Rosita++: Automatic Higher-Order Leakage Elimination from Cryptographic Code

Madura A. Shelton
University of Adelaide
Australia
madura.shelton@adelaide.edu.au

Łukasz Chmielewski
Radboud University and Riscure
The Netherlands
lukasz@cs.ru.nl

Niels Samwel
Radboud University
The Netherlands
nsamwel@cs.ru.nl

Markus Wagner
University of Adelaide
Australia
markus.wagner@adelaide.edu.au

Lejla Batina
Radboud University
The Netherlands
lejla@cs.ru.nl

Yuval Yarom
University of Adelaide
Australia
yval@cs.adelaide.edu.au

ABSTRACT

Side-channel attacks are a major threat to the security of cryptographic implementations, particularly for small devices that are under the physical control of the adversary. While several strategies for protecting against side-channel attacks exist, these often fail in practice due to unintended interactions between values deep within the CPU. To detect and protect from side-channel attacks, several automated tools have recently been proposed; one of their common limitations is that they only support first-order leakage.

In this work, we present Rosita++, the first automated tool for detecting and eliminating higher-order leakage from cryptographic implementations. Rosita++ proposes statistical and software-based tools to allow high-performance higher-order leakage detection. It then uses the code rewrite engine of Rosita (Shelton et al. NDSS 2021) to eliminate detected leakage. For the sake of practicality we evaluate Rosita++ against second and third order leakage, but our framework is not restricted to only these orders.

We evaluate Rosita++ against second-order leakage with three-share implementations of two ciphers, present and Xoodoo, and with the second-order Boolean-to-arithmetic masking, a core building block of masked implementations of many cryptographic primitives, including SHA-2, ChaCha and Blake. We show effective second-order leakage elimination at a performance cost of 36% for Xoodoo, 189% for present, and 29% for the Boolean-to-arithmetic masking. For third-order analysis, we evaluate Rosita++ against the third-order leakage using a four-share synthetic example that corresponds to typical four-share processing. Rosita++ correctly identified this leakage and applied code fixes.

1 INTRODUCTION

Cryptography is one of the main tools used to protect data, both in transit and at rest. With the increased proliferation of small computing devices into every aspect of modern life, secure cryptography is more important than ever. Traditionally, the security of cryptographic primitives was evaluated in terms of their mathematical function. However, in 1996 Kocher [36] demonstrated that the computation of a cryptographic primitive can interact with the environment in which it is computed. Such *side channels* can leak information about the internal state of the computation, leading to a potential collapse of the security of the implementation.

Since then, significant effort has been invested in researching side-channel attacks. On the offensive side, attacks have been demonstrated against various types of primitives, including symmetric ciphers [6, 46], public-key systems [28, 44] post-quantum cryptography [2], and non-cryptographic implementations [4, 60, 62, 67]. These attacks exploit various side channels, such as power consumption [37] electromagnetic emanations [24, 52], microarchitectural state [8, 27, 40], and even acoustic and photonic emissions [29, 38]. On the defensive side, proposals range over hardware designs that reduce emissions [15], software solutions that ensure secret-independent execution [27], adding noise to hide the signal [45], and information masking techniques [13, 35, 48].

Masking techniques, in particular, are considered promising because they provide a theoretical basis that guarantees protection. In a nutshell, these operate by splitting secrets into multiple *shares*, such that to recover a secret, an attacker needs to observe all of the shares that comprise the secret. For example, in order-d Boolean masking, a secret v is split into $d+1$ shares, $v_0, \ldots, v_d$ such that for $1 \leq i \leq d$, v_i is chosen uniformly at random, and v_0 is selected such that $v_0 = v \oplus v_1 \oplus v_2 \oplus \cdots \oplus v_d$. Such schemes are considered safe because attackers are limited in the number of observations they can made on the internal state of the implementations. Thus a d-order secure implementation which consists of $d+1$ shares, is secure against an attacker that can observe up to d internal values [35].

Despite the theoretical security, masked implementations often fail to provide the promised resistance in practice. A main cause for this failure is unintended interactions between values processed by the hardware, which allow an attacker to observe multiple shares with a single observation [3, 25, 49]. Thus, to protect against unintended interactions, designers need to first implement the cryptographic primitives aiming for best protection and then go through several rounds of evaluation. In each such round, the implementation is evaluated for the presence of leakage and then tweaked to eliminate observed leakage. The process usually repeats until no evidence of leakage is observed. This experimental process is expensive because it requires significant expertise, both in the design of cryptographic primitives, and in setting up and performing analysis of hardware measurements.

To reduce the effort required for producing side-channel resistant implementations, a designer may elect to use a leakage emulator [11, 42, 49, 65] instead of evaluating the hardware. A recent proposal goes one step further and suggests Rosita, a tool that combines a

leakage emulator with software manipulation techniques, providing automatic elimination of side-channel leakage [61]. However, one limitation of Rosita is that it only provides first-order security and cannot protect against higher-order attacks. Thus, in this paper we ask the following question:

Can we automatically detect and correct higher-order side-channel leakage from implementations protected with masking?

1.1 Our Contribution

In this work we present Rosita++, an extension to Rosita [61] that performs higher-order leakage detection and mitigation. At its core, Rosita++ extends the leakage detection and root-cause analysis capabilities of Rosita to support high-order analysis. It then uses the Rosita code rewrite engine to modify the evaluated implementation and eliminate leakage. While Rosita++ can analyse and fix code at any order, in this work we concentrate on second- and third-order leakage. We do not investigate orders higher than three for the sake of practicality. The complexity of third-order side-channel analysis is significant and the analysis requires tens of millions of traces (i.e. number of side-channel measurements). We expect that fourth-order analysis would require at least hundreds of millions of traces (i.e. months of trace acquisition with a similar setup to ours), making such analyses impractical in many scenarios.

Implementing Rosita++ is far from straightforward. The main appeal of high-order secure implementations is that high-order analysis is significantly more complex then first-order analysis. In particular, we identify three main challenges: the impact of the quadratic (for second order) and cubic (for third order) increases in trace lengths on the statistical tools used for the analysis, the explosion in the amount of data that needs be processed both due to the increase in trace length and because of the required increase in the number of required traces, and the complexities involved in performing multivariate root-cause analyses.

To address these challenges, we develop statistical software tools that allow robust and efficient high-order leakage analysis. Our software tools can combine and analyse millions of traces each with thousands of sample points and perform efficient bivariate and trivariate analysis on the combined traces. We believe that these tools are of independent value for the side-channel community and could be used for high-order analysis in a wide-range of cases.

We assess the second-order effectiveness of Rosita++ with three cryptographic primitives, which represent different points in the design space of symmetric cryptography. Present [56] is a popular lightweight block cipher with a traditional substitution-permutation network design. We extend the two-share present implementation of Sasdrich et al. [56] to support three shares. In contrast, Xoodoo [19, 20] is a modern cryptographic primitive that underlies multiple higher-level primitives [19]. We implement a three-share version of Xoodoo, building on the non-linear χ layer from Keccak. Finally, we evaluate Boolean-to-arithmetic masking [32] which is a cryptographic building block that converts a Boolean mask to an arithmetic mask, and is often required in implementing side-channel resistant instances of cryptographic algorithms that mix Boolean and arithmetic operations, e.g., SHA-2 [47], ChaCha [5], Blake [1], Skein [23], IDEA [39], and RC6 [55]. We implemented the second-order Boolean-to-arithmetic masking of Hutter and Tunstall [33].

We show that Rosita++ removes all leakage detected in the real experiment up to 2 million traces in Xoodoo and Boolean-to-arithmetic masking. For present all but one leakage point were removed. Further, we find that Rosita++ only requires to emulate 500,000 traces to achieve the same level of protection as achieved by analysing 2 million side-channel traces from physical hardware. Rosita++ is available as an open-source project at https://github.com/0xADE1A1DE/Rositaplusplus.

In summary, in this work we make the following contributions:

- We explore automated tools for automatic second and third order side-channel detection and protection. (Section 3.1.)
- We develop statistical and software tools for addressing the challenges. (Sections 3.2 to 3.4.)
- We build Rosita++, the first tool to automatically detect and remove unintended high-order leakage, evaluate it on three cryptographic primitives and demonstrate its efficiency. (Section 4.)
- We made Rosita++ and the associated tools available as open source.

1.2 Organisation of this paper

Section 2 introduces the necessary background on side-channel attacks, masking, univariate and multivariate side-channel leakage assessment methods, leakage emulators and automatic countermeasures, and statistical tools that we use in this work. In Section 3, we describe the design for Rosita++ and in particular how we extend Rosita to higher orders and what the challenges we face. We also describe multivariate root-cause analysis and how Rosita improves the code security by code rewrites. Subsequently, in Section 4, we present the results of our evaluation, including both the emulation results and the complimentary side-channel measurement evaluation. Finally, in Section 5 we conclude the paper.

2 BACKGROUND

2.1 Side-Channel Attacks

Traditional cryptanalysis attacks aim to extract secrets from cryptographic algorithms by focusing on the mathematical aspects of such algorithms. Side-channel attacks, in contrast, focus on obtaining internal values processed by the algorithm, which are not expected to become public. This information is gained by exposing intermediate values of an algorithm through the process of collection and analysis of measurements of physical phenomena. Such phenomena include timing, power consumption, acoustics, electro-magnetic emanations or properties such as various internal states of CPU components.

