Towards Automatic Power Analysis Leakage Elimination

Thesis submitted for the degree of

Doctor of Philosophy

by

Madura Anushanga Shelton

Supervised by,

Dr Markus Wagner

Dr Yuval Yarom

School of Computer Science,
Faculty of Engineering, Computer & Mathematical Sciences,
The University of Adelaide.

August 15, 2022

Contents

List of Figures

List of Tables

List of publications

1. Madura A. Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. Rosita: Towards automatic elimination of power-analysis leakage in ciphers. In *NDSS*. The Internet Society, 2021b
   Author's contributions: Research and design of ELMO*. Design, implementation and evaluation of ROSITA and writing.

2. Madura A. Shelton, Lukasz Chmielewski, Niels Samwel, Markus Wagner, Lejla Batina, and Yuval Yarom. Rosita++: Automatic higher-order leakage elimination from cryptographic code. In *CCS*, pages 685–699. ACM, 2021a
   Author's contributions: Design, implementation of ROSITA++, TRACETOOLS and writing.

Abstract

Side channel attacks were first described on timing based information leakage in 1996 by Paul Kocher. Similar leakage can also be observed through physical properties such as electro-magnetic emanations, acoustics and power consumption. Masking is a countermeasure that splits sensitive values into separate values called shares. However, due to the existence of unintended interactions in hardware, masked implementations may fail to reach their advertised security level. We propose an emulation based approach to find and eliminate leakage caused by unintended interactions.

This thesis presents three main contributions, ELMO*, Rosita and Rosita++. ELMO* is a modified version of ELMO (McCann et al. USENIX Security 2017) which can emulate leakage from unintended interactions realistically. Rosita and Rosita++ are two code rewriting tools that can fix univariate and multivariate leakage by using emulated leakage from ELMO*. We tested Rosita on first-order masked implementations of AES, ChaCha and Xoodoo and the slowdown incurred by the fixes were 21.3%, 75% and 32.3%. Rosita eliminated more than 90% of all observed leakage on a STM32F030 Discovery Evaluation board. We evaluated Rosita++ on second-order masked implementations of Boolean-to-arithmetic masking, PRESENT and Xoodoo where it eliminated all leakage in Boolean-to-arithmetic and Xoodoo implementations. The slowdown incurred by fixes applied were 36%, 29% and 189%. It was also evaluated on a third-order masked synthetic example using 30 million power traces recorded from the target device, where Rosita++ fixed all detected leakage. Our contributions have been presented at NDSS 2021 and CCS 2021 conferences.

Declaration

I certify that this work contains no material which has been accepted for the award of any other degree or diploma in my name, in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text. In addition, I certify that no part of this work will, in the future, be used in a submission in my name, for any other degree or diploma in any university or other tertiary institution without the prior approval of the University of Adelaide and where applicable, any partner institution responsible for the joint-award of this degree.

The author acknowledges that copyright of published works contained within the thesis resides with the copyright holder(s) of those works.

I also give permission for the digital version of my thesis to be made available on the web, via the University's digital research repository, the Library Search and also through web search engines, unless permission has been granted by the University to restrict access for a period of time.

15/08/2022

Date

Acknowledgements

I would like to offer my sincerest gratitude to both of my supervisors Dr Markus Wagner and Dr Yuval Yarom, without whom this thesis would not have been possible. It has been an honour to work under the dedicated guidance offered by them. The lessons learnt during this period have had a profound effect on me in both academic and non-academic life.

I am grateful to my collaborators with whom I worked in this project, Professor Lejla Batina, Łukasz Chmielewski, Niels Samwel and Dr Francesco Regazzoni. They were very kind in answering my many questions and collaborating with me in dire times that COVID-19 brought. I recognise the financial and student support offered by the University of Adelaide to fund and support me along this period. Thanks also to all the staff members of the School of Computer Science for their support and care. Specifically, Danny Di Giacomo from the School of Electrical and Electronic Engineering for his help and providing lab equipment. I am also grateful to my close friends for their support and attention offered during this period.

I am very grateful to my beloved wife Gothami Abayawickrama for her support, encouragement and sacrifices during my studies. Finally, I would like to thank my parents, Dewage Shelton and Chitrani Cooray who inspired and encouraged me throughout my life.

Chapter 1

Introduction

The introduction of side-channel attacks by Kocher in 1996 ushered in a new wave of attacks against cryptographic implementations. These attacks try to reveal the secret values of cryptographic devices by targeting their implementation details. In contrast, traditional attacks on cryptography focus on breaking the mathematical properties that cryptographic algorithms were based on. Side-channel attacks can be much more effective in breaking cryptographic implementations since they employ additional information that is emitted from an implementation to break it. Such information is gained from multiple avenues such as power consumption [Kocher et al., 1999], electromagnetic emanations [Quisquater and Samyde, 2001; Gandolfi et al., 2001] and acoustics [Genkin et al., 2014].

In the modern world, a large number of people depend on the security guarantees offered in devices such as credit cards, mobile phones, car keys and electronic safes. All such devices use some form of cryptographic implementation to offer security guarantees. When these devices are not protected against the threats of side-channel attacks, they become potentially unsafe to use. Therefore, the vendors of such devices require methods of securing these devices. Securing devices involves converting devices that leak side-channel information to ones that do not. This is a tedious and costly manual process that involves running multiple physical experiments, which require expertise in multiple fields such as computer science and electronics engineering. Securing a device involves multiple evaluations and manual application of code fixes such that the final implementation is leak-free. To this end, leakage emulation has provided a satisfactory answer to securing a device by approximating

leakage of hardware and software implementations.

Varying levels of success have been achieved by using emulators for detection of leakage [Debande et al., 2012; McCann et al., 2017; Corre et al., 2018]. Countermeasures for leakage have also been proposed in literature by introduction of noise to the power rails [Das and Sen, 2020], code polymorphism [Amarilli et al., 2011; Agosta et al., 2012] and information hiding [McEvoy et al., 2009; Tiri et al., 2002]. *Masking* is another type of countermeasure that was proposed by Messerges [2000]. It is a theoretical countermeasure that splits each intermediate value of a cipher into separate values called *shares* using random values. Typically, the advertised security level of masking is not reached in practice due to unintended interactions that happen in the hardware [Renauld et al., 2011; Balasch et al., 2015]. We observed that there existed a gap in research for automatically applying countermeasures to mitigate unintended interactions that happen in masked software implementations. Therefore, this work tries to answer the following problem,

> Is it possible to automatically mitigate leakage present in theoretically masked software implementations based on leakage emulation?

The core intuition of this work is based on the discovery of a methodology to insert code segments that mitigates leakage while preserving the original functionality. Another major finding is the usage of a linear regression based power model to discover root causes for leakage. Regression based power models are particularly beneficial due to their ease of adaptability to different hardware. Previously, regression based profiled power models were only used to emulate the power values of a device. The emulator that we base our work on is ELMO which was proposed by McCann et al. [2017]. ELMO is a profiling based emulator that uses a power model based on linear regression. We extended the leakage reproduction capabilities of ELMO's power model to match the realistic levels that are required to practically use it as an emulator that drives detection of root causes for leakage. Our main contribution in this area is the introduction of *internal state-based* leakage to the emulator. Internal states are specific values that are held in the device a long time after an instruction has been executed. Such states interact with newly executed instructions and end up exposing sensitive values. We also implemented previously discovered effects such as overwrite effects and memory bus related effects in ELMO* so that the emulated

leakage is in par with the leakage of the real device.

This thesis presents two methods which can be used to find root causes for leakage that depend only on emulated power values for their function. The first method analyses the statistical significance of the individual leakage that is contributed from each term of the power model. Although this method is efficient in detecting root cause leakage in univariate evaluations, it is not efficient for multivariate leakage evaluations. This lead us to introduce the second root cause detection method which searches for terms that cause leakage by elimination. In this method, the power values from the power model are evaluated while eliminating a single term from the power model at a time.

We implemented these methods in two open-source code rewriting tools called Rosita[1] and Rosita++[2]. We evaluated both of these tools with practical cipher implementations which feature a diverse set of cryptographic primitives such as AES, ChaCha, PRESENT and Xoodoo. Our evaluations show that all detected leakage using ELMO* can be fixed by using the methods detailed in this thesis. However, address-bus related leakage is not emulated by ELMO*. Address-bus related leakage is caused by interactions that happen in the address bus of a device. These are dependent on the different addresses that are used to address different parts of memory. If these addresses contain sensitive values or use shares of a sensitive intermediate value in such a way that they combine through the address bus, it will lead to information leakage that will not be detected by our emulation. A generic fix for such a leakage cannot be devised as the code rewrite engine requires additional implementation specific information such as the amount of memory that is accessed by the address bus related instructions. This is also not possible with the current code rewrite system.

We do not propose Rosita and Rosita++ as replacements to real evaluation of a device. Emulation based leakage evaluation will always be inferior to an evaluation of a real device while having benefits like reduced cost, reduced noise and parallelised evaluations. The aim of our work is to complement a leakage evaluator's tool set by introduction of automated code rewrites for detectable leakage using emulators.

[1] https://github.com/0xADE1A1DE/Rosita
[2] https://github.com/0xADE1A1DE/Rositaplusplus

Roadmap and main contributions of thesis

The main contributions of this thesis are in the areas of leakage emulation and automatic application of countermeasures. The rest of this thesis is structured as follows:

Chapter 2 Background. This chapter introduces preliminaries required to understand the contributions featured in this thesis. It introduces the reader to topics such as power analysis-based leakage, statistical leakage evaluation and past literature on leakage emulators and countermeasures for power-analysis-based leakage.

Chapter 4 Towards better leakage emulation. This chapter presents ELMO*, which is an extended version of the profiling based leakage emulator ELMO [McCann et al., 2017]. This chapter also presents a methodology of finding specific leakage types for a given Instruction Set Architecture (ISA) depending on the micro-architecture of the device. The effectiveness of the new power models are evaluated using nested model evaluation using F-tests.

Chapter 5 Automatic mitigation of univariate leakage. Presents Rosita, a tool that can find and eliminate univariate leakage detected by ELMO*. Rosita introduces a novel root cause detection method for univariate leakage using only emulated traces. Additionally, this chapter presents a methodology for finding new countermeasures that can be used as code rewrite patterns. The countermeasures that were generated through this methodology are used to fix leaky code segments found by Rosita. A discussion of the implementation details of Rosita follows. Finally, Rosita is evaluated using code blocks of first order masked implementations of AES [Daemen and Rijmen, 2002], ChaCha [Bernstein, 2008] and Xoodoo [Daemen et al., 2018a].

Chapter 6 Automatic mitigation of multivariate leakage. Presents Rosita++, a tool that can fix multivariate leakage. Multivariate leakage is leakage that stems from multiple sample points. Reasons why Rosita cannot be used to detect root causes for multivariate leakage are discussed in this chapter. We present a new root cause detection method called *elimination of terms*

in ROSITA++. ROSITA++ is evaluated with code segments from two 3-share implementations of PRESENT [Bogdanov et al., 2007] and Xoodoo [Daemen et al., 2018a] and a 3-share cryptographic primitive for converting between Boolean and arithmetic masking [Goubin, 2001].

Chapter 7 Conclusions. Concludes this thesis by discussing the limitations and future work in the area of emulation based automatic countermeasures for power analysis-based leakage.

Chapter 2

Background

2.1 Cryptology background

Cryptology is the study of secretive communication and storage methods and their design. Cryptology encapsulates the fields of *cryptography* and *cryptanalysis*. Cryptography deals with the design and implementation of methods used for transforming legible data (i.e. plaintext) to non-legible data called ciphertext and vice-versa. Cryptanalysis studies methods of breaking cryptographic methods so that adversaries can recover messages. Until the 20th century, cryptology remained a field that was predominantly used by governments and military establishments. Towards the latter part of the 20th century, cryptography found more adoption in personal electronic device such as personal computers and mobile phones, mainly due to the wide adoption of the Internet.

2.1.1 Cryptography

Earliest known uses of cryptographic algorithms are reported from 4000 years ago from Egypt. Julius Caesar (100 BC–44 BC) introduced the now famous shifted alphabet cipher. The Caesar cipher is a *substitution cipher* based on substituting letters from a mapping which is held secret [Kahn, 1996]. Following the major contributions of Marian Rejewski [Rejewski, 1981] and Alan Turing in breaking encrypted Nazi German communications [Turing and Copeland, 2004], cryptology was widely adopted as a field of great value that demonstrates the power of modern nations. Further developments such as electronic-fund transfer, authenticated messages and

internet services have made cryptography into a field that the modern world cannot function without.

Ciphers were placed on solid mathematical foundations following major improvements done in mid 20th century [Kahn, 1996]. Modern cryptographic algorithms depend on mathematical proofs for their security. The secrecy levels offered by cryptographic algorithms are mainly divided into two types called *perfect secrecy* and *provable secrecy*.

Perfect secrecy. A cipher has perfect secrecy, if given any ciphertext, no additional information is gained about the plaintext. This means that if the cipher has the same number of valid keys as the number of all possible plaintexts for each ciphertext, it is a cipher with perfect secrecy. Even if an adversary attacks this cipher with unlimited computational power through a brute-force approach, all possible messages will have the same probability. Therefore, ciphers with perfect secrecy are protected against adversaries with unlimited computational power. In 1949, Claude Shannon proved that the one-time pad does not provide any information regarding the plaintext other than its length and that this property makes it a cipher which offers perfect secrecy [Shannon, 1949].

Provable secrecy. Some algorithms depend on problems that are known to be mathematically hard to solve such as the discrete-logarithm problem. For sufficiently large input values, the calculation of an answer to the discrete-logarithm problem takes an impractically long time. If a cryptographic scheme is built on top of such a problem, then adversaries will have to solve the yet unsolved mathematical problem behind it. This guarantees the cryptographic scheme a certain level of secrecy which is called provable secrecy. However, such schemes are relative to the current state of knowledge and require replacement when such hard-problems become easier to solve following new discoveries. One such discovery is the discovery of quantum computers. A quantum computer would easily break cryptographic algorithms based on discrete-logarithm as they can solve it within practically achievable time limits compared to classical computers [Shor, 1994]. Post-quantum cryptography deals with cryptographic algorithms that are still harder to solve even using quantum computers. At the time of writing, all of this is under experimentation and current state-of-art quantum computers need to be developed significantly to achieve the performance

levels that are required to attack pre-quantum cryptographic schemes. For example, the quantum algorithm that can be used to attack the integer factorisation problem of RSA (Rivest Shamir Adleman) requires $2n + 2$ quantum bits to attack RSA keys of n bit length. In practice, the recommended minimum key length for RSA is 2048 [National Institute of Standards and Technology, 2015b]. The latest quantum computer at time of writing is only capable of operating at a quantum bit length of 76 [Zhong et al., 2020].

2.1.1.1 Symmetric key cryptography

Symmetric key ciphers use the same key for encryption and decryption operations. Examples for some symmetric ciphers include One-time pad, DES (Data Encryption Standard), AES (Advanced Encryption Standard), ChaCha, RC4 and all classical cryptographic algorithms. Symmetric ciphers can be classified as *stream* ciphers or as *block* ciphers. Stream ciphers operate on a single symbol at a time and block ciphers operate at a set of symbols at a time. If the length of the message is not a multiple of the block size, it may be padded with symbols before the encryption according to a given mode of operation.

A limitation with symmetric ciphers is that they must be paired with a secure key-exchange protocol to be useful. Otherwise, the key itself would have to be communicated in a secure channel. This is a recent requirement, as modern communication channels such as radio communication and wired communication networks are inherently public. It is easy for a remote eavesdropper, who has access to the communication medium, to extract such messages.

2.1.1.2 Asymmetric key cryptography

Asymmetric key cryptography, or public-key cryptography uses different keys for encryption and decryption operations. The key generation sequence for an asymmetric cipher always generates two keys. One is used for the encryption and the other for decryption. Popular examples include RSA and elliptic Curve cryptography based algorithms.

In addition to encryption and decryption of messages, asymmetric ciphers are used to offer authentication to messages similar to a signature. The signing party would sign a message with a key and keeps it private while publishing the verification key. The private key is held secret by the signing party and the public key is accessible

to anyone. This lets a receiver verify a message with the public key, it is impossible for them to impersonate the original sender as they cannot derive or have access to the privately held key.

2.1.2 Cryptanalysis

Cryptanalysis is dedicated to the analysis of cryptographic algorithms and protocols. Specifically, cryptanalysis focuses on methods for breaking ciphers. The first known cryptanalysis method is *frequency analysis* and was introduced by Al-Kindi around 800. Frequency analysis uses the relative frequencies of the appearance of letters in a language to break substitution ciphers. Traditionally, cryptanalysis was limited only to mathematical and statistical methods. With the increasing use of machines for carrying out cryptographic workloads, the information leaked as result of implementation weaknesses help drive attacks against them. *Side-channel* and *fault injection* attacks are examples of exploiting weaknesses in implementations. Side-channel attacks exploit a cryptographic device's interactions with its operating environment. The correlation between sensitive information and the recorded measurements aids in revealing the sensitive information that is processed by a device. Examples for recorded measurements include sound, electromagnetic emanations, power consumption and latency measurements. Fault injection attacks exploit weaknesses by operating a device at unusual operating conditions. These conditions lead the device to act differently to what it is designed for. For example, Choukri and Tunstall [2005] demonstrated that the number of AES rounds could be reduced to one by introducing a glitch on the microprocessor of a smart card. This significantly reduces the security offered by AES.

Cipher implementations that restrict access to the details of how they operate (e.g. source code) are said to offer security through obscurity. If a cipher is designed with the assumption of secrecy of the algorithm and or implementation details are preserved, this in turn becomes a weakness of the cipher when an adversary gets hold of the implementation details. Therefore, this is an inferior design technique to be used for implementing ciphers. In the 19th century Auguste Kerckhoffs formulated a principle which states that the key should be the only part of a cryptosystem that should be private. This method of designing a cryptosystem concentrates all secrecy to the key. Especially in the information age where computers operate on code that

could trivially be reverse engineered, this design pattern is regarded as the standard way of designing ciphers.

2.1.3 Side-channel based attacks

In 1996, Kocher demonstrated that RSA keys and Diffie-Hellman exponents could be revealed through measuring the time required to perform private key operations. Such *side-channels* can leak information about the internal state of the computation, leading to a potential collapse of the security of a cryptographic implementation.

Since then, significant effort has been invested in researching side-channel attacks. Attacks have been demonstrated against various types of primitives, including symmetric ciphers [Bernstein, 2005; Moradi et al., 2013], public key systems [Messerges et al., 1999; Genkin et al., 2016] post quantum cryptography [Aysu et al., 2018], and non-cryptographic implementations [Batina et al., 2019; Yan et al., 2020; Shahverdi et al., 2020; Shusterman et al., 2019]. These attacks exploit various side-channels, such as power consumption [Kocher et al., 1999] electromagnetic emanations [Quisquater and Samyde, 2001; Gandolfi et al., 2001], micro-architectural state [Ge et al., 2018; Lou et al., 2021; Bertoni et al., 2005], and even acoustic and photonic emissions [Genkin et al., 2014; Krämer et al., 2013]. Side-channel attacks work by correlating the recorded measurements of phenomena with the intermediate values of cryptographic algorithm that is executed within a device. This is done through the statistical analysis of recorded measurements.

2.1.4 Countermeasures against side-channel attacks

Proposals have been put forward for hardware designs that reduce emissions [Chen and Zhou, 2006], software solutions that ensure secret-independent execution [Ge et al., 2018], adding noise to hide the signal [Moradi and Mischke, 2013; Das and Sen, 2020], code polymorphism [Amarilli et al., 2011; Agosta et al., 2012], information hiding [McEvoy et al., 2009; Tiri et al., 2002] and information masking techniques [Chari et al., 1999; Nikova et al., 2006] where secret values are split to multiple unrelated values.

Full mitigation of such attacks is not possible as there always can be some other phenomenon that the implementers of the cryptographic system did not implement mitigations for or were aware of. However, a significant level of protection can be

gained by implementing defences for known side-channel attacks.

2.2 Statistical background

This section introduces statistical concepts that are used in this work. Throughout the rest of this work, we use upper case Latin letters to denote random variables and lower case of the same letter to denote a value that the random variable may take. A bar over a random variable denotes the mean of the random variable, for example, $\overline{X}$ denotes sample mean of X. Some common functions that are defined over random variable(s) are the expected value $E(X)$, the variance $Var(X)$ and the covariance $Cov(X, Y)$. Population mean and standard deviation is denoted by μ and σ.

2.2.1 Probability distributions

2.2.1.1 Normal distribution

A normal distribution is characterised by its probability density function ($\phi(x)$) shown in Equation 2.1, where the population mean is μ, population variance is σ^2 and x is a value of a continuous random variable. A random variable X, being normally distributed is denoted as $X \sim N(\mu, \sigma^2)$.

$$\phi(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{2.1}$$

2.2.1.2 Student's t-distribution

A Student's t-distribution with v degrees of freedom is defined by its probability density function ψ_v in Equation 2.2, where Γ is the gamma function.

$$\psi_v(x) = \frac{\Gamma\left(\frac{v+1}{2}\right)}{\sqrt{\pi v}\Gamma\left(\frac{v}{2}\right)} \left(1 + \frac{x^2}{v}\right)^{-\frac{v+1}{2}} \tag{2.2}$$

The Gamma function is defined as shown in Equation 2.3 for all complex numbers (z) except when the real part is 0 or negative.

$$\Gamma(z) = \int_0^\infty x^{z-1} e^{-x}\, dx \tag{2.3}$$

The Cumulative Distribution Function (CDF) of the Student's t-distribution ψ_v is

given by CDF_t as depicted in Equation 2.4.

$$CDF_t(x, v) = \int_{-\infty}^{x} \psi_v(x)\, dx \tag{2.4}$$

Relation to the normal distribution. Given a random variable X, μ and σ are population mean and standard deviation. The quantity q in Equation 2.5 follows a normal distribution when n samples are sampled from the population.

$$q = \frac{\overline{X} - \mu}{\sigma/\sqrt{n}} \tag{2.5}$$

Student's t-distribution arises when the population variance (σ^2) is approximated by a sample variance (s^2). The quantity t in Equation 2.6 follows a Student's t-distribution with $n - 1$ degrees of freedom. As shown in Figure 2.1, a Student's t-distribution gets closer to the standard normal distribution as its degrees of freedom increases.

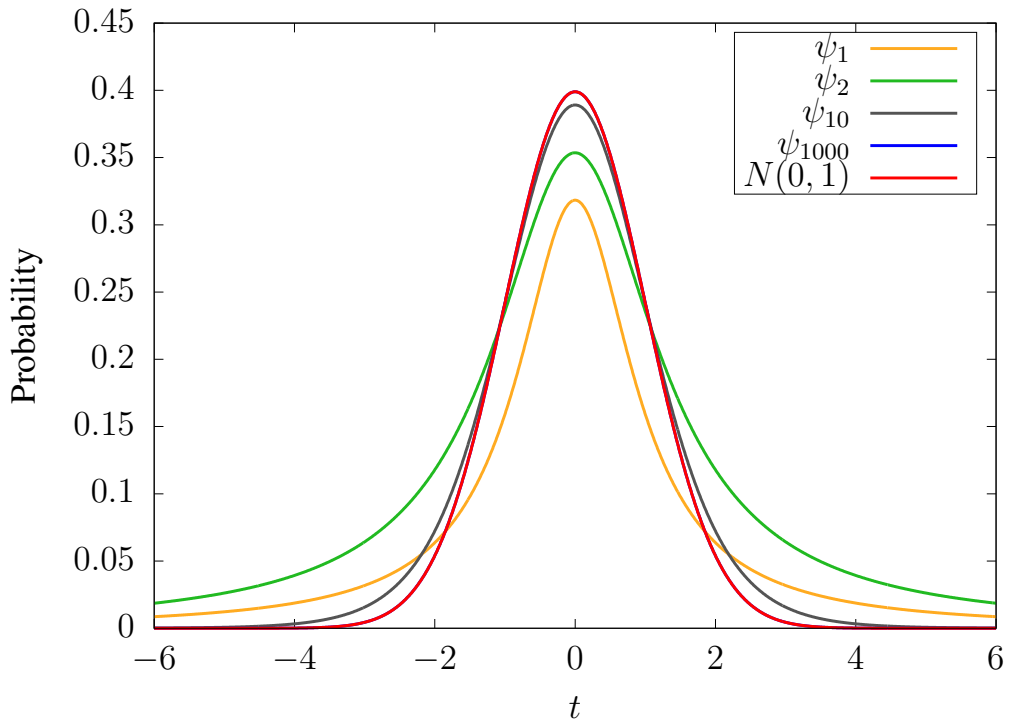


Figure 2.1: Student's t-distributions with differing degrees of freedom compared to the standard normal distribution.

$$t = \frac{\overline{X} - \mu}{s/\sqrt{n}} \tag{2.6}$$

2.2.2 Hypothesis testing

Hypothesis testing is a process used to test the truth of some supposition or a proposed explanation made with limited evidence. The explanation must be testable and falsifiable. The process of hypothesis testing uses the experiment data to prove either that a research hypothesis holds or not without a reasonable doubt.

The default decision of this test is called the *null hypothesis*. The null hypothesis (H_0) states that any difference shown in the experiment data that shows that the research hypothesis is true is due to statistical error and that there is no significant difference between the data collected when the research hypothesis is true or false. In contrast, the *alternative hypothesis* (H_1) states that there is significant evidence that the research hypothesis is true.

When using hypothesis testing with experiment data, the evidence is acquired by sampling values (i.e. measurements) from different distributions. The standard method of performing such a test is to sample values from the same random variable of two instances of the same experiment namely, *control* and *test*. The control experiment represents the null hypothesis. In other words, this experiment represents what happens when nothing special is done. The test experiment represents the alternative hypothesis. Alterations to the experiment dictated by the research hypothesis are applied to the test experiment. This must be done in such a way that the test experiment is minimally different from the control experiment otherwise measurements will include effects other than the ones that the evaluator is interested to observe. The result of the experiment depends on the evaluation of the two distributions of data collected from the above explained experiments. A standard way of comparing two distributions is to compare their means. This is referred to as the *testing of means* [Hardle et al., 2015, Chapter 9.4].

2.2.2.1 Student's t-test

The Student's t distribution can be used to measure the significance of the difference of means. There are three main configurations in which means are tested, they are known as two-sided, right-sided and left-sided tests.

Two-sided test.

$$H_0 : \overline{X} = \mu_0, \quad H_1 : \overline{X} \neq \mu_0 \tag{2.7}$$

The null hypothesis for a two-sided test states that the two distributions are the same. We assume that the sample under test is a part of a hypothetical population. $\overline{X}$ is the sample mean and s is the sample variance and the hypothetical population has mean μ_0. Under above conditions, the quantity t in Equation 2.8 is Student's t-distributed. Value of t quantity is referred to as the t-value of the t-test. It is assumed that the population data is normally distributed and that the variances are homogeneous which means that the variances of different samples from the population are equal.

$$t = \frac{\overline{X} - \mu_0}{s/\sqrt{n}} \tag{2.8}$$

The rejection of the null hypothesis is done in favour of the alternative hypothesis under a maximum allowable p-*value*. This is referred to as the significance level and is denoted by α. The p-value of a given statistic (i.e. t-test value) is the probability of observing a statistic at least as extreme or more extreme than the given statistic.

