A customizable side-channel modelling and analysis framework in Julia

Simon F. Schwarz
Robinson College

UNIVERSITY OF CAMBRIDGE

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 19, 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: ??

Signed:

Date:

Abstract

Write a summary of the whole thing. Make sure it fits in one page.

Contents

List of Figures

List of Tables

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 tries to address this gap by providing a purely software-side solution for generating and analyzing side-channel information, removing the need for expensive hardware and specialized knowledge. In particular, since this project scales effortlessly, results can be used in the context of teaching side-channel security without providing hardware to every student.

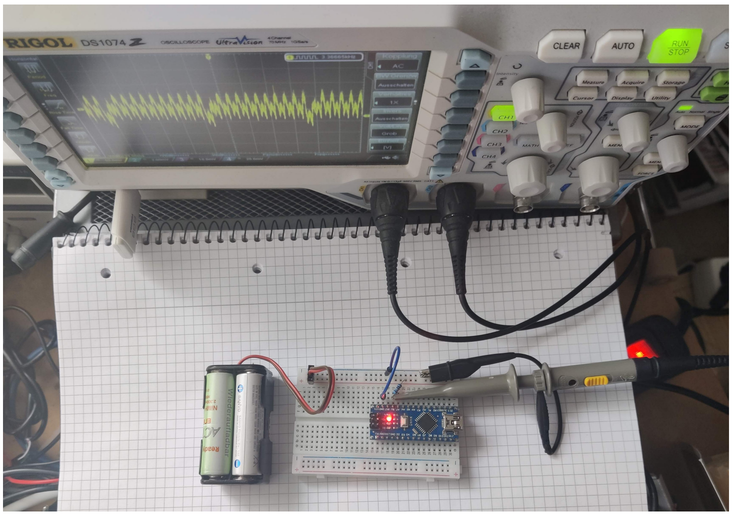

Figure 1.1: Real-world side-channel analysis: An oscilloscope measuring the power of a microchip.

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 ciphers 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.

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.

Equipped with a convenient way to record leakage emissions from a cryptographic algorithm, our framework provides various methods for analyzing the recorded data with respect to side-channel security.

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: "An adversary is unable to efficiently infer the key given only a pair of plaintext and corresponding ciphertext". Modern cryptographic algorithms like the Advanced Encryption Standard (AES) [9] or SPECK [10] 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 high or low wires. When an operation on the number is performed the level of the wires then change accordingly. In most hardware processors, pulling a line high (or even low) consumes more energy than not changing the state at all. Hence, it stands to reason that, for example, computing $(0000)_2 + (0001)_2$ needs less power than computing $(1010)_2 + (1101)_2$. 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. A prominent example is the NSA's TEMPEST program [3] which extracted sensitive information from electromagnetic emissions.

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

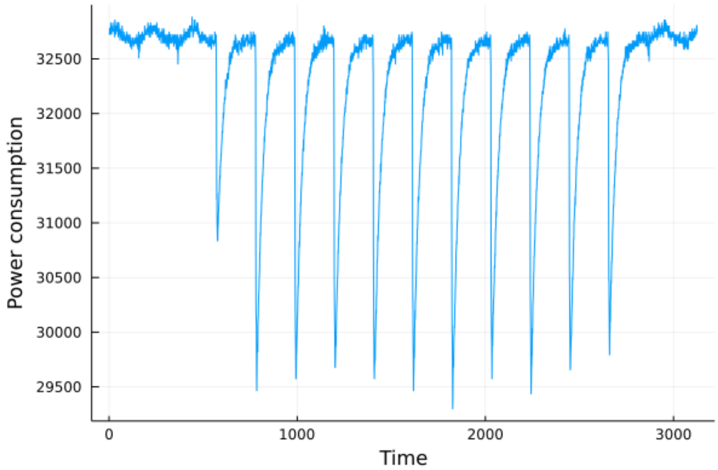

Figure 2.1: Power consumption during the execution of an AES encryption. The first peak is caused by loading the data, the subsequent ten peaks correspond to the ten rounds of AES. Data source: [1]

directly taken from the output of the Analog-Digital-Converter used to record the trace.