In 1996, Kocher [36] was the first to publish an exploit of side-channel leakage to recover secret information that was processed by a cryptographic algorithm. The algorithm in question was implemented with high performance in mind, and therefore ran in non-constant time; this allowed the timing differences for different inputs to be exploited. Subsequently, Kocher et al. [37] used side-channel information from power consumption to recover secret information in a new type of attack called Differential Power Analysis (DPA). In DPA, an attacker calculates a differential trace by finding the difference between averages of measured traces of

a certain bit being 1 or 0 given a plaintext and a guessed part of the key. With an incorrect guess for part of the key the sum of all difference of averages along the trace would converge to zero while for a correct guess this converges to a non-zero value. The model that Kocher et al. [37] used assumes that each individual bit of an intermediate value contributed to the power consumption of the device such that (with enough traces) it could be revealed. By extending the same idea to the power consumption of a register, we arrive at the Hamming weight model. This model states that the consumed power is proportional to the number of bits that are set [43].

In another type of attack, called Correlation Power Analysis (CPA), Brier et al. [10] used the correlation coefficient as a side-channel distinguisher, i.e. the statistical method used for the key recovery. CPA allows an attacker to recover parts of a key that is used in a cipher by using a known plaintext attack: samples measured using a single probe are correlated against a synthetic power value that is generated from an intermediate value calculated for all values that a subkey can take. Commonly, the power model used for CPA is either Hamming weight or Hamming distance. In the Hamming distance model the consumed power is proportional to the number of different bits between two intermediate values. Such leakage can occur in practice when an intermediate value stored in a register is overwritten with another value.

2.2 Side-Channel Leakage Assessment

Side-channel leakage assessment measures how vulnerable a device is to side-channel attacks. This cannot be an exhaustive assessment, as it is impossible to try all possible attacks on a device. However, such assessment is still valuable to the manufacturers of secure devices as they can benchmark a level of security of devices during the design and manufacturing process.

In side-channel leakage assessment, the main question we try to answer is whether the evaluated device shows significant leakage. Therefore, a device must show statistically significant leakage to be classified as insecure. Standards such as International Standard ISO/IEC 17825:2016(E) [34] build on a methodology called Test Vector Leakage Assessment (TVLA) that was initially presented by Goodwill et al. [31]. The TVLA methodology uses Welch's t-test [66] to detect statistical differences between sample distributions that are measured when the device processes different inputs. Two main test configurations are specified: the fixed vs. random configuration and the fixed vs. fixed configuration. The reason for calculating such differences is that a protected cipher implementation should not be emitting any information that would let an evaluator differentiate the data it processes. If the calculated difference is statistically insignificant, the device is regarded as side-channel free in the context that it was tested on. It has been demonstrated that the results of t-tests should not be misinterpreted as a single test that decides if a device is secure or not [64]. Specifically, the result only suggests that the t-test failed to find leakage for the specific fixed inputs used and number of traces collected from the device. For different fixed input values or for a greater number of traces significant leakage could be observable.

Welch's t-test defines a statistic called the t-value which is calculated from the means ($\overline{X}_1$ and $\overline{X}_2$) and variances (s_1^2 and s_2^2) of distributions of collected traces at a given sample point. The t-statistic follows a Student's t-distribution with v degrees of freedom. Given the number of samples in each distribution as n_1 and n_2, the t-value (t) and degree of freedom (v) are calculated as:

$$t = \frac{\overline{X}_1 - \overline{X}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} \quad \text{and} \quad v = \frac{\left(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right)^2}{\frac{\left(\frac{s_1^2}{n_1}\right)^2}{n_1 - 1} + \frac{\left(\frac{s_2^2}{n_2}\right)^2}{n_2 - 1}}.$$

The t-value tells us how significantly different the two distributions are. The hypothesis that these two distributions originate from the same population needs to be rejected with some given level of confidence to show that they are different. This process is known as hypothesis testing. Hypothesis testing is the scientific method of ruling out hypotheses by rejecting them based on significant evidence against them. The null hypothesis is the hypothesis that we assume to be correct by default. In TVLA, we assume that the device is not leaky until evidence, such as a significant t-value from the t-test rejects it in favour of the alternative hypothesis. The alternative hypothesis here states that the two sample distributions are statistically different, which implies that the considered device is leaky. The threshold value of 4.5 for significant leakage is chosen at a significance level (α) of 0.00001 under the assumptions that $s_1 \approx s_2$ and $n_1 \approx n_2$, such that the total number of traces ($n_1 + n_2$) is greater than 1,000 [57].

Using naive methods to compute t-values may result in numerical errors due to cancellation effects [12]. Schneider and Moradi [57] demonstrated computational improvements to overcome such issues. They also suggested online calculation of t-values in a single pass, speeding up the calculations compared to more naive methods. Another common concern with the evaluation of masked implementations is the typical t-test threshold of 4.5. This value assumes a single independent t-test. This threshold value is inadequate for large numbers of sample points, as the possibility of false positives (i.e. classification of leakage at sample points as significant when there is no actual leakage) increases due to the increased number of tests. Ding et al. [22] discussed this further and proposes a method to increase the t-test threshold according to the degree of freedom of the t-test [57] and number of samples.

2.3 Masking Techniques and Higher-Order Side-Channel Attacks

To protect ciphers against side-channel attacks, a technique called masking [13, 43] has been proposed. With masking, a sensitive intermediate value is split into multiple parts by using additional randomness. The additional random values are referred to as *masks* and the values that the original value is split into are referred to as *shares*. Depending on the order of masking, the number of shares increases. For example, in a dth order masking scheme there are $d + 1$ shares in use. Only when all of the shares are combined, the original value can be revealed.

Since masked implementations are secure against traditional attacks such as DPA and CPA, these attacks have been generalised to overcome masking by exploiting several leakage points simultaneously. Generally, a $(d + 1)$th order attack aims at breaking a dth order masked implementation. Such attacks first combine leakage occurring in $d+1$ intermediate operations and then a classical attack

such as CPA can be applied to recover the key. By increasing the number of shares, an implementer can increase the work that is required for an attack exponentially [13].

In particular, Ishai et al. [35] show that a masked implementation with $d + 1$ shares is secure against side-channel attacks in the d-probing model. The d-probing model considers an adversary that can only learn up to d intermediate values that are produced during the cryptographic computation. The model is usually considered a good approximation for modelling higher-order attacks.

Even though masking techniques can be theoretically secure against wide-range of side-channel attacks, many practical effects, such as glitches [14, 41] or transitional effects [3] that can void the countermeasure and still leak the secret information. In such cases, dth order implementations reveal their secret values at dth or lower order analysis. Renauld et al. [53] attribute this effect to breaching the Independent Leakage Assumption (ILA), which states that all related shares should be manipulated independently. Even though the ILA is assumed in theoretical cryptography, in reality this assumption does not hold due to the way that modern computers work. For example, to increase performance and reduce manufacturing costs, modern CPUs reuse many of their internal components without resetting or wiping them. Balasch et al. [3] demonstrated that transitional effects can be destructive to masked implementations as they halve the effective order of the analysis required when the leakage is modelled with a Hamming-distance leakage model. In the Hamming weight model, only a single intermediate value is considered as a sample at a sample point. In contrast, the Hamming distance model uses the bit difference between two intermediate values for a sample.

To measure the effectiveness of an implemented countermeasure such as masking, one needs to look into the leakage assessment of cryptographic devices.

2.4 Higher-Order Side-Channel Leakage Assessment

As discussed before, increasing the number of shares significantly increases the attack complexity, and information from multiple samples needs to be combined to reveal leakage of higher-order implementations.

In contrast to univariate analysis, where each sample point is analysed independently of other points, higher-order analysis takes into account the joint leakage of two or more sample points. This is similar to using multiple probes with respect to the model of Ishai et al. [35]. A combination function is typically used to combine mean-centered samples, and leakage assessment is then carried on the resulting combination. Following Prouff et al. [51], we choose the 'product of samples' combination function, In case of multivariate t-tests, the result of the combination is used as input to a first-order t-test and analysed similar to the univariate case [57].

Let us consider a set of n side-channel measurements T_i, $0 \leq i < n$, which are known as *traces*. Each trace contains m sample points denoted as $t_i^{(j)}$, for $0 \leq j < m$ with sample means denoted by $\mu^{(j)}$, Then the mean centered product of a given subset of sample points $\mathcal{J}$, is given by:

$$C_i = \prod_{j \in \mathcal{J}} \left(t_i^{(j)} - \mu^{(j)} \right). \tag{1}$$

When $|\mathcal{J}| = 2$ the combinations generated are called *bivariate* and when $|\mathcal{J}| = 3$ they are called *trivariate*.

Usually we need to consider all possible subsets $\mathcal{J}$ in a given trace $T_i = t_i^{(0)}, \ldots t_i^{(m-1)}$ to detect the leakage using t-test. Therefore, the complexity increase from using this approach higher-order leakage assessment is by a factor of $\binom{m}{|\mathcal{J}|}$, which is exponential for small values of $|\mathcal{J}|$.

2.5 Leakage Emulators and Automatic Countermeasures

Due to the high costs associated with evaluations that use real devices, implementers of cryptographic code are inclined to use emulators to determine leakage of a device [11]. The first use of such an emulator is evidenced within the PINPAS project [21], which had as the goal to emulate power analysis leakage in Java cards.

The most accurate method to emulate leakage is circuit-level emulation. While accurate, it is also very slow due to the very realistic reproduction of internal effects. Earlier generations that emulate leakage for software implementations used the cipher source code written in an high-level language [54, 65]. However, such implementations are inadequate for detecting leakage stemming from breaches of the ILA. In addition, compilation can also introduce breaches of ILA. Consequently, recent leakage emulators tend to use machine code as input rather than high-level source code [18, 42, 49]. Papagiannopoulos and Veshchikov [49] developed an automated methodology for detecting violations of the ILA in AVR assembly. They investigate the effects of the ILA violations on an AVR microcontroller, ATMega163. By enforcing the ILA, the authors produce a first-order secure S-box for the RECTANGLE [68] cipher.