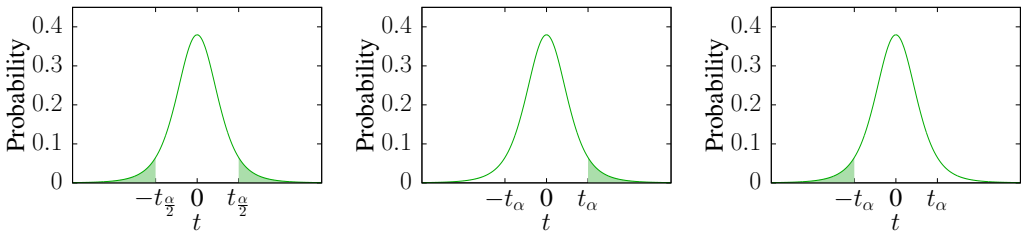
The p-value of a t-test is given by the area under the curve when the t-value is larger than the t-value at significance level α. This t-value is denoted by t_α. If the p-value is smaller, an evaluator is more confident $(1 - p)$ in rejecting the null hypothesis. Therefore, at a significance level of α the respective confidence level is $1 - \alpha$.

For a two-tailed t-test, the regions under the curve that are considered for the significance level are situated at ends of the Student's t-distribution curve. These are analogous to $\overline{X} > \mu_0$ and $\overline{X} < \mu_0$ as shown by Figure 2.2a. For the two-sided test, the confidence and p-value probabilities are split equally. Therefore, the respective t-value at a significance level of α for the two-sided test is $t_{\alpha/2}$.

Right-sided test.

$$H_0 : \overline{X} \leq \mu_0, \quad H_1 : \overline{X} > \mu_0 \tag{2.9}$$

In a right-sided t-test, as the sample mean $(\overline{X})$ gets larger, probability of observing an even more extreme statistic reduces. Therefore, only the right tail of the t-distribution is considered in the right-sided test. This is depicted by the right side tail



(a) Two-sided t-test. (b) Right-sided t-test. (c) Left-sided t-test.

Figure 2.2: Different configurations of a t-test.

of Figure 2.2b.

Left-sided test.

$$H_0 : \overline{X} \geq \mu_0, \quad H_1 : \overline{X} < \mu_0 \tag{2.10}$$

In contrast, in the left-sided test, only the left tail of the t-distribution is considered. As the sample mean gets smaller, probability of observing an even more extreme statistic reduces. This is depicted by the left side tail of Figure 2.2c.

2.2.2.2 Welch's t-test

Welch's t-test extends the mean comparison methodology used in Student's t-test for testing samples from two different populations. D is an estimator which is the mean difference of two samples from respective populations, X_1 and X_2.

$$D = \overline{X}_1 - \overline{X}_2$$

The comparison is done against D and a hypothetical population mean difference, ω_0. Therefore, the null hypothesis and the alternative hypothesis for the two-sided test changes as follows,

$$H_0 : D = \omega_0, \quad H_1 : D \neq \omega_0 \tag{2.11}$$

The variance of the hypothetical mean difference population is derived from the two populations. The two populations are assumed to be normally distributed and independent of each other. When n_1 and n_2 are population sizes, the following quantity of t has a Student's t-distribution with $n_1 + n_2 - 2$ degrees of freedom [Hardle et al., 2015].

$$t = \frac{\overline{X}_1 - \overline{X}_2 - \omega_0}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}}$$

Bernard Lewis Welch proposed approximating the population variances by using sample variances. Then, t can be approximated by a Student's t-distribution given by,

$$t = \frac{\overline{X}_1 - \overline{X}_2 - \omega_0}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}$$

with v degrees of freedom, where,

$$v = \frac{\left(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right)^2}{\frac{1}{n_1-1}\left(\frac{s_1^2}{n_1}\right)^2 + \frac{1}{n_2-1}\left(\frac{s_2^2}{n_2}\right)^2}.$$

2.2.2.3 Two One-Sided t-tests

The Two One-Sided t-test (TOST) is used to test for the equivalence between two populations with lower (ω_l) and upper (ω_u) boundaries of population mean differences. Similar to Welch's t-test, it introduces an estimate (D) as the difference of sample means,

$$D = \overline{X}_1 - \overline{X}_2.$$

Given that $\omega_l < \omega_u$, null and the alternative hypotheses used in TOST are defined as follows,

$$H_0 : D \leq \omega_l \text{ or } \omega_u \leq D\,,$$
$$H_1 : \omega_l < D \text{ and } D < \omega_u.$$

The null hypothesis covers the scenarios where the difference of means lie outside the boundaries and the alternative hypothesis covers the scenario of it lying within

the boundaries, which is interpreted as the two distributions are equivalent. We use the following t quantities as discussed earlier in Section 2.2.2.2, but here we use the boundary values instead of mean difference of the hypothetical population.

$$t = \frac{\overline{X}_1 - \overline{X}_2 - \omega_l}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}} \tag{2.12}$$

$$t = \frac{\overline{X}_1 - \overline{X}_2 - \omega_u}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}} \tag{2.13}$$

Similar to the Student's t-test and Welch's t-test, we now have two individual t-tests that both of which need to pass with a high confidence such that we can reject the null hypothesis with high confidence.

2.3 Power analysis based side-channel leakage

This section discusses side-channel attacks and mitigations for power analysis attacks. Power analysis is the methodology in which power consumption values of devices are recorded and analysed. In a side-channel leakage related context, power analysis is performed with the intention of retrieving internally processed sensitive information from a device. This section discusses side-channel attacks and evaluation methods based on power analysis.

2.3.1 Sources of information leakage through power consumption

Power consumption of a single bit. Contemporary computational electronic devices store intermediate values in memory units called *registers* during execution. Each bit in a register is stored in a digital sequential circuit called a *latch*. In a nutshell, a latch includes digital circuitry that enables it to store one of two stable states. These states can be switched from one to the other by sending a specific signal to the latch. The current state of a latch depends on its previous and current inputs. The Figure 2.3 shows a Set Reset (SR) latch, a type of latch that is commonly used to implement registers in computational devices. When the Set (S) input is set to high, the SR latch's output (Q) sets to high and stays *latched* in high level until high input

is received at Reset (R) input. The latched state is used for storage of a single bit.

The current technology used to implement these devices is called Complementary Metal Oxide Semiconductor (CMOS). CMOS is a fabrication process that uses Metal Oxide Semiconductor Field Effect Transistors (MOSFET) to build integrated circuit chips. Latches too are CMOS devices that have MOSFET transistors in them. Any changes done to the state of a latch translate to switching some transistors on or off. Therefore, the states that a latch stores (i.e. 0 or 1 bit) represent distinct power consumption levels.

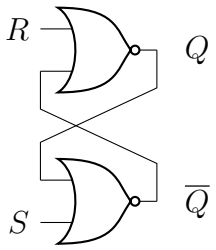


Figure 2.3: SR Latch.

Power consumption of a register. The power consumption of a register is the total consumption of all the bits of the register. This value is proportional to the number of bits that are set in a register which is the Hamming weight of the value that is set in the register. It also depends on the Hamming distance of new and old values when new values are being written to the register.

Glitching effects in logic cells. Glitching effects are observed at unstable states of logic cells. Due to propagation delays it takes some time for a logic cell to stabilise its output logic levels. In the meantime, it may output values that are *glitchy*. These values may combine input values of the logic cell in unnecessary ways and some of these combinations may also end up in information leakage.

Acquisition of power traces. The acquisition of power traces refers to the collection of power consumption measurements from a physical device. The power consumption of a device can be measured either by directly measuring the current through a current probe, voltage across a shunt resistor or indirectly by measuring the electromagnetic field intensity near to the device. Since we are interested in the power consumption while some computation is carried out on the device, the measurements need to

be collected within the time frame which the computation takes place. This can be done by using an oscilloscope that supports recording measurement traces to storage or sends the measurements to a separate acquisition device [Mangard et al., 2007, Section 3.4]. A description of the physical experiment setup that we used is in Section 3.2.

2.3.2 Simple power analysis attack

Power traces are power consumption measurements recorded within a given time frame. When observing a power trace recorded from a device that had run different code segments, the difference between each code segment could be easily distinguished visually. This is due to the nature of the time dependent patterns that are present in the power consumption signal. Whenever a different code segment is executed, it is reflected in the power traces. This effect can be used to break implementations of ciphers that have input dependent branches in them. This is referred to as a Simple Power Analysis (SPA) attack.

2.3.3 Differential power analysis attack

In contrast to SPA, Differential Power Analysis (DPA) involves analysis of a collection of power traces from code that has a constant control flow (i.e. code with no conditional branches dependent on input). It exploits the biases between the power consumption values when the processor executes the same operations in each run of a cipher with different values. This methodology was first used to extract sensitive data from a cipher by Kocher et al.. They demonstrated their attack on the Data Encryption Standard (DES) cipher. The key idea behind DPA is the exploitation of the power consumption of a single bit. A DPA attack exploits the correlation between the power consumption and the value of a specific bit in multiple executions of the same operation with different inputs.

The first step of conducting a DPA attack is to collect power values for different inputs (i.e. plaintexts) at a point of time when the selected bit is operated on. Second, the value of the selected bit is calculated for a set of all possible *subkeys* and for all inputs used for the previous test. A subkey is a part of the key which is then enumerated through all possible values to generate all possible hypothetical values for the selected bit. The subkey that generates the hypothetical bit values with the

highest correlation to the observed power values is the correct subkey. This process can be extended to other subkeys depending on the cipher to reveal the full key.

2.3.4 Correlation power analysis attack

Correlation Power Analysis (CPA) extends the understanding of the power consumption of a device by introducing better approximations through power models. Two such models are Hamming weight model and Hamming distance model [Messerges, 2000]. Hamming weight is the number of set bits of a binary number and the Hamming distance is the number of corresponding bits that are different between two binary values that have the same bit count.

CPA employs the Pearson's correlation coefficient (r) shown in Equation 2.14 to determine the correlation between the power traces and the hypothetical power values from a specific power model (e.g. Hamming weight or Hamming distance). Here, x_i and y_i denote individual power samples and the hypothetical values. $\overline{x}$ and $\overline{y}$ denote the mean of each distribution of samples. n is the number of samples.

$$r = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \overline{y})^2}} \tag{2.14}$$

2.3.5 Test Vector Leakage Assessment

In previous sections, multiple methods of attack were discussed in the context of attacking power side-channel leakage of cipher implementations. However, such methods only allow specific attacks on a given implementation. The assessment of the security of a given device can only be done by a generic evaluation methodology that does not limit itself to exploitation based on a certain model or intermediate values. One such method is Test Vector Leakage Assessment (TVLA).

Test Vector Leakage Assessment (TVLA) was first introduced in Goodwill et al. [2011] as a methodology for measuring power analysis side-channel leakage of a given device. An advantage of TVLA is that it can be used both with or without a power model. When used with a power model, the test is said to be run on a *specific* configuration and the result shows whether the leakage can be exploited or not. In a *non-specific* configuration, a power model is not used, and the result gives a level

of confidence that helps the evaluator conclude whether the device under test emits leakage. This method does not offer insight into how this leakage can be exploited or whether it can be exploited at all [Schneider and Moradi, 2015].

TVLA is based on statistical hypothesis testing using a two-sided Welch's t-test, discussed in Section 2.2.2. The quantity t in Equation 2.15 follows a Student's t-distribution with v degrees of freedom when X_1 and X_2 are sample distributions with their means following a normal distribution and when ω_0 is a constant value. The standard deviations of X_1 and X_2 are s_1 and s_2, and the number of samples in each sample distribution is n_1 and n_2.

$$t = \frac{\overline{X}_1 - \overline{X}_2 - \omega_0}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} \tag{2.15}$$

$$v = \frac{\left(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right)^2}{\frac{1}{n_1 - 1}\left(\frac{s_1^2}{n_1}\right)^2 + \frac{1}{n_2 - 1}\left(\frac{s_2^2}{n_2}\right)^2} \tag{2.16}$$

Recall that in hypothesis testing using two-sided t-tests, the null hypothesis of $\overline{X}_1 - \overline{X}_2 = \omega_0$ is rejected in favour of the alternative hypothesis ($\overline{X}_1 - \overline{X}_2 \neq \omega_0$) when the absolute value of t is above a certain t-value *threshold* that is set according to a given significance level, α. The t-test is used with ω_0 set to 0 on both specific and non-specific configurations of TVLA. In other words, this means that the t-test tests for the inequality of means of X_1 and X_2.

The Cumulative Distribution Function (CDF) for the Student's t-distribution, which is the area under the curve is calculated from Equation 2.17, where $\psi_v(t)$ is the Probability Density Function (PDF) for a t-distribution with v degrees of freedom (see Section 2.2.1.2 for details).

$$CDF_t(t_0, v) = \int_{-\infty}^{t_0} \psi_v(t)\, dt \tag{2.17}$$

For a two-sided t-test, the p-value is given by Equation 2.18.

$$p = 2(1 - CDF_t(|t|, v)) \tag{2.18}$$

If the p-value is less than or equal to a certain significance level (α), the null hypothesis is rejected at significance level α. The t-value at significance level α can be inferred from the inverse of the CDF as $CDF_t^{-1}(1-\alpha/2, v)$. Typically, TVLA uses a threshold value $t = 4.5$ under the assumption of a significance level (α) of 0.00001 and $v > 1000$ [Schneider and Moradi, 2015].

On a specific t-test, the two sample distributions (X_1 and X_2) are selected according to an intermediate value calculated from a model. If the t-test value shows statistical significance, then a suitable DPA attack can be carried out to extract the key.

In contrast, a non-specific t-test has no explicit model. In a non-specific t-test, the sample distributions are grouped based on the kind of input supplied to the cipher. Mainly, two such input types exist, one is *fixed vs. random* input where a set of power traces are recorded while fixed input is processed and the same number of traces are recorded while the device processes random inputs. The other input kind is *fixed vs. fixed* where power traces are recorded while the device processes two different fixed inputs. Example t-test values generated from 10000 power traces (5000 with fixed input and 5000 with random input) collected in the fixed vs. random input configuration for an unprotected AES implementation is shown in Figure 2.4, which shows significant leakage around the SubBytes procedure.

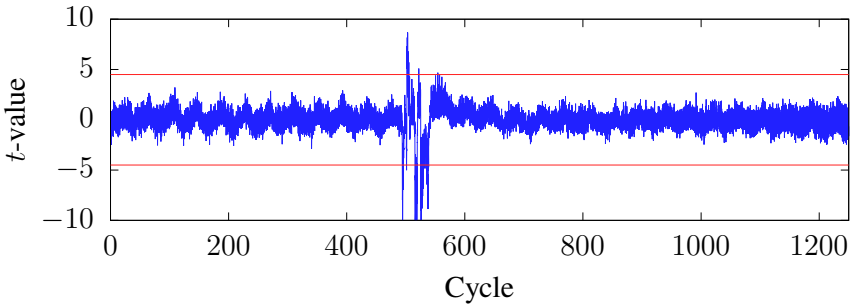


Figure 2.4: t-values for the first round of AES.

There are a number of things to be concerned when conducting a power analysis experiment to record samples that will be used for TVLA. First, the input for each run of the cipher's code segment must be chosen from a random coin flip. This is done in order to remove any bias that stems from the order of execution and to minimise the effects of persisted state left over from the previous run of the code segment.

Therefore, the inputs should be sent to the target device in an non-deterministic and randomly-interleaved fashion [Schneider and Moradi, 2015].

Second, the analysis of TVLA should also be done with care as both false-positives and false-negatives could be present. As TVLA is not an exhaustive test, it cannot guarantee that a certain implementation does not leak sensitive information. It depends on the number of power traces recorded and the inputs used as fixed inputs.

The amount of statistical error for rejecting the null hypothesis increases as the number of observed sample points increase. This increases the number of false-positives. Therefore, increasing the t-test threshold accordingly has been suggested in many studies [Balasch et al., 2015; Ding et al., 2017]. Ding et al. suggest updating the significance level for a t-test for a single sample point to α' where the significance level for all sample points is α. The new significance level (α') is given by, $1-(1-\alpha)^{\frac{1}{m}}$. Therefore, the new threshold ($t_{\alpha'}$) is ${CDF_t}^{-1}(1-\alpha'/2, v)$.

2.3.6 Modelling leakage from multiple probes

Usually, a single probe is used to sample power consumption values. As deeper and more fine-grained information is revealed from the processor, it becomes increasingly hard to defend against such attacks. To reason about the security of a processor, a theoretical model was required where an attacker's strength could be evaluated with respect to a given implementation. Such a model was first presented in Ishai et al. [2003]. This model assumes that an attacker has access to power values from only t wires of a Boolean circuit. This is called the *t-probing* model. By dividing the information of each wire in the circuit into $t+1$ or wires, the implementation can be made secure under the t-probing model. The reason behind the enhanced security is that the attacker is now required to gather information from $t+1$ or more wires and should *combine* all of them to gain the original value. The combination is done through a predetermined combination function. Commonly used combination functions are the product or difference between mean centred power samples [Prouff et al., 2010; Schneider and Moradi, 2015]. The t-probing model can be applied to software implementations by assuming that an attacker is limited to observe the combined values at t sample points of a power trace. For example, dth-order Boolean masked implementations split each of their secret intermediate values into $d+1$ parts called shares using random values (see Section 2.3.7 for details). An attacker needs

to observe all shares to reconstruct the original secret value.

2.3.7 Countermeasures for power analysis attacks

One of the earliest methods used against side-channel attacks is the introduction of noise to the emitted signals. This was achieved in many forms including hardware noise injection [Das and Sen, 2020] and code polymorphism [Amarilli et al., 2011; Agosta et al., 2012]. Another approach is to make the leakage independent of the intermediate values. Changes to the underlying logic circuit implementations significantly lowers the correlation between the emitted signal and the intermediate values that are being processed. This is referred to as *balancing* [McEvoy et al., 2009; Tiri, 2010]. *Masking* [Messerges, 2000] is another type of countermeasure that can be applied in hardware and software implementations. It is an established method for mitigating power analysis side-channel attacks on ciphers [Rivain and Prouff, 2010; Balasch et al., 2015]. In masking, a sensitive value is split into two or more parts called *shares* [Messerges, 2000; Chari et al., 1999]. One of the methods of splitting sensitive values is called *Boolean masking*. The Boolean masking scheme splits a secret value (s) into $k+1$ shares, $s_0, s_1, ..., s_k$ where for $1 \leq i \leq k$, s_i is chosen uniformly at random and s_0 is chosen such that $s_0 = s \oplus s_1 \oplus s_2 \oplus \cdots \oplus s_k$. This scheme ensures that the sensitive value can be only recovered by observing all shares. Another method is to use addition in modulo 2^k for some $k \in \mathcal{Z}^+$ instead of the exclusive or operator. This is referred to as *arithmetic* masking [Messerges, 2000]. However, practical masking based implementations often fail to offer their advertised theoretical levels of security due to unintended interactions that breach an assumption called the Independent Leakage Assumption (ILA).

The Independent Leakage Assumption states that manipulation of each share must be separate from all other shares [Balasch et al., 2015; Renauld et al., 2011]. The existence of *glitches* and register overwrites reduce the effective number of shares by combining subsets of shares. As these breaches happen unintentionally, they are referred to as *unintentional ILA breaches*.

Verification of masked implementations can help identify weakly masked implementations and fix them. Such issues have been previously demonstrated to be automatically verifiable by using tools such as SLEUTH [Bayrak et al., 2013]. However, as SLEUTH depends only on data and control flow, it is not aware of additional

micro-architectural interactions that may take place. It has been demonstrated that due to the inherent resource sharing of hardware, there can be combinations of shares that cannot be deduced from the assembly code alone [Meyer et al., 2020].

A number of studies have demonstrated application of countermeasures to fix leakage caused by ILA breaches [Papagiannopoulos and Veshchikov, 2017; Meyer et al., 2020]. Studies that do not model or depend on physical power traces cannot effectively mitigate ILA breaches as such effects depend on physical characteristics of devices and vary significantly between different devices. For example, Wang et al. [2019] demonstrate a methodology that checks for register overwrites but does not include checking of interactions at the micro-architecture level. Micro-architectural operations within a microprocessor such as overwriting of internal pipeline registers have been found to cause ILA breaches [McCann et al., 2017; Corre et al., 2018]. Seuschek and Rass found that no-op instructions caused accesses to the register file which leads to leakage of secret values in an Atmel AVR microcontroller unit. They observed that even instructions that do not change the processor state such as no-op instructions can cause leakage due to the internal architecture of a CPU. The analysis of a bit-sliced implementation of AES S-box in Seuschek et al. [2017] show leakage leakage on an ARM Cortex-M0 CPU. Their analysis revealed significant leakage as the authors of the AES S-box [Schwabe and Stoffelen, 2016] had not taken micro-architectural effects into account. It is evident from these studies that detailed leakage characteristics cannot be exposed by only analysing software machine code without some description about the hardware it runs on.

2.3.7.1 Automatic application of countermeasures

Application of masking to an unprotected cipher implementation mitigates the risk of mistakes due to human error and saves development and testing times for large cipher implementations. Compiler assisted masking [Moss et al., 2012] introduces a code repair system that is based on heuristics. A known set of inconsistencies are replaced with secure alternatives. SC-Masker [Eldib and Wang, 2014] is another tool that can automatically apply masking to an unprotected cipher. It achieves this by replacing code segments with functionally equivalent masked code segments. The masked code segments are generated straightforwardly when the functionality of a code segment is linear. When it is non-linear, an SMT solver based inductive

synthesis is used to find an optimal masked code segment.

Recall from the earlier discussion in this section that even theoretically secure masked implementations can still leak due to ILA breaching effects in hardware. Therefore, effective means of eliminating ILA breaches are needed. One such method is to use code rewrites that block the leaky interaction. Another method is to write software in an extended ISA that supports masking at an instruction level [Gao et al., 2021a]. Methods such as register reallocation and *random precharging* has been used in past literature [Bayrak et al., 2011; Tillich and Großschädl, 2007] for code rewrites. Random precharging repeats an instruction that is already a part of a leaky interaction with randomly selected operand values such that the overall program functionality is not affected. This resets the internal state that was stored earlier in the microprocessor [Tillich and Großschädl, 2007]. Seuschek et al. [2017] proposes leakage aware scheduling and register allocation, where instructions are scheduled so that the transition-based leakage between them are eliminated. FENL [Gao et al., 2020b] extends the ISA by additional instructions that clear internal states when called. However, such instructions need to be manually applied to an implementation.

2.4 Emulation based power analysis leakage evaluation

Emulation is the reproduction of exact or approximate output for a certain process without actually depending on it. An emulator is expected to produce the exact output or a similar one by any means possible. On other hand, simulation involves in modelling the internal states of the original process. Emulation and simulation are both used in modelling circuits due to the excessive costs in creating prototype circuits for each iteration of design changes. The cost reductions can also be applied to power analysis due to the physical experiments that are costly and tedious to perform without expertise in multiple fields. Simulation of electronic components can be done with a simulator called Simulation Program with Integrated Circuit Emphasis (SPICE) [Nagel and Pederson, 1973]. Although such a simulator can produce accurate power consumption for smaller circuits efficiently based on analog simulations [Regazzoni et al., 2009; Tiri and Verbauwhede, 2005; Kamel et al., 2014; Arribas et al., 2018], simulating larger circuits can become considerably slower. This

makes SPICE simulations comparably slower to their digital simulator counterparts.

Software leakage emulators generate power traces directly from the machine code and additionally from a description of the hosting hardware, albeit not as descriptive as what SPICE may require. The first such emulator was PINPAS [den Hartog et al., 2003] which was capable of emulating power traces for code that ran on smart-cards. Emulators based on profiling [Debande et al., 2012] emerged following the introduction of stochastic models for profiling based attacks by Schindler et al. [2005]. SILK was presented by Veshchikov [2014] as a leakage emulator based on high-level abstraction of the source code. This approach is ineffective in emulating ILA breaching effects because it only operates on the source code level. Architectural and micro-architectural effects that cause ILA breaches are not exposed in such emulations. Unlike SILK, SAVRASCA [Veshchikov and Guilley, 2017] is an emulator based on machine code and thus does not suffer from this issue. Additionally, it supports detection of register overwrite based leakage through Hamming distance leakage model for the new and old values for registers. SLEAK [Walters et al., 2014] is a leakage simulator based on the Gem5[1] simulator which simulates information leakage through Hamming weights of values stored in registers. ASCOLD [Papagiannopoulos and Veshchikov, 2017] successfully emulates micro-architectural effects of reading from and writing to memory. Although ASCOLD can be used to detect ILA breaches dependent on a micro-architecture, it is not an emulator based on a profiled model. Due to this, its results may include false positive leakages that are not observed in a real device. Additionally, it also requires the sensitive values to be tagged for its operation. ELMO [McCann et al., 2017] is a profiling based leakage emulator that was introduced for the ARM Cortex-M0 and M4 CPUs based on an existing ARM instruction emulator[2]. ELMO can emulate micro-architectural leakage based on neighbouring instructions. This kind of emulation can mimic micro-architectural effects of internal CPU pipeline register overwriting and generates power traces that closely match physical observations. ELMO is discussed in detail in Section 2.4.1. MAPS [Corre et al., 2018] extends this idea further by basing its power model entirely on the Hardware Description Language (HDL) source code for the ARM Cortex-M3 CPU core. This means that MAPS can track all values stored in internal registers

[1] https://www.gem5.org/
[2] https://github.com/dwelch67/thumbulator

during the execution of a program binary. Although not portable to other CPU architectures with unknown CPU core implementations, MAPS provides insight to the level of leakage information that could be extracted if processor implementation details were known. Gao and Oswald [2021] introduce a completeness test for leakage models aiming to improve accuracy of leakage emulators. This test uses collapsed models to determine if a certain model captures all leakage present in power traces. This completeness test can be carried out without access to hardware descriptions of a particular device. ABBY [Bazangani et al., 2021] is a Multi-Layer Perceptron (MLP) based power emulation model which is capable of generating comparable and even better results when compared to regression based emulation models such as ELMO. Recently introduced COCO [Gigerl et al., 2021] makes use of a hardware simulator to emulate leakage for software. It accomplishes this by emulating the software on hardware using Verilator [Snyder]. When supplied with the hardware description netlist, Verilator creates a software counterpart for it. This software counterpart can be compiled and run to simulate hardware that simulates a device made with the netlist. When Verilator is used with netlists of CPU cores, it is possible to simulate the entire functionality of a CPU. COCO feeds the masked software cipher implementation as input to this simulator along with annotated registers in the netlist. The annotations mark the registers which initially hold the shares. Another tool called REBECCA was used to determine whether the simulation leaked information or not. REBECCA [Bloem et al., 2018] is a formal verifier that can determine whether masked hardware implementations are insecure. ACA [Yao et al., 2020], SCRIPT [Nahiyan et al., 2020] and CASCADE [Sijacic et al., 2020] are similar tools that were recently introduced that use hardware descriptions as input.

2.4.1 ELMO

ELMO is a leakage emulator for the ARM Cortex family of processors [McCann et al., 2017]. In a nutshell, ELMO uses a power model that is based on Multiple Linear Regression (MLR) to emulate power traces of a given program that is executed on an ARM Cortex core. The profiling stage of ELMO adapts the model to the target device that it is supposed to emulate. Therefore, profiling should be performed before emulating any code on a new device. Afterwards, the profiled model is used to

generate the power traces for any given binary that is run through an ARM emulator. The specific emulator that is used in ELMO limits itself to the ARM Thumb 16 bit ISA.

ELMO's power model. The power model of ELMO is a linear expression that evaluates to the value of the differential voltage between the two terminals of a shunt resistor fitted across the power supply terminal of a certain microprocessor. This expression contains independent variables that take values of intermediate values from a running instance of the target program. The coefficients of this linear expression are determined by a linear regression using the Ordinary Least Squares (OLS) estimate. The measurement data is fitted to Equation 2.19 where $\boldsymbol{y}$ is a column vector of dependent variables (i.e. voltage across the shunt resistor), $\boldsymbol{\delta}$ is a column vector of constant terms (i.e. intercept terms) $\boldsymbol{X}$ is the matrix of independent variables (i.e. linearised[3] intermediate values from program) and $\boldsymbol{\epsilon}$ is a column vector of error terms. The estimated values in $\boldsymbol{\beta}$ and $\boldsymbol{\delta}$ are used in the final power model to estimate a value for the differential voltage.

$$\boldsymbol{y} = \boldsymbol{\delta} + \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon} \tag{2.19}$$

Profiling the model. The experiments conducted in the profiling stage provide measurements for the regression process. Empirically based classes of instructions were used instead of the actual instructions to reduce the number of actual instructions that are needed for the experiments. A reduced set of five representative instructions have been selected based on statistical evidence from empirical data. All triplet combinations of these five instructions ($5^3 = 125$) are used to gather measurements to profile the model. Each combination is run with 1000 different inputs. Five acquisitions for each input are averaged to lower independent noise [McCann et al., 2017].

