Template attacks

Recall that template attacks consist of two phases. Firstly, n side-channel vectors $x_i^m \in \mathbb{R}^k$ are collected for each operation O_m. In our case, O_m is an operation processing the integer value m, for example, a load of the value m from memory. The collection of the vectors x_i^m is realized with an instance of the `GenericLog` type that can be constructed with the `StochasticLog` function:

```
function StochasticLog(val, logging_array, mean_for_value, noise_for_value)
  reduction_function = m -> mean_for_value(m) + rand(noise_for_value(m))
  GenericLog(val, logging_array, reduction_function)
end
```

Besides the usual arguments `val` and `logging_array` (cf. section 4.1.2), this method receives two interesting arguments:

- `mean_for_value` must be a function returning the vector $\bar{x}_m$ for a value m.
- `noise_for_value` is a function returning a representation of the covariance matrix $\mathbf{\Sigma}_m$ for value m. Then, a vector from this distribution can be sampled from $\mathbf{\Sigma}_m$ with zero mean by calling `rand(noise_for_value(m))`.

In total, for each processed value m, a vector is sampled from the distribution defined by $(\bar{x}_m, \mathbf{\Sigma}_m)$ and appended to the logging array. With this method, vectors can be collected for arbitrary operations. However, in practice, most template attacks target single load instructions. In this case, it is necessary to collect a set of vectors for all possible loaded values k. For example, if only a single byte is loaded, then vectors for $0 \leq k \leq 255$ must be collected. Our framework provides multiple pre-implemented methods for collection which are described in detail in the "Template attacks" chapter of the documentation.

In the second phase, the sampled vectors are analyzed to draw conclusions about the operation. This process is implemented in:

```
template_core_attack(profiled_vectors::AbstractMatrix, inputs::AbstractVector,
↪  attack_vectors::AbstractMatrix)
```

The function takes the following arguments:

- `inputs` is a list containing N entries, where N is the number of sampled leakage vectors. At position i shall be the input that was used to generate the i-th vector.
- `profiled_vectors` is an $M \times N$ matrix. The column `profiled_vectors[:,i]` should contain the data that was generated with input i.
- `attack_vectors` is an $M \times K$ matrix, where K is the number of traces collected from the attacked device, stored in column-major order. All traces must be generated with the same secret input $v\star$.

This function returns a list containing the tuple (p_v, v), where p_v is the likelihood of $v = v\star$. The returned list is sorted by decreasing p_v.

4.3.4 Key combination

Recall from section 2.3.4 the notion of *divide and conquer* attacks, where a key is split into smaller parts (i.e. single key bytes) that are recovered. Later, those key bytes are merged to form the whole key again. The simplest key combination strategy is to take the most likely option for every separate key part. However, this yields incorrect final keys even if only a single key byte is wrong. Moreover, all attacks described in the past sections can order values based on decreasing likelihood. This ordering is not used if only the most likely option is taken.

To solve this problem, our framework provides the `LikelyKey` interface which implements a key enumeration technique. When iterating over a `LikelyKey` object, all possible combined keys are iterated in order of descending overall likelihood. Consider the following example, where two key parts are combined. We know that in the first key part, 1 is the most likely option, followed by 2, which itself is more likely than 3. In the second key part, 4 is more likely than 5 and 6. Then, the iteration over the combined key looks as follows:

```
julia> k = LikelyKey([[1, 2, 3], [4, 5, 6]])
julia> for x = k
         print(x); print(" ")
       end
[1, 4] [2, 4] [1, 5] [3, 4] [2, 5] [1, 6] [3, 5] [2, 6] [3, 6]
```

Internally, this is realized by implementing Julia's iteration interface, i.e. providing the following methods:

```
Base.iterate(k::LikelyKey)
Base.iterate(k::LikelyKey, state::Stack{Int})
```

```
Base.length(k::LikelyKey)
Base.eltype(k::LikelyKey)
```

A detailed explanation on the implementation of those methods can be found in the chapter "Key combination" of the documentation.

4.4 Technical details

4.4.1 Installation

This project is bundled as a Julia package. Thus, it can easily be installed using Julia's built-in package manager Pkg.

To install the project, the following code should be executed in the Julia REPL:

```
julia> using Pkg
julia> Pkg.add(PackageSpec(url="https://github.com/Riscure/Jlsca"))
julia>
↪  Pkg.add(PackageSpec(url="https://github.com/parablack/CryptoSideChannel.jl"))
```

Alternatively, the package can be directly installed from the `Pkg` command line interface:

```
julia> ]
(@v1.6) pkg> add https://github.com/Riscure/Jlsca
(@v1.6) pkg> add https://github.com/parablack/CryptoSideChannel.jl
```

Note that one dependency, `Jlsca`, has to be added manually. This, unfortunately, is a restriction in the current version of `Pkg` since the `Jlsca` package is unregistered.

Afterwards, the package can be brought into scope by typing

```
julia> using CryptoSideChannel
```

Loading `CryptoSideChannel` implicitly also loads all submodules described in the sections above:

```
julia> names(CryptoSideChannel)
8-element Vector{Symbol}:
 :AES
 :CPA
 :DPA
 :Logging
```

```
:Masking
:SPECK
:TemplateAttacks
```

4.4.2 Development

Git: Version control of this project was done with Git. The Git repository is hosted on GitHub at `https://github.com/parablack/CryptoSideChannel.jl`.