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 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 [13]. 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

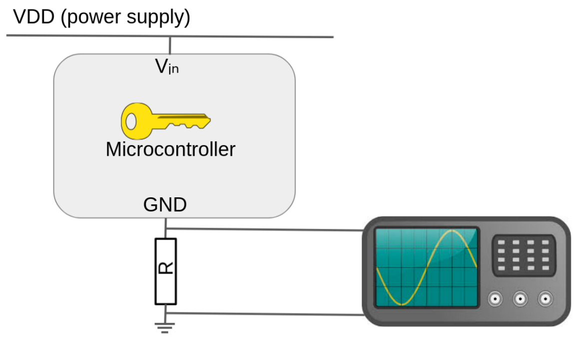

Figure 2.2: Schematic wiring for collecting power traces with an oscilloscope.

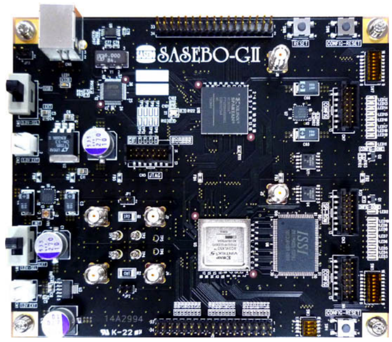

Figure 2.3: The SASEBO side-channel evaluation board [2].

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*

[14] and *amplitude-only correlation* [15]. Both methods employ results from a Fourier transformation to re-align traces based on similarity.

2.3 Attacks

Historically, the first formal analysis of side-channel attacks has been introduced by Kocher in 1996 [16], 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 [17]. 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 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.

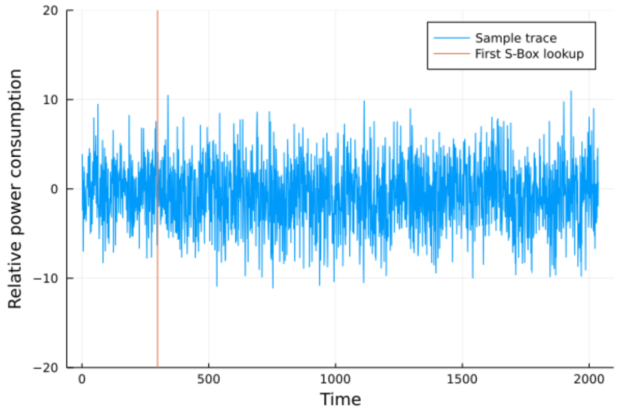

Figure 2.4: A recorded power trace, and the point in time where the first AES S-Box lookup is performed.

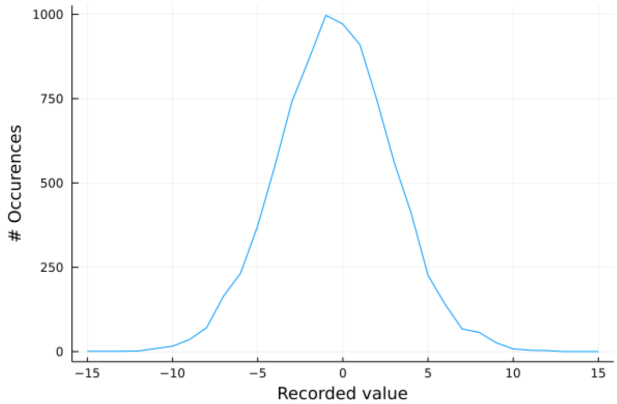

Figure 2.5: Power consumption of all traces at the first S-Box lookup.

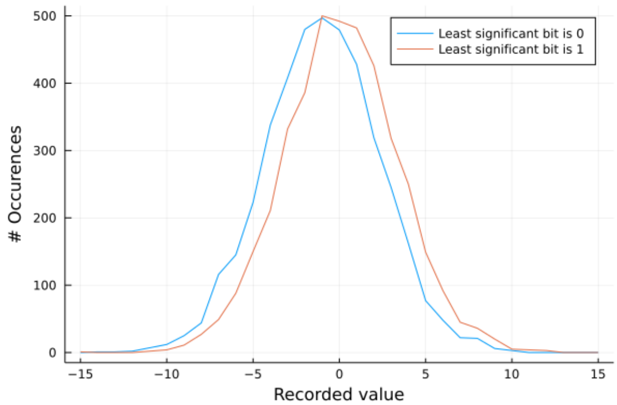

Figure 2.6: Power consumption for the two partitions at the first S-Box lookup.