The independent variables included in $\boldsymbol{X}$ are determined by the evaluation of nested models using the F-test. This process prevents the model from including terms

[3]In accordance with the assumptions of MLR, the independent variables are required to have a linear relationship to the dependent variable. When this is not the case, the independent variables are adjusted so that a linear relationship is formed

that do not have a significant contribution to the dependent variable ($\boldsymbol{y}$). It starts with a power model that has a few terms and is then tested against larger models by adding terms one by one. Observe that the nested model can be obtained by removing the newly added term. A basic model is listed in Equation 2.20 where $\boldsymbol{O_i}$ refers to ith operand value of the currently executing instruction and $\boldsymbol{T_i}$ refers to the bit transitions between ith operand values of the current instruction and previous instruction. The intuition behind starting with this base model is based on empirical evidence [McCann et al., 2016].

$$\boldsymbol{y} = \boldsymbol{\delta} + [\boldsymbol{O_1}|\boldsymbol{O_2}|\boldsymbol{T_1}|\boldsymbol{T_2}]\,\boldsymbol{\beta} + \boldsymbol{\epsilon} \tag{2.20}$$

Improving the model using nested models. The power model is further expanded to the model shown in Equations 2.21 and 2.22 by evaluation of nested models. The final ELMO power model depends on previous and subsequent instructions of the currently executing instruction. Two power model equations are used in ELMO due to different classes of instructions cluster their behaviour to two models. ELMO internally uses five different power models based on Equations 2.21 and 2.22 with constant terms and coefficients obtained for the five different instruction classes through MLR.

$\boldsymbol{A_i}, \boldsymbol{B_i}, \boldsymbol{C_i}$: Row vectors of i^{th} operand of the previous, current, and subsequent instructions $(i \in \{1, 2\})$

$\boldsymbol{I_p}, \boldsymbol{I_s}$: Row vectors of dummy variables for representing the type of previous and subsequent instruction

$\boldsymbol{e_i}$: Column vector with i^{th} row element set to 1 and all other values set to 0

y : Differential voltage

$\boldsymbol{\beta}$: Column vector of linear regression coefficients

δ : Intercept

$HW(\boldsymbol{x})$: Hamming weight of $\boldsymbol{x}$

$[X|Y]$: Concatenation of $\boldsymbol{X}$ and $\boldsymbol{Y}$

$$\boldsymbol{T}_i = \boldsymbol{A}_i \oplus \boldsymbol{B}_i$$

$$\boldsymbol{H_{i,j}} = [\boldsymbol{B}_1\boldsymbol{e}_i \times \boldsymbol{B}_1\boldsymbol{e}_j | \boldsymbol{B}_2\boldsymbol{e}_i \times \boldsymbol{B}_2\boldsymbol{e}_j | \boldsymbol{T}_1\boldsymbol{e}_i \times \boldsymbol{T}_1\boldsymbol{e}_j | \boldsymbol{T}_2\boldsymbol{e}_i \times \boldsymbol{T}_2\boldsymbol{e}_j]$$

$$\boldsymbol{D} = [\boldsymbol{B}_1 | \boldsymbol{B}_2 | \boldsymbol{T}_1 | \boldsymbol{T}_2]$$

$$\boldsymbol{D}_k = [HW(\boldsymbol{B}_1)\boldsymbol{I}_k | HW(\boldsymbol{B}_2)\boldsymbol{I}_k | HW(\boldsymbol{T}_1)\boldsymbol{I}_k | HW(\boldsymbol{T}_2)\boldsymbol{I}_k], \ (k \in \{s, p\})$$

$$\begin{aligned}\boldsymbol{H} = [&\boldsymbol{H_{1,2}} | \boldsymbol{H_{1,3}} | \boldsymbol{H_{2,4}} | \ldots | \boldsymbol{H_{1,32}} | \\ &\boldsymbol{H_{2,3}} | \boldsymbol{H_{2,4}} | \ldots | \boldsymbol{H_{2,32}} | \\ &\ldots | \boldsymbol{H_{i,i+1}} | \ldots | \boldsymbol{H_{i,32}} | \\ &\ldots | \boldsymbol{H_{32,32}}]\end{aligned}$$

$$y = \delta + [\boldsymbol{I}_p | \boldsymbol{I}_s | \boldsymbol{D} | \boldsymbol{D}_p | \boldsymbol{D}_s] \boldsymbol{\beta} \tag{2.21}$$

$$y = \delta + [\boldsymbol{I}_p | \boldsymbol{I}_s | \boldsymbol{D} | \boldsymbol{D}_p | \boldsymbol{D}_s | \boldsymbol{H}] \boldsymbol{\beta} \tag{2.22}$$

Chapter 3

Overview

This chapter presents a high-level design for the leakage mitigation workflow and the physical experiment setup used in experiments done in this thesis. The leakage mitigation workflow is intended to be used by side-channel leakage evaluators find and fix power analysis based leakage.

3.1 Design

We suggest an automated emulator driven workflow that augments an evaluator's tool set in producing secure cipher implementations. Traditionally, leaky cipher implementations were secured through a manual process where human effort was required in both evaluation and implementation of countermeasures.

Our workflow is shown in Figure 3.1. It aims to eliminate leakage stemming from unintentional ILA breaching effects in masked implementations. ELMO* is a modified version of ELMO presented in McCann et al. [2017]. We selected ELMO as the basis of our work since ELMO had the most realistic reproduction of leakage effects with comparison other solutions at the start of our project. ELMO* can emulate leakage from additional effects such as memory bus and register overwrite leakage that were missing from ELMO. Rosita and Rosita++ are code rewriting frameworks presented in Chapters 5 and 6 which can detect root causes of leakage and apply code fixes to mitigate them automatically. The code fixes are applied until all detected leakage converges to zero at a specified trace count level. Although the final implementation has a significantly reduced amount of leakage, the final implementation is required to be evaluated on hardware due to differences between

emulated leakage and the leakage from the real device.

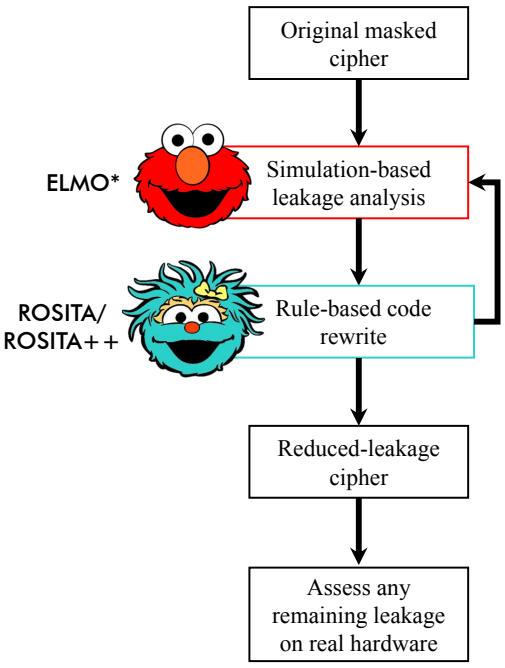


Figure 3.1: ROSITA and ROSITA++ workflow.

3.2 Experiment setup

We detail the steps we used for setting up an experiment for evaluation of a real device in this section. Single core devices are preferred to multicore devices as their power consumption could be trivially mapped to what happens on the CPU core at a given time. However, this does not mean that power analysis cannot be used on multicore devices. Memory caches, CPU pipelines and out-of-order execution affect power analysis negatively by making it harder to reveal what is executing on the CPU at a given time. Therefore, by choosing a simpler embedded system, many of the technical difficulties are mitigated.

First, a measurement probe should be attached into the power rail of the CPU to collect power consumption samples. The most common method of achieving this is through a shunt resistor. To measure the voltage across the resistor, we used a specialised probe called a differential probe. Differential probes are active devices that detect the voltage at their input and produce a proportional voltage difference

at their output. A non-invasive current measuring probe could also be used for the same purpose.

Second, we used an oscilloscope collect the power samples. The differential probe was fitted to the oscilloscope. The trigger of the oscilloscope was set up such that it triggers right before the code segment under test is executed. As we had control of the software that ran on the device under test, the trigger was introduced from the code. Otherwise, a repeating power value pattern should be found so that the start of the code segment can be determined. We set the sampling rate of the oscilloscope to a value several times higher than the clock rate of the device under test to capture a detailed signal. The captured signals are referred to as *power traces*.

Figure 3.2 shows a photograph of the measurement setup used for the experiments described in this work. The physical device used for all tests in this work is the STM32F030 Discovery evaluation board by ST Microelectronics. The ARM Cortex-M0 CPU in the STM32F030 can run at a clock rate of 48 MHz. To capture detailed signals at a moderate speed, we reduced it to 8 MHz as was used by McCann et al. [2017]. Similarly, we disconnected one of the two power inputs of the System on Chip (SoC) and attached a 330 Ω shunt resistor to the second power input, following the experiment setup in McCann et al. [2017]. Additionally, to avoid switching noise, we used batteries to power the evaluation board. A PicoScope 6404D oscilloscope was used with a Pico Technology TA046 differential probe connected to the oscilloscope via a Langer PA 303 preamplifier, to measure the voltage drop across the shunt resistor as a proxy for the power consumption of the SoC. We sampled every 12.8 ns, which, with a clock rate of 8 MHz, is roughly 9.77 samples per clock cycle. The samples are 8 bit wide and our PicoScope can store up to two billion samples before running out of memory.

To orchestrate the experiments we used a PC that controls both the oscilloscope and the evaluation board. The PC generated all of the randomness needed by the experiments and communicated it to the evaluation board via a serial connection. Communication was carried out in batches to speed up the process. A diagram of this setup is shown in Figure 3.3. C_i and P_i denote the flow of ciphertexts and plaintexts. All randomness used in the experiments was taken from `/dev/urandom`, which is considered to be cryptographically secure. The GPIO pins of the evaluation board were used to trigger the collection of traces and adequate padding was added near

the triggers to improve the signal.

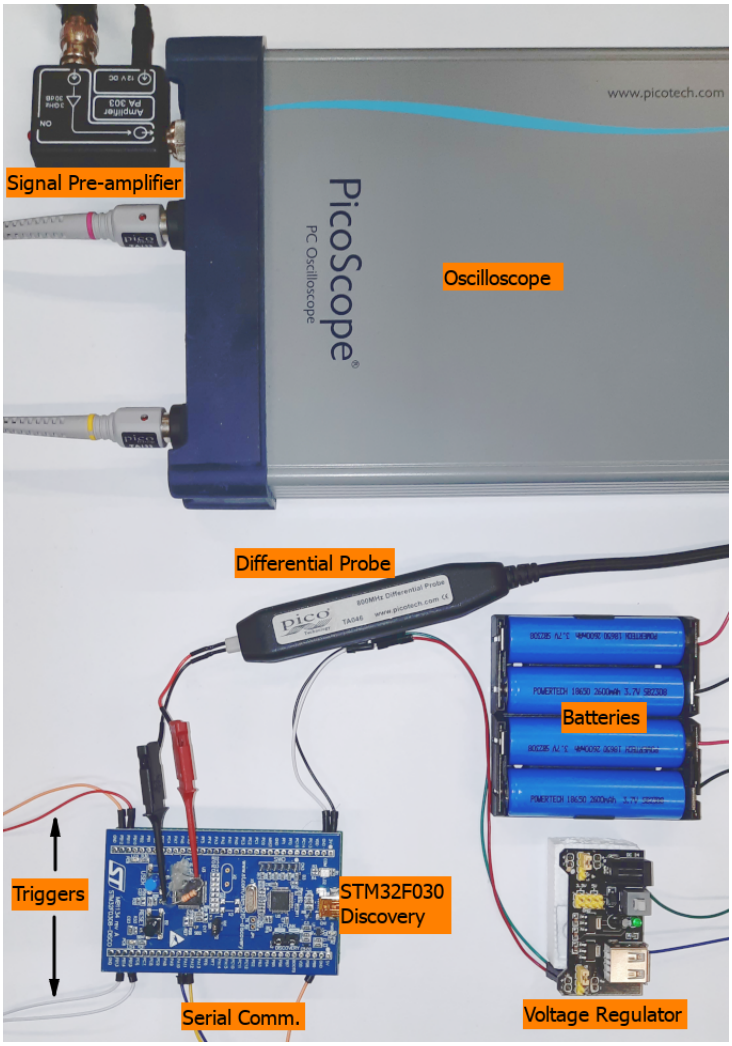


Figure 3.2: The measurement setup used in this work.

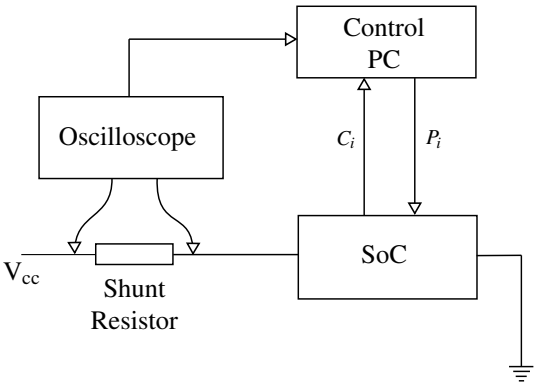


Figure 3.3: A diagram of the measurement setup.

Chapter 4

Towards better power analysis leakage emulation

ELMO [McCann et al., 2017] emulates the voltage between the terminals of a shunt resistor fitted across the power pin of an ARM CPU for a given software implementation based on the input values of three consecutive instructions at a time. The generated power traces were analysed with TVLA to generate leakage traces that were used to localise the leaky instructions. This offers a mechanism to correct leakage in cipher implementations free of physical experiments. However, the model must be able to reliably emulate leakage observed in a real device so that it could be effectively used in leakage detection. Our experiments with ELMO showed discrepancies between the emulated and the real leakage. Our experiment setup is based on the STM32F030 Discovery Evaluation board and details of the experiment setup are discussed at Section 3.2. Figure 4.1 shows the t-values from a fixed vs. random t-test for the code segment shown in Listing 4.1 that implements a 2-share Boolean masking scheme. `r1` and `r4` were initialised with one share each. `r2` was initialised with a writable memory location. `r3` and `r7` were initialised with unrelated values. The results in Figure 4.1 show that ELMO's power model is insufficient to model leakage between `str` and `eors` instructions.

```
1    str r1, [r2]
2    movs r7, r7
3    movs r7, r7
```

```
4     movs r7, r7
5     movs r7, r7
6     movs r7, r7
7     eors r3, r4
```

Listing 4.1: Evaluating interactions between the `str` and the `eors` instructions.

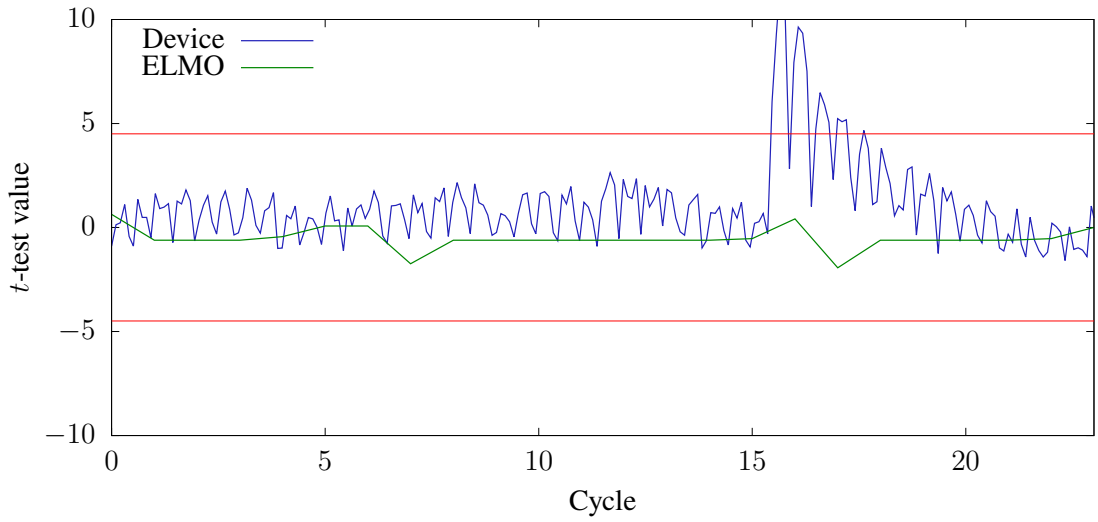


Figure 4.1: t-test results for Listing 4.1.

As evident from Figure 4.1, the CPU leaks information through interactions between instructions that have executed many cycles before and the currently executed one. Experiments showed that such effects prevailed even when the gap between the instructions were larger than the CPU pipeline size. Further experimentation showed that many instructions (e.g. ALU and load/store instructions) interacted in a similar way.

This section discusses the causes for discrepancies between ELMO and results from the real device. We then propose modifications to ELMO such that these missing leakages are emulated realistically. Finally, we evaluate the new additions using the F-test based nested model evaluations that were also used in McCann et al. [2017].

4.1 Leakage causes

Information leakage happens due to the following leakage causes as demonstrated in previous literature. Most prominent leakage types are value-based and transition-based leakages [Balasch et al., 2015].

4.1.1 Value-based leakage

Value-based leakage is observed when the power consumption of a device at a single sample point correlates with a single intermediate value. The Hamming weight of an intermediate value is an example for value-based leakage. The value of a single intermediate value is what leaks.

4.1.2 Transition-based leakage

Leakage caused due to interaction between two intermediate values are called transition-based leakage. Transition-based leakage is also important for leakage emulation as their existence further reduces the security order of a cipher implementation. Balasch et al. [2015] theoretically proved that existence of transition-based leakage reduces the security of an implementation from dth-order to $\lfloor \frac{d}{2} \rfloor$-order. Additionally, recent practical evaluations done by Gao et al. [2020a] show that the reduction can be much larger than just a factor of two. Such effects drastically reduce the work that an attacker needs to carry out to recover sensitive information. The reason for this is that a single sample point holds information from multiple intermediate values when run on a CPU with transitional interactions. If run on a hypothetical CPU without such interactions, an attacker needs to combine power values from all individual sample points to gain the same information. We note that ELMO does support a specific type of transitional effect based leakage through its three consecutive instruction based model as it considers the Hamming distance between the operands of previous and current instructions.

We describe below in detail the types of transition-based leakage that we found to be not supported by ELMO. We later continue to modify ELMO so that these effects are adequately emulated.

4.1.2.1 Internal state related leakage

When registers that hold intermediate values that are not cleared when new intermediates are processed, they contribute to transition-based leakage between the two intermediates. Updating the value of a register causes power consumption that correlates with the Hamming distance between the two intermediate values, as explained in Section 2.3.3. Such interactions happen through the internal registers within the CPU pipeline, or through any other means of holding intermediate values from previously executed instructions. Generally, this work refers to such leakage as *internal state* related leakage.

We designed an experiment to search for such leakage on our test device. In this experiment, we selected two instructions from the instruction set and tried to find interactions between them. The experiment runs the example test code segment shown in Listing 4.1. Prior to running the code segment, registers `r1`, `r3`, `r4` and `r7` were loaded with unrelated random values and `r2` was loaded with an address that pointed to a random value. A set of padding instructions were used to guarantee separation of the instructions so that any leakage that is observed is not related to known interactions that happen within the CPU pipeline (i.e. interactions between instructions that run on consecutive cycles). `movs r7, r7` was used as the padding instruction. We used a custom no-op instruction because it takes the same time to execute as any other `movs` instruction. All inputs (i.e. all register values) to the device were recorded and 10000 power traces were collected. The experiment setup is detailed in Section 3.2. Then, a Correlation Power Analysis (CPA) attack was run against the measurement values that were gathered. We evaluated Hamming distances of different pairs of inputs. We observed a high correlation was observed between the Hamming distance of values set to `r4` and `r1` and the power samples collected at the `eors` instruction. The correlation graph shown in Figure 4.2 shows highest correlation at cycle 25, this is when the `eors` instruction was executed.

This experiment confirms that some internal state has been set by the `str` instruction and when the `eors` instruction ran, and that its second operand interacted with the previously set internal state. By changing the two instructions at top and bottom of the code segment listed at Listing 4.1 we confirmed that such effects are widely prevalent in ALU instructions and in instructions related to memory bus operations. We continued running a fixed vs. random t-tests based on the same code

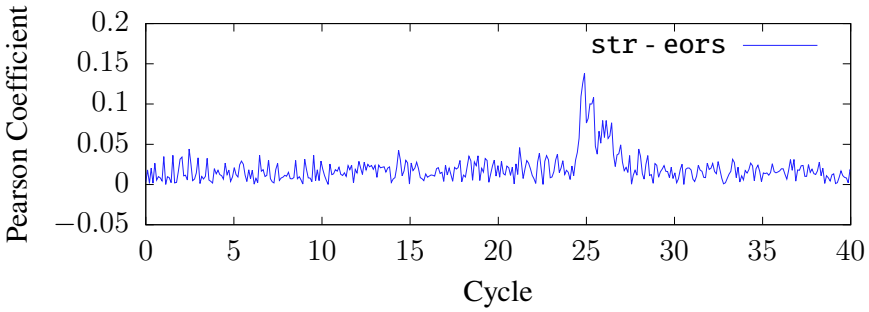


Figure 4.2: Pearson correlation coefficient for the internal state test.

segment on all possible pairs of instructions between a subset of the ARM Thumb instruction set. Table 4.1 shows the results that we obtained for these experiments (i.e. whether the leakage was significant or not), the column wise instructions are the first instruction and row wise ones are the second instruction with respect to the code listing at Listing 4.1. We initially used 10000 power traces for this TVLA procedure. The tests that result in no significant leakage were retested with 50000 traces to increase the confidence of the lack of leakage.

These observations lead us to believe that state related effects are predominantly happening due to a small subset of internal registers that are shared by several instructions. For example, all ALU instructions show similar leakage characteristics with other instructions. Different versions of `str` and `ldr` also showed similar characteristics. We recognised that there are at least four distinct state interaction types as highlighted in Table 4.1. The colours for highlighting were selected according to the interactions that an instruction from each row participates in. Through this method, we recognise that there are four distinct classes of internal state setting instructions which are `eors` (representative of all ALU instructions), `str`, `ldr` and `mov`.

4.1.2.2 Effects related to the memory bus

When loading from or storing to memory, the value of the storage element is overwritten internally, leaking the Hamming distance between the previous and the new value. Consequently, when consecutively writing to or reading from memory, care should be taken to only access non-secret values or unrelated secret values. We

	eors	adds	ands	bics	cmps	mov	orrs	subs	lsls	rors	lsrs	str	strb	strh	ldr	ldrb	ldrh	muls	pop	push
adds	●	●	●	●	●	●	●	●	●	●	●	●	●	●					●	●
ands	●	●	●	●	●	●	●	●	●	●	●	●	●	●					●	●
eors	●	●	●	●	●	●	●	●	●	●	●	●	●	●					●	●
bics	●	●	●	●	●	●	●	●	●	●	●	●	●	●					●	●
cmps	●	●	●	●	●	●	●	●	●	●	●	●	●	●					●	●
mov	●	●	●	●	●	●	●	●	●	●	●	●	●	●						
orrs	●	●	●	●	●	●	●	●	●	●	●	●	●	●					●	●
subs	●	●	●	●	●	●	●	●	●	●	●	●	●	●					●	●
lsls	●	●	●	●	●	●	●	●	●	●	●	●	●	●					●	●
rors	●	●	●	●	●	●	●	●	●	●	●	●	●	●					●	●
lsrs	●	●	●	●	●	●	●	●	●	●	●	●	●	●					●	●
muls	●	●	●	●	●	●	●	●	●	●	●	●	●	●					●	●
str	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●		●	●
strb	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●		●	●
strh	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●		●	●
ldr												●	●	●	●	●	●		●	●
ldrb												●	●	●	●	●	●		●	●
ldrh												●	●	●	●	●	●		●	●
pop	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●		●	●
push	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●		●	●

Table 4.1: State interactions between the second operands of instruction pairs. The circles which pairs of instructions interacted through internally stored state.

note that the storage element could be the contents of the addressed memory itself, where the power leakage correlates with changing the contents of the memory bus.

It is important to note that the storage element always stores a 32 bit word. Thus, when loading or storing a byte, the whole 4 byte aligned 32 bit word that contains the byte is moved to the storage element. This may create memory interaction between memory operations that seem completely unrelated. For example, consider the code in Listing 4.2. In this example we assume that memory locations 0x300 and 0x400 both contain one share each from a 2-share Boolean masked implementation. The code in this example performs two memory operations, the first stores a byte into

```
1  movs r3, 0x303
2  movs r4, 0x402
3  movs r7, r7
4  movs r7, r7
5  movs r7, r7
6  movs r7, r7
7  movs r7, r7
8  strb r5, [r3]
9  movs r7, r7
10 movs r7, r7
11 movs r7, r7
12 movs r7, r7
13 movs r7, r7
14 ldrb r6, [r4]
```

Listing 4.2: Example of word interaction

address 0x303 and the second reads a byte from location 0x402. We note that none of these locations contains secret data, and the data stored is also not secret. However, the store operation loads the 32 bit word in memory locations 0x300–0x303 into the memory bus, and the following load operation replaces the contents with the 32 bit word in memory location 0x400–0x403. This causes an interaction between the data in memory locations 0x300 and 0x400, leaking the Hamming distance between the values stored in these locations. We believe that this behaviour is explained by glitching effects in the ARM memory system. Since individual bytes are combined to make the requested data word in the ARM memory system, glitching effects in the combination circuitry may cause them to interfere with each other [Furber, 2000, Figure 8.1].

Value-based leakage in memory operations. Another issue found in the memory bus is the interaction between the bytes of words loaded from or stored to memory. Specifically, our analysis shows that when memory data is accessed, consecutive bytes in the word interact with each other. Thus, if a word contains multiple bytes that are all masked with the same mask, loading it from or storing it to memory will leak the Hamming distance between consecutive bytes. We note that due to the

memory bus storage element described above, the leakage occurs even if the memory access operations access a single byte of a 32 bit word.

Store Latch. We also found that storing a value of a register to memory results in potential interactions between the value of that register and the second argument of subsequent ALU instructions, such as `eors`. However, if the contents of the register changes between the `str` and the ALU instruction, the second argument of the ALU instruction interacts with the *updated* value of the register rather than with its original value.

```
1  str  r5, [r3]
2  movs r7, r7
3  movs r7, r7
4  movs r7, r7
5  movs r7, r7
6  movs r7, r7
7  movs r5, r2
8  movs r7, r7
9  movs r7, r7
10 movs r7, r7
11 movs r7, r7
12 movs r7, r7
13 eors r1, r4
```

Listing 4.3: Store latch example.

For example, the code in Listing 4.3 stores the value of `r5` to memory (Line 1). It then updates the value of `r5`, moving the contents of `r2` to it (Line 7). Finally, it calculates the exclusive-or of `r1` and `r4`. Our experiments show leakage at Line 13, which correlates with the Hamming distance between the original values of `r2` and `r4`. Interestingly, we note that the update of the interacting register takes one cycle to become effective. That is, removing Lines 8–12 in Listing 4.3 removes the interaction between the original values of `r2` and `r4`, but leaves an interaction between the original values of `r5` and `r4`.

```
1 ...
2 movs r7, r7
3 movs r7, r7
4 movs r1, r3
5 movs r7, r7
6 movs r7, r7
7 ...
```

Listing 4.4: Code segment with overwrite effects

We believe that the processor maintains a reference to the most recently stored register. This reference is used as an input to a multiplexor that selects the contents of the referenced register. We believe that a glitch on the bus causes interference between the contents of referenced register and the second argument of subsequent instructions, explaining the leakage we observe [Furber, 2000, Figure 4.6].