Unit testing: Unit testing was performed with the built-in `Test` package. Single unit tests are bundled to test sets using the SafeTestsets.jl package. In total, more than 50000 unit tests (most of them, however, are automatically generated) in 11 test sets are present.

Continuous integration: Building and testing the project is automated using continuous integration (CI). For this process, Travis CI is used. The CI pipeline is run every time a commit to master is detected. Currently, the integration process builds the project on Julia version 1.6 (this is the latest stable version) and on the most recent nightly build of Julia. This ensures compatibility with the current and new versions of Julia.

Documentation: The documentation consists of multiple HTML files that are automatically created using Documenter.jl. This process is automated in the continuous integration pipeline. After every push to the Git repository, Travis CI automatically re-builds the documentation and deploys it to a special branch of the repository. This branch is then served with GitHub pages at `https://parablack.github.io/CryptoSideChannel.jl/dev/`.

Chapter 5

Evaluation

This chapter evaluates our framework in two dimensions: Section 5.1 presents the outcome of the different attacks that are implemented. Next, section 5.2 discusses how to integrate already existing Julia implementations of cryptographic algorithms into this framework.

5.1 Evaluation of attacks

In this section, we will look at the results of the attacks that are implemented in our framework. Firstly, we look at a correlation power attack against AES in 5.1.1. This section, also, describes how our CPA implementation can be used with data collected from a real-world device running AES. Next, we present a correlation power attack against the SPECK cipher in section 5.1.2. There, we will also highlight some interesting differences to an attack against AES. Lastly, section 5.1.3 describes the results of our template attacks, also taking real-world data into account. Note that DPA attacks against AES were already presented in section 2.3.1 and, thus, will be omitted in this chapter.

5.1.1 CPA against AES

Recall that for a correlation power attack against AES, a leakage model W is required. Given this model, let W_R denote the power estimate for the calculation of a value R. In the following example, we will use the Hamming weight model, i.e. $W_R = H(R)$. Now, we can target the computation of the first S-Box output $S[X_i \oplus K_i]$ (recall that we targeted the same value in DPA). For our attack, we fix $i \in \{1, \dots, 16\}$. Then, for each different i-th input byte $X_i \in \{0, \dots, 255\}$, we can calculate $W_{S[X_i \oplus K_i]}$ if the i-th key byte K_i is known. Since $K_i \in \{0, \dots, 255\}$ is only a single key byte, this value can be brute-forced easily.

For each trace, $W_{S[X_i \oplus K_i]}$ is a trace-specific power estimation. Now, at the point in the trace where $S[X_i \oplus K_i]$ is calculated, we expect the measured values to correlate with the predicted power. For example, figure 5.1 shows the power estimate for the correct

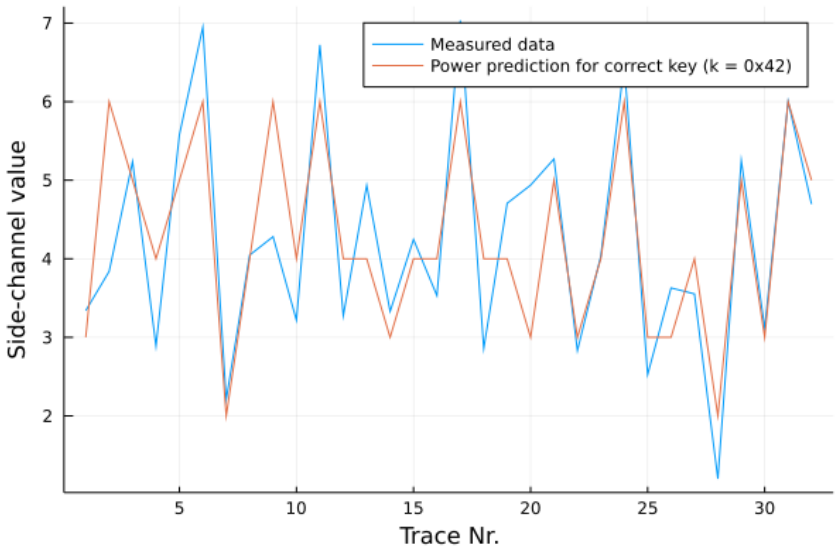

Figure 5.1: Comparison between the power prediction for the correct key and the actual measured power at the point of the first S-Box computation ($t = 298$). Both are strongly correlated ($\rho = 0.87$).

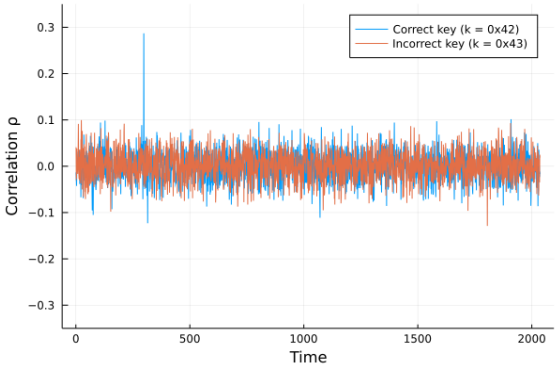

Figure 5.2: Correlation for the correct and for an incorrect key against time. The correct key exhibits a spike in correlation at the computation of the first S-Box output ($t = 298$).