With Coco [30], it is possible to formally verify a masked implementation down to the gate-level when the netlist of the CPU is available. A major difference between the construction of other leakage emulators and Coco is that Coco uses a software tool called Verilator to convert Verilog hardware descriptions of the CPU into C++. This enables the construction of a detailed emulator and offers fine-grained information about the execution. It collects power information for each gate and then uses a SAT-solver to find the leaky gates. While Coco finds the exact gates that are contributing to the leakage, it does not provide an automated fixing mechanism.

McCann et al. [42] demonstrated an emulator named Elmo that emulates leakage based on machine instructions. The emulation uses a statistical model that is profiled using real traces. This makes it specific for the device it was profiled on. Elmo currently supports ARM Cortex-M0 and ARM Cortex-M4 processors.

The recently introduced Rosita [61] aims to automate the process of producing masked first-order implementations. Rosita uses leakage information from an improved version of Elmo [42], which the authors call Elmo*, to emulate the power consumption of the target device running the software. It then uses TVLA [31] on the emulated traces to detect instructions that leak information. When leakage is detected, Rosita performs root cause analysis to identify the cause of the leakage. Specifically, it performs t-test analysis on emulated traces of each of the components of the Elmo* model, identifying a components that show evidence of leakage. Based on the root cause, Rosita applies rewrite rules, modifying the cipher

code with the aim of eliminating the leakage. The process repeats until either no more rules can apply or no leakage is evident.

Similarly, Gao et al. [26] have demonstrated an Instruction Set Extension (ISE) to RISC-V Instruction Set Architecture (ISA). The ISE guarantees that internal states that cause leakage are cleared acting as a barrier instruction when used in sensitive programs.

2.6 Testing for Statistical Equivalence of Distributions

In Section 3.4 we use a statistical equivalence test during root-cause analysis to determine which parts of the code contribute to the leakage; In this section, we describe the statistical method we use for equivalence testing.

The aim of statistical equivalence tests is to determine how probable it is that two sampled distributions originate from the same population. Observe that this is the opposite of what Welch's t-test offers. The null hypothesis of an equivalence test is that the two distributions are different and we expect to reject it in favour of the alternative hypothesis which states that the distributions are the same with a given significance level. One such equivalence test is the Two One Sided t-test (TOST) [50, 58]. As the name indicates, TOST uses two one-sided t-tests to test whether the two distributions are equivalent. TOST is a parameterised test that requires a lower bound and upper bound for the mean difference of the two distributions under test as parameters. Two individual t-tests determine whether the mean difference is lower than the upper bound and whether it is higher than the lower bound with a given level of significance (α). Passing both t-tests indicates that the mean difference is between the lower and upper bounds with the given significance level.

However, TOST in its original form has a limitation when it comes to the evaluation of the mean differences of two distributions: when these mean differences are close or equal to the boundary values, the TOST concludes that the distributions are not equivalent. This happens due to the t-value of the individual t-tests resulting in values closer to 0 when the mean differences are close to boundary values. In the paradigm where TOST is commonly used (e.g. in drug test trials), the boundaries are regarded as the worst values that the mean difference can get. However, in equivalence testing for engineering, we expect a test which accepts boundary values and also the values which are closer to the boundaries.

To mitigate this limitation, Pardo [50] proposed the following formulas that compute new boundaries ($\overline{X}_H$ and $\overline{X}_L$) given a target mean difference (μ), where s and n are standard deviation and cardinality of the mean differences distribution. t_α is the one sided t-test value at a significance value of α.

Selecting a critical region with α significance level such that $\overline{X}_H$ is higher than μ is given as

$$\overline{X}_H = \mu + t_\alpha \frac{s}{\sqrt{n}} \tag{2}$$

and such that $\overline{X}_L$ is lower than μ is given as

$$\overline{X}_L = \mu - t_\alpha \frac{s}{\sqrt{n}}. \tag{3}$$

Using the confidence interval of $\overline{X}_L$ and $\overline{X}_H$ instead of having arbitrarily defined values for mean difference boundaries overcomes the above-stated limitation.

3 ROSITA++ DESIGN

Past solutions that aim to automate leakage detection [21, 42, 49, 59, 65] and correction [61] focus on first-order leakage. As the security of cipher implementations can be increased by employing more shares in their masking schemes, there is a need for emulators and countermeasures that can work with multivariate leakage. In this section we describe how Rosita++, our solution for this need, works. We outline the main challenges in performing higher-order analysis and proceed to describe our approaches for addressing these challenges.

3.1 Challenges for Higher-Order Analysis

The core extension required for Rosita++ to support higher-order leakage detection and mitigation is support for multivariate analysis. Specifically, instead of looking for instructions that show indication of leakage, we need to look for combinations of instructions that *together* show indication of leakage.

Schneider and Moradi [57] suggest a methodology for performing multivariate analysis. Their approach is to generate artificial, multivariate traces from the original univariate traces. For that, the original traces are first preprocessed by calculating the average value for each sample point and subtracting the average from the corresponding point in each trace. As Equation 1 shows, Each point in an artificial trace represents a tuple of points in the original trace, where the value associated with the artificial point is the product of the values for the corresponding points in the original trace.

Our approach for performing higher-order analysis is to replace the use of TVLA in Rosita with the Schneider-Moradi methodology. However, while seemingly straightforward, the approach raises significant challenges.

Challenge C1: Statistical confidence with multivariate traces

The artificial d-variate traces have an artificial sample for each combination of d samples in the original traces. Consequently, the length of the multivariate traces grows exponentially with the length of the original traces. For traces of length n, the multivariate traces have a length of $\binom{n}{d}$ samples.

The de-facto standard statistical test used in TVLA is to reject the null hypothesis, i.e. report leakage, when the absolute value of the t-test is above a threshold of 4.5, achieving a significance level of 0.00001. This test, however, fails to account for the multiple comparisons performed in TVLA, where a statistical test is performed independently on each sample point. For a small number of points, the effect of multiple comparisons is negligible. When the trace length increases, multiple comparisons result in false positives, showing leakage where no leakage exists.

Challenge C2: Increased data size

Another issue with multivariate analysis is the increase in the volume of data that needs to be processed compared with first-order univariate analysis. Three factors contribute to this increase. First,

due to the effects of noise, the number of traces required for statistical analysis grows exponentially with the order of analysis [13]. Secondly, as discussed, the artificial multivariate traces are significantly longer than the original traces. Thirdly, to increase the statistical confidence while handling Challenge C1 without missing leakage we need to increase the number of traces we process.

Because Rosita++ repeatedly evaluates implementations, there is a need for efficient methods for handling the increased amount of data with minimal impact on analysis time.

Challenge C3: Multivariate root-cause analysis

The third challenge we face relates to performing the root-cause analysis. Rosita performs the analysis using a t-test on each of the Elmo* model components. Such an approach can detect univariate leakage. However, detecting multivariate leakage necessitates evaluating combinations of components. A brute-force approach that evaluates a t-test statistics on every combination of components is computationally expensive, particularly considering the increased number of traces, as described in Challenge C2. Thus, new techniques for root-cause analysis are required.

We now discuss how Rosita++ addresses these challenges.

3.2 Achieving Statistical Confidence

As discussed, Challenge C1 is that, due to the exponential increase in the number of sample point per trace, the t-test threshold of 4.5 is no longer appropriate. This mostly affects the traces collected from the physical experiment where we collect longer traces (10 times more samples) to reduce the effects of noise. To demonstrate the false positives we collect 500,000 bivariate traces of a three-share implementation of Xoodoo (further described in Section 4.3) running on a STM32F030 Discovery evaluation board, where all inputs are drawn uniformly at random. The experiment setup we used is described in Section 4.1. We then split these arbitrarily into two populations, and perform a bivariate t-test analysis, comparing these populations with a threshold of 4.5. As Figure 1 shows, despite the populations being sampled from the same distribution, several false positives are present.

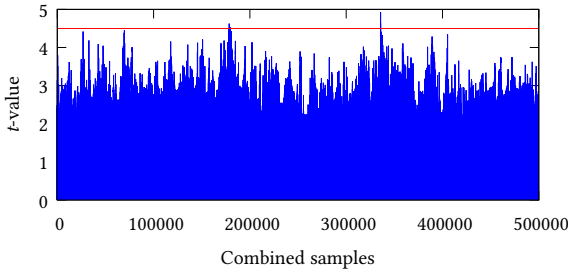


Figure 1: A t-test threshold value of 4.5 for a bivariate analysis with 1000 samples with all inputs being random.

For engineers, these false positives are typically of low impact. Experienced engineers can typically identify false positives, e.g. by observing the context. Alternatively, repeating the test can confirm true positives.

Automatic tools, such as Rosita++, do not have the experience or the insight, and must rely on statistical tools for handling false positives. If Rosita++ is used with long code segments these false positives will also be present in its leakage analysis. Therefore, in Rosita++, we adopt the approach of Ding et al. [22], who propose increasing the threshold to reduce the probability of false positives. Specifically, Ding et al. provide a formula to calculate the threshold given the number of samples and a desired significance level α. We apply the formula to the length of the bivariate trace aiming for a significance level of 0.00001. This ensures that the probability of a false positive error is less than .001%, which we consider negligible. For the traces in Figure 1 we would use a threshold of 6.71, which is clearly above the largest peak in the figure. Hence, at this threshold, the analysis does not indicate any leakage, which is expected considering that the two populations are drawn from the same distribution.