Overwrite effects. When register values are being overwritten with new values the Hamming distance between the old value and the new value is leaked [Papagiannopoulos and Veshchikov, 2017]. It was possible to reproduce these effects in our experimental setup by conducting a CPA attack on 1 million power traces that were gathered from our experiment setup (see Section 3.2 for details on setup). The code segment in Listing 4.4 was used as the victim. `r1` and `r3` were loaded with one share each from a 2-share Boolean masked scheme. The CPA attack was carried out on the Hamming distance between the two shares which would mean that the two shares were combined. The resulting Pearson's correlation coefficients plot is shown in Figure 4.3.

Inter-bus effects in the CPU pipeline. According to ELMO's power model, it is known that the register values of two neighbouring instructions interact in order (i.e. the neighbouring instructions' operand values interact when they use the same bus) which results in the leakage of their Hamming distance. Our tests show that inter-bus related interactions can also happen. We conducted a CPA attack on the code segment in Listing 4.5 by collecting 50000 power traces on our experiment setup (see Section 3.2 for details on setup). We loaded `r3` and `r1` with the shares of

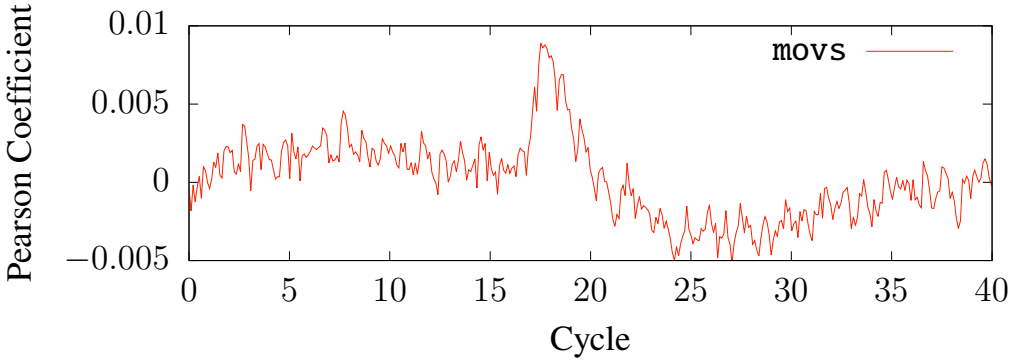


Figure 4.3: Pearson correlation coefficient for overwrite effect tests.

```
1 ...
2 movs r7, r7
3 movs r7, r7
4 ands r4, r3
5 ands r1, r2
6 movs r7, r7
7 movs r7, r7
8 ...
```

Listing 4.5: Code segment with potential interbus interaction

a 2-share Boolean masking scheme. `r4` and `r2` were initialised with unrelated values. The Pearson's correlation coefficient graph is shown in Figure 4.4. Hence, the first operand of the first instruction also interacts with the second instruction's second operand and vice-versa. Therefore we conclude that all possible combinations of the operand values of two neighbouring instructions can lead to leakage.

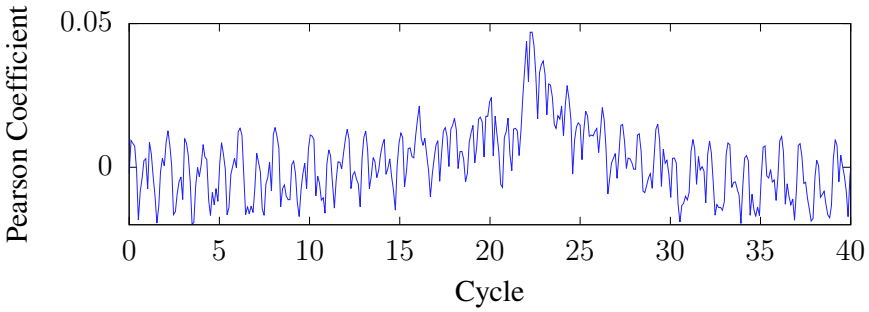


Figure 4.4: Pearson correlation coefficient for the cross operand test.

4.2 Extending the ELMO power model

The previous section discusses limitations of ELMO. This section aims to extend ELMO so it can be modified to increase its accuracy. Recall from Section 2.4.1 that ELMO's power emulation is based on a Multiple Linear Regression (MLR) model that is profiled initially by a set of real traces. The model is profiled on five representative instructions instead of running the test for all possible combinations of instructions. The five representative instructions (i.e. `eors`, `lsls`, `str`, `ldr` and `muls`) are used to create combinations of three consecutive instruction triplets. The six operands used in all of them are set with random values in the profiling stage. It is assumed that all instructions are two operand instructions to ease model building. The power traces gathered from running these code segments on the real device are used to profile the ELMO model.

The power model of ELMO is built by adding new terms to a base model. The new models are evaluated with the F-tests based nested model evaluation methodology to guarantee that the newly added terms are more explanatory than a previous model. The original ELMO model shown in Equations 4.1 and 4.2, is profiled by only using five distinct instructions to generate the real traces. These are `eors`, `lsls`, `str`, `ldr` and `muls`. This was done to reduce the huge number of combinations of instructions if all 21 instructions have been used. These representative instructions have been selected by conducting statistical clustering of the leakage that is shown by each instruction [McCann et al., 2017]. The model that is fitted to `eors`, `str` and `ldr` is shown in Equation 4.1, and the model that is fitted to `lsls` and `muls` is Equation 4.2.

Matrices and row vectors are denoted by uppercase Latin letters and column vectors by lowercase letters.

$\boldsymbol{A_i}, \boldsymbol{B_i}, \boldsymbol{C_i}$: Row vectors of i^{th} operand of the previous, current, and subsequent instructions $(i \in \{1, 2\})$

$\boldsymbol{I_p}, \boldsymbol{I_s}$: Row vectors of dummy variables for representing the type of previous and subsequent instruction

$\boldsymbol{e_i}$: Column vector with i^{th} row element set to 1 and all other values set to 0

y : Differential voltage

$\boldsymbol{\beta}$: Column vector of linear regression coefficients

δ : Intercept

$HW(\boldsymbol{x})$: Hamming weight of $\boldsymbol{x}$

$[X|Y]$: Concatenation of $\boldsymbol{X}$ and $\boldsymbol{Y}$

$$
\begin{aligned}
\boldsymbol{T}_i &= \boldsymbol{A}_i \oplus \boldsymbol{B}_i \\
\boldsymbol{H_{i,j}} &= [\boldsymbol{B}_1\boldsymbol{e}_i \times \boldsymbol{B}_1\boldsymbol{e}_j | \boldsymbol{B}_2\boldsymbol{e}_i \times \boldsymbol{B}_2\boldsymbol{e}_j | \boldsymbol{T}_1\boldsymbol{e}_i \times \boldsymbol{T}_1\boldsymbol{e}_j | \boldsymbol{T}_2\boldsymbol{e}_i \times \boldsymbol{T}_2\boldsymbol{e}_j] \\
\boldsymbol{D} &= [\boldsymbol{B}_1 | \boldsymbol{B}_2 | \boldsymbol{T}_1 | \boldsymbol{T}_2] \\
\boldsymbol{D}_k &= [HW(\boldsymbol{B}_1)\boldsymbol{I}_k | HW(\boldsymbol{B}_2)\boldsymbol{I}_k | HW(\boldsymbol{T}_1)\boldsymbol{I}_k | HW(\boldsymbol{T}_2)\boldsymbol{I}_k], \ (k \in \{s, p\}) \\
\boldsymbol{H} &= [\boldsymbol{H_{1,2}} | \boldsymbol{H_{1,3}} | \boldsymbol{H_{2,4}} | \ldots | \boldsymbol{H_{1,32}} | \\
&\qquad \boldsymbol{H_{2,3}} | \boldsymbol{H_{2,4}} | \ldots | \boldsymbol{H_{2,32}} | \\
&\qquad \ldots | \boldsymbol{H_{i,i+1}} | \ldots | \boldsymbol{H_{i,32}} | \\
&\qquad\qquad \ldots | \boldsymbol{H_{32,32}}]
\end{aligned}
$$

$$y = \delta + [\boldsymbol{I}_p | \boldsymbol{I}_s | \boldsymbol{D} | \boldsymbol{D}_p | \boldsymbol{D}_s] \, \boldsymbol{\beta} \tag{4.1}$$

$$y = \delta + [\boldsymbol{I}_p | \boldsymbol{I}_s | \boldsymbol{D} | \boldsymbol{D}_p | \boldsymbol{D}_s | \boldsymbol{H}] \, \boldsymbol{\beta} \tag{4.2}$$

Below we describe a set of new terms that extend Equations 4.1 and 4.2 so that the effects discussed in Section 4.1.2 are emulated by the ELMO power model.

$$\boldsymbol{O} = [HW(\boldsymbol{B}_1 \oplus \boldsymbol{B}_2)] \tag{4.3}$$

$$\boldsymbol{X} = [HW(\boldsymbol{A}_1 \oplus \boldsymbol{B}_2)|HW(\boldsymbol{A}_2 \oplus \boldsymbol{B}_1)] \tag{4.4}$$

$$\boldsymbol{S} = [\boldsymbol{B}_2 \oplus \boldsymbol{S}_0|\boldsymbol{B}_2 \oplus \boldsymbol{S}_1|\boldsymbol{B}_2 \oplus \boldsymbol{S}_2|\boldsymbol{B}_2 \oplus \boldsymbol{S}_3] \tag{4.5}$$

We introduced terms for overwrite effects related leakage ($\boldsymbol{O}$), inter-bus effects leakage ($\boldsymbol{X}$) and set of four new terms for state related leakage ($\boldsymbol{S}_i, i \in [0, 3]$) as shown in Equations 4.3 to 4.5. The value for $\boldsymbol{S}_i$ is set according to the last instruction that had set some internal state (see Section 4.1.2.2). Empirical testing showed that the leakage from the state set by the first operand ($\boldsymbol{B}_1$) was insignificant. Therefore, only the second operand ($\boldsymbol{B}_2$) is considered for the power model.

Representing `movs` in the model. Our experiments showed that the `movs` instruction is not adequately represented in ELMO even though it is quite frequently used in cipher implementations. `movs` is represented as `lsls rd, rn, 0x0` in ARM Thumb 16 bit ISA. In the original ELMO model, this instruction is not emulated. Due to the significant overwrite effects observed in `movs` instruction (shown in Figure 4.3), it is required to be emulated. However, adding a new instruction group will increase the total number of distinct instructions to six and the total number of combinations would be $6^3 = 216$. This nearly doubles the number of combinations that need to be considered. Therefore, we used a separate instruction group for `movs` by moving `muls` to the `eors` instruction group. This was done primarily to reduce modelling overhead while keeping a reasonable approximate for `muls`.

Changes in the profiling stage. The introduction of new terms necessitated some changes in the profiling stage. Originally, ELMO was profiled by collecting measurements for runs of combinations of three consecutive instructions. There are 125 such combinations as discussed in Section 2.4.1. Since a term for internal state ($\boldsymbol{S}$) is added, these 125 combinations are required to be collected once per each kind of state that was detected in Table 4.1. This means any previous state should be reset and specific state should be set before running each of the 125 combinations

of instructions. Originally, 1000 inputs were used with each repeated for five times and then averaged to reduce effects of noise. Data was collected under all possible state configurations with the four different kinds of state discovered from Table 4.1. The MLR is then profiled using the measurements as input to the dependent variable and the register values from the instruction emulator as input to the independent variables.

Nested model evaluations. From earlier discussion it follows that the effects in Equations 4.3 to 4.5 should be added to Equations 4.1 and 4.2. We use the F-test based nested model evaluations to measure increases or decreases of the explanatory power of these equations after adding new terms. Considering two models where a reduced model (R) is nested within the full model (F), the F-statistic is calculated as follows,

$$F = \frac{\left(\frac{RSS_R - RSS_F}{p_F - p_R}\right)}{\left(\frac{RSS_F}{n - p_F}\right)}$$

Where RSS_x is the residual sum of squares of the model x, p_x is the number of parameters of model x ($p_R < p_F$) and n is the sample size used to profile the model. The null hypothesis for the F-test is that the added terms have no effect. Therefore, the quantity F follows an F-distribution with $(p_F - p_R, n - p_F)$ degrees of freedom. The original power model of ELMO proposed Equation 4.1 to emulate power for `eors`, `str` and `ldr` instructions and Equation 4.2 to emulate power for `lsls` and `muls` instructions[1]. The results we obtained for nested F-tests are shown in Table 4.2 and Table 4.3. We used a significance level of $\alpha = 0.05$ similar to McCann et al. [2017]. The first column of Tables 4.2 and 4.3 describes the added parameters for each F-test. A comma separates the reduced model and the full model. The common parameters for both reduced and full models are not shown to increase clarity. Table 4.2 shows the F-test results for extending Equation 4.1 and Table 4.3 shows results of F-tests done for extending Equation 4.2.

Extending Equation 4.1 results in two equations due to the addition of $\boldsymbol{X}$ failing to reject the null hypothesis for `ldr` at a significance level of $\alpha = 0.05$. The null hypothesis is rejected for `eors` and `str` instructions with high significance. This

[1] https://github.com/sca-research/ELMO/blob/master/ModelBuildingCode/ReleaseModelMethod.m

results in Equation 4.6 for emulating power for `ldr` instruction and Equation 4.7 emulating power for `eors` and `str` instructions.

Tested effects (df1,df2)	`eors`	`str`	`ldr`
,$\boldsymbol{S}$ (128,624703)	219.857	244.238	213.403
$\boldsymbol{S}$,$\boldsymbol{S}\|\boldsymbol{O}$ (1,624702)	186.313	146.995	9.733
$\boldsymbol{S}\|\boldsymbol{O}$,$\boldsymbol{S}\|\boldsymbol{O}\|\boldsymbol{X}$ (2,624700)	559.864	78.991	2.653

Table 4.2: F-statistics for nested model evaluations on Equation 4.1.

Extending Equation 4.2 results in the rejection of the null hypothesis for all tested effects with high significance as shown in Table 4.3. Therefore `lsls` and `movs` instructions are emulated by Equation 4.8.

Tested effects (df1,df2)	`lsls`	`movs`
,$\boldsymbol{S}$ (128,623711)	207.857	275.540
$\boldsymbol{S}$,$\boldsymbol{S}\|\boldsymbol{O}$ (1,623710)	120.799	33.424
$\boldsymbol{S}\|\boldsymbol{O}$,$\boldsymbol{S}\|\boldsymbol{O}\|\boldsymbol{X}$ (2,623708)	490.775	1400.513

Table 4.3: F-statistics for nested model evaluations on Equation 4.2.

$$y = \delta + [\boldsymbol{I}_p|\boldsymbol{I}_s|\boldsymbol{D}|\boldsymbol{D}_p|\boldsymbol{D}_s|\boldsymbol{S}|\boldsymbol{O}]\,\boldsymbol{\beta} \tag{4.6}$$

$$y = \delta + [\boldsymbol{I}_p|\boldsymbol{I}_s|\boldsymbol{D}|\boldsymbol{D}_p|\boldsymbol{D}_s|\boldsymbol{S}|\boldsymbol{O}|\boldsymbol{X}]\,\boldsymbol{\beta} \tag{4.7}$$

$$y = \delta + [\boldsymbol{I}_p|\boldsymbol{I}_s|\boldsymbol{D}|\boldsymbol{D}_p|\boldsymbol{D}_s|\boldsymbol{H}|\boldsymbol{S}|\boldsymbol{O}|\boldsymbol{X}]\,\boldsymbol{\beta} \tag{4.8}$$

Figure 4.5 shows that ELMO* is capable of emulating leakage realistically for Listing 4.1.

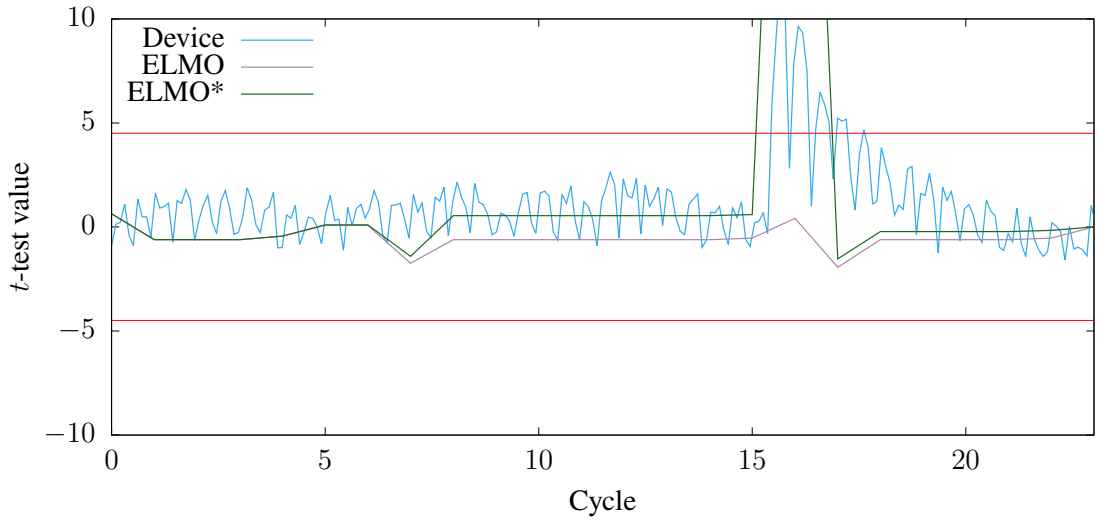


Figure 4.5: t-test results for Listing 4.1 with ELMO*.

Chapter 5

Mitigation of univariate leakage

This chapter aims to eliminate univariate leakage by automatically applying code fixes such that unintended ILA breaches are eliminated. In doing so, we introduce a methodology to find root causes of univariate leakage. This lets our code rewriting tool Rosita know where and how to apply code fixes.

Univariate leakage is leakage that can be explained using univariate statistical analysis. Univariate statistics model data using a single variable [Kachigan, 1986]. Explanation of leakage is typically done under a hypothetical power model. A power model describes the relation between some input value(s) and the instantaneous power consumed by a device when processing those inputs [Mangard et al., 2005]. This means that the cause(s) for a leakage can only be described relative to the inputs used in a specific power model. Given that an evaluator has access to where in code specific information is originating from, they are able to apply corrective fixes such that the leakage is eliminated.

TVLA can be used to detect specific points of leakage that produce leakage through the power traces generated from a power model such as ELMO*. Given that all input for ELMO* is supplied through an ARM instruction emulator that executes a masked program, the flow of information through register values to the power model can be traced back to code. By combining knowledge from the instruction emulator and ELMO* we claim that it is possible to find root causes for leakage using only the emulator's execution trace and the TVLA results.

We further claim that detected leakage can be fixed by using code patterns from a library of patterns only if the leakage is caused by *unintended interactions*. We

define unintended interactions in the context of masked software implementations as interactions between the register values of a masked software implementation which are not part of the algorithm. Specifically, the intention in this context is defined by the functionality of the assembly code of an implementation, rather than the intention of the programmer. This means that the values stored in registers by an implementation must contain at least one share less than the total number of shares of the masked implementation. If any such value contains all the shares, it means that the implementation is an insecure masked implementation. Possible causes for such scenarios are programmer error and compiler optimisations. This chapter discusses how to find root causes for univariate leakage through the use of ELMO* and how to apply countermeasures to such leakage.

5.1 Discovery of root causes using ELMO*

ELMO* emulates the power consumption of a target device at the execution of each instruction of a program. The differential voltage (v) across a shunt resistor across the power rail of a CPU is approximated by the following simplified multiple linear regression model shown in Equation 5.1. Independent variables, and coefficients are represented by x_i, β_i for $0 < i < n$. β_0 is the intercept. The independent variables x_i take values from the emulator that executes the program code. Equation 5.1 is a simplified version of the more complex multiple linear regression model that was introduced in Chapter 4. This simplified version is used as a replacement for the more complex one.

$$v = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_{n-1} x_{n-1} \tag{5.1}$$

We claim that univariate TVLA evaluations of individual terms (i.e. $\beta_i x_i$) will give us information about which individual independent variable(s) are causing the leakage. The intuition behind this claim is based on the fact that leakage observed from the sum of terms can be explained by leakage of at least one term. A limitation of this method is that there exist scenarios where TVLA of v values would show leakage and that no individual TVLA of $\beta_i x_i$ would show leakage. Consider the scenario where the sum of two or more terms are leaky and the individual terms themselves are not. Even-though hypothetical, the existence of such cases limits the

discovery of all causes for leakage. The cost of computation for this method can be reduced through parallelisation since each individual TVLA evaluation can be run independently of others.

***Unintentional* ILA breaches.** ILA breaches happen when shares of a masked implementation are combined. Given a theoretically masked implementation written in assembly with care to keep all shares separate, *unintentional* ILA breaches may originate from unintended interactions between the intermediate values. Such interactions are superfluous and do not affect the functionality of a program in any way. We claim that these are the only leakages that will be present in a correctly masked assembler implementation. The reasoning behind this claim is as follows. Considering that each register value in the program is at least one share less than the total number of shares, any leakage detected must be caused due to the interaction between two or more of the register values. As long as this combination is not stored in a register (i.e. never used in the algorithm), such leakage can be removed without affecting the functionality of a program.

5.2 Countermeasures for ILA breaching

Protection of masked cipher implementations can be improved by changing code segments such that the leakage is removed without affecting the code's functionality [Papagiannopoulos and Veshchikov, 2017]. However, the limitations in synthesis of such code segments are not well understood. Usually, the synthesis is limited to a few example cases that practical evaluations can be performed for [Papagiannopoulos and Veshchikov, 2017; Bayrak et al., 2011, 2015]. In contrast, we suggest generating code patterns based on *dominating instructions* discussed in detail in Section 5.2.1 which can overwrite state set by previously executed operations.

Power analysis side-channel leakage can be observed in both protected and unprotected implementations. Even though it is obvious why an unprotected implementation may leak, the reasons why already protected implementations leak are not straightforward. A theoretically protected implementation, for example a dth order masked implementation is not expected to leak below order d under the ISW probing model [Ishai et al., 2003]. In other words, if a sensitive value has been split into $d + 1$ shares (i.e. in dth order masking), any number of observations

below $d + 1$ should not leak information (see Section 2.3.6). However, it is evident from our discussion in Section 2.3.7 that unintended interactions inside a CPU may inadvertently combine certain shares, resulting in a breach of the ILA. Therefore, a theoretical dth-order masked implementation will no longer be fully protected under dth-order attacks.

Unintended interactions end up combining shares that are assumed to be separately operated on by a masked cipher implementation. Such leakage stem from transition-based effects (see Section 4.1.2). As any ILA breaching interaction is not part of the cipher algorithm, rewriting such code segments without the ILA breaches will not affect the functionality of the cipher.

Value-based leakage is another type of leakage that can be observed in a masked implementation. In value-based leakage, a single intermediate value is the cause of the leakage. This means that, if value-based leakage is observed at orders below d for a dth-order masked implementation, there must exist a point in time when a register is loaded with an intermediate value that has at least two shares combined. Due to any single intermediate value being part of the algorithm, there exists no simple rewrite that would remove leakage without major changes to the algorithm. Due to above reasons, ROSITA only protects against ILA breaching effects that are not part of the masked cipher implementation.

The information needed to distinguish between transition-based and value-based leakage is already available from the power model. It follows from discussion of root cause analysis in Section 5.1 that root causes for each observed leakage from emulated power traces can be traced back to individual terms in the power model. Since the power model is evaluated on intermediate values generated from the execution of the assembly code, the root cause can be further traced back to the exact intermediate values in the assembly implementation of the cipher. Individual statistics of the values of each term reveal their participation for a particular leakage observed at a sample point. By tagging each of the participating terms with a label that uniquely identifies each term, it is possible to distinguish the kind of leakage that a specific leak belongs to.

5.2.1 Dominating instructions

It is possible for the internal state discussed in Section 4.1.2.1 to be overwritten by the same operations being executed later with the same registers which have different values in them. Thus far, the investigations conducted do not disprove this hypothesis. To test whether there is any interaction between the state types, we conducted the same fixed vs. random t-test that was carried out for Table 4.1 with the code segment shown in Listing 5.1. Similar to the previous experiment, registers `r2` and `r4` were initialised with one share each from a masked 2-share Boolean masking scheme. All other registers were initialised with unrelated random values. Without Line 6, the two `str` instructions would interact with each other through the memory bus (See Section 4.1.2.2 for details). In this experiment we were interested in whether the introduction of unrelated random values through the `eors` instruction at Line 6 would be able to erase the internal state that interacts with Line 11. We observed that `eors` was unable to stop the interaction between the two `str` instructions. We then continued to use different instruction pairs in place of `str` and `eors`.

The results obtained not only show that the states interact with each other, but also some instructions set *stronger* state that can erase previously set state. We designate these instructions as *dominating instructions*. The results we obtained are depicted in Table 5.1. The circles show equivalent instructions that fail to overwrite the internal state set by the other instruction. The triangles point to the instruction that is capable of overwriting the internal state. In other words, the triangles point to the dominant instruction. The interaction between ALU instructions show that none of the internal state set by them persists beyond any other ALU instruction. But `str`, `pop` and `push` instructions are capable of overwriting state set by any ALU instruction. Similarly, between `str` and `ldr` instructions, `str` is the dominant instruction.

```
1 str r1, [r2]
2 movs r7, r7
3 movs r7, r7
4 movs r7, r7
5 movs r7, r7
6 eors r5, r6
```

```
7  movs r7, r7
8  movs r7, r7
9  movs r7, r7
10 movs r7, r7
11 str r3, [r4]
```

Listing 5.1: Evaluating interactions between the `str` and the `eors` instructions.

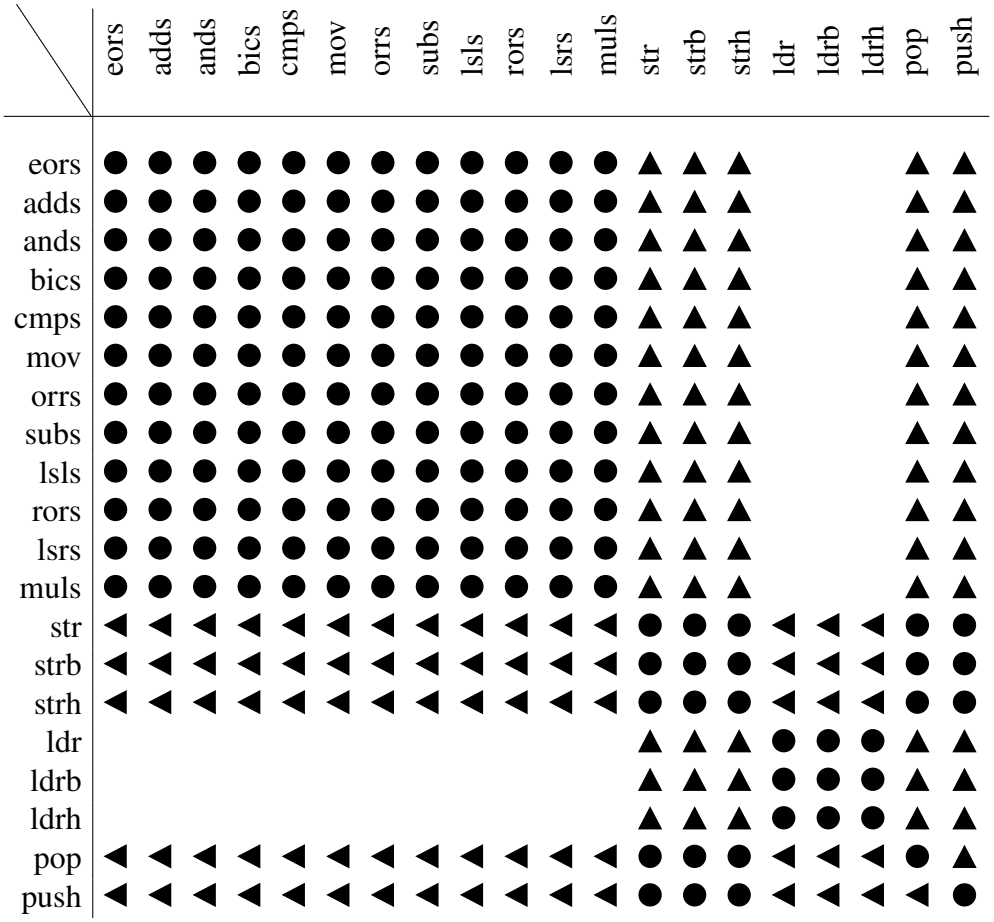


Table 5.1: State interactions between the second operands of instruction pairs. Triangles point to the dominating instruction. Circles indicate instruction pairs which overwrite internal state set by each other.