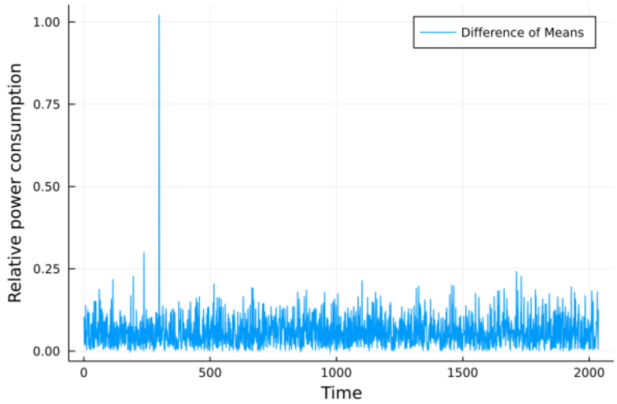

Figure 2.7: DOM against time between the two positions. The spike marks the time of the first S-Box lookup.

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 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

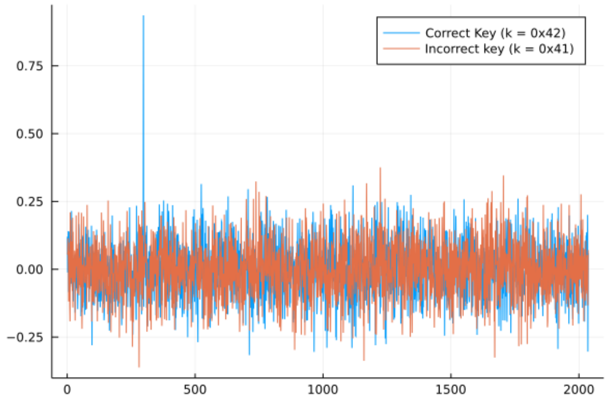

Figure 2.8: DOM against time for two key guesses. The spike for the correct key guess is at the position where the first S-Box output is calculated.

equation for the first S-Box output I_i at position i in the first round is:

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

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 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 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 $16 \cdot 2^8$ queries in total. This is a major improvement compared to a naive brute-force attack, which would need up to $2^{8 \cdot 16}$ tries.

2.3.2 CPA

Correlation Power Analysis was published 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 present 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 that is processed R, and should return a side-channel estimate W_R.

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 pull a line high or low. 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

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

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

Example: Attack against AES

Given a power estimation model, we can target the first S-Box output as in DPA. Let W_R denote the power estimation to calculate value R. Consider only the i-th position in AES ($1 \leq i \leq 16$). Then, for each different input $X_i \in \{0, \ldots, 255\}$, we can calculate $W_{S[X_i \oplus K_i]}$ if K_i is known. Since $K_i \in \{0, \ldots, 255\}$ is a single key byte, this value can be brute-forced.

For each trace, $W_{S[X_i \oplus K_i]}$ is a trace-specific power estimation. Now, at the point where $S[X_i \oplus K_i]$ is calculated, we expect the measured values to correlate with the power estimation. For example, Figure 2.9 shows the power estimation for the correct 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 clear correlation is not necessarily required for this attack to work.

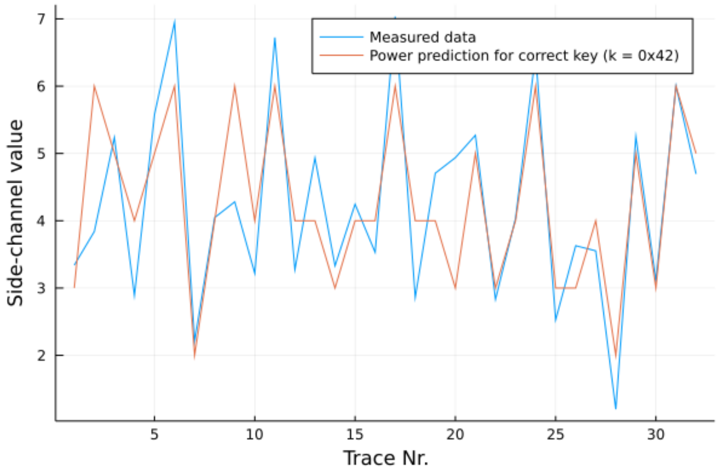

Figure 2.9: Comparison between the power prediction for the correct key, and the actual measured power at the point of the first S-Box computation ($t = 298$). Both are strongly correlated ($\rho = 0.87$).