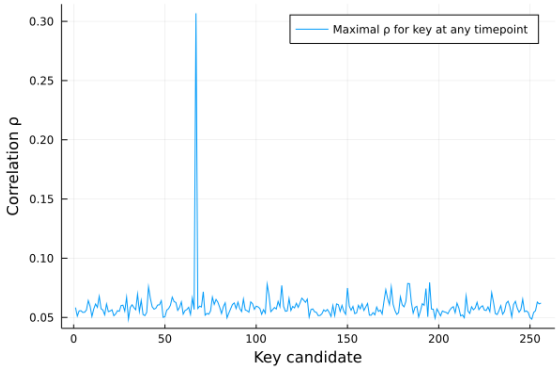

Figure 5.3: Maximal correlation for all key-byte values on real-world AES data. The correct key byte is $0x42$.

key and the measured values at the point where $S[X_i \oplus K_i]$ is calculated. Note that the correlation is very clear in this picture for visualisation purposes. However, such a strong correlation is not necessarily required for this attack to work.

As in DPA, the point where the first S-Box computation occurs is not known beforehand. Hence, the correlation is established for every point of time in our traces. Figure 5.2 shows the correlation against time for the correct key, as well as for an incorrect key. The correlation with the estimate for the correct key spikes at the point where the first S-Box output is calculated. Note that the correlation with the correct key also has minor spikes afterwards, which result from the computation of values directly dependent on the targeted S-Box output.

Similar to DPA, the correct key bytes is expected to have a higher spike in the correlation

at the targeted time point than other bytes at any time point. Thus, to infer the correct key, it is sufficient to consider the maximal absolute correlation at any time point. Figure 5.3 shows the maximal correlation at any time point for all key candidates, with a clear spike at the correct key $k = 0x42$. Finally, key bytes can be iterated in decreasing likelihood with the technique described in 4.3.4 to recover the full AES key.

CPA with real-world AES data

In the above attack, all attack traces have been sampled with our framework. However, our code can also be used to attack real-world traces of AES operations. For this purpose, we use the unmasked AES power trace data provided by [1]. An example trace of this data set was already shown as an introductory example in figure 2.1. Notably, these traces provide power data for an AES decryption process instead of encryption. Thus, the last round key of AES is recovered with our CPA attack. The original key is, afterwards, obtained with `AES.inv_key_expand`.

In total, our framework is able to reconstruct the last round key (and, thus, the original key) with about 2000 traces. However, clearer results appear when more traces are used. Figures 5.4 and 5.5 show statistics of the reconstruction of the first round key byte with 7000 traces.

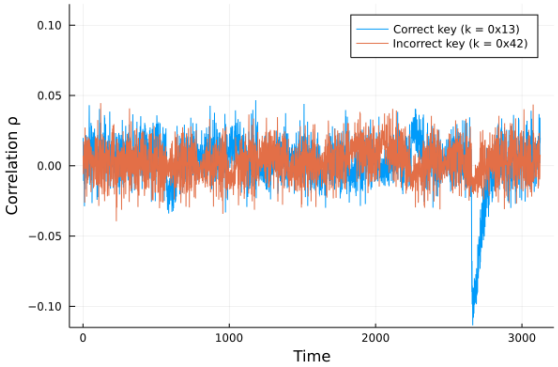

Figure 5.4: Correlation for the correct and for an incorrect key against time on real-world AES data.

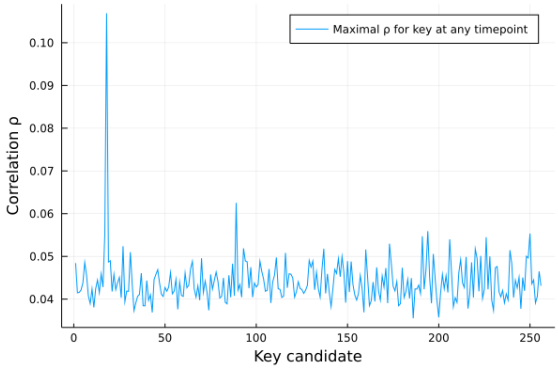

Figure 5.5: Maximal correlation for all key-byte values on real-world AES data. The correct key byte is `0x13`.

The code for parsing the provided data and running the attack can be found in the folder `test/aes_realword/`.

5.1.2 CPA against SPECK

Recall from 4.2.2 that it is necessary to attack multiple 64-bit round keys to obtain the overall 128-bit SPECK key. It is possible to reconstruct the whole key with knowledge of the first two round keys, as described in [33]. Thus, our CPA attack will be targeting the first two round keys of SPECK.