3.3 Handling Large Datasets

As mentioned, several aspects of multivariate analysis result in a significant increase in the size of data that Rosita++ needs to process. First, for given mean and variance, the Welch t-value grows linearly with the square root of the size of the population. Consequently, when increasing the threshold we need a quadratic increase in the number of traces to achieve the same detection sensitivity. Second, the length of the multivariate artificial traces is several orders of magnitude longer than the original univariate traces. Third, due to the effects of noise, detecting higher-order leakage is inherently harder then detecting first-order leakage. The combined effect of these changes is that the amount of data that Rosita++ needs to process is several orders of magnitude larger than that of Rosita. When evaluating the final version of code produced by Rosita++ on real hardware, the same issue gets even more apparent because we use longer traces as mentioned in Section 3.2.

While we are aware of works that have performed analyses at scales similar and even larger than our work [16, 17], we could not find public tools that perform such analyses, or even performance figures for the analysis. Free tools such as Jlsca[1], Scared[2], SCALib[3] seem to only offer limited capabilities. To address this challenge we developed analysis tools from the ground up. Our analysis tools avoid the overhead of storing the artificial traces (i.e. multivariate combinations) by calculating them on the fly. The tools are multi-threaded, allowing a significant speed-up, and the data is divided point-wise between the threads, so that each thread only accesses a limited subset of the original traces' samples.

We acknowledge that the approach is fairly straightforward, but we believe that the contribution is important for practical future research into bivariate analysis.

3.4 Multivariate Root-Cause Analysis

The third challenge for Rosita++ is performing root-cause analysis on multivariate traces. The Elmo* linear regression model consists of 28 term components, each modelling a different micro-architectural effect. When Rosita performs univariate root-cause

[1] https://github.com/Riscure/Jlsca
[2] https://gitlab.com/eshard/scared
[3] https://github.com/simple-crypto/SCALib

analysis, it calculates the Welch t-value for each component separately, where the leaky components are identified by observing significant t-values.

While this approach works well for univariate leakage detection, adapting it to multivariate leakage is not trivial. The main reason is that, in multivariate analysis, there is no single cause for leakage.

As shown by Equation 1, a multivariate sample point is a combination of many samples in the original trace. In Elmo*, each of the original samples is calculated from the sum of 28 model components. Searching for a combination of d samples using a method similar to the one used for univariate evaluation would require evaluating 28^d combinations. Even for the bivariate case of $d = 2$, the process is very inefficient with the large number of traces that need to be processed due to increase of order [13].

To avoid searching the whole space of pairs of model components, Rosita++ uses two new methods for finding the components that contribute to the leak. The *component elimination* method tests whether removing a model component removes the leakage. While efficient, this approach may sometimes fail. In the case of such a failure, Rosita++ reverts to a *Monte-Carlo method*, which tests random combinations of components looking for evidence of component leakage. We refrain from using the Monte-Carlo method by default due to its inefficiency and the instability inherent in a randomised process. We now describe these two methods in detail.

Component Elimination The basic idea behind the component elimination method is to identify components that contribute to the leakage by removing one component at a time from the multivariate sample combination function (which is shown in Equation 1); we then evaluate the combination with removed component for absence of leakage. If the removal of a component leads to the absence of leakage at a previously leaky point, this means that the removed component contributed to the leakage. When this process ends, Rosita++ has a set of components that contribute to the leakage. Rosita++ can now apply fixes using the approach of Rosita.

More specifically, component elimination consists of the following steps. First, each component value of Elmo* is recorded with the component index, sample index, and the trace index. There exists 28 different components in Elmo*, the sample index is the array index of the instruction when the emulated code segment is unrolled into individual instructions. The trace index is a number identifying each run of the fixed vs. random test. All of these values are stored in a 3D matrix that is denoted by $\boldsymbol{L}$.

Second, the multivariate leaky points for the implementation are found by running the t-test on the final power value of Elmo* generated while running the code segment in a fixed vs. random input configuration.

Finally, Algorithm 3.1 is run to find the leaky components at the leaky points recognised in the previous step. The two utility functions that are used by Algorithm 3.1 are Normalised Product of Samples (NPS) and NotLeaky. The first function, NPS returns the combined traces for a given set of sample points and a given set of components. In a nutshell, NPS returns the results of Equation 1 for an arbitrary set of components and an arbitrary set of sample points. As the name suggests, the NotLeaky function differentiates between trace sets which are significantly similar and ones which are not. NotLeaky requires an additional run of the code segment with all random input configuration instead of a fixed vs. random input configuration. This run collects information required to calculate the mean differences and variances required by TOST.

Algorithm 3.1 Find Leaky Components

$\boldsymbol{L}$: A 3D matrix with component values for Elmo* organised by trace index, sample index and component index.
$\mathcal{S}$: Set of d sample points that participate in the leakage.
C: Set of all components that are in Elmo*.
NPS($\boldsymbol{L}, \mathcal{S}, C$): Normalised Product of Samples. Returns the normalised product of the power samples from reduced models which only contain a given set of components (C) at some given sample points ($\mathcal{S}$) from a 3D matrix that holds component samples ($\boldsymbol{L}$).
NotLeaky(Y): Determine the absence of leakage using TOST.
$\odot$: Elementwise multiplication operator.

```
1:  function FLC(L, S, C)
2:      r ← {}
3:      for s ∈ S do
4:          x ← NPS(L, S \ s, C)
5:          for t ∈ C do
6:              u ← C \ t
7:              y ← NPS(L, s, u)
8:              z = y ⊙ x
9:              if NotLeaky(z) then
10:                 r ← r ∪ {(s, t)}
11:             end if
12:         end for
13:     end for
14:     return r
15: end function
```

While the component elimination method is efficient, it may sometimes fail. For example, if multiple model components leak the same share, removing any one of these components will not eliminate the leak. Similarly, TOST may fail to demonstrate the equivalence of the two distribution even when removing a model component eliminates the leak.

The Monte-Carlo Method In the Monte-Carlo approach we run a preset number of random experiments where, in each experiment, we select a random subset of the model components, and perform the t-test on hypothetical power traces with only the selected components. For each component, we keep track of the number of random experiments it participates in and how many of those experiments indicate significant leakage. After we repeat the experiment a preset number of times, we arrive at a subset of components that contribute significantly more to the leakage.

To select the preset number of random experiments we first performed an analysis for a code segment from Xoodoo cipher (shown in Listing 4) only by using Monte Carlo method to detect and remove leakage. We use the chosen preset number in all our subsequent experiments. We first gathered 100,000 traces from this cipher implementation and performed the initial leakage analysis. Initially, it had 45 total leakage points. Figure 2 shows the reduction of remaining leaky points as we gradually increase the number of Monte Carlo experiments starting from 10. Figure 2 shows that increasing number of experiments improves detection of root causes, but after about 30 experiments the reduction of leakage is nearly

constant[4]. Therefore, we decided to settle at using 50, slightly more than 30 for the sake of certainty, as the preset experiment count for our experiments.

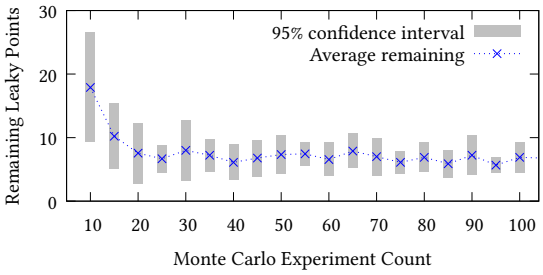


Figure 2: Effectiveness in removing leakage of Monte Carlo method for increasing number of experiments

Observe that both component elimitation and the Monte-Carlo method are independent of the security order. We evaluate both methods in the second and third order in Section 4.

In our experiments we find that we need to fallback to the Monte-Carlo method in four out of 16 root-cause detections in Xoodoo, in 70 out of 262 in PRESENT, and in one out of 15 in the Boolean-to-arithmetic conversion algorithm.

3.5 Code Rewrite

After finding the root cause of the leakage, ROSITA++ selects the code-rewrite rule that best match the detected root cause using the code-rewrite engine of ROSITA.

In a nutshell, ROSITA reserves the register `r7`, which it initialises with a random value. When an unintended interaction is detected, the code rewrite engine inserts instructions that use `r7` to eliminate the interaction. For example, when the detected interaction is caused by a pipeline register that is updated by two consecutive instructions, ROSITA inserts the instruction `mov r7, r7`, to buffer between the interacting instructions. Similarly, when the leakage is from an interaction with the memory subsystem, ROSITA inserts the pair of instructions `push {r7}` followed by `pop {r7}`, which wipes the internal state of the memory pipeline. Many other fine-grained fixes are used to erase internal state set by other instructions (i.e. instructions related to ALU's operations).

Observe that we use the same code rewriting engine that was initially designed for fixing univariate leakage. We find that it is usable as is to also fix multivariate leakage becasue the output of our root-cause detection algorithm matches the format of the original ROSITA output. The downside of reusing the code rewriting engine is that we may miss opportunities for addressing multiple leaks with a single fix. We leave optimising the code-rewrite engine to future work.

4 EVALUATION

In this section we evaluate the effectiveness of ROSITA++ in eliminating leakage. First, we describe our physical experiment setup. Second, we present toy Boolean masked examples of second and third order where ROSITA++ fixes a single leaky point. Third, we present the evaluation results of ROSITA++'s emulation process and root cause detection. Finally, we demonstrate effectiveness of ROSITA++ on practical code segments implemented with 3-shares. Due to practical reasons, we limit the discussion to second and third order. We note that ROSITA++ can detect and apply fixes at any order.

[4]We have noticed no further leakage reduction even for 1000 experiments.

4.1 Experimental Setup

Our experimental hardware setup is depicted in Figure 3. For evaluation we use the STM32F030 Discovery evaluation board by ST Microelectronics, which features an ARM Cortex-M0 based on STM32F030R8T6 System-on-Chip (SoC), running at 8 MHz. To avoid switching noise, we power the evaluation board with batteries instead of a mains-connected power supply.