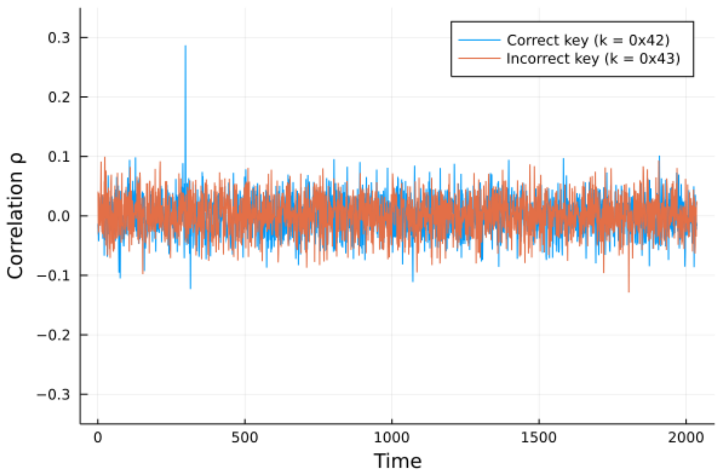

Figure 2.10: Correlation for the correct and for an incorrect key against time. The correct key exhibits a spike in correlation at the computation of the first S-Box output ($t = 298$)

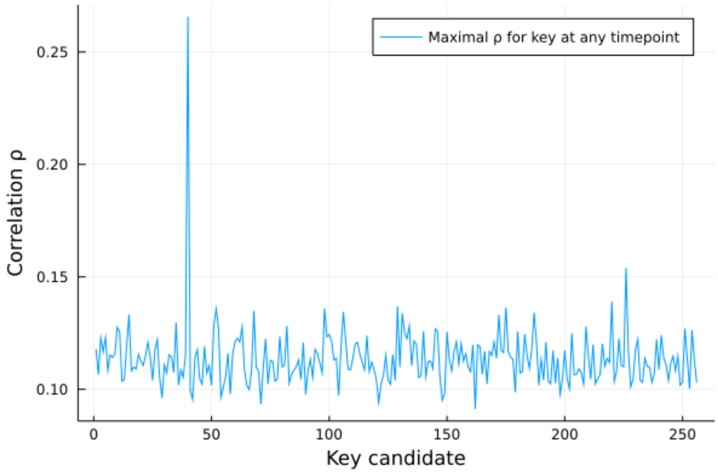

Figure 2.11: Correlation for all keys

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 2.10 shows 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, likely key bytes are expected to have higher spikes in the correlation at the targeted time point. Thus, to infer the correct key, it is sufficient to consider the maximal absolute correlation at any time point. Figure 2.11 shows the maximal correlation for all key candidates, with a clear spike at the correct key $k = 0x42$.

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 noise out by averaging, template attacks actively classify 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:

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

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.

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 are:

- *SPA on AES Key-Expansion*: In [18], 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 [19]. Subsequently, collision attacks have been generalized to cover AES in [20], and improved by Bogdanov, Kizhvatov, and Pyshkin in [21]. 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. [21] 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 [22]. 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 [23]. 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, 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 is rather hard to implement and often slows 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 [24]. 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 randomness of the 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 [25] suggests that for a table lookup $Y = T[X]$, with $X = A_X \oplus M_X$ a table T' with the following specification is computed:

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

Now, a masked table lookup can be performed on A_X. The resulting value itself is then masked with M'_X, which can be either a fresh random value, or $M'_X = M_X$ if no re-randomisation is desired. However, note that calculating T requires $\mathcal{O}(|T|)$ steps. Hence, it is 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 [26] 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

$$V = A_V + M_V \mod 2^k$$

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 boolean 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 [25]. However, his conversion algorithm was again vulnerable to differential power attacks. Subsequently, Goubin proposed in [27] 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 [28]. 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 [29] 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, [30] 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 [30] 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 unifies 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.

At its core, Julia uses a dynamic type system with support for polymorphism. More details on Julia's type system can be found in section 3.1. A distinctive feature of Julia is the *multiple dispatch* paradigm, which is used in combination with the type system to dynamically dispatch type-specific code. A detailed explanation of multiple dispatch can be found in section 3.2.