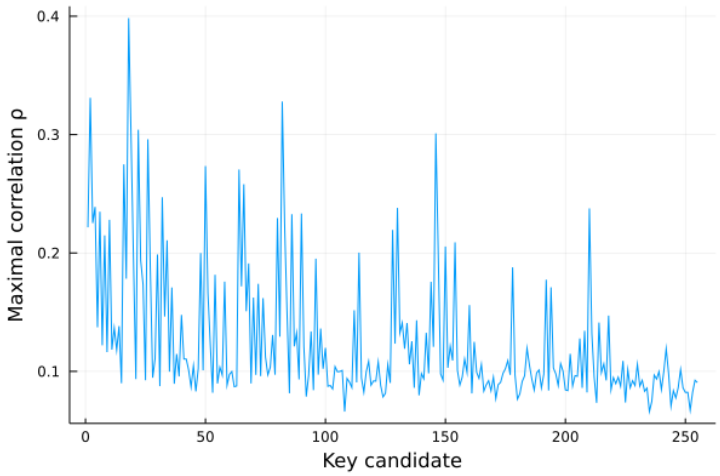

Figure 5.6: Correlation in SPECK for different key bytes (Correct key: `0x12`).

Firstly, we need to obtain the first round key. To this end, we target the intermediate value I_1 with

$$I_1 = ((P_1 \gg 8) + P_2) \oplus K_1$$

where the input consists of the plaintext tuple (P_1, P_2) and the first 64-bit round key is K_1. To recover K_1 efficiently, we split it into eight bytes, called K_1^i for $1 \leq i \leq 8$, which we will attack separately.

Figure 5.6 shows the correlation for different values of K_1^1, where all other K_1^i are set to zero. Note that, in this plot, multiple spikes are observable. The biggest spike corresponds to the actual value of $K_1^1 =$ `0x12`, while the other spikes are caused by the fact that I_1 depends *linearly* on K_1^1. Thus, for example, there are other observable spikes at `0x12 + 0x10`, `0x12 + 0x20`, and `0x12 + 0x40`. This is because, in those values, only a *single bit* differs from the actual key. Since the $\oplus$-operation in our power estimate is linear, the single bit difference propagates up to the final correlation. Hence, candidates that are similar to our actual key will show a strong correlation as well. In contrast, the AES power consumption was calculated after the S-Box lookup. Since this lookup is non-linear, there is no linear relationship between the key and the estimated power consumption. Thus, in AES, only the correct key should exhibit clear spikes in correlation.

After K_1 is obtained, the second round key K_2 must be reconstructed. To this end, we target the intermediate value

$$I_2 = ((I_1 \gg 8) + (I_1 \oplus (P_2 \gg 3)) \oplus K_2$$

Note that, since K_1 is already reconstructed, I_1 is a constant in this equation. With the same technique outlined above, it is possible to reconstruct the 64-bit value K_2 by splitting it into eight bytes.

If K_1 and K_2 are recovered, the overall secret SPECK key (S_1, S_2) can be determined

with the following equations:

$$S_1 = ((K_2 \oplus (K_1 \ll 3)) - K_1) \ll 8$$
$$S_2 = K_2$$

5.1.3 Template attacks

For a template attack, first, templates for all operations must be worked out. This is done by analyzing profile traces for each possible operation O_k. In our implemented attack, we use a pooled covariance matrix. Thus, the templates have the form $(\bar{x}_m, \mathbf{\Sigma})$ where $\bar{x}_m$ is the mean of all observed vectors for operation k. With the pooled covariance matrix, our templates can be interpreted as points in a d-dimensional coordinate system. For example, different templates for a single load instruction of a 3-bit value correspond to the stars in figure 5.7.

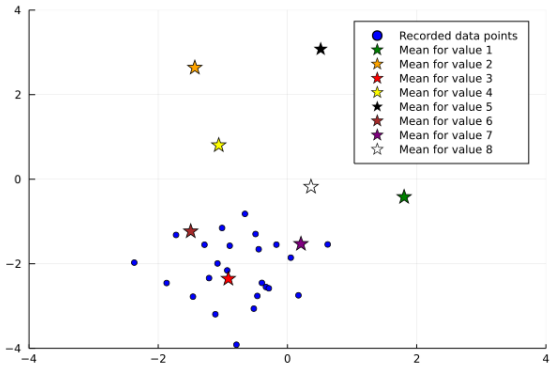

Figure 5.7: Recorded values and templates for a sample 2-dimensional recording. In this example, the covariance matrix is diagonal and has only one eigenvalue. The value loaded was 3.

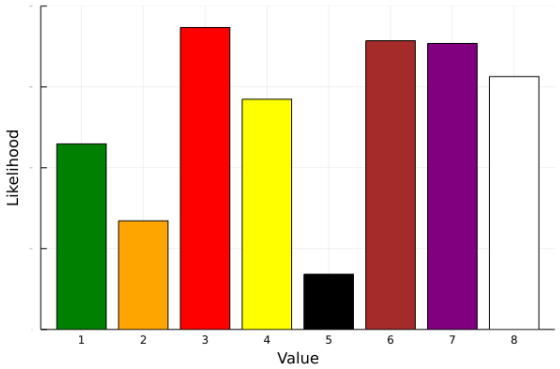

Figure 5.8: Likelihood for each different value. The overall likelihood scales logarithmically with the probability.