To measure the power consumption of the evaluation board, we introduce a shunt resistor across one of its power terminals. We measure the voltage drop across the shunt resistor with a PicoScope 6404D oscilloscope, configured at a sampling rate of 78.125 MHz (12.8 ns sample interval) which translates to roughly 9.77 samples per clock cycle. The voltage is measured with a PicoTechnology TA 046 differential probe connected to the oscilloscope via a Langer PA 303 preamplifier.

We use two of the I/O pins of the board to trigger the acquisition. One indicates trace start and the other indicates the end. To increase signal stability, interrupts are disabled for the duration of each trace, using `__disable_irq()` before the start trigger and `__enable_irq()` after the end trigger.

To orchestrate the experiment, we used a PC with a serial connection to our device under test. The PC controls all aspects of the experiment, and in particular it selects the type of the experiment (i.e. fixed vs. random) and the randomness used. The tested device is oblivious to the type of experiment and uses the inputs received from the PC. To reduce the communication overhead the PC uses bulk transfer to send the inputs for multiple successive experiments, which the device executes sequentially.

We post-processed the traces to improve signal quality. Firstly, we aligned the traces statically using a correlation-based alignment, reducing sample drift. We then used a highpass filter to remove frequencies below 400 KHz. Before filtering, the signal was zero padded to avoid introduction of transients [63].

4.2 Evaluation of second and third-order Boolean masked toy example

```
1    ; nop padding
2  ldrb r4, [r1]
3  push {r7}
4  pop {r7}
5    ; nop padding
6  ldrb r5, [r2]
7  ldrb r6, [r3]
8    ; nop padding
```

Listing 1: A Toy Example (second order)

Before we evaluate ROSITA++ on real-world software examples, we demonstrate its effectiveness on a toy example, shown in Listing 1. The code presents a typical operation in second-order protected

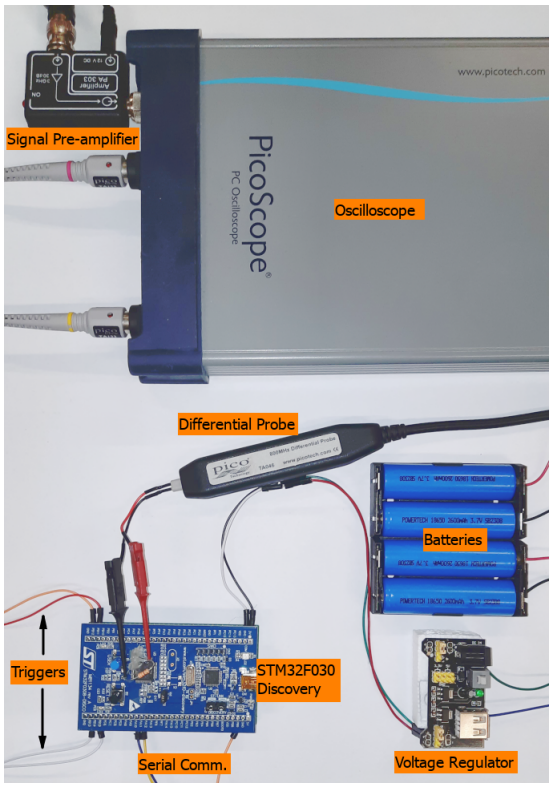


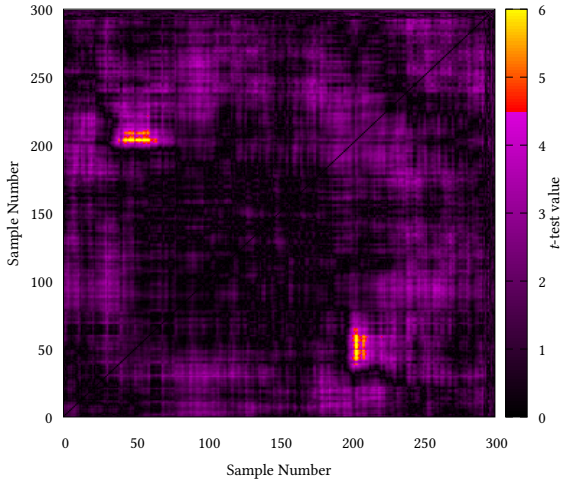
Figure 3: Measurement setup

implementation that uses Boolean masking. Specifically, it assumes that registers `r1`, `r2`, and `r3` contain the addresses of three shares that represent a secret value. The code uses three `ldrb` instructions to load the masked value into three registers, `r4`, `r5`, and `r6`. We note that the code is nominally second-order secure, because all instructions process at most one share of the secret. However, as we see below, unintended interactions between the load instructions at Lines 6 and 7 result in second-order leakage.

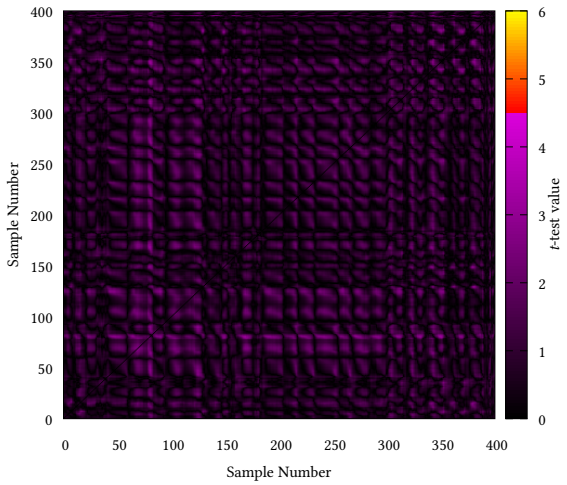
To avoid first-order leakage through a combination of the three load instructions, we separated the first load (Line 2) from the rest of the code. We added the `push` and `pop` instructions in Lines 3 and 4 to remove interactions between the first load and the following two loads. (See Shelton et al. [61] for details.) We further added sequences of nine `nop` instructions (concretely, `mov r7, r7`) to avoid unintended interaction through the processor's pipeline and to achieve a clear temporal separation between the loads. Last, we add short sequences of `nop` instructions around the code to create a temporal separation between the measured code and the triggers.

In Figure 4a we see the results of bivariate leakage analysis on two million traces collected using our experimental setup. The figure is a heatmap, where the X and Y axes indicate the samples that are combines to create the artificial bivariate sample. The colour of each combined sample indicates the magnitude of the fixed vs. random t-test analysis for the combined sample. The figure is symmetric across its main diagonal.

Examining the figure we find that there are two regions that show a t-test value above our threshold of 4.5. These occur at the



(a) Before applying code fixes. Leakage is visible around coordinates (50,200) and (200,50). t-value peak: 6.86.

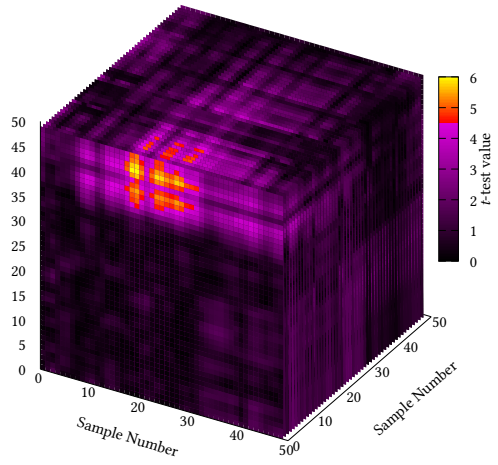


(b) After applying code fixes no leakage is present (t-value peak: 3.15)

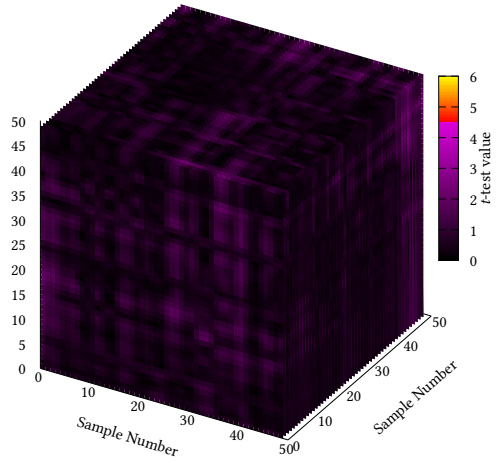
Figure 4: Evaluating a toy example.

combinations of samples around 50, which corresponds to Line 2 of Listing 1, and sample 200, which corresponds to Line 7. Running Rosita++ also shows that the combination of Line 2 and Line 7 leaks. Root-cause analysis shows that Lines 6 and 7 interact both through the processor pipeline and through the memory bus.