5.3 Synthesis of protective code segments

According to the results shown in Table 5.1, it is clear that dominating instructions or instructions that set equivalent state can be used to reset internal state by using random operands. Therefore, new instructions with random values can be used to reset internal states set by previous instructions. However, this modification must be done with care so that the register values are preserved after the new instruction is added.

Note that we assume that the CPU executes instructions in-order rather than possibly, out-of-order. In an out-of-order CPU, the order of the execution of instructions is not guaranteed. Therefore, ordering of the machine code is not strictly adhered. We also assume that each run of the same instruction uses the same internal components, this may not hold for CPUs that support speculative execution. Therefore, the fixes proposed in this work cannot be used on such CPUs. The simple ARM Cortex-M0 CPU that is used in this work is a non-speculative, in-order CPU.

The insertion of instructions with random data brings some technical challenges. First, a free register needs to be reserved to hold the new random value. Optimally, multiple random values may be used, but as a proof-of-concept, this work only uses a single random value that refreshes on each round of cipher invocation. Reserving a free register needed for this process is done manually for assembly implementations. For C implementations, we use dedicated flags, such as `-ffixed-`*`register`* in GCC[1] to generate code that does not use the reserved register. Reserving only one register minimises the performance impact due to memory spills.

Second, instructions that require destination registers (e.g. `adds`, `eors`) need an additional output register for the output to be written without affecting the dummy random value. Using the same register for input and output means that the random value needs to be reloaded from memory. Otherwise, registers that are already in use need to be spilled over to memory making the code segment modification complex. Additionally, such code segments will have a lower performance due to the use of additional instructions.

[1] https://gcc.gnu.org/onlinedocs/gcc/Code-Gen-Options.html#Code-Gen-Options

5.3.1 CPU pipeline leakage

Consider the code segment before fixes in Listing 5.2 where `r1` and `r3` each hold a single share of a 2-share Boolean masked implementation. `r0` and `r2` have been set to 0 before execution of Listing 5.2. The combination of the values of `r1` and `r3` causes leakage that is detectable through a CPA attack. The reason for this unintended interaction is the overwriting of internal pipeline registers that hold operand values [Gao et al., 2021b]. This results in a transition-based leakage where power consumption is correlated to the Hamming distance between the two shares in `r1` and `r3`. Such leakage can be mitigated by introducing a dummy random value (e.g. `movs r7, r7` where `r7` is preloaded with some random value) that breaks the overwrite between the two share values. It is introduced in between the leaky two instructions. With the dummy random value's introduction, the power draw will be correlated with the Hamming distances between dummy random value and `r1`, and `r3` individually.

Another kind of leakage that can be detected from the CPU pipeline is leakage from inter-bus interactions. An example inter-bus leakage with the same inputs used for the earlier example (shown in Listing 5.2) is shown in Listing 5.3. In this example code segment, the same two registers, `r1` and `r3` shows significant leakage when those two registers hold one share each. The same kind of leakage was also observed when the pair of registers that hold the shares and registers that hold unrelated values (`r0` and `r2`) were swapped. Our tests showed that inter-bus leakage can also be mitigated by introducing a dummy instruction with random values.

In a nutshell, transition-based leakage observed in the CPU pipeline is eliminated by introducing instructions that process random values. This results in a power draw that correlates with the Hamming distance with the random value instead of a sensitive intermediate value. This kind of protection is referred to as *random precharging* in literature [Tillich and Großschädl, 2007; Bayrak et al., 2011, 2015; Papagiannopoulos and Veshchikov, 2017].

5.3.2 Overwrite effects based leakage

Consider the before fixes code segment in Listing 5.4 where the shares of a Boolean masked implementation are stored in `r1` and `r3`. Due to overwriting of `r3`, the power draw will correlate with the Hamming distance between the two shares. As a fix for

```
movs r0, r1
movs r2, r3
```

(a) Before.

```
movs r0, r1
movs r7, r7
movs r2, r3
```

(b) After.

Listing 5.2: Example leakage from internal pipeline registers and fix

```
movs r1, r0
movs r2, r3
```

(a) Before.

```
movs r1, r0
movs r7, r7
movs r2, r3
```

(b) After.

Listing 5.3: Example inter-bus leakage from CPU pipeline and fix.

such leakage, the original value at the destination register can be replaced by a random value. The fix is shown highlighted in Listing 5.4. As the destination register's value is not important to the result of `movs r3, r1`, the leakage is mitigated while the functionality of the code segment is preserved.

Similarly, the same effect can be observed in code segment in Listing 5.5 where a register value of `r0` and memory pointed to by `r1` interacts. Similar to the previous case, writing a random value over the memory that holds the share can mitigate such leakage.

```
movs r3, r1
```

(a) Before.

```
movs r3, r7
movs r3, r1
```

(b) After.

Listing 5.4: Overwrite effects related to `movs` and fix.

```
str r0, [r1]
```

(a) Before.

```
str r7, [r1]
str r0, [r1]
```

(b) After.

Listing 5.5: Overwrite effects related to `str` and fix.

5.3.3 Arithmetic and Logic Unit leakage

Leakage that originate from state set by Arithmetic and Logic Unit (ALU) instructions can be fixed as shown in Listings 5.6 and 5.7.

We propose using `rors` instruction to fix this kind of leakage. The reasoning is as follows. As mentioned in Section 5.3, under the assumptions of in-order and non-speculative execution by the CPU, using the same instruction with random values as operands will clear internal state set by the previous instruction. Through results observed from Table 5.1 it follows that the internal state set by all ALU instructions are equivalent. The only remaining constraint is that the destination register should not destructively modify the random value. Therefore, we selected `ror` as a suitable instruction to be used to fix ALU state leakage.

```
lsls r0, r1, #8
...
...
...
lsls r2, r3, #8
```

(a) Before.

```
lsls r0, r1, #8
...
ror r7, r7
...
lsls r2, r3, #8
```

(b) After.

Listing 5.6: Example `lsls` leakage and fix.

5.3.4 Memory subsystem leakage

The leakage stemming from the memory subsystem manifest themselves mainly through state and overwrite based interactions. Similar to how state related leakage was fixed in Section 5.3.3, state related leakage from the memory subsystem can

```
eors r0, r1
...
...
...
eors r2, r3
```

(a) Before.

```
eors r0, r1
...
ror r7, r7
...
eors r2, r3
```

(b) After

Listing 5.7: Example `eors` leakage and fix.

be fixed by inserting dominating instructions that do not destructively modify the input random value. Suitable candidates according to Table 5.1 are `str` and a pair of `push` and `pop`. If `str` is used, a temporary variable is required to write the random value from `r7`. Since registers are limited, the address of this temporary variable is required to be loaded from memory using a `ldr` instruction. Due to restrictions in the architecture it takes even more instructions to guarantee this address is loaded from a point that is reachable within offset limits of `ldr` [Furber, 2000, Section 3.2]. Therefore, the most convenient code pattern to use is `push` and `pop` instructions with a random value. Both instructions are such that the stack pointer is preserved in the code that surrounds them.

An example usage of a similar fix is shown in Listing 5.8. In the before fixes code segment shown in Listing 5.8, state set by the first `ldr` instruction interacts with the state set by the second `ldr` instruction. `r1` and `r3` hold addresses of two distinct memory locations which hold the two shares of a 2-share Boolean masked implementation. `r0` and `r2` have been initialised to zero prior to running the code segment. Optionally, if overwrite based leakage is detected between `r2` (i.e. when `r2` holds a share) and the value pointed to by `r3`, the `r2` register can be used with `pop` instruction instead of `r7` to clear both leakages from addition of single code segment.

```
ldr r0, [r1]
...
...
...
...
ldr r2, [r3]
```

(a) Before.

```
ldr r0, [r1]
...
push {r7}
pop {r7}
...
ldr r2, [r3]
```

(b) After.

Listing 5.8: Example leakage and fix for `ldr` state.

5.4 Applying code fixes to masked code

The procedure of applying of code fixes is depicted in Figure 5.8. The procedure starts by emulating a masked cipher implementation at some arbitrary number of traces. Then, the code fixes for mitigation of detected leakages are applied by Rosita. If there is no leakage observed at a higher number of power traces, then the process stops. Otherwise, it continues to apply fixes until the leaky points count is zero.

Even with a wide range of code patterns it is still possible to find leakage that does not match any pattern in the library. In such a situation, Rosita will warn about consecutive fixes failing to eliminate leakage. Manual intervention may then be needed to introduce new code fixing patterns. However, our evaluations performed on Rosita with AES, Xoodoo and ChaCha showed such failures are rare after the pattern library is complete with code patterns for first two implementations.

The level of protection offered by the code fixes is defined with respect to the number of traces that were used for the emulated experiments. Rosita does not offer any protection above the number of traces that have been emulated. Further, due to the intricacies of the fixed vs. random t-test, multiple fixed inputs are required to guarantee that the coverage of detected leakage is increased.

Rosita cannot fix leakages that are value-based. Such leakage can be detected by the label that the leaky terms are associated with, if the label belongs to one of the value-based terms then the masked implementation needs to be modified through other means to guarantee that each intermediate value only contains at most $d - 1$ shares in a d-share masked implementation.

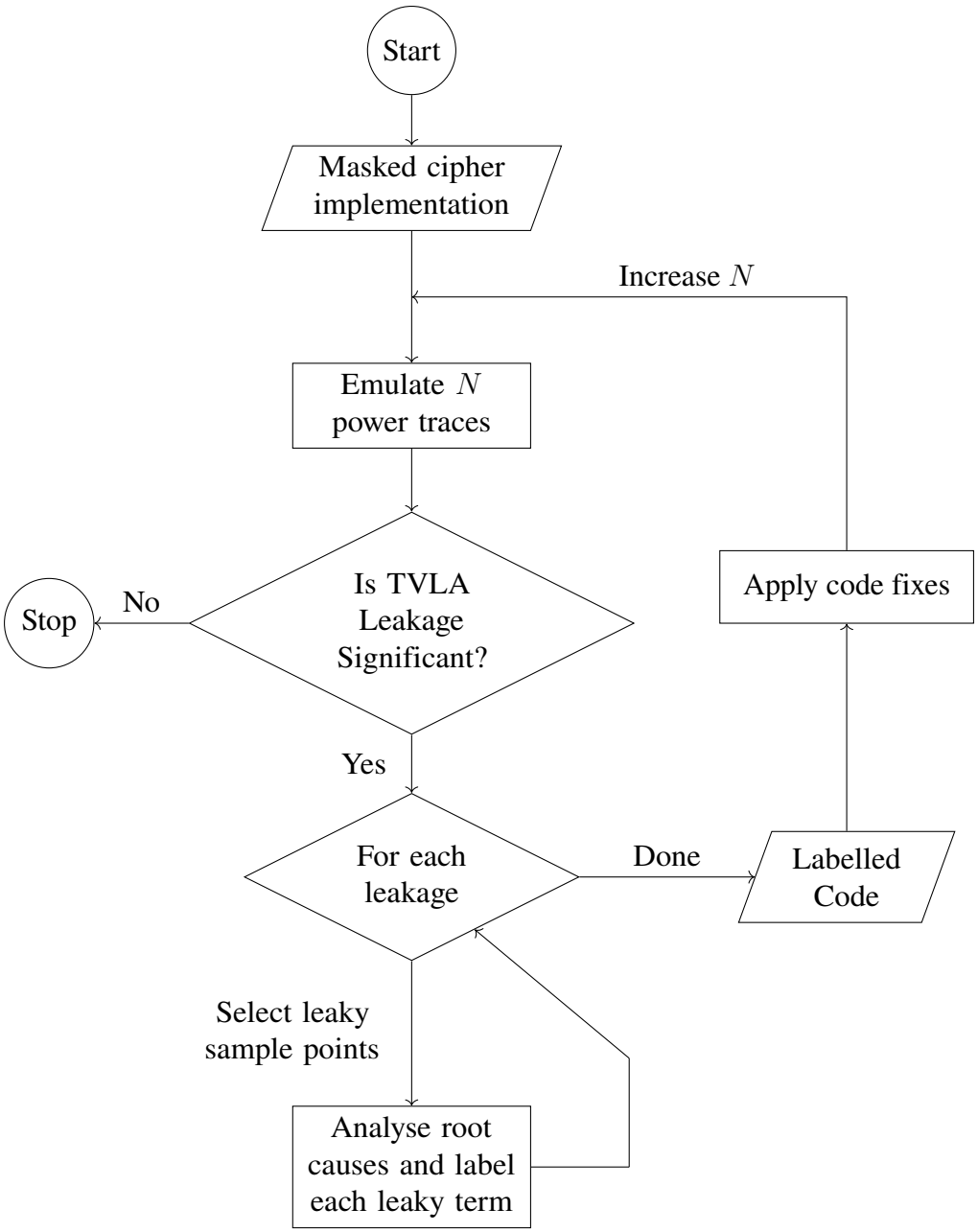


Figure 5.8: ROSITA mitigation application flow.

5.5 Implementation

The functionality of Rosita is divided across a set of three executables. These are as follows,

1. `elmo` - Hosts the emulator from the original ELMO project by McCann et al. [2017] and the modifications listed in this work to facilitate ELMO*.
2. `emulatetraces` - Handles building the software implementation and invocation of `elmo`.
3. `rosita` - Handles application of code fixes after the analysis of the t-test values generated in earlier step.

Rosita requires an evaluator to present the code to be evaluated as a *project*. A Rosita project is a bundle of files which are required for the successful production of an executable. Rosita also requires the project to hold information related to the execution of non-specific TVLA (see Section 2.3.5 for details). This includes information such as the total number of experiments (i.e. number of emulated traces) and build flags for the compiler.

The block diagram in Figure 5.9 shows the data flow within Rosita. The Project Builder component in the `emulatetraces` program is responsible for building an executable binary file from a Rosita project. Rosita currently only supports C and assembly source code files. The setup code for an experiment must be written in C and the segment of code under test can be written in either assembly or in C. However, C compilers might break masked implementations as a byproduct of code optimisations. The leakage stemming from such breakage can be detected, but cannot be fixed by Rosita. Care must be taken such that C implementations are either compiled with masking aware compilers [Moss et al., 2012] or to rewrite in assembly.

The Project Builder first updates the number of traces listed in the C source files. This is done through a reserved format of C style comments. An example is shown in Listing 5.9. The Project Builder component updates the content within each `@NTRACES{` and `}` comment with the trace count that is governed by `emulatetraces`. A similar method is used to introduce arbitrary fixed inputs to the masked implementation.

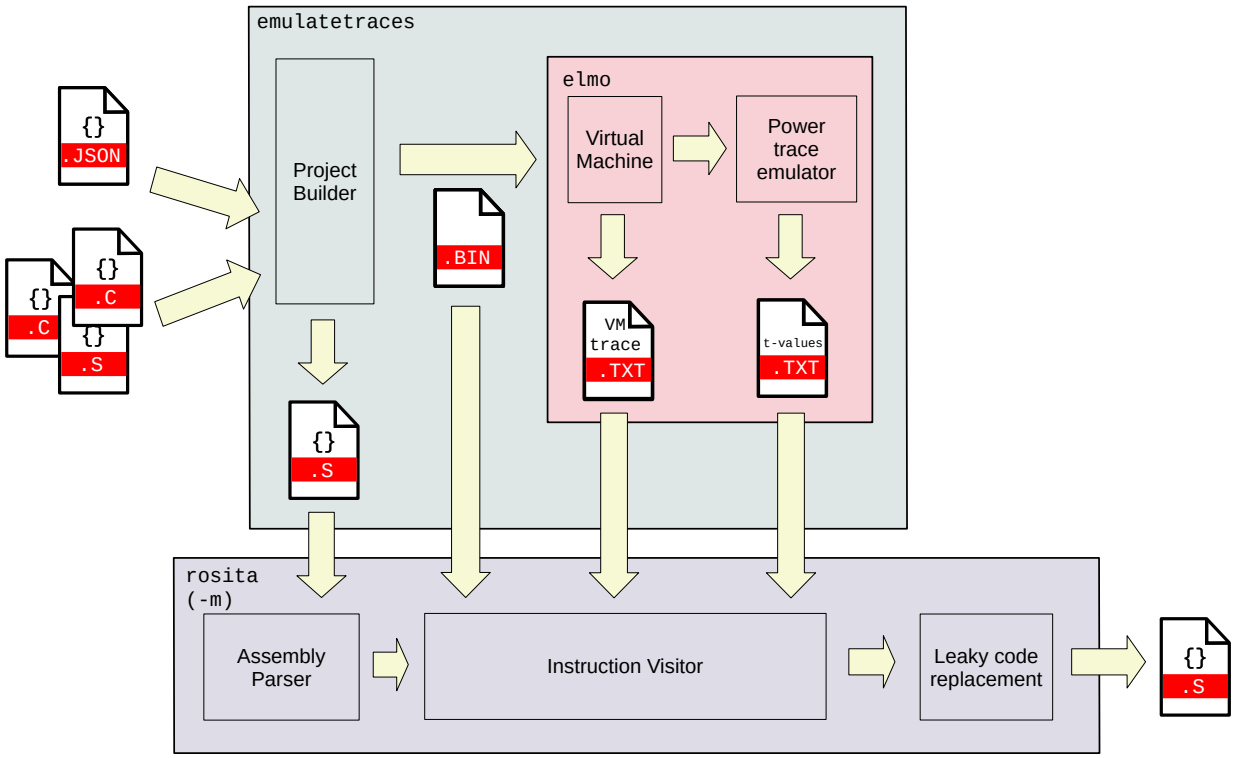


Figure 5.9: Diagram of data flow across the Rosita tool set.

```
#define NTRACES /*@NTRACES{*/ 10000 /*}*/
static const uint8_t fixedinput[PLT_SZ] = {
   /*@FIXED_INPUT{*/ 0xda, 0x39, 0xa3, 0xee, 0x5e, 0x6b,
   0x4b, 0x0d, 0x32, 0x55, 0xbf, 0xef, 0x95, 0x60, 0x18,
   0x90/*}*/ };
```

Listing 5.9: Example C source comment

A project in Rosita consists of a JSON (JavaScript Object Notation) file that describes all other files of the project and build configurations. This file lists compiler

flags that are needed to build a project along with some key value pairs required for Rosita's functionality. The field `c_tmpl_files` is where an evaluator should list the C file(s) that include the comment format listed in Listing 5.9. This file also includes the reserved register's name and the flag that needs to be used for the C compiler for explicit reservation of it. The reserved register must not be used in hand written assembly code. Examples of such build files can be found in the test project directories in the open source repository of Rosita[2].

Once the project is built, the resulting executable is emulated with `elmo`. `elmo` is a modified version of the emulator that is shipped with ELMO[3]. We applied a few modifications to the emulator to facilitate the operation of ELMO*. Similar to the original implementation of `elmo`, the code segment under test needs to be surrounded with function calls to `starttrigger` and `endtrigger`. These functions signal the start and end of the code segment under test. Once the emulation is completed, an emulator execution trace and a file with all the t-values for the code segment under test are generated by `elmo`. The t-values file holds the t-values for final differential voltages and t-values for individual terms as discussed in Section 5.1. `emulatetraces` program exits once the emulation is done.

`rosita` can be used with the flag `-a` to list the emulator's execution trace and the t-values generated from the fixed vs. random t-tests. This output highlights any significantly leaky instructions (i.e. $|t| > 4.5$) if found. Rosita works on the assembly code that is produced from the Project Builder. All C code is converted to assembly code first by execution of `gcc` with `-S` flag and then the GNU Assembler (`as`) is used to create object files from assembly files. Assembly Parser component is a custom parser that parses each line of assembly code by using regular expressions to split each assembly code line into instruction mnemonic and operands. The parser logic is implemented separately from the specialised logic required to parse the assembly of a particular ISA. We implemented ARM ISA parsing logic in `ARMParseMode.py` as the device that we used of our evaluation is based on the ARM ISA. The common parser logic is implemented in `ASMParser.py`. The Instruction Visitor component combines information from the emulator execution trace with the code from assembly files. This is required as the code fixes are applied to the assembly files and Rosita

[2] https://github.com/0xADE1A1DE/Rosita
[3] https://github.com/sca-research/ELMO

needs to know where the leaky instructions are located in assembly files.

An invocation of `rosita` with `-m` flag is shown at the bottom part of Figure 5.9. This flag switches `rosita` into code rewrite mode. Code rewriting begins with the same functionality as the listing of the leaky instructions but additionally, continues to label the leaky instructions according to the individual terms with significant t-values. Once the labels are applied, the code patterns for each label is matched through a prioritised logic. This logic is called the *matching logic* and is listed in separate code files for each micro-architecture. Matching logic is dependent on micro-architecture of a device. Two devices based on different micro-architectures may respond differently to the same code fixing pattern. Therefore, we opted to introduce all such patterns related to a particular device in a single source file. This helps Rosita to be easily adapted to different micro-architectures. The corresponding file for the ARM architecture is `ARMMatcher.py`[4]. Once this process ends, the assembly file that contains the code segment surrounded by `starttrigger` and `endtrigger` will have the code fixes applied to it.

The final step is to check if the changes done to the masked implementation actually reduced the leakage. This is done by emulation of the fixed version by running `emulatetraces` with the `--from-asm` option. This flag forces `emulatetraces` not to rebuild the project from scratch. Therefore, the emulation run will use a binary that is produced with the new assembly file. This process is repeated until all leakage at a certain trace count is removed.

5.6 Evaluation

We evaluated Rosita on a range of ciphers which have different kinds of round functions. First, selected code segments from a set of masked cipher implementations were protected using the code fixes that were applied by Rosita. Then, the resulting code segments were run on a physical device and the power traces collected from the physical device was evaluated using TVLA in the same configuration as used by Rosita to apply fixes. These evaluations were run on a STM32F030 Discovery evaluation board by ST Microelectronics. Refer to Section 3.2 for a detailed description of the experiment setup that was used. The resulting t-values from TVLA procedure were finally used to draw the plots shown in Section 5.6.2.1. We used a desktop PC with

[4] https://github.com/0xADE1A1DE/Rosita/blob/master/ROSITA/ARMMatcher.py

an Intel Core i9-10900K CPU and 32 GBs of RAM to run Rosita and the analysis of the power traces from the real device.

5.6.1 Implementations under test

We selected masked implementations of AES, ChaCha and Xoodoo to conduct our evaluations. Source code for these implementations is available under `./TESTS` path in the Rosita source code repository[5].

5.6.1.1 AES

AES is a block cipher first introduced by Daemen and Rijmen [2002]. It is one of the most commonly used ciphers in practice. We used the table-lookup based Boolean masked cipher implementation of AES-128 by Yao et al. [2018] in our AES tests. This implementation is closely related to the masked AES implementation presented in Mangard et al. [2005, Figure 9.1]. Due to the inter-byte interactions that we discussed in Section 4.1.2.2 (and also described by Gao [2019]), we modified the implementation so that it used 32 bit random values to mask 32 bit values. The code segment of interest was limited to the first round of AES-128.

5.6.1.2 ChaCha

ChaCha is a prominent example of an ARX (Addition-Rotation-Xor) stream cipher [Bernstein, 2008]. This was the main reason for its selection as ChaCha's construction is significantly different to the other algorithms considered. ChaCha is very efficient in software and widely used in TLS implementations. An efficient masked version written in assembly for ARM Cortex-M3 and Cortex-M4 processors was published by Jungk et al. [2018]. We ported their implementation into our platform which is an ARM Cortex-M0. As this implementation was done in ARM assembly, the code was rewritten in such a way that it never used `r7` so that it could be used as the reserved register for Rosita. This implementation has 20 rounds in it and therefore is referred to as ChaCha20. The code segment that was used for our evaluation only included the first quarter round of ChaCha20.

[5] https://github.com/0xADE1A1DE/Rosita

5.6.1.3 Xoodoo

Recent ciphers, mostly permutations, can be implemented efficiently with only bitwise Boolean instructions and (cyclic) shifts, e.g., Keccak-p, the permutation underlying SHA-3 [Dworkin], Ascon [Dobraunig et al., 2016], Gimli [Bernstein et al., 2017] and Xoodoo [Daemen et al., 2018b]. They all have a nonlinear layer of algebraic degree 2 and hence allow very efficient masking. Among those, we chose Xoodoo because it is the simplest of all and it lends itself to efficient implementations for 32 bit architectures.

Xoodoo was proposed recently by Daemen et al. [2018a] for use in authenticated encryption modes [Daemen et al., 2018b]. We used the optimised and non-masked implementation of Xoodoo from Bertoni et al. and we implemented the 2-share Boolean masking scheme of the non-linear layer χ as suggested by Bertoni et al. [2011]. In contrast to what Bertoni et al. [2011] mention, we initialised the state with fresh randomness for each trace to keep it consistent with the implementation of AES, even though this is not required.

5.6.2 Results

5.6.2.1 Univariate TVLA evaluation

The graphs in this section show the performance and effectiveness of Rosita. The cipher implementations listed in Section 5.6.1 were protected using Rosita. This was done by gradually increasing number of emulated traces from 50,000 to 1,000,000. The number of leaky points discovered and the number of leaky points remaining after fixes are shown in Figures 5.12, 5.16, 5.20 and in Figures 5.13, 5.17, 5.21.

The implementations were run before and after the fixes were applied on a physical device, the Welch's t-test results from these experiments are shown in Figures 5.10, 5.14, 5.18 and in 5.11, 5.15, 5.19. Leakage detected using the Welch's t-test values calculated at each sample point from the power traces collected from the physical device. One million power traces were gathered from each physical experiment. Refer to Section 3.2 for more information on our experiment setup.

First round of AES-128. Significant leakage was observed around the ShiftRows operation of masked AES in Figure 5.10 from one million power traces gathered from our test device. We used one million emulated power traces to apply fixes to

this implementation, the time spent on emulation and root cause detection is shown in Figure 5.13. It took less than one hour to emulate find root causes for one million power traces for the first round of masked AES. The number of leaky points and remaining leaky points after applying fixes is shown in Figure 5.12. There were no new leaky points found after 150,000 power traces. The t-test values observed from the same code segment after application of code fixed by ROSITA are shown in Figure 5.11. We observed significant leakage that was not emulated by ELMO*. We investigated the reason behind such leakage and found out that it is address bus related leakage from the SUBBYTES section of AES. The leakage could be mitigated by manually applying address bus resetting instructions that loaded values from random points from the S-box. The level of knowledge that ROSITA requires to apply such a mitigation automatically is significantly higher than the level of knowledge it currently has about a certain software implementation. Currently, ROSITA does not have an apparatus for tracking any buffer related memory operations. ROSITA was able to fix 93.1% of the observed significant leakage from the physical device at 1 million power traces.