Now, after the templates are established, a set of N attack vectors $V_1, \ldots V_N$ is sampled from the attacked device. Those attack vectors correspond to the blue dots in figure 5.7. Then, for each template, all V_i are then analyzed with respect to their likelihood. In section 2.3.3, this was done by computing the probability $p_m(V_i)$ under the multivariate normal distribution defined by the template $(\bar{x}_m, \mathbf{\Sigma})$. Mathematically, the overall probability of observing all vectors $V_1, \ldots, V_N$ under the template $(\bar{x}_m, \mathbf{\Sigma})$ would then be $\prod_{i=1}^{N} p(V_i)$. However, note that the values of $p(V_i)$ are often small (clearly, $0 \leq p(V_i) \leq 1$ since $p(V_i)$ is a probability). If many of such values are multiplied, it is likely that the resulting number cannot be stored due to a floating-point underflow. For example, consider the distribution from figure 5.7. There, even for p defined by the correct template $(\bar{x}_3, \mathbf{\Sigma})$, the average of $p(V_i)$ is about 10^{-1}. For a realistic number of 1000 sampled attack traces, the overall probability is already 10^{-1000}. This number, however, cannot be represented as a `Float64`, as their smallest representable value is $5 \cdot 10^{-324}$:

```
julia> nextfloat(0.0)
5.0e-324
```

Thus, we do not compute probabilities, but instead use a logarithmic likelihood function $l(V_i) = \log p(V_i)$. Hence, the overall likelihood of observing $V_1, \ldots, V_N$ becomes $\sum_{i=1}^{N} l(V_i)$. Figure 5.8 plots this likelihood for the 8 different values in this example, given the measurements from figure 5.7.

Template attacks against real-world data

The template attacks implemented in our framework have been tested against the "Grizzly" dataset provided by Choudary [34]. This data records a single load instruction on an 8-bit Atmel CPU. Our implementation was able to reconstruct the loaded value with few bits of entropy left.

The code for parsing the data and running the template attacks can be found in the folder `test/template-grizzly/`.

5.2 Extending foreign cipher algorithms

A goal of this project was to create a generic framework, which is able to record traces for third-party ciphers without major modifications. Currently, the following modification may be necessary:

- Remove all type annotations containing plain integer types. For example, methods that are restricted to `UInt8` or `Integer` arguments must be made more general. This can be achieved in two ways: Either, the argument type can be removed (or, equivalently, changed to `Any`). However, this may introduce method ambiguities if the corresponding function already has a method with an overlapping argument signature. In this case, the argument type can be changed to `Union{Integer, GenericLog, Masked}`.
- Remove explicit casts to integer types. For instance, a code line `x = Int64(x)` must be replaced. A generic way to perform casting is to use Julia's built-in promotion instead. For example, an integer `x` can be extended to an `Int64` by calculating `x = x | 0`, since the implicit type of `0` is `Int64`.
- If the code uses non-standard operations on integers, it is necessary to define those for the `GenericLog` and `Masked` type. For logging, this construction should be similar to the code in section 4.1.2. In contrast, it is not easily possible to give a template for masking operations, as the implementation heavily depends on the arithmetic or algebraic properties of the operator.

5.2.1 Integrating the `AES.jl` implementation

There already exists an implementation of the Advanced Encryption Algorithm in Julia, called `AES.jl`, that is available on GitHub. This code can be instrumented to generate traces of AES within our framework with a few modifications. Thus, although this framework already includes an implementation of the AES algorithm, other implementation variants can be easily integrated as well.

Overall, the modified source code is obtained by removing about fifty type annotations and replacing about ten explicit integer casts. A complete `diff` of the code is included in appendix A.

5.2.2 Integrating a RSA implementation

A proof-of-concept RSA implementation can be obtained from GitHub. This code can be refactored to support our framework by removing about ten annotated `Integer` types. A complete `diff` of all changes that need to be applied can be found in appendix B.

5.3 Creation of student exercises

One purpose of this project is to create side-channel data for student practicals. One such exercise has already been used as a supplement to my talk in the "Hardware security" class. In this exercise, students are provided with 2^{14} AES side-channel traces. The trace values are the Hamming weight of all intermediate values, where a random number sampled from a normal distribution ($\mu = 0$, $\sigma = 3$) is added to each value. The intended solution is to implement either a DPA or a CPA attack.

The exercise can be accessed at `https://parablack.github.io/CryptoSideChannel.jl/exercise/`. It is also included in the folder `sample_exercise/` of the source code submission.

Chapter 6

Summary & future work

6.1 Summary

In this work, we presented a generic Julia framework for modelling side-channel security. The resulting framework focuses on side-channel generation and analysis:

Firstly, our framework allows for the easy recording of side-channel data when only a Julia implementation of the cryptographic algorithm is provided. To this end, we created custom types that behave similarly to integers but dispatch different methods when operations are executed upon them. With such types, it is possible to create traces that log information about the values stored in our custom type. Furthermore, our framework allows us to automatically record masked traces by stacking custom masking types on top of our logging implementation. We provide two pre-implemented ciphers, namely AES and SPECK, and verify that they can be analyzed with respect to side-channel security using this framework. Due to the different nature of the two ciphers, this shows that our framework is flexible and can analyze and protect ciphers using lookup tables (AES) as well as ciphers that mix arithmetic and bitwise operations (SPECK). Furthermore, we have demonstrated how third-party cipher algorithms can be instrumented easily to generate side-channel traces. This process only requires few modifications to the original algorithm's code.