To fix the leakage, Rosita++ first inserts a `mov r7, r7` instruction between Line 6 and Line 7, and repeats the analysis to check that the leakage has been eliminated. Finding that there is still leakage through the memory bus, Rosita++ further adds a



(a) Before applying code fixes. t-value peak of 6.85 at (42,28,7)



(b) After applying code fixes. t-value peak of 3.31 at (19,0,6)

Figure 5: Evaluating a toy example.

combination of `push` and `pop` instructions, producing the code in Listing 2. Rosita++ required 200,000 emulated traces to apply fixes for this implementation. Running the bivariate analysis on the code shows no evidence of second-order leakage, as shown in Figure 4b.

```
    ; nop padding
mov r7, r7
ldrb r4, [r1]
push {r7}
pop {r7}
  ; nop padding
ldrb r5, [r2]
mov r7, r7
push {r7}
pop{r7}
ldrb r6, [r3]
  ; nop padding
```

Listing 2: Fixed Toy Example

We extended the second order Boolean masked implementation shown in Listing 1 to the third order by introducing another share to it. The code for this implementation is shown in Listing 3. Similar to the second order version, we intentionally design the example to leak operand information from the last two `ldr` instructions. This implementation was fixed by Rosita++ with two million emulated traces. To detect leakage in the physical device we had to collect 30 million traces. Figure 5a shows the detected leakage from a 3rd order t-test. A first order t-test was run on the combined traces that were combined using Equation 1 with a window of 50 samples. In contrast, Figure 5b shows the results of the t-test that was run on 30 million side-channel traces taken from the physical experiment after applying Rosita++'s fixes.

```
    ldr r3, [r1,#0]
    push {r7}
    pop {r7}
    ; nop padding
    ldr r4, [r1,#4]
    push {r7}
    pop {r7}
    ; nop padding
    ldr r5, [r1,#16]
    ldr r6, [r1,#20]
    ; nop padding
```

Listing 3: A Toy Example (third order)

Comparing Emulated and Real Traces: To better understand the relationship between emulated and real traces, we compared the leakage observed in the traces in terms of signal-to-noise ratio (SNR). For this experiment we used 20,000 random input traces coming from the emulation and the real experiments using the code segment shown in Listing 1; we chose this number because it is sufficient to find leakage in the emulated traces using TVLA. We computed SNR for the leaking values that need to be combined for bivariate analysis: hamming weight (HW) of 4 bytes of $r1$ and HW of 4 bytes of $r2 \oplus r3$. For the real experiments these values were between 0.041 and 0.063 for the bytes of $r1$ and between 0.012 and 0.014 the bytes of $r2 \oplus r3$. We could not compute the SNR directly for the emulated traces since the emulation is deterministic and therefore, noise-free. We added a sufficient amount of noise to generate similar SNR to the real experiments. We used Gaussian Noise with means 0 and standard deviation of 0.25% of the signal amplitude for the bytes of $r1$ and 0.1% for the bytes of $r2 \oplus r3$. We do not know from where this leakage difference is exactly coming from, but we suspect that we simply found a slight difference between the emulated and the real measurements.

We conclude that if we introduce between 0.1% and 0.25% ratio of noise to the emulated traces then we obtain a similar SNR to the real traces. Moreover, we can use 25 times less traces than in the

$$a_{0,0} = a_{0,0} \oplus (\neg a_{1,0} \wedge a_{2,0}) \oplus (a_{1,0} \wedge b_{2,0}) \oplus (b_{1,0} \wedge a_{2,0})$$
$$b_{0,0} = b_{0,0} \oplus (\neg b_{1,0} \wedge b_{2,0}) \oplus (b_{1,0} \wedge c_{2,0}) \oplus (c_{1,0} \wedge b_{2,0})$$
$$c_{0,0} = b_{0,0} \oplus (\neg c_{1,0} \wedge c_{2,0}) \oplus (c_{1,0} \wedge a_{2,0}) \oplus (a_{1,0} \wedge c_{2,0})$$

Listing 4: Xoodoo code segment under test

real experiment to detect leakage using TVLA, since we can detect leakage using emulation with 20,000 traces and we need 500,000 in the real experiments.

4.3 Evaluated Cryptographic Implementations

We now turn our attention to more realistic examples. Before performing the evaluation we use Rosita to detect and eliminate any first-order leakage from the code. We further perform a first-order fixed vs. random TVLA with 2,000,000 traces on the real hardware to verify that no first-order leakage is detected. For the evaluation, we use Rosita++ to detect and correct second-order leakage for 500,000 simulated traces. We then collect 2,000,000 power traces from each of the original and the fixed software, and perform bivariate second-order analysis to identify any leakage. We evaluate two cryptographic implementations and one cryptographic primitive, which we describe below.

Xoodoo Xoodoo was proposed by Daemen et al. [19] and a reference implementation is available from Bertoni et al. [7]. We converted this code to a three-share implementation based on the Threshold Implementation (TI) approach [48]. TI schemes were proposed to prevent the leakage from "glitches" that can occur in hardware implementations. The concept is accomplishing the goal of masking through a number of shares with some additional properties. Specifically, the non-completeness property of TI enforces that no operation should involve more than two shares.

Xoodoo's state is 48 bytes in length. The state is divided into three equal blocks called *planes*, each consisting of four 32-bit words. $x_{i,j}$ denotes the j^{th} 32-bit word of the i^{th} plane of share x, where $x \in \{a, b, c\}$. Listing 4 shows the algorithm segment that we analyse, which forms part of the start of the Xoodoo χ function. Our initial C implementation demonstrated first-order leakage caused by the optimiser merging shares. We therefore manually implemented the code in assembly, ensuring that shares are not merged.

Present Present is a block cipher based on a substitution permutation network, which was proposed by Bogdanov et al. in [9]. It has a block size of 64-bit and the key can be 80 or 128 bits long. The non-linear layer is based on a single 4-bit S-box facilitating lightweight hardware implementations.

We implemented present with side-channel protection in software based on TI with three shares, as described by Sasdrich et al. [56, Alg. 3.2]. Thus, at least in theory, the implementation should not leak in the first order. We used the code shown in Listing 5 that implements a part of the present S-box, involving 3 shares x^1, x^2, x^3 and the lookup table T. The table is an 8-bit to 4-bit lookup table where the inputs are two 4-bit nibbles. Each table lookup used to compute t^i is repeated 16 times to cover the complete 64-bit shares.

Observe that threshold implementations with three shares provides provable first-order security, but only limited protection against the second-order attacks [48]. Therefore, we can expect

$$t^3 = \mathtt{T}(x^1, x^2)$$
$$t^2 = \mathtt{T}(x^3, x^1)$$
$$t^1 = \mathtt{T}(x^2, x^3)$$

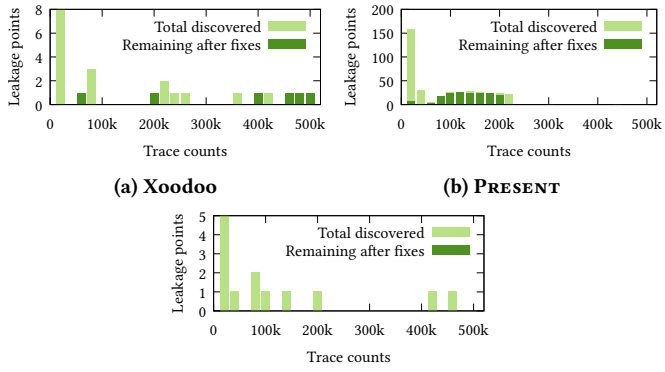
Listing 5: Present code segment under test

that diminished second-order leakage may occur for both Xoodoo and present implementations.

Second-order Boolean-to-arithmetic masking Boolean-to-arithmetic masking [32] is a cryptographic building block that converts a Boolean mask to an arithmetic mask. It is often used in side-channel resistant implementations of cryptographic algorithms that mix Boolean and arithmetic operations (for example, ChaCha [5]). We implement and evaluate the second-order Boolean-to-arithmetic masking of Hutter and Tunstall [33, Alg. 2].

In our evaluation this procedure takes as input boolean shares $x' = x \oplus r_1 \oplus r_2$, where x, r_1 and r_2 are random in $\mathbb{Z}_{2^{32}}$. For side-channel protection, the procedure uses three additional masks γ_1, γ_2, and α also random in $\mathbb{Z}_{2^{32}}$. It computes $x'' = x + s_1 + s_2$, where x'', s_1, and s_2 are the output arithmetic shares. This implementation is proven to be second-order secure in [33] and therefore, we do not expect to see leakage in an implementation protected with Rosita++.

4.4 Emulation results



(a) Xoodoo **(b) Present**

(c) Boolean-to-arithmetic

Figure 6: Discovered and remaining leakage points when fixed code from previous iteration is used as input to the next iteration of Rosita++

We used Rosita++ to fix the leakages that were discovered in the code segments introduced above in Section 4.3. Specifically, we focus on leakage discovered by a bivariate fixed vs. random t-test.

To analyse the relationship between the number of traces and leakage discovery, we ran Rosita++ on the unprotected ciphers, varying the number of traces from 20,000 to 500,000 at steps of 20,000. In each iteration we used the output of the previous iteration as the input. Each iteration performed emulation and root-cause detection. The emulation results are shown in Figure 6. This proves to be more efficient than running Rosita++ a single time with a large number emulation traces. The reason for the efficiency of the iterations based method with gradually increasing trace counts is that leakage is fixed as it is detected so that large numbers of traces

Implementation	Emulation time	Root Cause Det. time
Xoodoo	1:35:41	3:12
PRESENT	1:55:19	24:46
Boolean to arithmetic	1:08:19	1:07

Table 1: Time taken for emulation and root-cause detection

Implementation	Unprotected size (cycles)	Protected size (cycles)	Increase
Xoodoo	56	76	36%
PRESENT	114	330	189%
Boolean to arithmetic	75	97	29%

Table 2: Performance overhead of fixes

Trace set	Samples	Wall Clock Time
Xoodoo unprotected	1000	4:51
Xoodoo protected	1400	33:50
Present unprotected	1400	28:31
Present protected	3500	7:02:00
Boolean to arithmetic unprotected	1000	4:18
Boolean to arithmetic protected	1200	8:51

Table 3: Bivariate analysis time

are not required for fixing all the leakage points that are detected. Table 1 shows the emulation and root cause detection time when fixed code is used from the previous iteration. Table 2 shows the performance overhead of the code fixes.