3.1 Types

Julia is a *dynamically* typed language. Hence, types of objects may not be known at compile time. 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 allowing explicit *type annotations*. Besides the performance improvements, these constructs serve an even more important purpose, since they allow for *multiple dispatch*. A detailed explanation of multiple dispatch can be found in 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. A special abstract datatype is the `Any` type, which is a supertype of all other types. In contrast, concrete types are always instantiable and are either composite or primitive. Primitive types allow for declaring values that operate directly on bits, while composite types contain a collection of named fields. In this thesis, we will focus on declaring custom composite types. Concrete types can subtype other abstract types, but cannot be a subtype of a concrete type itself. For example, the subtyping hierarchy of numerical types 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 are bits types. Thus, all concrete numerical types are bits types as well. Furthermore, numerical composite types like `Complex` numbers are bits-types as well.

Note that *closures* that only capture global variables are bits-types as well. This can be, for example, used for referencing mutable objects like arrays:

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

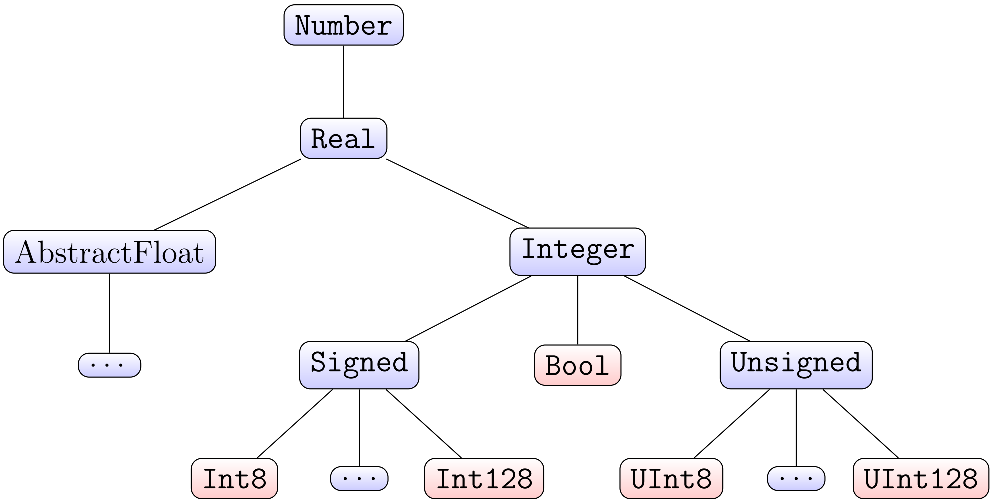

Figure 3.1: Hierarchy of Julia's numerical types. Abstract types are blue, concrete types are red.

```
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}`.

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 the use of multiple different methods for a single function. This paradigm is called *multiple dispatch*. 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. Thus, 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 at compile time. Hence, the actual dispatch can only be performed 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

4.1 Custom types

This section first describes in section 4.1.1 how custom integer types can be added to the Julia language, focusing on useful properties for cryptography. Afterwards, in section 4.1.2 we look at a logging datatype. Lastly, section 4.1.3 considers a masking datatypes.

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:

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

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

```julia
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 procedures 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
```

Extending this to other operators is, of course, extremely tedious and results in large code duplicates. This can be avoided by the use of metaprogramming to generate those functions. Consider the following, equivalent function 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 by using Julia's metaprogramming:

```julia
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 metaprogramming as well. Here, we iterate over all (binary) operators that we want to define in another for-loop:

```julia
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

- Arithmetic
- Bitwise Operators

- Comparison
- Array accesses
- Randomness

Subtypes of `Integer`

A canonical question to ask is whether the new `MyInt` type should be a subtype of `Integer`. At first glance, this would seem like the right choice. However, establishing this subtype relationship poses the following two issues:

First, 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

function Base.getindex(v::AbstractArray, i::MyInt)
    return v[i.value]
end

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

v[i]
```

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

```
struct MyInt <: Integer
    value::Int
end
```

```julia

function Base.getindex(v::AbstractArray, i::MyInt)
    return v[i.value]
end

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

v[i]
```