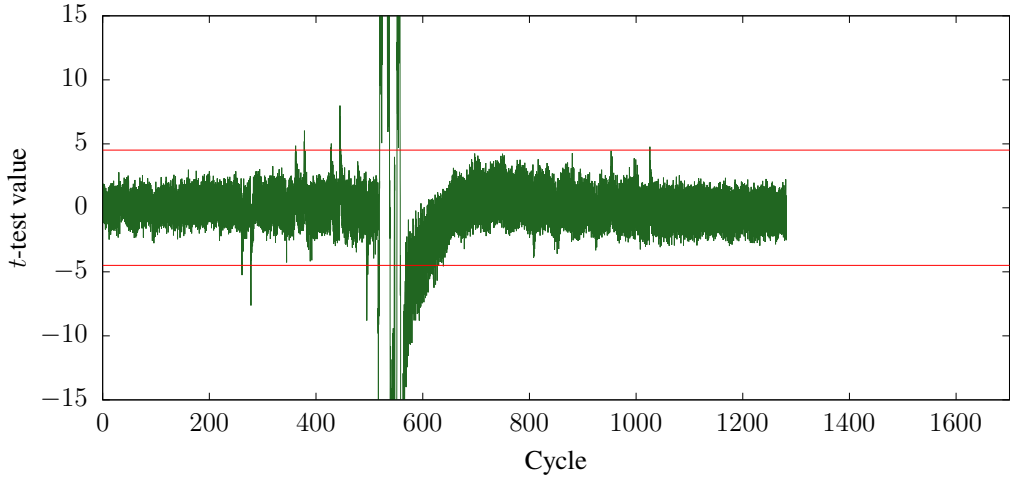


Figure 5.10: t-test values for AES before applying countermeasures.

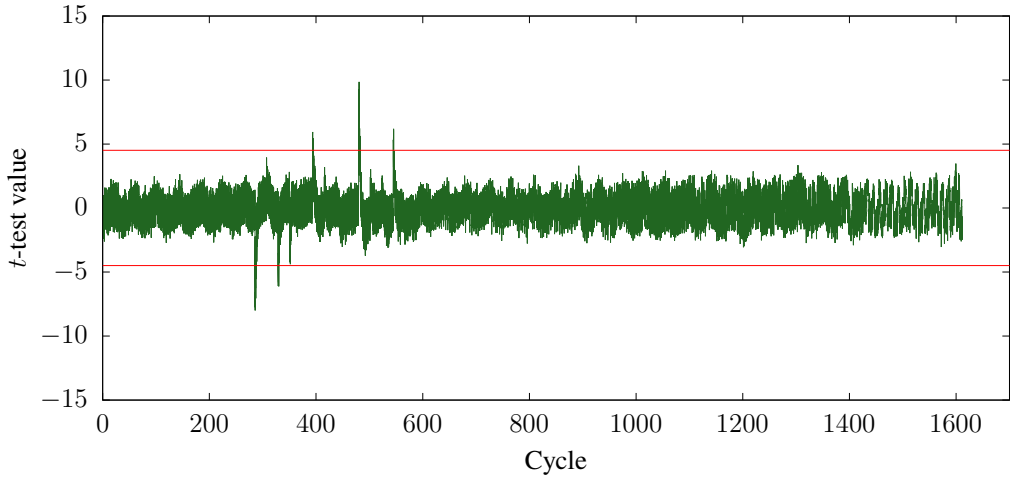


Figure 5.11: t-test values for AES after applying countermeasures.

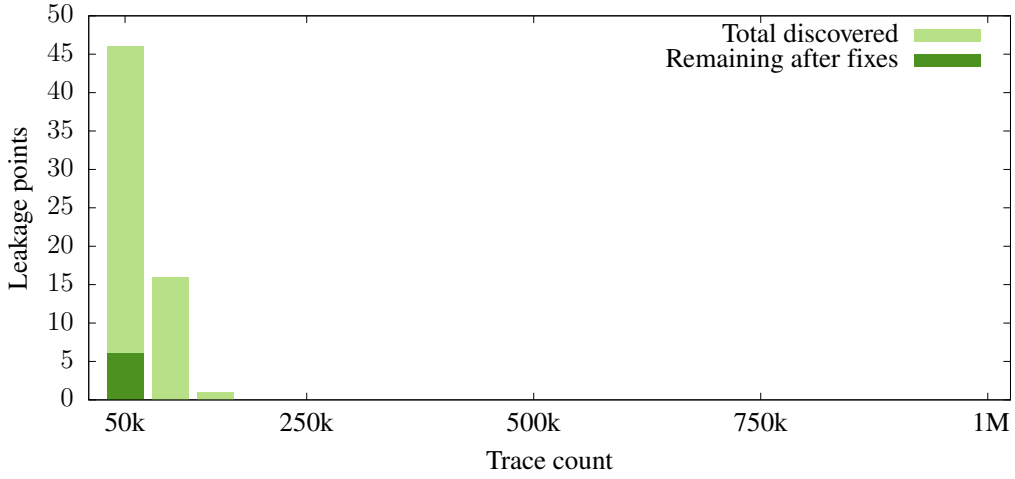


Figure 5.12: Number of leaky instructions before and after fixes for AES.

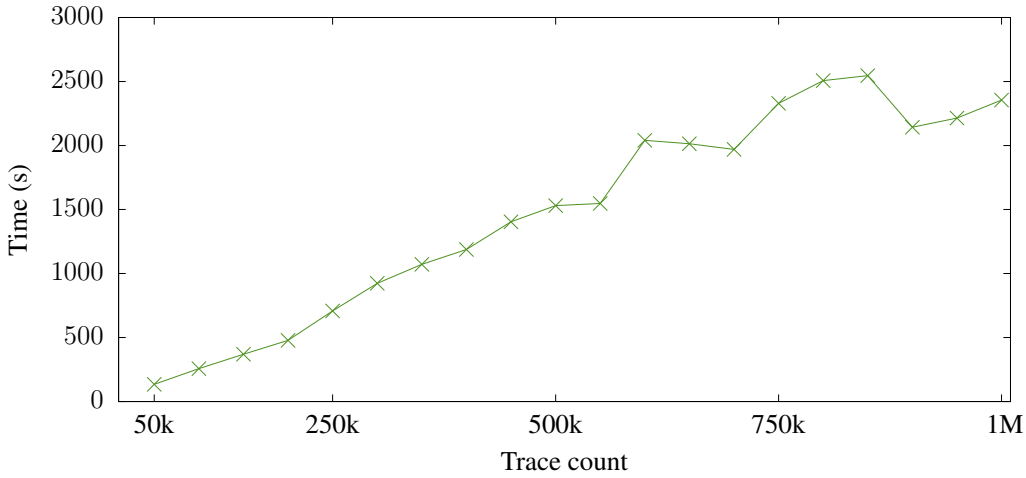


Figure 5.13: Time taken to emulate and analyse AES by `elmo`.

First quarter round of ChaCha. We detected the highest levels of leakage from all three implementations in the first quarter round of ChaCha as shown in Figure 5.14. Similar to the observations from the physical experiment, the emulated traces found 208 significant leaks. The time spent by Rosita on emulation and analysis of ChaCha is shown in Figure 5.17. It took less than two hours to emulate and analyse leakage from one million power traces. The number of leaky points and remaining leaky points after applying fixes are shown in Figure 5.16. We found no new leaky points after 150,000 traces. Rosita was able to fix 99.7% of the significant leakage detected

from physical device at 1 million power traces.

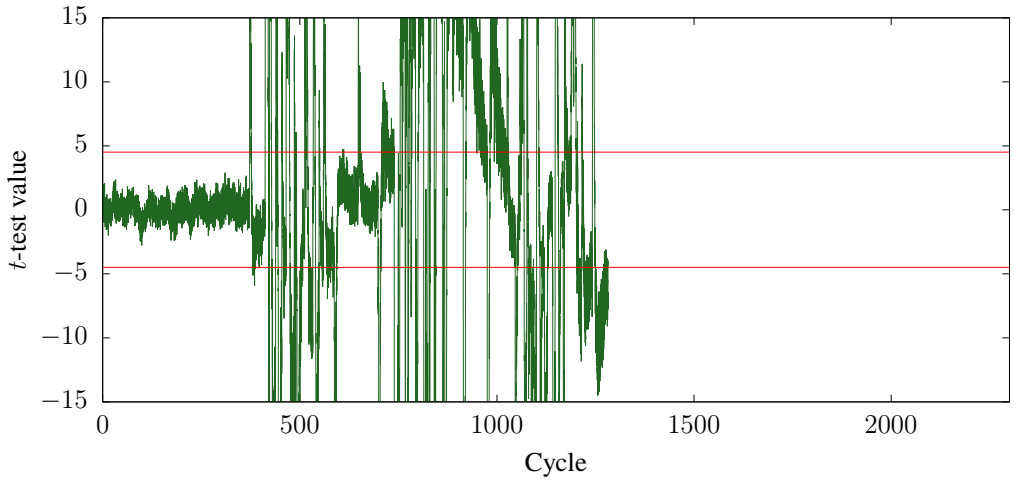


Figure 5.14: t-test values for ChaCha before applying countermeasures.

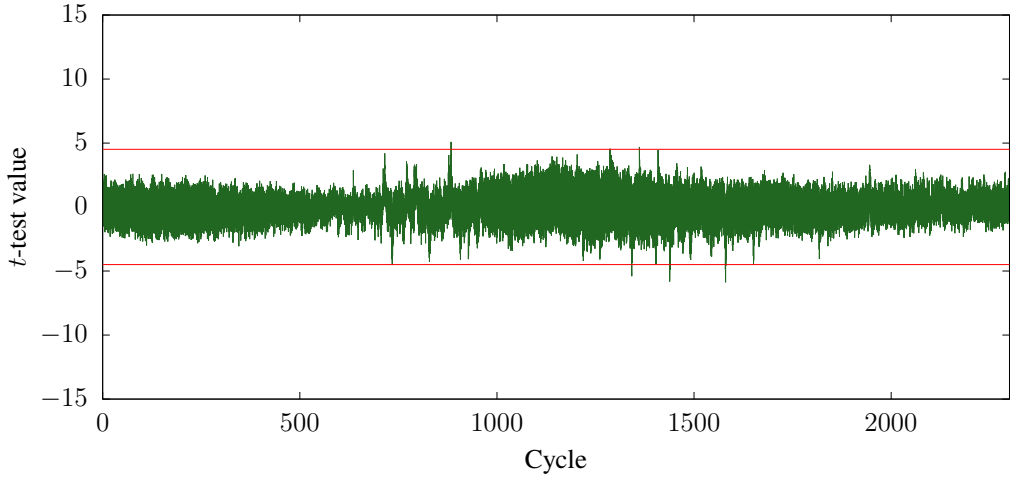


Figure 5.15: t-test values for ChaCha after applying countermeasures.

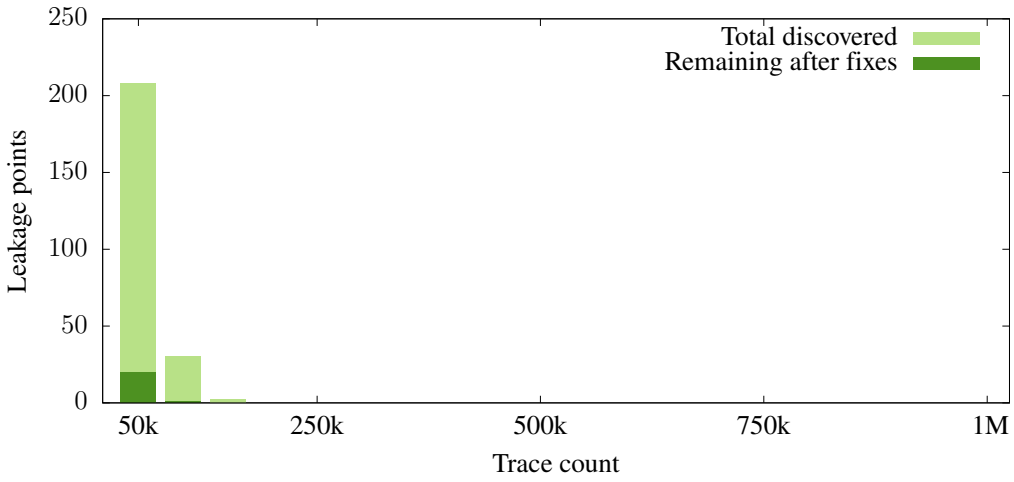


Figure 5.16: Number of leaky instructions before and after fixes for ChaCha.

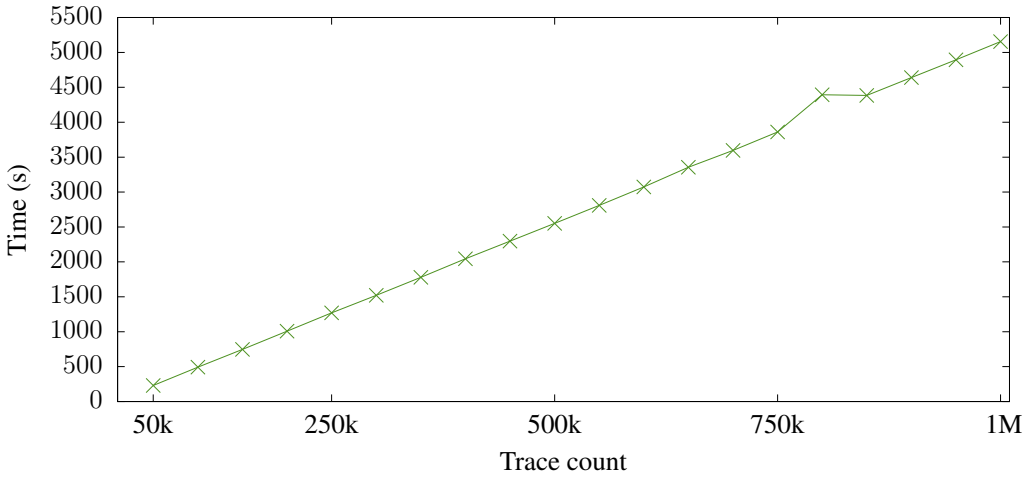


Figure 5.17: Time taken to emulate and analyse ChaCha by `elmo`.

First round of Xoodoo. Significant leakage was detected around the cycles that executed the χ function of Xoodoo as shown in Figure 5.18. The time spent by Rosita on emulation and analysis of Xoodoo is shown in Figure 5.21, and the number of leaky points and remaining leaky points after applying fixes are shown in Figure 5.20. It took less than an hour to emulate and analyse power traces from Xoodoo. Similar to both AES and ChaCha implementations, there was no new leakage found beyond 150,000 power traces for Xoodoo. We then ran the fixed implementation on our test device and gathered power traces from the physical experiment to draw Figure 5.19.

The single leaky point that is left over in Figure 5.19 is due to a leaky point that had multiple leaky terms which destructively interfered in the detection the leakage in emulated traces. We believe that effects similar to a glitch may have exposed this leakage. This means that each of the leaky terms that were found in emulation manifest themselves individually in the real experiment. ROSITA was able to fix 91.6% of the significant leakage detected from physical device at 1 million power traces.

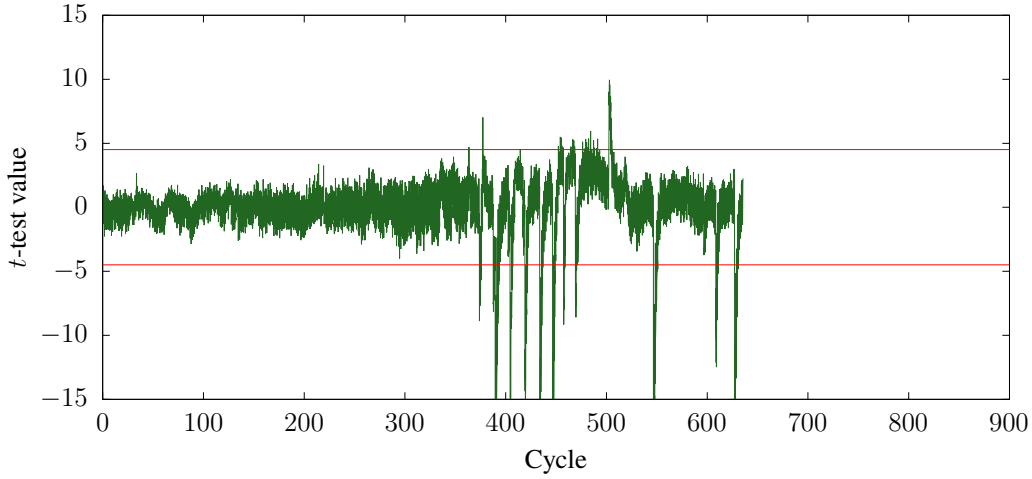


Figure 5.18: t-test values for Xoodoo before applying countermeasures.

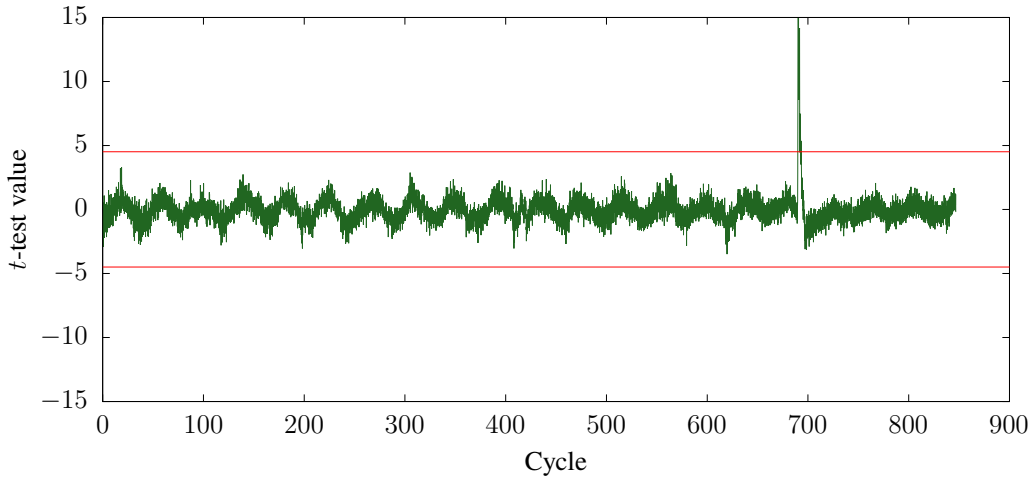


Figure 5.19: t-test values for Xoodoo after applying countermeasures.

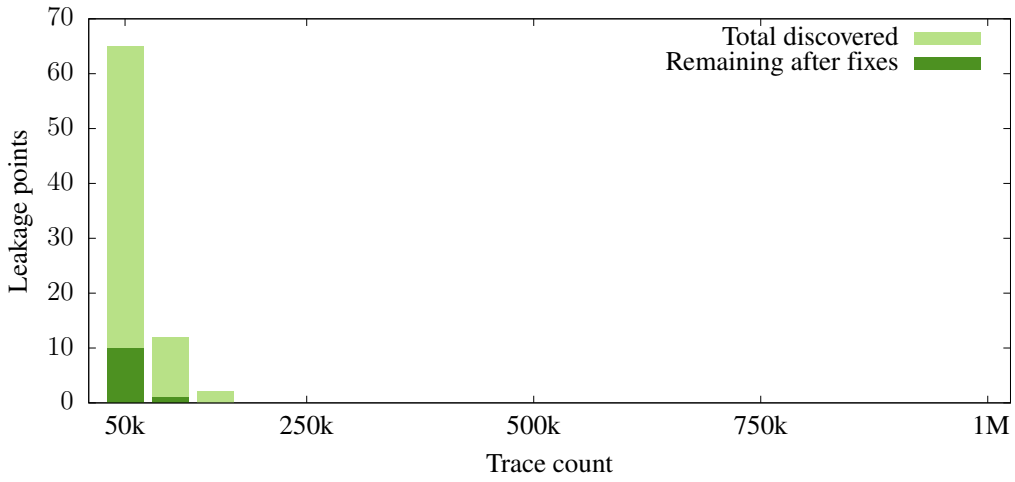


Figure 5.20: Number of leaky instructions before and after fixes for Xoodoo.

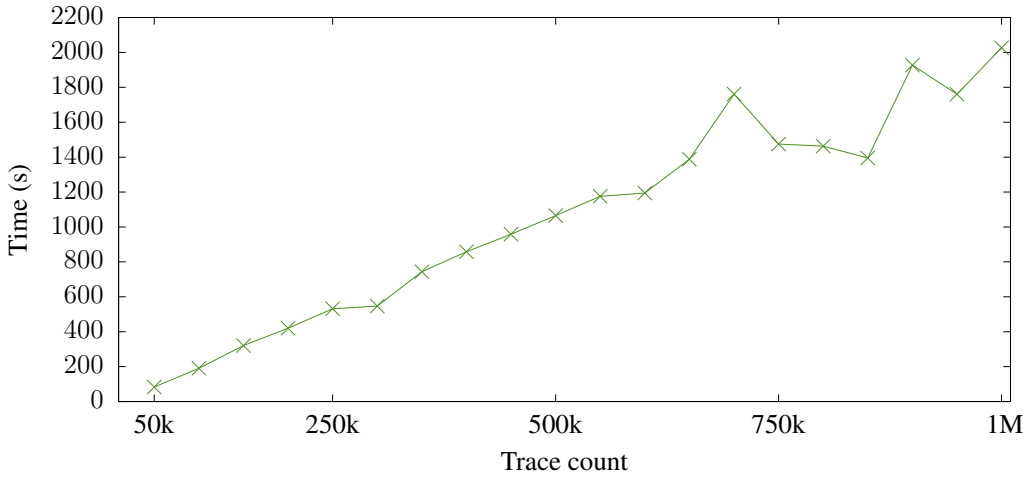


Figure 5.21: Time taken to emulate and analyse Xoodoo by `elmo`.

5.6.2.2 10 fixed-inputs TVLA univariate evaluation

The TVLA results in this section are combined results from multiple physical experiments that were run with multiple fixed inputs. Using multiple fixed inputs increases the coverage of the detected leakage. Therefore, the fixes applied are more effective than using a single fixed input when using the same number traces. Figures 5.22, 5.24, 5.26 show leakage from multiple fixed inputs before applying code fixes by Rosita. Figures 5.23, 5.25, 5.27 show leakage after the fixes were applied by Rosita. We used 10,000 power traces with 10 fixed-inputs to draw

these plots. The overhead added to each of the masked implementations due to the code fixes were 20.5% for AES, 65.1% for ChaCha and 35% for Xoodoo. The address bus related leakage that was observed at one million traces in Figure 5.10 and Figure 5.11 have become less significant due to the lesser number of traces used for drawing Figure 5.22 and Figure 5.23.

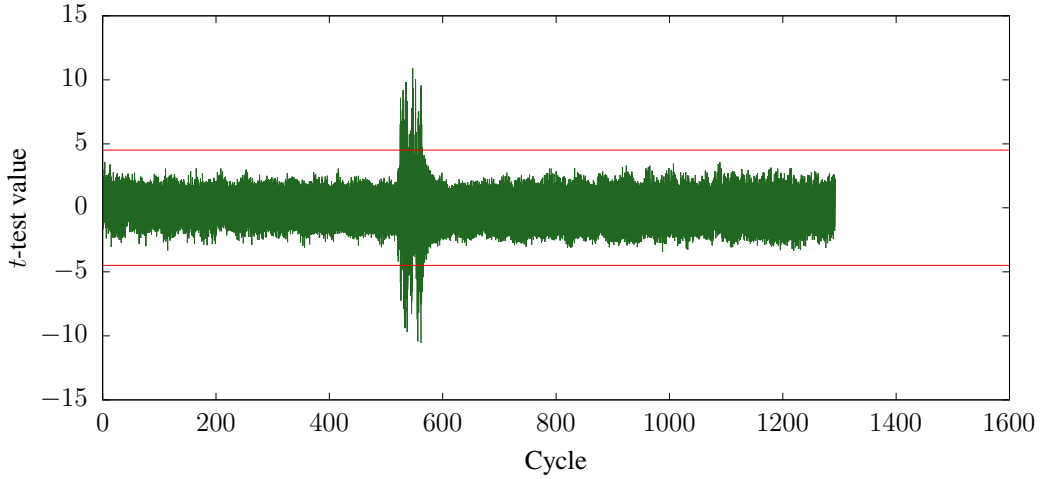


Figure 5.22: Maximum t-test values of 10 fixed inputs for AES first round before applying countermeasures.

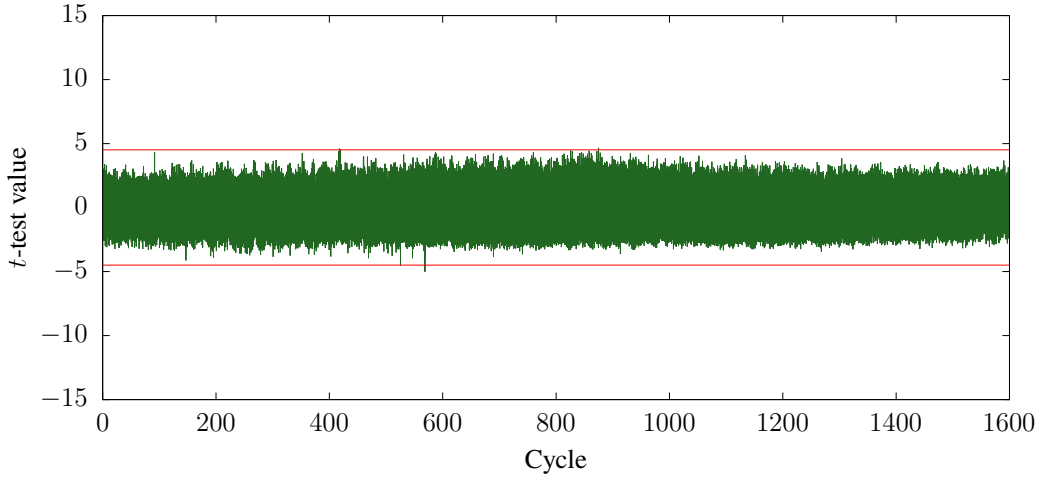


Figure 5.23: Maximum t-test values of 10 fixed inputs for AES first round after applying countermeasures.

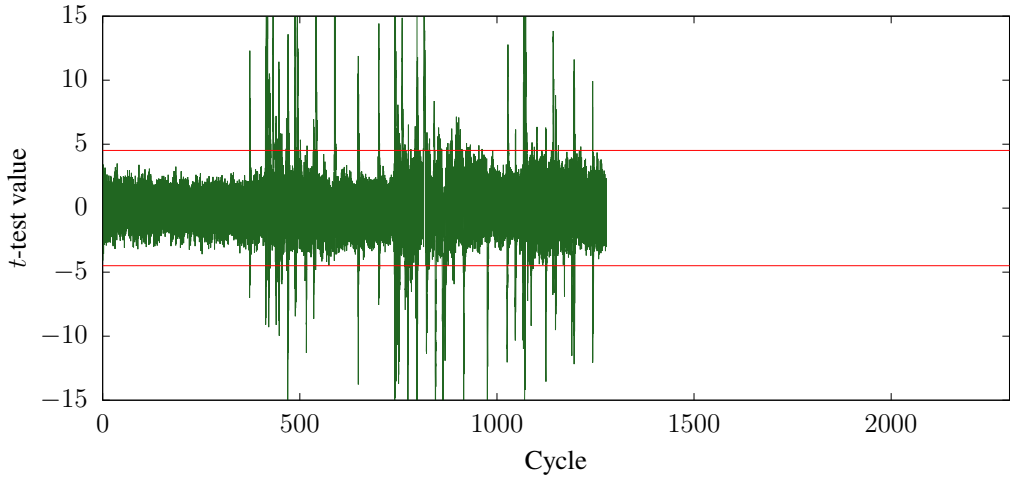


Figure 5.24: Maximum t-test values of 10 fixed inputs for ChaCha first round before applying countermeasures.