We observe that after emulation 500,000 traces for the fixed vs. random t-test, there was only one remaining leakage in the Xoodoo masked implementation. PRESENT and Boolean-to-arithmetic implementations did not have any remaining leakage points. However, when running the physical experiments we observed that the remaining leakage in Xoodoo was not significant.

```
1  ldrb  r2, [r4, #16]
2  lsls  r1, r1, #4
3  adds  r1, r3, r1
4  ldrb  r0, [r1, r2]
```

Listing 6: Leaky code segment of fixed PRESENT

4.5 Physical experiment results

Figure 7 compares the t-test values of side-channel traces for the three ciphers before and after ROSITA++, as measured on the physical device. The top row (Figures 7a to 7c) show the leakage of the original implementations, whereas the bottom row (Figures 7d to 7f) shows the leakage after applying ROSITA++. The three implementations were protected using 500,000 emulated traces. Collecting the traces took around 8 hours for PRESENT and for Boolean-to-arithmetic, and around 9:30 hours for Xoodoo, which requires significantly more mask bytes, slowing down the communication with the PC.

As the figures show, for Xoodoo and the Boolean-to-arithmetic masking conversion, ROSITA++ completely eliminates leakage. However, for PRESENT some leakage is not fixed. Further analysis shows that this leakage is caused by interactions through the address bus. Listing 6 shows the first leaky segment of the code corresponding to samples (700, 440) in Figure 7e. We confirmed this leakage through correlation based testing against actual share values and their combinations. The registers used for addressing in the `ldrb` instruction at Line 4 carry one share each. Our investigation showed that sample 440 originates from this point. Additionally, the missing share is provided by the instruction that corresponds to sample 700. Both points show high correlation to the share values. Therefore, this leakage becomes observable as second-order leakage. We confirmed this leakage pattern by reproducing the same effect in a separate fixed vs. random experiment which has only two shares that is used in an `ldrb` instruction for addresses. It showed significant first order leakage at 200,000 traces.

Because our tooling does not detect address leakage, this code cannot be currently corrected. Moreover, we suspect that that the leakage might be present here due to the used threshold implementation algorithm and therefore, solving it is out of the scope of this work.

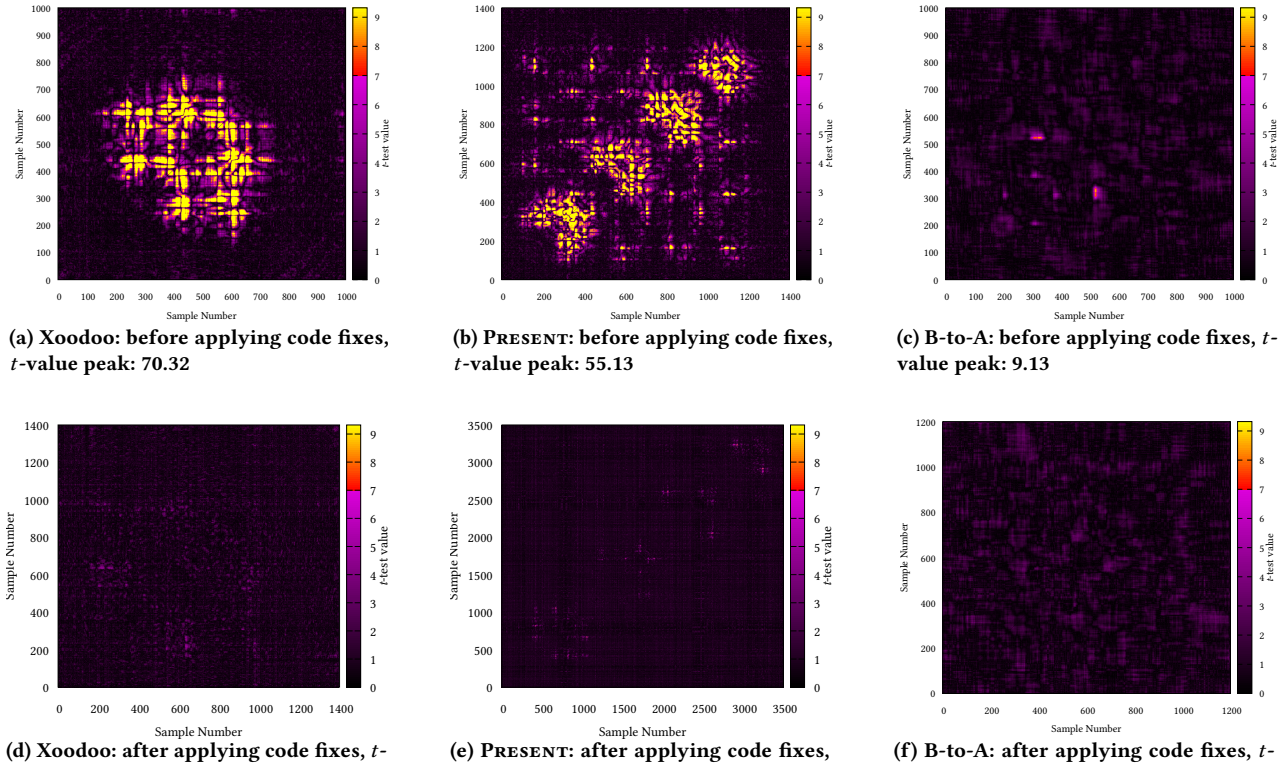
4.6 Tools for Leakage Analysis

We now present the performance of our second-order analysis tools. We run the tools on a desktop computer, featuring an Intel Core i9-10900K CPU and 32 GB of memory. We spawn 10 threads and perform bivariate analysis of four cryptographic implementations. For each implementation we use our measurement setup to collect 2M traces from the real experiments, which we analyse to draw the heatmaps shown in Figure 7. The results are shown in Table 3. The number of threads used can be changed to fit the underlying hardware, the thread count is dependent on the equal sized splits that are done along the sample axis. It is given by $S(S+1) \div 2$ where S is the number of equal sized splits. For our runs, S was set at 4. Without parallelisation the run time will be 8 times slower if run in a single thread as 4 out of 10 of the threads do half of the work.

5 CONCLUSIONS

Since the introduction of side-channel attacks, implementation security of embedded devices has been under the immense scrutiny and constant threat of being exploited. Even with theoretically sound measures such as masking, the devices tend to exhibit some leakages in practice due to unintended interactions in hardware. Mostly manual evaluation involving a tedious decision process and applying fixes to such "leaky" implementations have since been adopted. Some automatic countermeasures have also been developed, but all of them target univariate leakage.

In this work, we set out to automate the detection and application of fixes for high order secured implementations through multivariate analysis. We have demonstrated that it is possible to fix almost all detected leakage for three second-order masked implementations using our root cause analysis. Furthermore, we have shown practically that our methodology also is applicable for the third order analysis. It is a significant improvement over previous automatic countermeasure application methods due to its simplicity.



(a) Xoodoo: before applying code fixes, t-value peak: 70.32

(b) PRESENT: before applying code fixes, t-value peak: 55.13

(c) B-to-A: before applying code fixes, t-value peak: 9.13

(d) Xoodoo: after applying code fixes, t-value peak: 6.44

(e) PRESENT: after applying code fixes, t-value peak: 12.38

(f) B-to-A: after applying code fixes, t-value peak: 3.91

Figure 7: Evaluation of three cryptographic primitives (B-to-A stands for Bollean to arithmetic)

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their insightful comments and recommendations.

This work was supported by ARC Discovery Early Career Researcher Award number DE200101577; ARC Discovery Projects numbers DP200102364 and DP210102670; the Blavatnik ICRC at Tel-Aviv University; European Commission through the ERC Starting Grant 805031 (EPOQUE) of Peter Schwabe; and gifts from Facebook, Google and Intel.

Parts of this work were carried out while Yuval Yarom was affiliated with CSIRO's Data61.

REFERENCES

[1] Jean-Philippe Aumasson, L. Henzen, W. Meier, and R. Phan. 2009. SHA-3 proposal BLAKE.

[2] Aydin Aysu, Youssef Tobah, Mohit Tiwari, Andreas Gerstlauer, and Michael Orshansky. 2018. Horizontal side-channel vulnerabilities of post-quantum key exchange protocols. In *HOST*. 81–88.

[3] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. 2015. On the Cost of Lazy Engineering for Masked Software Implementations. In *CARDIS*. 64–81.

[4] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. 2019. CSI NN: Reverse Engineering of Neural Network Architectures Through Electromagnetic Side Channel. In *USENIX Security Symposium*. 515–532.

[5] Daniel Bernstein. 2008. ChaCha, a variant of Salsa20.

[6] Daniel J Bernstein. 2005. Cache-timing attacks on AES. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[7] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. [n. d.]. The eXtended Keccak Code Package (XKCP). https://github.com/XKCP/XKCP

[8] Guido Bertoni, Vittorio Zaccaria, Luca Breveglieri, Matteo Monchiero, and Gianluca Palermo. 2005. AES Power Attack Based on Induced Cache Miss and Countermeasure. In *ITCC*. 586–591.

[9] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. 2007. PRESENT: An Ultra-Lightweight Block Cipher. In *CHES*. 450–466.

[10] Eric Brier, Christophe Clavier, and Francis Olivier. 2004. Correlation Power Analysis with a Leakage Model. In *CHES*. 16–29.

[11] Ileana Buhan, Lejla Batina, Yuval Yarom, and Patrick Schaumont. 2021. SoK: Design Tools for Side-Channel-Aware Implementations. *IACR Cryptol. ePrint Arch.* 2021 (2021), 497.

[12] Tony F. Chan, Gene H. Golub, and Randall J. Leveque. 1983. Algorithms for Computing the Sample Variance: Analysis and Recommendations. *The American Statistician* 37, 3 (1983), 242–247. https://doi.org/10.1080/00031305.1983.10483115 arXiv:https://www.tandfonline.com/doi/pdf/10.1080/00031305.1983.10483115

[13] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. 1999. Towards Sound Approaches to Counteract Power-Analysis Attacks. In *CRYPTO*. 398–412.