However, this second block produces an error on execution:

```
ERROR: LoadError: 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, since `Array <: AbstractArray`, we must define another method `getindex(::Array, ::MyInt)`. However, it is not sufficient to define this single method to work with every possible `AbstractArray`, since a corresponding method must be defined for every concrete subtype of `AbstractArray`. However, this process cannot be completed at compile-time, since new subtypes may be added dynamically. For example, the `StaticArrays` package provides a type `StaticArray{...} <: AbstractArray{...}`. Hence, declaring our new type as a subtype of `Integer` requires additional method declarations that may be not even known at compile-time.

Another argument against subtyping the `Integer` is the plurality of abstract integer types: One benefit of subtyping is compatibility to code that restricts arguments to `Integer` types. In this code, type annotations do not need to be changed to work with this framework. However, many projects restrict

Operation	Resulting A_Z	Resulting M_Z
$Z = X \oplus Y$	$A_X \oplus A_Y$	$M_X \oplus M_Y$
$Z = X \ \& \ Y$	$(A_X \ \& \ A_Y) \oplus (A_X \ \& \ M_Y) \oplus (M_X \ \& \ A_Y)$	$M_X \ \& \ M_Y$
$Z = X \mid Y$	$(A_X \& A_Y) \oplus (A_X \& M_Y) \oplus (M_X \& A_Y)$	$M_X \ \& \ M_Y$

function arguments to, for example, `Signed` or `Unsigned`. Thus, those types have to be manually exchanged again.

Following the two arguments outlined above, we will not use subtypes of `Integer` throughout this project. Instead, our type declarations will not have any specific supertype.

4.1.2 Logging

4.1.3 Masking

Boolean Masking

Include this table, or something similar.

Arithmetic Masking

Conversion

Problems with high-level masking

TODO copy from docs

4.1.4 Obtaining traces

One of the main features of this project is the ability to obtain traces of an arbitrary program working with integers. For example, this feature can be used to obtain traces from a 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 idea 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. Whenever a value `x` is processed, the value `reduction_function(x)` is appended to the logging array. In this example, we choose `reduce_function = x -> Base.count_ones(x) +` which computes the Hamming weight with uniform noise.

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

```
trace = []

function encrypt_collect_trace(pt::MVector{16, UInt8})
    global trace
    trace = []
    clos = () -> trace

    d = Distributions.Normal(0, 2)
    reduction_function = x -> Base.count_ones(x) + rand(d)

    key_log = map(x -> Logging.SingleFunctionLog(x, clos, reduction_function), SECRET_KEY)
    pt_log  = map(x -> Logging.SingleFunctionLog(x, clos, reduction_function), pt)

    AES.AES_encrypt(ptl, kl)
    # or SPECK.SPECK_encrypt(ptl, kl)

    return copy(trace)
end
```

Masked traces

Collecting masked traces is a very similar process. The only difference is that the logging datatype of input and key must be encapsulated in a masking datatype:

```
coll = []

function encrypt_collect_masked_trace(pt::MVector{16, UInt8})
    global coll
    global key
    coll = []
    clos = () -> coll

    reduce_function = x -> Base.count_ones(x)

    kl = map(x -> Masking.BooleanMask(Logging.SingleFunctionLog(x,
    ↪   clos, reduce_function)), key)
    ptl = map(x -> Masking.BooleanMask(Logging.SingleFunctionLog(x,
    ↪   clos, reduce_function)), pt)

    unwrap = x -> Logging.extractValue(Masking.unmask(x))
    output = unwrap.(AES.AES_encrypt(ptl, kl))