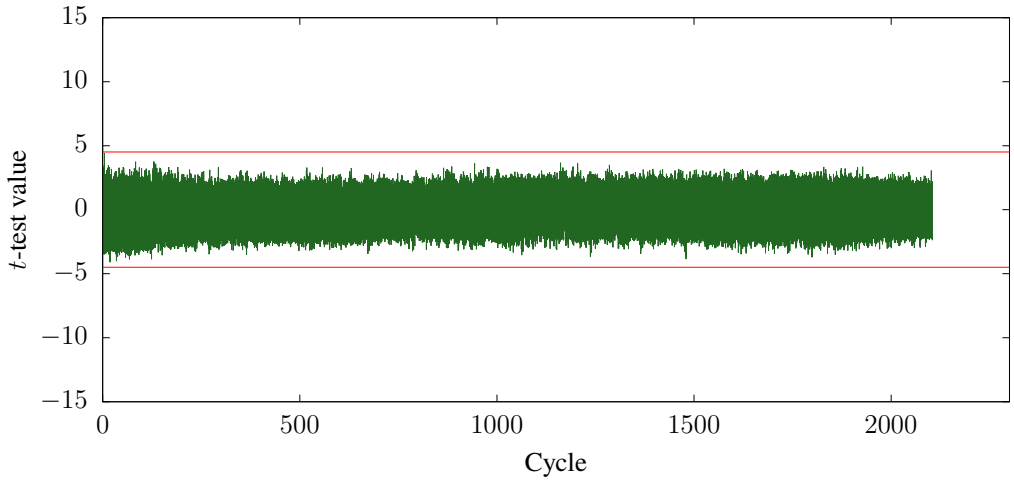


Figure 5.25: Maximum t-test values of 10 fixed inputs for ChaCha first round after applying countermeasures.

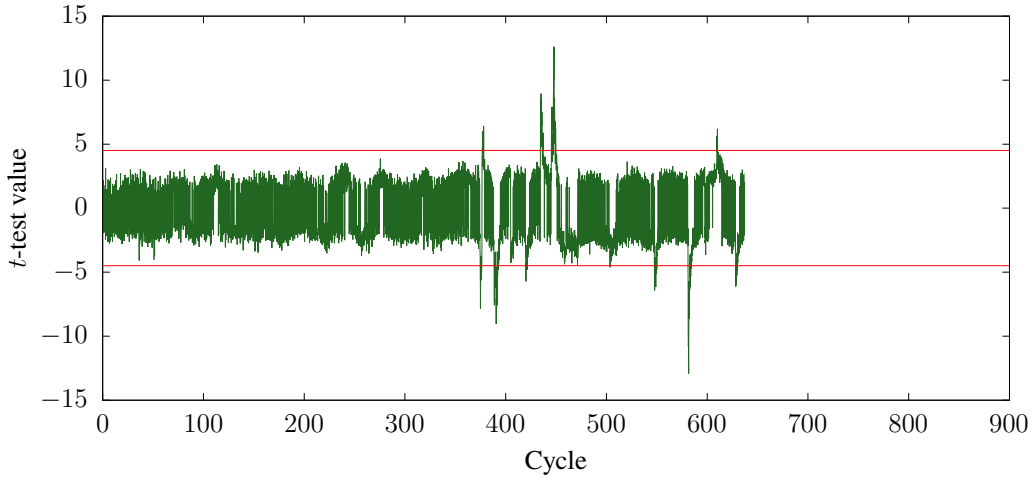


Figure 5.26: Maximum t-test values of 10 fixed inputs for Xoodoo first round before applying countermeasures.

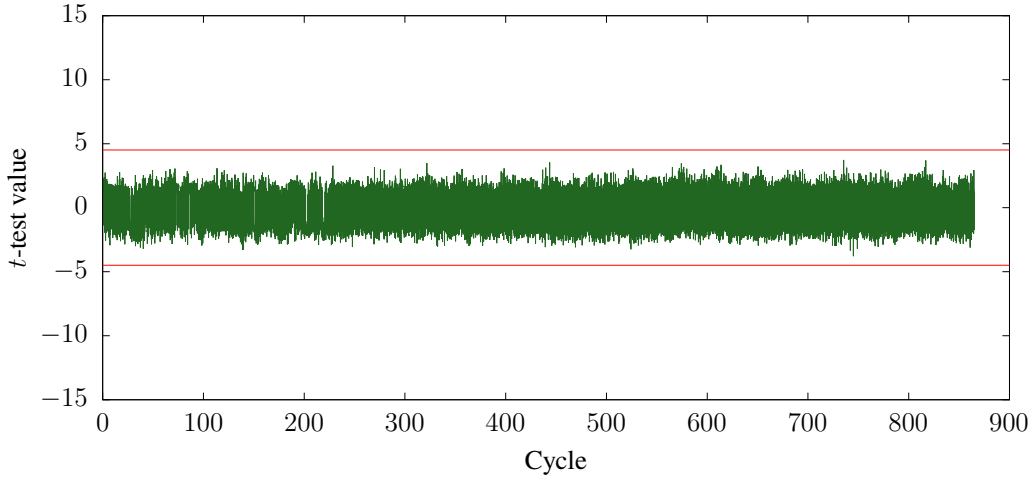


Figure 5.27: Maximum t-test values of 10 fixed inputs for Xoodoo first round after applying countermeasures.

5.6.3 Discussion

Rosita was capable of eliminating more than 91% of leakage observed from the physical experiments in all masked cipher implementations that we tested. Xoodoo ended up as the lowest performing implementation due to the fact that it initially had comparatively lower number of leakages and the fact that due to the high significance of the leakage it was detected on many cycles. The highest percentage of eliminated leakage was observed in ChaCha which is 99.7%. Rosita only required less than 250,000 emulated traces to detect all practical leakage that were detected from one million power traces. We highlight this as a major benefit of using emulation instead of physical experiments where it significantly reduces the time that is required to conduct experiments. It took us around four hours to collect one million power traces from the physical experiment, in the worst case, the emulation of ChaCha took 35% of that time to emulate and analysis of the same number of power traces (without the the use of multi-threading). We also ran the same experiment in a multi-threaded setting with 10 threads where each emulator instance emulated 1/10 of power traces. This setup took 11.75% of the time that the physical experiment took to produce the same number of traces. In this setting, the emulated traces were first written to disk and then the analysis was done by a separate process. In contrast, the single-threaded approach did not require disk usage as the first order univariate t-test could be run online. Multi-threaded performance could be further improved by introducing single pass first order univariate t-tests for multiple trace partitions [Schneider and Moradi, 2015]. We believe doing so will further improve the performance due to the removal of the dependency on slow disk storage.

Table 5.2 shows the number of cycles in original implementations, number of cycles in the fixed implementations and overhead added on each cipher implementation due to Rosita's code fixes. These results show that Rosita has been successful in eliminating univariate leakage from practical masked cipher implementations. The overheads shown in Table 5.2 come from additional instructions added by Rosita. However, some initial overhead is added to the original implementations as they are required to be compiled with a reserved register allocated to Rosita (i.e. `r7`). If all registers were available, the resulting programs would be shorter. This limitation can be resolved by introducing dedicated instructions to clear micro-architectural state

as shown in Gao et al. [2020b]. The number of instructions added by disabling the use of `r7` was nine and two instructions for AES and Xoodoo. This was calculated by compiling the C implementation source with and without the `--ffixed-r7` flag. Since ChaCha was originally written in assembler without the use of `r7`, the overhead for ChaCha implementation could not be determined. The results in Table 5.2 are from emulations that used only a single fixed input, therefore it fails to eliminate all leakage that is detected from the physical experiment. However, this can be mitigated by using many fixed inputs for the emulation as shown in the results in Section 5.6.2.2. We believe Rosita fails to discover all leaky points from a single fixed input due to noise level differences in emulation and in the physical experiments.

In comparison with manually applied code fixes, Rosita's code fixes do not offer any globally optimised code generation. Rosita is only focused on eliminating leakage from instructions with a limited local scope around the leaky instruction. Globally optimised code generation would enable a reduction in overhead as register allocation could be performed optimally considering the whole program.

Function	**Original Cycles**	**Fixed Cycles**	**Overhead**	**Percentage of eliminated leakage**
AES	1285	1559	21.3%	93.1%
ChaCha	1322	2313	75%	99.7%
Xoodoo	637	843	32.3%	91.6%

Table 5.2: Results of running Rosita to automatically fix masked implementations of AES, ChaCha, and Xoodoo.

Chapter 6

Mitigation of multivariate leakage

This chapter discusses methods that apply code fixes to eliminate multivariate leakage. Rosita, which was introduced in Chapter 5 is capable of fixing leakage in first-order masked implementations. To increase the security offered by masking, cipher implementers often incorporate masking schemes of higher-orders [Rivain and Prouff, 2010; Cnudde et al., 2015; Hutter and Tunstall, 2019]. The methods used for root cause detection in Rosita take impractical amount of time if used to fix higher-order leakage. In this chapter we present Rosita++ which solves this problem.

We first describe what multivariate leakage is, and how to assess it. Then we discuss two new root cause detection algorithms that are orders more efficient than the naive method used in Rosita. These are elimination of terms (Section 6.2) and a Monte Carlo experiment based method (Section 6.3). Finally, we detail the implementation of Rosita++ and evaluate it using second- and third-order masked code.

6.1 Multivariate leakage assessment

Multivariate leakage assessment analyses information leakage from multiple sample points simultaneously. In contrast, univariate leakage assessment only uses information from a single sample point. Detection of multivariate leakage is useful in evaluating higher-order masked implementations as higher-order leakage cannot be detected by univariate methods. We require multivariate root cause detection to apply code fixes based on multivariate leakage. In this section we introduce multivariate leakage assessment and discuss why the naive root cause detection Rosita cannot be

optimised to detect multivariate root causes.

Multivariate leakage is defined as leakage detected from the combination of power values from a set of sample points. The normalised product of sample values is the commonly used function for combining the sample values [Prouff et al., 2010]. The combined sample value from normalised d sample points, $\mathcal{J} = \{j_0, j_1, ..., j_{d-1}\}$ is given by q in Equation 6.1 where $y^{(j)}$ is the sample value at the jth sample point and $\mu^{(j)}$ is the mean of all sample values observed at sample point j.

$$q = \prod_{j \in \mathcal{J}} \left(y^{(j)} - \mu^{(j)} \right) \tag{6.1}$$

Similar to the univariate case (see Chapter 5), $y^{(j)}$ takes values from the following multiple linear regression model, where $y^{(j)}$ is the differential voltage across a shunt resistor across the power rail. Independent variables and coefficients of the regression model are x_k and β_k for $0 < k < n$. β_0 is the intercept.

$$y^{(i)} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_{n-1} x_{n-1} \tag{6.2}$$

What is different from the univariate case is that the values of $y^{(j)}$ from different sample points are used as input to Equation 6.1 to create a combined power value q, which is then evaluated using univariate TVLA. This means that if we follow the same methodology as mentioned in Chapter 5, there will be n^d total combinations between the terms. This is due to each emulated power value at a different instruction (i.e. sample point) is considered distinct. Given that ELMO* has 26 terms in total, running such a number of t-tests is inefficient even for smaller values of d. Additionally, due to the requirement of exponentially more power traces to detect leakage as d increases in multivariate settings [Chari et al., 1999], this method becomes even more impractical for multivariate root cause detection.

We introduce two methods for multivariate leakage root cause detection. The first method depends on removing a single term from Equation 6.1 at a time and then reevaluating the leakage trying to find instances where the removal of a term converts leaky set of samples to a non-leaky set of samples. To alleviate the limitations of this method, we also propose a second method for root cause detection based on Monte Carlo experiments.

6.2 Elimination of terms

Elimination of terms is a novel method for efficient root cause detection in multivariate leakage that was introduced in ROSITA++. A term is removed from all $y^{(j)}$ and $\mu^{(j)}$ where $j \in \mathcal{J}$ that participate in multivariate sample q and then the fixed and random input power value distributions are tested for the absence of leakage. As only one term is removed at a time from $y^{(j)}$, the number of statistical checks required drops from n^d to nd.

A core observation for this approach is that TVLA cannot be used for determining the absence of leakage as it was used in detection of root causes in univariate leakage (see Chapter 5). In elimination of terms, the null hypothesis states that the two distributions are different (i.e. leaky). Therefore, the Welch t-test is not suitable for testing the equivalence of two distributions because it is designed only to measure statistical difference between distributions. Failure to show that two distributions are different does not demonstrate that they are the same. Instead, we use the Two One-Sided t-tests (TOST) procedure [Schuirmann, 1987] for testing whether distributions are equivalent.

In a nutshell, TOST determines if the mean difference between two distributions falls within two boundary values determined by a certain level of significance (see Section 2.2.2.3 for a detailed description of TOST). The power values needed for TOST are collected by running the same experiment as for the univariate case which was run in a fixed vs. random input configuration. The two boundary values used in TOST are determined from power samples collected from running the same experiment in an all random input configuration. This is done by first recording all term values separately with ELMO* in a fixed vs. random input setting and then again when all inputs are random. The emulated traces for fixed vs. random and random vs. random configurations result in emulated traces which are stored in a format we refer to as the TERMTRACES format, as shown in Figure 6.1. $\boldsymbol{L}$ is a three dimensional array with trace number, sample number and term number as its dimensions. We refer to the trace array from the fixed vs. random input configuration as $\boldsymbol{L_{FR}}$ and the trace array from the random vs. random input configuration as $\boldsymbol{L_{RR}}$. Since we know and are able to reset the internal state of the emulator between cipher invocations, the order of the tests do not matter. Therefore, in fixed vs. random input configurations,

the traces from fixed input tests are written to the file first and the traces from random input trace are written after that.

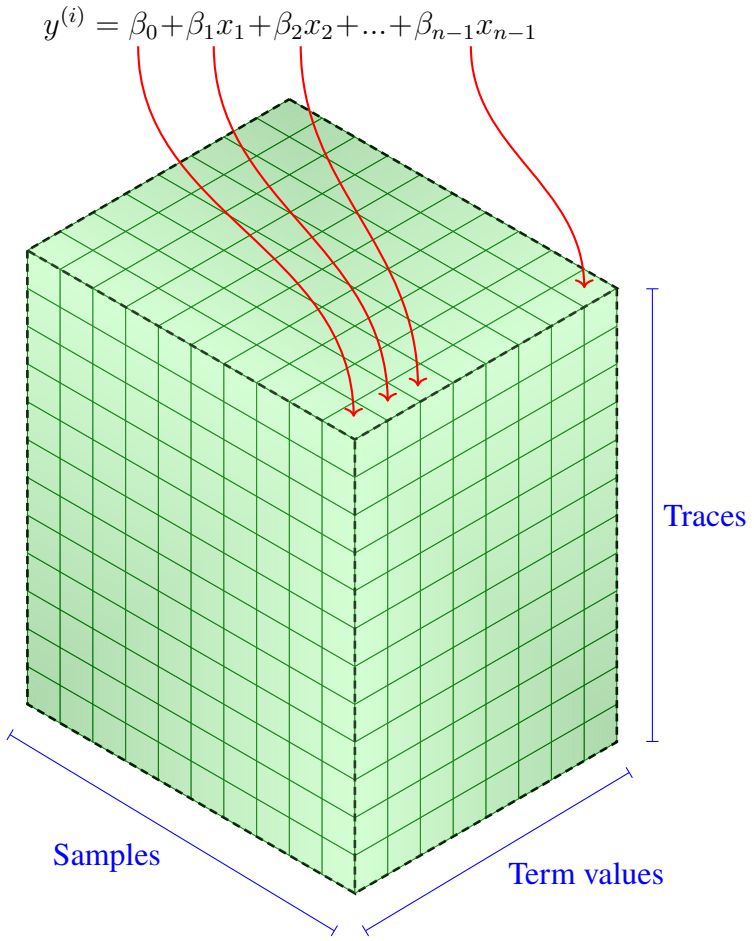


Figure 6.1: Mapping of term values to $\boldsymbol{L}$.

We then use the traces collected from the emulation tests to find leaky terms. This process determines the set of terms that contribute to a leakage from each sample point. The algorithm we use for finding leaky terms is shown in Algorithm 6.3. The FLT algorithm removes a single term from a single sample point at a time and calculates the differential voltage ($y^{(i)}$) at that sample point. Differential voltage values from d sample points are then combined to a single array of normalised power values, $\boldsymbol{Y}$. This is done through the NPS (Normalised Product of Samples) function

shown in Algorithm 6.1. It calculates the normalised product from the power value of reduced models at given set of sample points only using a given set of terms. The corresponding combination of normalised power values at same sample points from $\boldsymbol{L_{RR}}$ results in $\boldsymbol{X}$.

Algorithm 6.1 Normalised Product of Samples.

NORMALISE($\boldsymbol{Y}$) Normalises $\boldsymbol{Y}$ array by subtracting the mean from all elements.

```
1:  function NPS(L, S, T)
2:      i ← 0
3:      V ← [|L[:, 0, 0]|, |S|]
4:      for s ∈ S do
5:          V[:, i] ← 0
6:          for t ∈ T do
7:              V[:, i] ← V[:, i] + L[:, s, t]
8:          end for
9:          V[:, i] = NORMALISE(V[:, i])
10:         i ← i + 1
11:     end for
12:     W ← V[:, 0]
13:     for j ← 1 to |S| do
14:         W ← W ⊙ V[:, j]
15:     end for
16:     return W
17: end function
```

Given two distributions, $\boldsymbol{X_1}$ and $\boldsymbol{X_2}$, the TOST is used in ISEQUIVALENT function to test for equivalence. From the discussion in Section 2.2.2.3 it follows that when t_0 and t_1 from Equations 6.3 and 6.4 are both above a certain threshold value, t_β, the two distributions $\boldsymbol{X_1}$ and $\boldsymbol{X_2}$ are determined to be equivalent. Here, s_1 and s_2 are standard deviations of the two distributions, n_1 and n_2 are the number of samples in each distribution and ω_l and ω_u are the boundary values for the TOST.

$$t_0 = \frac{\overline{\boldsymbol{X}}_1 - \overline{\boldsymbol{X}}_2 - \omega_l}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} \tag{6.3}$$

Algorithm 6.2 Is Equivalent

$\textsc{Select}(\boldsymbol{Y}, g)$	Selects elements that belong to the group g. In case of TermTraces format, the traces are divided as two halves for each input configuration. Hence, first and second halves of traces are returned when $g = 0$ and $g = 1$.
$\textsc{Mean}(\boldsymbol{Y})$	Mean of $\boldsymbol{Y}$.
$\textsc{Std}(\boldsymbol{Y})$	Standard deviation of $\boldsymbol{Y}$.

function IsEquivalent($\boldsymbol{X_{RR}}, \boldsymbol{X_{FR}}$)
  $\boldsymbol{X_1} = \textsc{Select}(\boldsymbol{X_{RR}}, 0)$
  $\boldsymbol{X_2} = \textsc{Select}(\boldsymbol{X_{RR}}, 1)$
  $\mu = \textsc{Mean}(\boldsymbol{X_1} - \boldsymbol{X_2})$
  $w = \textsc{Std}(\boldsymbol{X_1} - \boldsymbol{X_2})$
  $\omega_u = \mu + t_\alpha w / \sqrt{|\boldsymbol{X_1}|}$
  $\omega_l = \mu - t_\alpha w / \sqrt{|\boldsymbol{X_1}|}$
  $\boldsymbol{Y_1} = \textsc{Select}(\boldsymbol{X_{FR}}, 0)$
  $\boldsymbol{Y_2} = \textsc{Select}(\boldsymbol{X_{FR}}, 1)$
  $s = \sqrt{\frac{\textsc{Std}(\boldsymbol{Y_1})^2}{|\boldsymbol{Y_1}|} + \frac{\textsc{Std}(\boldsymbol{Y_2})^2}{|\boldsymbol{Y_2}|}}$
  $t_0 = \left(\textsc{Mean}(\boldsymbol{Y_1}) - \textsc{Mean}(\boldsymbol{Y_2}) - \omega_l\right) / s$
  $t_1 = \left(\omega_u - \left(\textsc{Mean}(\boldsymbol{Y_1}) - \textsc{Mean}(\boldsymbol{Y_2})\right)\right) / s$
  return $(t_\beta < t_0) \wedge (t_\beta < t_1)$
end function

$$t_1 = \frac{\omega_u - (\overline{\boldsymbol{X}}_1 - \overline{\boldsymbol{X}}_2)}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} \tag{6.4}$$

Determining the boundary values to be used in a TOST is done primarily according to two paradigms [Pardo, 2013]. The first is the *bioavailability* paradigm. In the bioavailability paradigm, the boundaries are regarded as undesirable values for rejecting the null hypothesis. The tests are constructed to fail (i.e. fail to reject the null hypothesis) with a high probability when the mean differences are closer to the boundary values. The second approach, *quality engineering*, considers the boundary values as desirable values and therefore is less strict on the failure of the test when mean differences are closer to boundary values. As their names suggest they are used in their respective fields and are constructed towards the goals of each field.

We selected the *quality engineering* approach to determine boundary values for our TOSTs as we do not regard the boundary values as undesirable. We calculate the

Algorithm 6.3 Find Leaky Terms

$\boldsymbol{L_{FR}}$	A TermTraces array that holds the traces from fixed vs. random test. See Figure 6.1 for the mapping of term values from Equation 6.2.
$\boldsymbol{L_{RR}}$	Same as $\boldsymbol{L_{FR}}$ but holds traces from random vs. random test.
$\mathcal{S}$	Set of d sample points that participate in the leakage.
$\mathcal{T}$	Set of all terms that are in ELMO*.
$\odot$	Elementwise multiplication operator.

```
1:  function FLT(L_FR, L_RR, S, T)
2:      r ← {}
3:      for s ∈ S do
4:          W ← NPS(L_FR, S \ s, T)
5:          for t ∈ T do
6:              u ← T \ t
7:              Z ← NPS(L_FR, s, u)
8:              X ← NPS(L_RR, S, u)
9:              Y ← Z ⊙ W
10:             if IsEquivalent(X, Y) then
11:                 r ← r ∪ {(s, t)}
12:             end if
13:         end for
14:     end for
15:     return r
16: end function
```

boundary values for the TOST from the data collected in $\boldsymbol{L_{RR}}$ trace set. The two power value distributions from the random vs. random input configuration are used because they are equivalent. The ideal distributions for this purpose are the ones that are measured with the same inputs but without the leaky interactions. As these ideal distributions are immeasurable, we use the non leaky power value distributions from the random vs. random input configuration. The boundaries are calculated as shown in Equations 6.5 and 6.6. The mean difference (μ) and the variance of differences (s) are calculated from $\boldsymbol{L_{RR}}$ power traces. n is the number of samples.

$$\omega_u = \mu + t_\alpha \frac{s}{\sqrt{n}} \tag{6.5}$$

$$\omega_l = \mu - t_\alpha \frac{s}{\sqrt{n}} \tag{6.6}$$

6.3 The Monte Carlo Method

While elimination of terms is efficient, it may sometimes fail. For example, if multiple model terms leak the same share, removing any one term will not eliminate the leak. In such cases the slower but more versatile method of Monte Carlo simulations is used to find the terms that are responsible for the leakage.

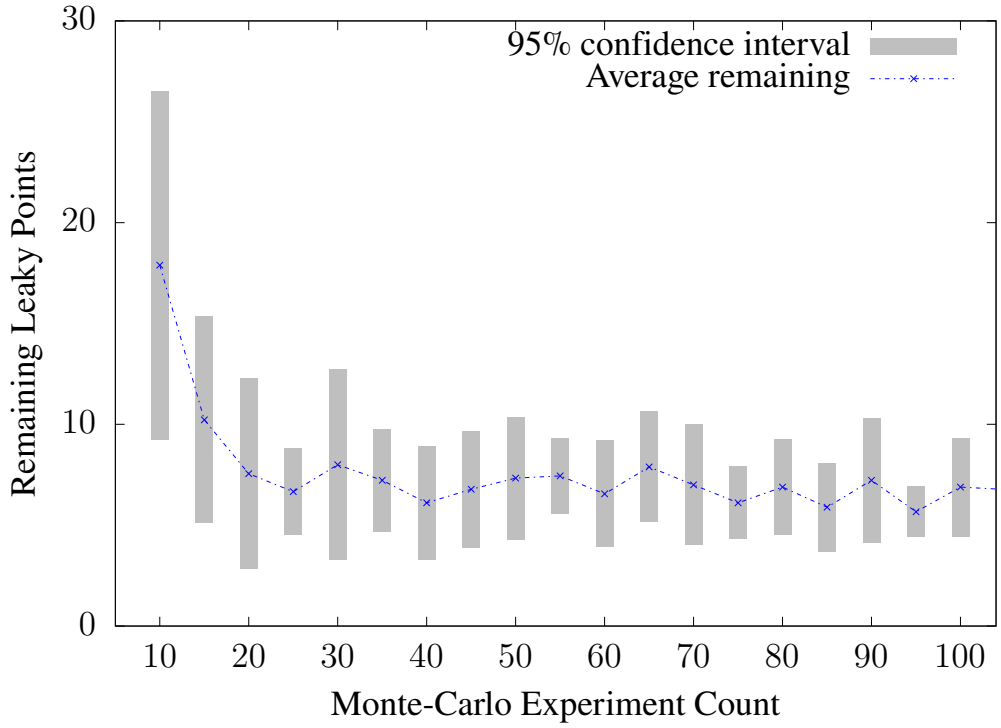


Figure 6.2: Effectiveness in removing leakage of Monte Carlo method for increasing number of experiments.

Monte Carlo simulations are a class of algorithms that use repeated random sampling to solve problems [Kroese et al., 2014]. In this approach, a preset number of random experiments are run where in each experiment a random subset of the model terms are evaluated using the t-test for leakage. We use the term *reduced model* to refer to a model created from such a subset of the terms. A scoring system awards a point for each term that participates in a reduced model that ends up being leaky. After a preset number of experiments are done, the terms which have outlying high scores are selected as terms that contribute most to the leakage.

The preset number of random experiments were selected from a performance analysis done for a code segment from a masked implementation of Xoodoo [Daemen et al., 2018a] cipher by only using Monte Carlo method to detect and remove leakage. After gathering 100,000 traces from this implementation and running the initial t-test it had 45 total leakage points. Figure 6.2 shows the reduction of remaining leaky points as the number of Monte Carlo experiments is gradually increased. Higher number of experiments improves detection of root causes, but after 50 or so experiments the reduction of leakage was nearly constant. After evaluation of 1000 experiments the improvements were minimal. Therefore, we selected 50 as the optimal experiment number.

6.4 Implementation

The implementation of Rosita++ is similar to the implementation of Rosita as detailed in Section 5.5. This section details the notable differences between the implementations of Rosita and Rosita++. Similar to Rosita, Rosita++ also includes the same set of executables `elmo`, `emulatetraces` and `rosita` (see Section 5.5 for details).

Additions to the flow of data within Rosita are shown in green in Figure 6.3. A significant difference in the output produced by `emulatetraces` is the additional recording of all emulated power values with individual values for each term of the power model. This means that the values of each term in Equation 6.2 is recorded. In the root cause detection method of elimination of terms used in Rosita++, the target executable is emulated twice in two different input configurations. One is fixed vs. random and the other is random vs. random. The additional run of a test with all random inputs is required by the elimination of terms algorithm discussed in Section 6.2.