[14] Zhimin Chen, Syed Haider, and Patrick Schaumont. 2009. Side-Channel Leakage in Masked Circuits Caused by Higher-Order Circuit Effects. In *ISA*. 327–336.

[15] Zhimin Chen and Yujie Zhou. 2006. Dual-Rail Random Switching Logic: A Countermeasure to Reduce Side Channel Leakage. In *CHES*. 242–254.

[16] Thomas De Cnudde, Begül Bilgin, Oscar Reparaz, Ventzislav Nikov, and Svetla Nikova. 2015. Higher-Order Threshold Implementation of the AES S-Box. In *CARDIS*. 259–272.

[17] Thomas De Cnudde, Oscar Reparaz, Begül Bilgin, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. 2016. Masking AES With d+1 Shares in Hardware. In *TIS@CCS*. 43.

[18] Yann Le Corre, Johann Großschädl, and Daniel Dinu. 2018. Micro-architectural Power Simulator for Leakage Assessment of Cryptographic Software on ARM Cortex-M3 Processors. In *COSADE (Lecture Notes in Computer Science, Vol. 10815)*. Springer, 82–98.

[19] Joan Daemen, Seth Hoffert, Gilles Van Assche, and Ronny Van Keer. 2018. The design of Xoodoo and Xoofff. *IACR Trans. Symmetric Cryptol.* 2018, 4 (2018), 1–38.

[20] Joan Daemen, Seth Hoffert, Gilles Van Assche, and Ronny Van Keer. 2018. Xoodoo cookbook. *IACR Cryptol. ePrint Arch.* 2018 (2018), 767.

[21] Jerry den Hartog, Jan Verschuren, Erik P. de Vink, Jaap de Vos, and W. Wiersma. 2003. PINPAS: A Tool for Power Analysis of Smartcards. In *IFIP SEC*. 453–457.

[22] A. Adam Ding, Liwei Zhang, François Durvaux, François-Xavier Standaert, and Yunsi Fei. 2017. Towards Sound and Optimal Leakage Detection Procedure. In *CARDIS*. 105–122.

[23] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. 2010. The Skein Hash Function Family. https://www.schneier.com/academic/skein/

[24] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. 2001. Electromagnetic Analysis: Concrete Results. In *CHES*. 251–261.

[25] Si Gao, Ben Marshall, Dan Page, and Elisabeth Oswald. 2020. Share-slicing: Friend or Foe? *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020, 1 (2020), 152–174.

[26] Si Gao, Ben Marshall, Dan Page, and Thinh Hung Pham. 2020. FENL: an ISE to mitigate analogue micro-architectural leakage. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020, 2 (2020), 73–98.

[27] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptogr. Eng.* 8, 1 (2018), 1–27.

[28] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. 2016. ECDSA Key Extraction from Mobile Devices via Nonintrusive Physical Side Channels. In *CCS*. 1626–1638.

[29] Daniel Genkin, Adi Shamir, and Eran Tromer. 2014. RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis. In *CRYPTO (1)*. 444–461.

[30] Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. 2020. Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs. , 1294 pages.

[31] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. 2011. A Testing Methodology for Side-Channel Resistance Validation. (2011).

[32] Louis Goubin. 2001. A Sound Method for Switching between Boolean and Arithmetic Masking. In *CHES*. 3–15.

[33] Michael Hutter and Michael Tunstall. 2019. Constant-time higher-order Boolean-to-arithmetic masking. *Journal of Cryptographic Engineering* 9 (06 2019). https://doi.org/10.1007/s13389-018-0191-z

[34] International Organization for Standardization. 2016. Testing methods for the mitigation of non-invasive attack classes against cryptographic modules. International Standard ISO/IEC 17825:2016(E).

[35] Yuval Ishai, Amit Sahai, and David A. Wagner. 2003. Private Circuits: Securing Hardware against Probing Attacks. In *CRYPTO*. 463–481.

[36] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*. 104–113.

[37] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential Power Analysis. In *CRYPTO*. 388–397.

[38] Juliane Krämer, Dmitry Nedospasov, Alexander Schlösser, and Jean-Pierre Seifert. 2013. Differential Photonic Emission Analysis. In *COSADE*. 1–16.

[39] Xuejia Lai and James L. Massey. 1991. A Proposal for a New Block Encryption Standard. In *Eurocrypt*. 389–404.

[40] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. 2021. A Survey of Microarchitectural Side-channel Vulnerabilities, Attacks and Defenses in Cryptography. *CoRR* abs/2103.14244 (2021).

[41] Stefan Mangard, Thomas Popp, and Berndt M. Gammel. 2005. Side-Channel Leakage of Masked CMOS Gates. In *CT-RSA*. 351–365.

[42] David McCann, Elisabeth Oswald, and Carolyn Whitnall. 2017. Towards Practical Tools for Side Channel Aware Software Engineering: 'Grey Box' Modelling for Instruction Leakages. In *USENIX Security Symposium*. 199–216.

[43] Thomas S. Messerges. 2000. *Power Analysis Attacks and Countermeasures for Cryptographic Algorithms*. Ph. D. Dissertation. University of Illinois at Chicago, USA.

[44] Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan. 1999. Power Analysis Attacks of Modular Exponentiation in Smartcards. In *CHES*. 144–157.

[45] Amir Moradi and Oliver Mischke. 2013. Comprehensive Evaluation of AES Dual Ciphers as a Side-Channel Countermeasure. In *ICICS*. 245–258.

[46] Amir Moradi, Oliver Mischke, and Christof Paar. 2013. One Attack to Rule Them All: Collision Timing Attack versus 42 AES ASIC Cores. *IEEE Trans. Computers* 62, 9 (2013), 1786–1798.

[47] National Institute of Standards and Technology. 2015. *Security Requirements for Cryptographic Modules*. Technical Report Federal Information Processing Standards Publications (FIPS PUBS) FIPS 180-4. U.S. Department of Commerce, Washington, D.C. https://doi.org/10.6028/NIST.FIPS.180-4

[48] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. 2006. Threshold Implementations Against Side-Channel Attacks and Glitches. In *ICICS*. 529–545.

[49] Kostas Papagiannopoulos and Nikita Veshchikov. 2017. Mind the Gap: Towards Secure 1st-Order Masking in Software. In *COSADE*. 282–297.

[50] Scott Pardo. 2013. *Equivalence and Noninferiority Tests for Quality, Manufacturing and Test Engineers* (1 ed.). CRC Press LLC, Philadelphia, PA.

[51] Emmanuel Prouff, Matthieu Rivain, and Régis Bevan. 2009. Statistical Analysis of Second Order Differential Power Analysis. *IEEE Trans. Computers* 58, 6 (2009), 799–811.

[52] Jean-Jacques Quisquater and David Samyde. 2001. ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards. In *E-smart*. 200–210.

[53] Mathieu Renauld, François-Xavier Standaert, Nicolas Veyrat-Charvillon, Dina Kamel, and Denis Flandre. 2011. A Formal Study of Power Variability Issues and Side-Channel Attacks for Nanoscale Devices. In *EUROCRYPT*. 109–128.

[54] Oscar Reparaz. 2016. Detecting Flawed Masking Schemes with Leakage Detection Tests. In *FSE*. 204–222.

[55] Ronald L. Rivest, M. J. B. Robshaw, R. Sidney, and Y. L. Yin. 1998. The RC6 Block Cipher. In *in First Advanced Encryption Standard (AES) Conference*. 16.

[56] Pascal Sasdrich, René Bock, and Amir Moradi. 2018. Threshold Implementation in Software - Case Study of PRESENT. In *COSADE*. 227–244.

[57] Tobias Schneider and Amir Moradi. 2015. Leakage Assessment Methodology - A Clear Roadmap for Side-Channel Evaluations. In *CHES*. 495–513.

[58] Donald J Schuirmann. 1987. A comparison of the two one-sided tests procedure and the power approach for assessing the equivalence of average bioavailability. *Journal of pharmacokinetics and biopharmaceutics* 15, 6 (1987), 657–680.

[59] Nader Sehatbakhsh, Baki Berkay Yilmaz, Alenka G. Zajic, and Milos Prvulovic. 2020. EMSim: A Microarchitecture-Level Simulation Tool for Modeling Electromagnetic Side-Channel Signals. In *HPCA*. 71–85.

[60] Aria Shahverdi, Mahammad Shirinov, and Dana Dachman-Soled. 2020. Database Reconstruction from Noisy Volumes: A Cache Side-Channel Attack on SQLite. *CoRR* abs/2006.15007 (2020).

[61] Madura A. Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. 2021. Rosita: Towards Automatic Elimination of Power-Analysis Leakage in Ciphers. In *NDSS*.

[62] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. 2019. Robust Website Fingerprinting Through the Cache Occupancy Channel. In *USENIX Security Symposium*. 639–656.

[63] Steven W. Smith. 1997. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, USA.

[64] François-Xavier Standaert. 2018. How (Not) to Use Welch's T-Test in Side-Channel Security Evaluations. In *CARDIS*. 65–79.

[65] Nikita Veshchikov. 2014. SILK: high level of abstraction leakage simulator for side channel analysis. In *PPREW@ACSAC*. 3:1–3:11.

[66] Bernard L Welch. 1947. The generalization of student's' problem when several different population variances are involved. *Biometrika* 34, 1/2 (1947), 28–35.

[67] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. 2020. Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures. In *USENIX Security Symposium*. 2003–2020.

[68] Wentao Zhang, Zhenzhen Bao, Dongdai Lin, Vincent Rijmen, Bohan Yang, and Ingrid Verbauwhede. 2015. RECTANGLE: a bit-slice lightweight block cipher suitable for multiple platforms. *Sci. China Inf. Sci.* 58, 12 (2015), 1–15.