    return (output, copy(coll))
end
```

4.2 Ciphers

In this project, two different ciphers are implemented, namely AES and SPECK. The following two sections describe the implementation of the ciphers in detail.

4.2.1 AES

The *Advanced Encryption Algorithm* [9] has been standardized in 2000. Currently, AES is widely used throughout different applications. AES is also a popular algorithm for hardware applications like Smartcards. For example, the very popular "MIFARE DESFire" series of access control cards supports AES. This makes AES a prime target for hardware security.

In this project, the AES algorithm is implemented in the file `src/cipher/AES.jl`. It provides the following interface 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 with AES, 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-/decryption process.

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

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

Key schedule

Another important property while analysing AES are methods to compute the key schedule for all rounds. The following two methods provide basic functionality for providing the key schedule:

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

Note that `inv_key_expand` is only needed when the last round of AES is attacked. For example, this is common if the decryption process is monitored. Since the last round of AES is the first round of the decryption process, the last round key of AES is obtained. Fortunately, the key expansion of AES is reversible. Thus, `inv_key_expand` provides the whole key schedule given only the *last* round key of AES.

4.2.2 SPECK

SPECK [10] is a light-weight block cipher proposed by the NSA in 2013. Internally, SPECK is an *add-rotate-xor* cipher. Thus, SPECK mixes (modular) arithmetic operations (addition) with bitwise operations (xor, bitwise rotation).

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 encrypted data in two shares of 64 bit.

Key schedule

As AES, SPECK uses a key schedule for each 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, containing each round key. Note that it this operation is not easily invertible from a single round key, since round keys consist only of 64 bits, while the SPECK key has 128 bits. Thus, recovering the original key from attacked round keys is more difficult, but nonetheless possible. This process in described in TODO ref CPA on SPECK / attacking mult rounds.

4.3 Attacks

4.3.1 DPA

4.3.2 CPA

CPA against AES

TODO ref or copy the whole chapter here

CPA against SPECK

TODO Note that the $\oplus$ operation is linear, and addition is almost linear.

TODO

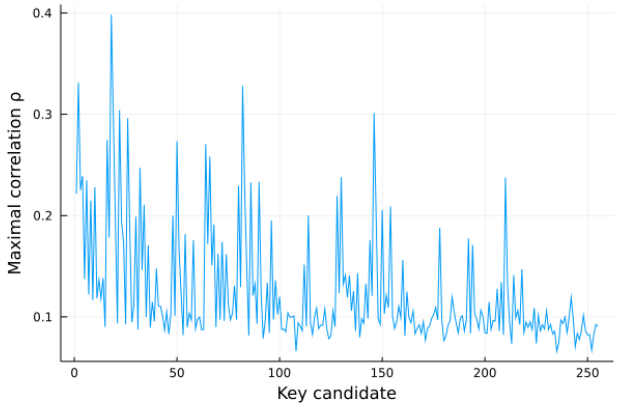

Figure 4.1: Correlation in SPECK for different key bytes (Correct key: `0x12`).

Figure 4.1 shows the correlation for different keys. Here, multiple spikes are observable. Note that the largest spike corresponds to the actual key byte `0x12` used. The other spikes are caused by the *linearity* of all operations. For example, there are other observable spikes at `0x12 + 0x10`, `0x12 + 0x20`, `0x12 + 0x40`. This is because in all of the previously mentioned values, only a *single bit* differs to the actual key value. Since all operations in our power estimate are (almost) linear, the single bit difference propagates up to the final correlation. Hence, candidates that are close to our actual key will show significant correlation as well. In contrast, the AES power consumption was calculated after the S-Box lookup. Since the S-Box lookup is non-linear, there is no linear relationship between the key and the estimated power consumption. Hence, only the correct key exhibits clear spikes in correlation.

... TODO

[31]

4.3.3 Key combination

TODO the two approaches: sort + prioqueue, stack approach by me. `LikelyKey` docs.

4.4 Documentation

A detailed project documentation with examples can be found here.

4.5 Technical details

4.5.1 Installation

The 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.

4.5.2 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 with the SafeTestsets.jl package. In total, more than 50000 unit tests (some of which are automatically generated) in 9 test sets are present.

Continuous integration Building and testing the project is automated using continuous integration (CI). For the CI 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 to 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-builts 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

Timing would be nice. + we need space for the nice pictures rendered by Plot

Take https://github.com/faf0/AES.jl/blob/master/src/aes-code.jl and show how easy to port? Take https://github.com/mkfryatt/BinaryECC and show how easy to port?

Chapter 6

Summary & conclusions

6.1 Summary

6.2 Future Work

SASCA = completely generic analysis on crypto algos? Infer constraints. Possible but hard?

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. Nacsim 5000: Tempest fundamentals. Partially declassified transcript: http://cryptome.org/nacsim-5000.htm, 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 [32], 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] 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.

[14] 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.

[15] 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.

[16] 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, Proceed-*

ings, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[17] 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.

[18] 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.

[19] 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.

[20] Kai Schramm, Gregor Leander, Patrick Felke, and Christof Paar. A collision-attack on AES: combining side channel- and differential-attack. In Joye and Quisquater [32], pages 163–175.

[21] 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.

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

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

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.