Secondly, this framework implements a range of side-channel attacks against different ciphers. We provide an implementation of differential power analysis (DPA) attacks, correlation power analysis (CPA) attacks, and template attacks. Furthermore, we provide an interface for recovering an overall key given multiple likely key parts. All those attacks can either be run against traces generated with our framework to verify the security of the implementation, or against real-world data collected with external hardware.

6.2 Future Work

Future work could extend this project in multiple directions:

- Recall from section 5.2 that some transformation steps need to be applied before foreign code is compatible with our framework. Most, or possibly all of those transformations could be automated with static code analysis (e.g. removing `Integer` type annotations). Such an automatic transformation on code- or AST-level could be explored in future work.
- In section 4.1.2, we described how custom methods can be added for logging. This is especially relevant for third-party integer functions that are not already included in our framework. A possible extension would be to automatically detect newly added integer methods in Julia during runtime. For those methods, then, instrumented logging versions could be added upon detection.
- While logging methods can likely be added automatically, methods for masking are harder to implement. A possible extension of this project could be to explore additional masked operations, or to extend masked operations to other numerical types. One starting point could be the support for masked floating-point numbers.
- The problem of compiler optimisations with respect to masking was discussed in section 4.1.3. It could be interesting to explore which compiler optimisations are especially problematic for masking approaches. These transformations then could be removed from the compilation pipeline.
- Recall the notion of *algebraic attacks* from section 2.3.4. In those attacks, additional constraints from the structure of the cipher are considered. A possible extension of this project could be to automatically infer "hard" constraints from the program flow. This can either be done with static analysis or with a similar duck-typing approach. The newly inferred constraints then could be used to implement, for example, a SASCA approach against some ciphers.

Bibliography

[1] Northeastern University. TeSCASE dataset. https://chest.coe.neu.edu/.

[2] Akashi Satoh, Toshihiro Katashita, and Hirofumi Sakane. Secure implementation of cryptographic modules — development of a standard evaluation environment for side channel attacks. *Synthesiology English edition*, 3:86–95, 04 2010.

[3] Maryland National Security Agency, Fort George G. Meade. TEMPEST: A signal problem. https://www.nsa.gov/Portals/70/documents/news-features/declassified-documents/cryptologic-spectrum/tempest.pdf, 1982.

[4] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[5] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In Joye and Quisquater [35], pages 16–29.

[6] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[7] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[8] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[9] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002.

[10] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK families of lightweight block ciphers. *IACR Cryptol. ePrint Arch.*, 2013:404, 2013.

[11] The OpenSSL Project. OpenSSL: The open source toolkit for SSL/TLS. www.openssl.org, April 2003.

[12] David Brumley and Dan Boneh. Remote timing attacks are practical. In *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*. USENIX Association, 2003.

[13] Giovanni Camurati, Sebastian Poeplau, Marius Muench, Tom Hayes, and AurÃ©lien Francillon. Screaming channels: When electromagnetic side channels meet radio transceivers. In *Proceedings of the 25th ACM conference on Computer and communications security (CCS)*, CCS '18. ACM, October 2018.

[14] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. PLATYPUS: Software-based power side-channel attacks on x86. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021.

[15] Naofumi Homma, Sei Nagashima, Yuichi Imai, Takafumi Aoki, and Akashi Satoh. High-resolution side-channel attack using phase-based waveform matching. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, volume 4249 of *Lecture Notes in Computer Science*, pages 187–200. Springer, 2006.

[16] Sylvain Guilley, Karim Khalfallah, Victor Lomné, and Jean-Luc Danger. Formal framework for the evaluation of waveform resynchronization algorithms. In Claudio Agostino Ardagna and Jianying Zhou, editors, *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication - 5th IFIP WG 11.2 International Workshop, WISTP 2011, Heraklion, Crete, Greece, June 1-3, 2011. Proceedings*, volume 6633 of *Lecture Notes in Computer Science*, pages 100–115. Springer, 2011.

[17] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[18] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*, volume 2523 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 2002.

[19] Marios O. Choudary and Markus G. Kuhn. Efficient, portable template attacks. *IEEE Trans. Inf. Forensics Secur.*, 13(2):490–501, 2018.

[20] Stefan Mangard. A simple power-analysis (SPA) attack on implementations of the AES key expansion. In Pil Joong Lee and Chae Hoon Lim, editors, *Information Security and Cryptology - ICISC 2002, 5th International Conference Seoul, Korea, November 28-29, 2002, Revised Papers*, volume 2587 of *Lecture Notes in Computer Science*, pages 343–358. Springer, 2002.

[21] Kai Schramm, Thomas J. Wollinger, and Christof Paar. A new class of collision attacks and its application to DES. In Thomas Johansson, editor, *Fast Software Encryption, 10th International Workshop, FSE 2003, Lund, Sweden, February 24-26, 2003, Revised Papers*, volume 2887 of *Lecture Notes in Computer Science*, pages 206–222. Springer, 2003.

[22] Kai Schramm, Gregor Leander, Patrick Felke, and Christof Paar. A collision-attack on AES: combining side channel- and differential-attack. In Joye and Quisquater [35], pages 163–175.

[23] Andrey Bogdanov, Ilya Kizhvatov, and Andrei Pyshkin. Algebraic methods in side-channel collision attacks and practical collision detection. In Dipanwita Roy Chowdhury, Vincent Rijmen, and Abhijit Das, editors, *Progress in Cryptology - INDOCRYPT 2008, 9th International Conference on Cryptology in India, Kharagpur, India, December 14-17, 2008. Proceedings*, volume 5365 of *Lecture Notes in Computer Science*, pages 251–265. Springer, 2008.

[24] Mathieu Renauld and François-Xavier Standaert. Algebraic side-channel attacks. *IACR Cryptol. ePrint Arch.*, 2009:279, 2009.

[25] Nicolas Veyrat-Charvillon, Benoît Gérard, and François-Xavier Standaert. Soft analytical side-channel attacks. *IACR Cryptol. ePrint Arch.*, 2014:410, 2014.

[26] Louis Goubin and Jacques Patarin. DES and differential power analysis (the "duplication" method). In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 1999.

[27] Thomas S. Messerges. Securing the AES finalists against power analysis attacks. In Bruce Schneier, editor, *Fast Software Encryption, 7th International Workshop, FSE 2000, New York, NY, USA, April 10-12, 2000, Proceedings*, volume 1978 of *Lecture Notes in Computer Science*, pages 150–164. Springer, 2000.

[28] Bruce Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. Twofish: A 128bit block cipher. 01 1998.

[29] Louis Goubin. A sound method for switching between boolean and arithmetic masking. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 3–15. Springer, 2001.

[30] Jean-Sébastien Coron, Johann Großschädl, Praveen Kumar Vadnala, and Mehdi Tibouchi. Conversion from arithmetic to boolean masking with logarithmic complexity. *IACR Cryptol. ePrint Arch.*, 2014:891, 2014.

[31] Thomas S. Messerges. Using second-order power analysis to attack DPA resistant software. In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings*, volume 1965 of *Lecture Notes in Computer Science*, pages 238–251. Springer, 2000.

[32] Kai Schramm and Christof Paar. Higher order masking of the AES. In David Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, volume 3860 of *Lecture Notes in Computer Science*, pages 208–225. Springer, 2006.

[33] Hasindu Gamaarachchi, Harsha Ganegoda, and Roshan Ragel. Breaking SPECK cryptosystem using correlation power analysis attack. *Journal of the National Science Foundation of Sri Lanka*, 45:393, 12 2017.

[34] Omar Choudary. Grizzly: power-analysis traces for an 8-bit load instruction. https://www.cl.cam.ac.uk/research/security/datasets/grizzly/, 2013. [Online; accessed 30-May-2021].

[35] Marc Joye and Jean-Jacques Quisquater, editors. *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, volume 3156 of *Lecture Notes in Computer Science*. Springer, 2004.

Appendix A

Code changes for AES

The original software has been taken from https://github.com/faf0/AES.jl at commit 92ea00536a7280c710f9dfa755e6ea3747b1b7aa. The following changes have been applied:

A.1 Changes to the file aes-code.jl

```
< function AESEncrypt(plain, key)
---
> function AESEncrypt(plain::Array{UInt8, 1}, key::Array{UInt8, 1})
92,94c94,96
< # Both the ciphertext and key are arrays holding bytes of type Any.
< # The returned plaintext is an array holding bytes of type Any.
< function AESDecrypt(cipher::Array{Any, 1}, key::Array{Any, 1})
---
> # Both the ciphertext and key are arrays holding bytes of type UInt8.
> # The returned plaintext is an array holding bytes of type UInt8.
> function AESDecrypt(cipher::Array{UInt8, 1}, key::Array{UInt8, 1})
101c103
< function AEScrypt(input, key)
---
> function AEScrypt(input::Array{UInt8, 1}, key::Array{UInt8, 1})
112c114
< function AESParameters(key)
---
> function AESParameters(key::Array{UInt8, 1})
128c130
< function KeyExpansion(key::Array, Nk::Int, Nr::Int)
---
> function KeyExpansion(key::Array{UInt8, 1}, Nk::Int, Nr::Int)
131c133
<         w = Array{Any}(undef, WORDLENGTH * Nb * (Nr + 1))
---
>         w = Array{UInt8}(undef, WORDLENGTH * Nb * (Nr + 1))
149c151
< function SubWord(w)
---
> function SubWord(w::Array{UInt8, 1})
151c153
<         map!(x -> SBOX[x + 1], w, w)
---
>         map!(x -> SBOX[Int(x) + 1], w, w)
155c157
< function RotWord(w)
---
> function RotWord(w::Array{UInt8, 1})
169c171
< function AESCipher(inBytes, w, Nr::Int)
---
> function AESCipher(inBytes::Array{UInt8, 1}, w::Array{UInt8, 1}, Nr::Int)
191c193
< function AESInvCipher(inBytes, w, Nr::Int)
```

```
---
> function AESInvCipher(inBytes::Array{UInt8, 1}, w::Array{UInt8, 1}, Nr::Int)
214c216
< function columnIndices(column)
---
> function columnIndices(column::Int)
219c221
< function rowIndices(row)
---
> function rowIndices(row::Int)
223c225
< function SubBytes(a)
---
> function SubBytes(a::Array{UInt8, 1})
227c229
< function InvSubBytes(a)
---
> function InvSubBytes(a::Array{UInt8, 1})
231c233
< function SubBytesGen(a, inv::Bool)
---
> function SubBytesGen(a::Array{UInt8, 1}, inv::Bool)
233c235
<       f = (x) -> box[(x) + 1]
---
>       f = (x) -> box[Int(x) + 1]
238c240
< function ShiftRows(a)
---
> function ShiftRows(a::Array{UInt8, 1})
242c244
< function InvShiftRows(a)
---
> function InvShiftRows(a::Array{UInt8, 1})
246c248
< function ShiftRowsGen(a, inv::Bool)
---
> function ShiftRowsGen(a::Array{UInt8, 1}, inv::Bool)
258c260
< function MixColumns(a)
---
> function MixColumns(a::Array{UInt8, 1})
262c264
< function InvMixColumns(a)
---
> function InvMixColumns(a::Array{UInt8, 1})
266c268
< function MixColumnsGen(a, inv::Bool)
---
> function MixColumnsGen(a::Array{UInt8, 1}, inv::Bool)
282c284
< function AddRoundKey(s, w)
---
```

A.2 Changes to the file `aesgf-code.jl`

```
81,82c79,80
< function gadd(a::UInt8, b::UInt8)
<   xor.(a , b)
---
> function gadd(a::Any, b::Any)
>   xor(a, b)
85c83
< function gadd(v::Array{UInt8})
---
> function gadd(v)
94,95c92,93
< function gsub(a::UInt8, b::UInt8)
<   xor.(a, b)
---
> function gsub(a::Any, b::Any)
>   xor(a, b)
99c97
```

```
< function gmul(a::UInt8, b::UInt8)
---
> function gmul(a::Any, b::Any)
123c121
< function gmulbits(a::UInt8, b::UInt8)
---
> function gmulbits(a::Any, b::Any)
144c142
< function gmulinv(a::UInt8)
---
> function gmulinv(a::Any)
149c147
<                 return ATABLE[256 - LTABLE[Int(a) + 1]]
---
>                 return ATABLE[256 - LTABLE[(a) + 1]]
```

Appendix B

Code changes for RSA

The original software has been taken from https://gist.github.com/johnmyleswhite/9302145. Furthermore, the original software has been very slightly modified to be compatible with Julia 1.6, i.e. the `@printf` macro was replaced with `println` and curly braces were replaced with `where` clauses. Those changes are purely of syntactical nature and do not appear in the `diff` below.

The following changes have been applied to make this code compatible with our framework:

```
10a11,12
> using CryptoSideChannel
>
12c14
< function extended_gcd(a::T, b::T) where {T <: Integer}
---
> function extended_gcd(a::T, b::T) where T
22c24
<     function modulo_inverse(a::Integer, n::Integer)
---
>     function modulo_inverse(a, n)
27c29
<     totient(p::T, q::T) where {T <: Integer} = (p - one(T)) * (q - one(T))
---
>     totient(p::T, q::T) where T = (p - one(T)) * (q - one(T))
30c32
<     square(x::Integer) = x * x
---
>     square(x) = x * x
33c35
<     function modulo_power(base::T, exp::T, n::T) where {T <: Integer}
---
>     function modulo_power(base::T, exp::T, n::T) where T
51c53
<     function is_legal_public_exponent(e::T, p::T, q::T) where {T <: Integer}
---
>     function is_legal_public_exponent(e::T, p::T, q::T) where T
56c58
<     function private_exponent(e::T, p::T, q::T) where {T <: Integer}
---
>     function private_exponent(e::T, p::T, q::T) where T
65c67
<     function encrypt(m::T, e::T, n::T) where {T <: Integer}
---
>     function encrypt(m::T, e::T, n::T) where T
74c76
<     function decrypt(c::T, d::T, n::T) where {T <: Integer}
---
>     function decrypt(c::T, d::T, n::T) where T
82,83c84,88
<     p = BigInt(41) # A "large" prime
<     q = BigInt(47) # Another "large" prime
---
```

```
>   arr = []
>   clos = () -> arr
>
>   p = Logging.FullLog(BigInt(41), clos) # A "large" prime
>   q = Logging.FullLog(BigInt(47), clos) # Another "large" prime
86c91
<   e = BigInt(7)                 # The public exponent
---
>   e = Logging.FullLog(BigInt(7), clos)  # The public exponent
89c94
<   plaintext = BigInt(42)
---
>   plaintext = Logging.FullLog(BigInt(42), clos)
```