All emulated power traces are stored on disk by appending new ones to the end of a trace file. These traces are processed sample-wise when root cause detection is done. A single sample point from all traces are accessed at once. Using column-major order when reading values that were written to disk in a row-major order incurs significant penalties in performance [Thiyagalingam et al., 2003]. As a solution,

we first transposed[1] the power traces matrix[2] offline and then used the resulting transposed matrix for root cause detection with Rosita++. Transposing resulted in a major performance gain for root cause detection with large numbers of emulated traces (e.g. more than two million). This process is handled by the new addition of the `transposer` component as shown in Figure 6.3.

The Leaky Term Searcher component in Rosita++ is in charge of conducting the search for leaky terms where multivariate leakage is detected from the emulation of power values in `emulatetraces`. This is done by either employing the method of elimination of terms or employing the Monte Carlo method. This process is a CPU intensive task with multiple read only references to the content of power value trace files (one each from fixed vs. random and random vs. random experiments) that were generated in the earlier step by `emulatetraces`. Therefore, we implemented this as a multi-threaded component with read only access to memory mapped regions of the two trace files.

The `rosita` program was modified to facilitate a second run as shown in the bottom part of Figure 6.3. Two runs are required due to the nature of multivariate root cause detection. The final result of multivariate root cause detection is not finalised until all combinations of the power values from different sample points have been analysed. Therefore, the labels that identify the leaky terms are stored in a separate file that is read by `rosita` in the next run, finishing the code rewriting operation by overwriting the leaky assembly code segments with the non-leaky code patterns listed the pattern library (i.e. `ARMMatcher.py`[3] for ARM ISA related code patterns) similar to the same operation in Rosita.

Similar to how Rosita worked (see Section 5.4 for details), Rosita++ also requires to be run in multiple iterations to guarantee that the leakage was actually fixed. The reason behind this is that at some trace count levels, the t-test value for the final differential voltage value remains leaky, but it is not enough for the root cause detection procedure. Therefore, the trace count needs to be increased in later

[1] https://github.com/0xADE1A1DE/Rositaplusplus/blob/master/PWMODEL/src/transposer.cpp

[2] We collapsed two dimensions (term and sample numbers) of the three dimensional array of power trace described in Algorithm 6.3 to one dimension and treated it as a matrix

[3] https://github.com/0xADE1A1DE/Rositaplusplus/blob/master/ROSITAPP/ARMMatcher.py

iterations to facilitate the successful detection of root causes.

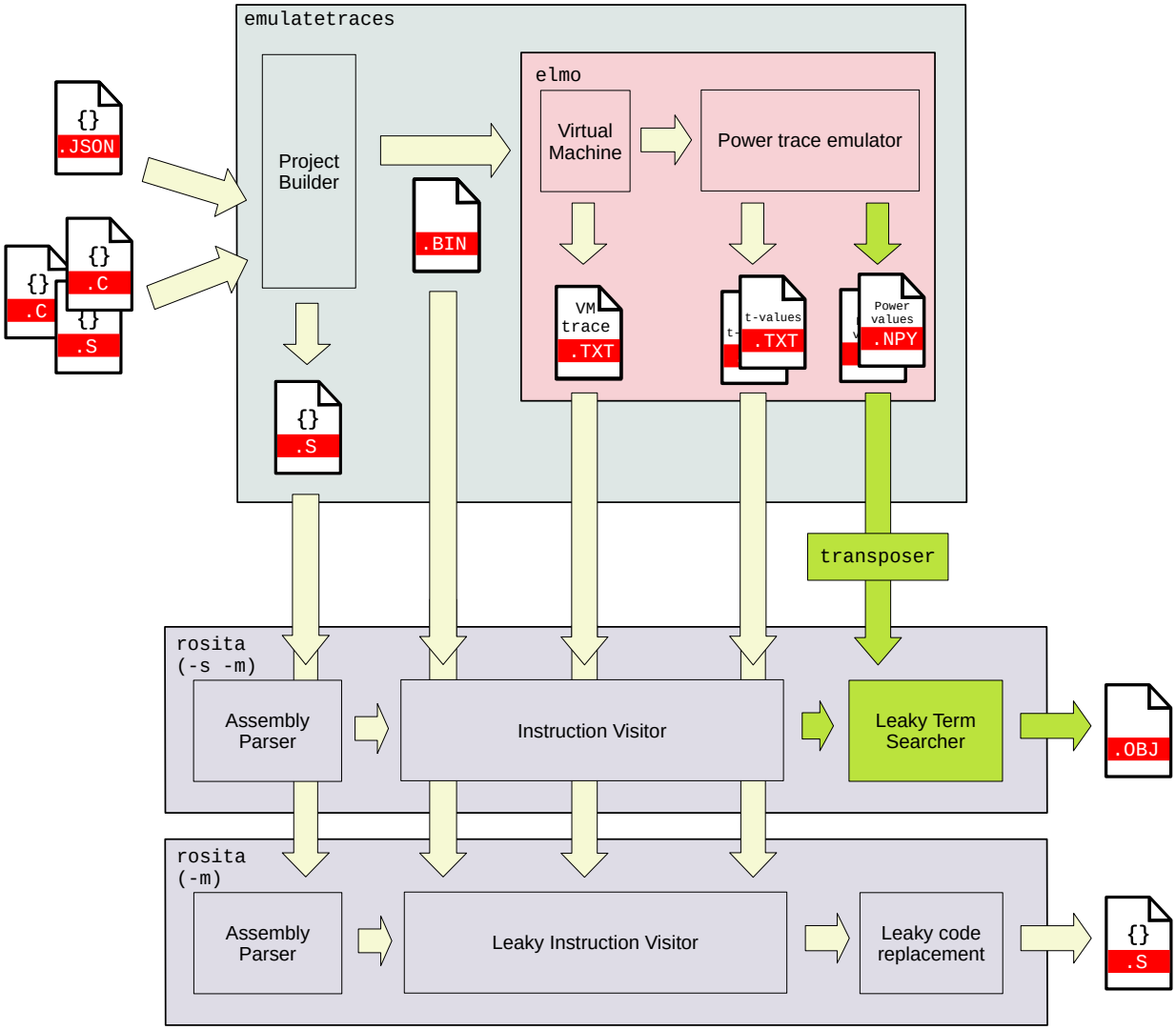


Figure 6.3: Pearson correlation coefficient for the cross operand test.

6.5 Evaluation

Rosita++'s evaluation was done on second order masked PRESENT and Xoodoo ciphers, and on a cryptographic primitive for converting between Boolean masking and arithmetic masking. For evaluation of third order masked implementations, we used an example implementation that uses four shares in a Boolean masking scheme. Before continuing with the multivariate evaluation, we tested all implementations for possible first order leakage. PRESENT and Xoodoo were implemented as threshold implementations. Note that threshold implementations with three shares provides provable first-order security, but only limited protection against the second-order attacks [Nikova et al., 2006]. The reason for this is that as many as two shares can be used in a single operation in a three share threshold implementation due to the non-completeness property of threshold implementations. Therefore, we can expect that diminished second-order leakage may occur for both PRESENT and Xoodoo implementations.

Increasing t-test threshold value. The typical t-test threshold used for TVLA is 4.5 [Goodwill et al., 2011]. However, as the number of sample points increase, the false-positive rate also increases. This is due to the fact that we observe the results of individual t-tests at each sample point. Therefore, as the number of observations increases, the threshold would need to be increased [Balasch et al., 2015; Ding et al., 2017]. We increased the t-test threshold for our tests following the method described in Section 2.3.5.

6.5.1 Implementations under test

This section lists the implementations we used for testing the effectiveness of Rosita++. All three share implementations were emulated for 500,000 power traces. This number of emulated traces were enough to fix the code segments under test such that that significant leakage was not observable at two million power traces from the physical setup. This is a significantly lower number of traces than the power traces we used to detect leakage the real device which was two million traces. The four share example implementation described below was emulated for only 200,000 when it detected the root causes for the leakage. It was possible to detect the leakage at an even lower number of power traces, but more power traces were required for the

root cause detection. This shows a significant reduction in noise from the emulated traces. The source code for all the implementations listed in this section are available in `./TESTS/` path at https://github.com/0xADE1A1DE/Rositaplusplus.

We used three cryptographic primitives, which represent different points in the design space of symmetric cryptography. The choices were also limited as second order masked implementations were scarce.

6.5.1.1 3-share Boolean-to-arithmetic conversion

Boolean-to-arithmetic mask conversion [Goubin, 2001] 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 such as SHA-2 [National Institute of Standards and Technology, 2015a], ChaCha [Bernstein, 2008], Blake [Aumasson et al., 2009], Skein [Ferguson et al., 2010], IDEA [Lai and Massey, 1991], and RC6 [Rivest et al., 1998]. We implemented and evaluated the second-order Boolean-to-arithmetic masking of Hutter and Tunstall [2019, Alg. 2].

The conversion procedure takes Boolean shares $x' = x \oplus r_1 \oplus r_2$, r_1 and r_2 as input, where r_1 and r_2 are random in $\mathbb{Z}_{2^{32}}$ and x is the secret value. The procedure uses three additional masks γ_1, γ_2, and α also random in $\mathbb{Z}_{2^{32}}$ for protecting the operation. 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 Hutter and Tunstall [2019] and therefore, we do not expect to see leakage in an implementation protected with Rosita++. We implemented this algorithm in assembly taking care to keep the shares separate.

6.5.1.2 3-share PRESENT

PRESENT is a block cipher based on a substitution permutation network, which was proposed by Bogdanov et al. [2007]. 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 threshold implementations with three shares, as described by Sasdrich et al. [2018, Alg. 3.2]. We used the code shown in Listing 6.1 that implements a part of the PRESENT S-box, involving three shares x^1, x^2, x^3 and the lookup table T. The table

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

Listing 6.1: PRESENT code segment under test.

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

Listing 6.2: Xoodoo code segment under test.

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.

6.5.1.3 3-share Xoodoo

Xoodoo was proposed by Daemen et al. [2018a] and a reference implementation is available from Bertoni et al.. Xoodoo [Daemen et al., 2018a] is a modern cryptographic primitive that underlies multiple higher-level primitives [Daemen et al., 2018b]. We implemented a three-share version of Xoodoo, building on the non-linear χ layer from Keccak.

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 6.2 shows the algorithm segment that we evaluated, which forms part of the start of the Xoodoo χ function. Our initial C implementation showed first-order leakage caused by the code optimiser merging shares. We therefore manually implemented the code under test in assembly, ensuring that shares were not merged.

6.5.1.4 4-share Synthetic Example

For an example third-order leakage analysis, we opted to use a four share synthetic example, chosen to reduce the large computational overhead to a manageable level. The example that we use is listed in Listing 6.3. All 32 bit shares are stored in the array that is pointed to by `r1`. These are loaded into the registers `r3`–`r6` using `ldr` instructions. The `push` and `pop` pairs that are in Lines 2 and 6 separate the interaction

between the `ldr` instructions. For more details about how we found out such effects, see Section 5.2.1. The `push` and `pop` instructions act as a barrier to the interactions that happen through the memory by resetting the internal state. Observe that there is no such barrier between Lines 9 and 10. This results in a combination of all shares at Line 10. In this evaluation we expect Rosita++ to first emulate this code segment correctly and detect multivariate leakage at Line 10 and then detect the root causes for the leakage and apply code fixes to remove it.

```
1  ldr r3, [r1, #0]
2  push {r7}
3  pop {r7}
4  ; nop padding
5  ldr r4, [r1, #4]
6  push {r7}
7  pop {r7}
8  ; nop padding
9  ldr r5, [r1, #16]
10 ldr r6, [r1, #20]
11 ; nop padding
```

Listing 6.3: Synthetic example.

6.5.2 Tools for leakage evaluation

We developed a high-performance, configurable set of tools for executing operations on signal trace files. These operations include online calculation of t-test values [Schneider and Moradi, 2015], performing online CPA attacks [Pebay, 2008], static alignment of signals using cross-correlation and application of filters to signal trace files. We note that there have been analyses of even larger sets of power traces than ours [Cnudde et al., 2015, 2016], but we failed to find any publicly available tools that fulfil all our analysis needs. Publicly available tools such as Jlsca[4], SCAred[5], Lascar[6] only offered limited capabilities. Our tool set, TraceTools is publicly available at https://github.com/0xADE1A1DE/tracetools.

[4] https://github.com/Riscure/Jlsca
[5] https://gitlab.com/eshard/scared
[6] https://github.com/Ledger-Donjon/lascar

TraceTools is written in C++ and offers multi-threaded support for calculating univariate and bivariate Welch's t-test values. Our tool uses multiple passes when required which are executed by reading the entire file from start to end. This process is significantly sped up by using solid state drives for storing the trace files. TraceTools was designed such that it acts as a library that offers utilities to various trace processing applications. TraceTools Trace processing activities such as Welch's t-test analysis or application of filtering to each trace are implemented as clients of the library. The design of TraceTools enables using command-line arguments to specify the properties and ranges of samples which are to be processed rather than using a script based approach used in commonly used SCA (Side Channel Analysis) tools such as Jlsca, Lascar or SCAred. This saves time that analysts spend on reading documentation and limits writing new code to when new functionality is required. Operations like the creation of nth-order traces, normalisation of traces and calculation of Pearson correlation coefficient for CPA attacks are implemented as separate building blocks that can be used in different analysis pipelines. Due to this, the creation of new workflows were made simpler.

We conducted a benchmarking test with a trace set containing 1,715,580 power traces each having 1,000 samples gathered from the physical experiment of Boolean to arithmetic mask conversion. All samples were stored in double precision floating point in Riscure Inspector TRS format[7] and as NumPy array on-disk format[8] . The time required to conduct univariate t-test by using each tool is shown in Table 6.1. We chose the univariate t-test as a benchmark test since it was supported by all tools with minimal adaption code. Jlsca was nearly as fast as TraceTools, taking only 1.5 times longer to conduct the univariate t-test results. Lascar and SCAred were significantly slower and took 18.4 and 9.6 times longer than TraceTools for the analysis. All tests were run in single-threaded mode in all tools as some do not support multi-threaded operation.

We also used the same trace set to run bivariate t-tests using TraceTools and Jlsca. As shown in Table 6.2, TraceTools performed twice as fast as Jlsca in single-threaded bivariate t-test analysis.

[7] https://github.com/Riscure/python-trsfile
[8] https://numpy.org/doc/stable/reference/generated/numpy.save.html

Tool	Wall Clock Time (s)	
	TRS	NumPy
TraceTools	4	5
Lascar	-	92
SCAred	-	48
Jlsca	6	-

Table 6.1: Univariate t-test analysis time.

Tool	Wall Clock Time
TraceTools	26:18
Jlsca	1:03:36

Table 6.2: Bivariate t-test analysis time.

We conducted CPA attacks based on the trace data from DPA Contest V2[9]. Table 6.3 shows the benchmark results we obtained for CPA attacks on the first 20,000 traces done using SCAred and TraceTools. SCAred is multi-threaded and uses all available CPU cores (i.e. eight in our test machine) for the attack whilst TraceTools only uses a single thread for CPA attacks. We listed both the wall clock time and total CPU time (taken from `/usr/bin/time`) which show that the total CPU usage of TraceTools' CPA attack is less than SCAred's.

Tool	Wall Clock (s)	CPU Time (s)
TraceTools	13.59	13.59
SCAred	6.43	22.32

Table 6.3: CPA attack time.

6.5.3 Results

6.5.3.1 Bivariate evaluation

This section lists the results gained from the evaluation of Rosita++. The bivariate leakage analysis done in this section uses product between samples from two

[9] https://www.dpacontest.org/v2/download.php

sample points was used as the combination function and univariate Welch's t-test on the combined traces. The heatmaps shown in Figures 6.4, 6.5, 6.8, 6.9, 6.12 and 6.13 are mirrored across the axis through $(0, 0)$. These were drawn from the leakage that resulted from two million power traces each which were gathered from the implementations listed in Section 6.5.1 run on a fixed vs. random input configuration on a ST Microelectronics STM32F030 Discovery evaluation board. We used TraceTools for the evaluation of the traces that were collected from our physical test setup. Refer to Section 3.2 for details on our test setup. The wall clock time that each analysis took with TraceTools is shown in Table 6.4. Each result in Table 6.4 is from analysing two million power traces from the real device. We used a desktop PC with an Intel Core i9-10900K CPU and 32 GBs of RAM to run analysis using TraceTools and to run Rosita++. The massive increase in time required for the bivariate t-test analysis is due to the large increase in combinations when number of samples increase from 1000 to 3500 (i.e. ${}^{3500}C_2/{}^{1000}C_2 \approxeq 12$).

Trace set	**Samples**	**Wall Clock Time**
Xoodoo original	1000	4:51
Xoodoo fixed	1400	33:50
PRESENT original	1400	28:31
PRESENT fixed	3500	7:02:00
Boolean-to-arithmetic original	1000	4:18
Boolean-to-arithmetic fixed	1200	8:51

Table 6.4: Bivariate analysis time.

Boolean-to-arithmetic mask conversion. We tested the entire implementation of Boolean-to-arithmetic mask conversion algorithm as described in Section 6.5.1. The heatmap in Figure 6.4 shows the univariate t-test values for the combined power traces from the physical experiment before applying fixes using Rosita++. The absolute peak t-test value observed from these values was 9.13 which meant that, despite our efforts to keep shares separate, unintended interactions occurred in the assembly code implementation of the Boolean-to-arithmetic conversion. We then used Rosita++ to apply fixes to this implementation. The times taken to emulate and analyse the three share Boolean-to-arithmetic implementations are shown in Figure 6.7 and Figure 6.6. The analysis time is the time spent on root cause detection. The variations in time

depicted in Figure 6.7 follows the number of leaky points detected at each trace count level. The final heatmap in Figure 6.5 shows that ROSITA++ was successful in eliminating all detected leakage. This was evident from the absolute peak t-test value of Figure 6.5 which was 3.91.

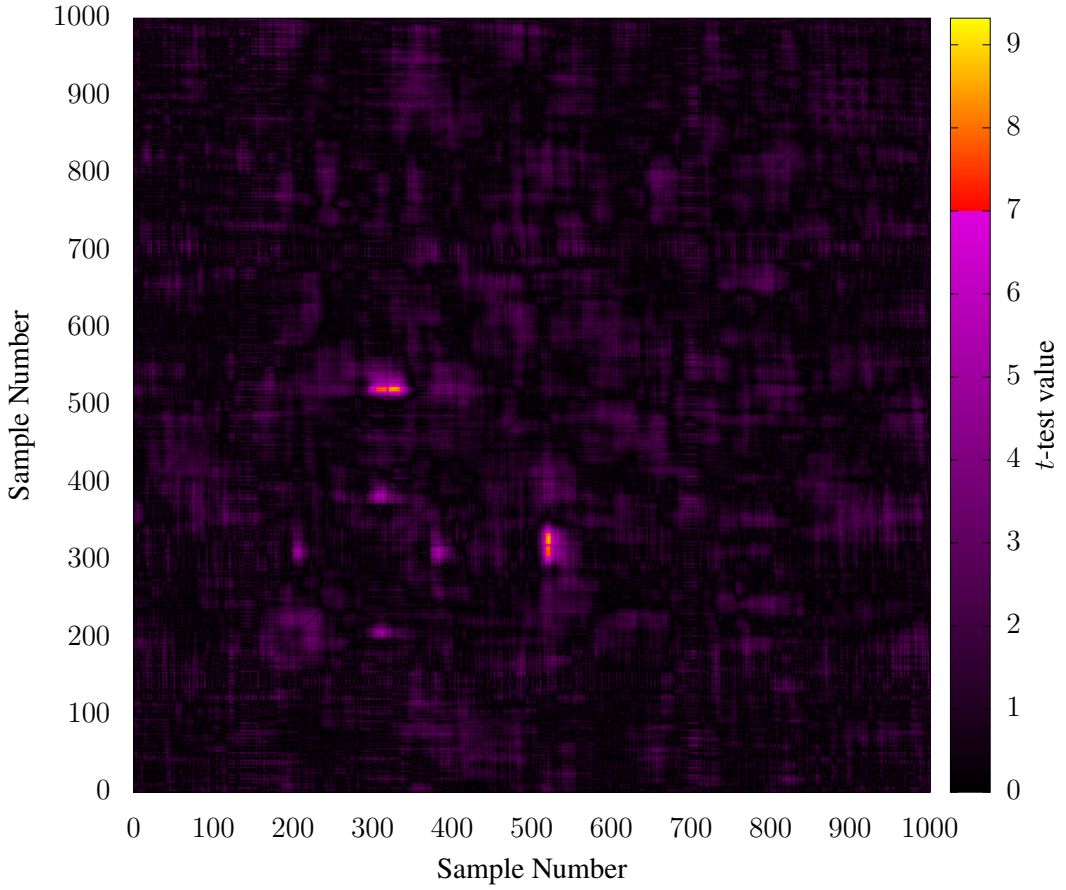


Figure 6.4: t-test values for Boolean-to-arithmetic before applying code fixes, peak t-test value: 9.13.

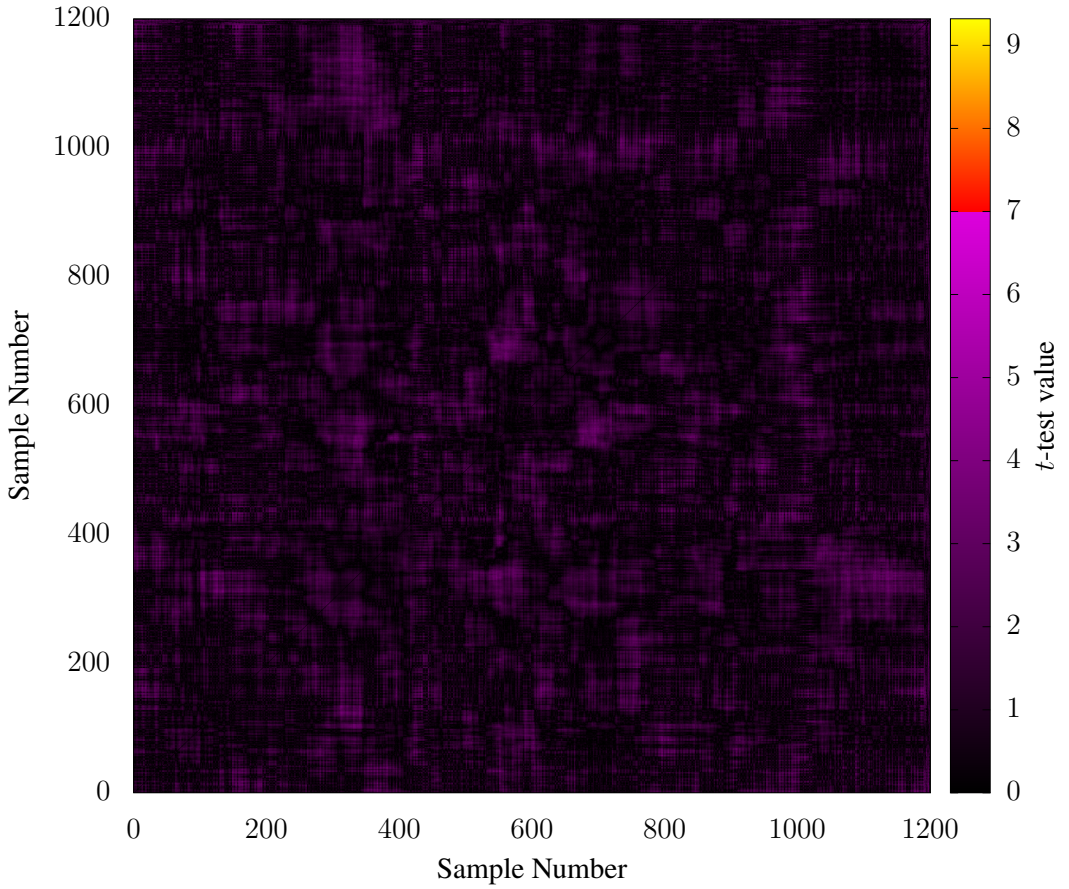


Figure 6.5: t-test values for Boolean-to-arithmetic after applying code fixes, peak t-test value: 3.91.

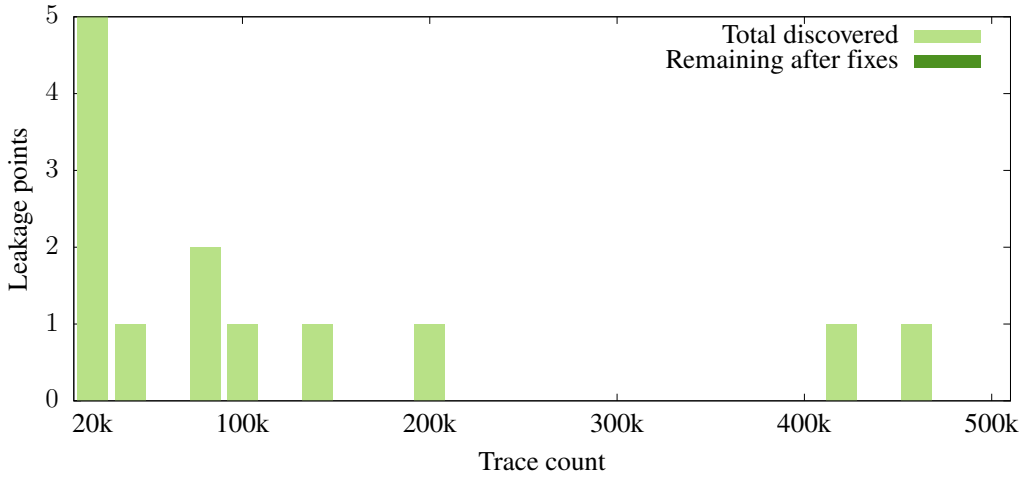


Figure 6.6: Number of leaky instructions before and after fixes for Boolean-to-arithmetic.

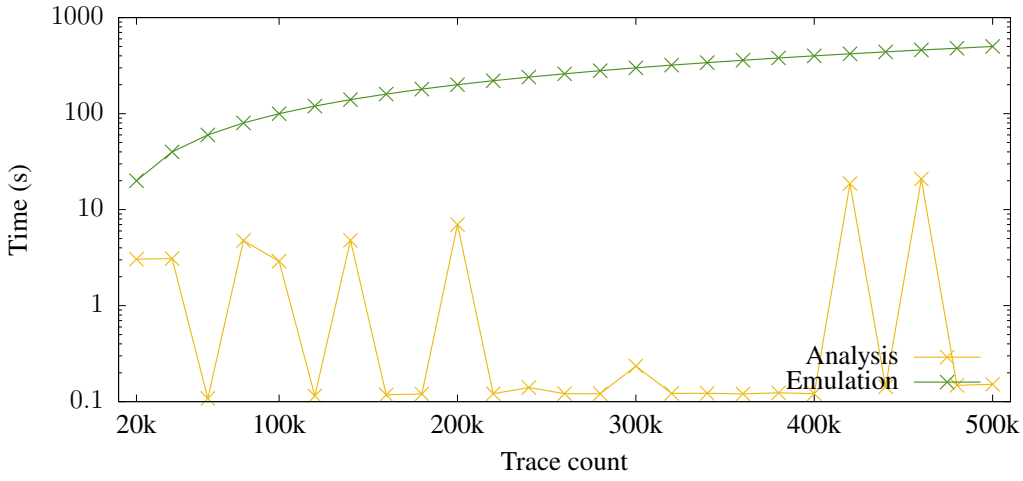


Figure 6.7: Time taken to emulate Boolean-to-arithmetic by `elmo`.

PRESENT. The part of the S-box look up of a three share PRESENT implementation listed in Section 6.5.1 was also tested by us on our test device. This resulted in the heatmap shown in Figure 6.8. The maximum t-test value observed was 55.13. This three share implementation was then fixed through Rosita++, the time taken for emulation and analysis is shown in Figure 6.11 and the number of detected and fixed leaks are shown in Figure 6.10. The time for analysis depicts the difference in time spend on root cause detection when there are many leaky points versus when there are a few leaky points.

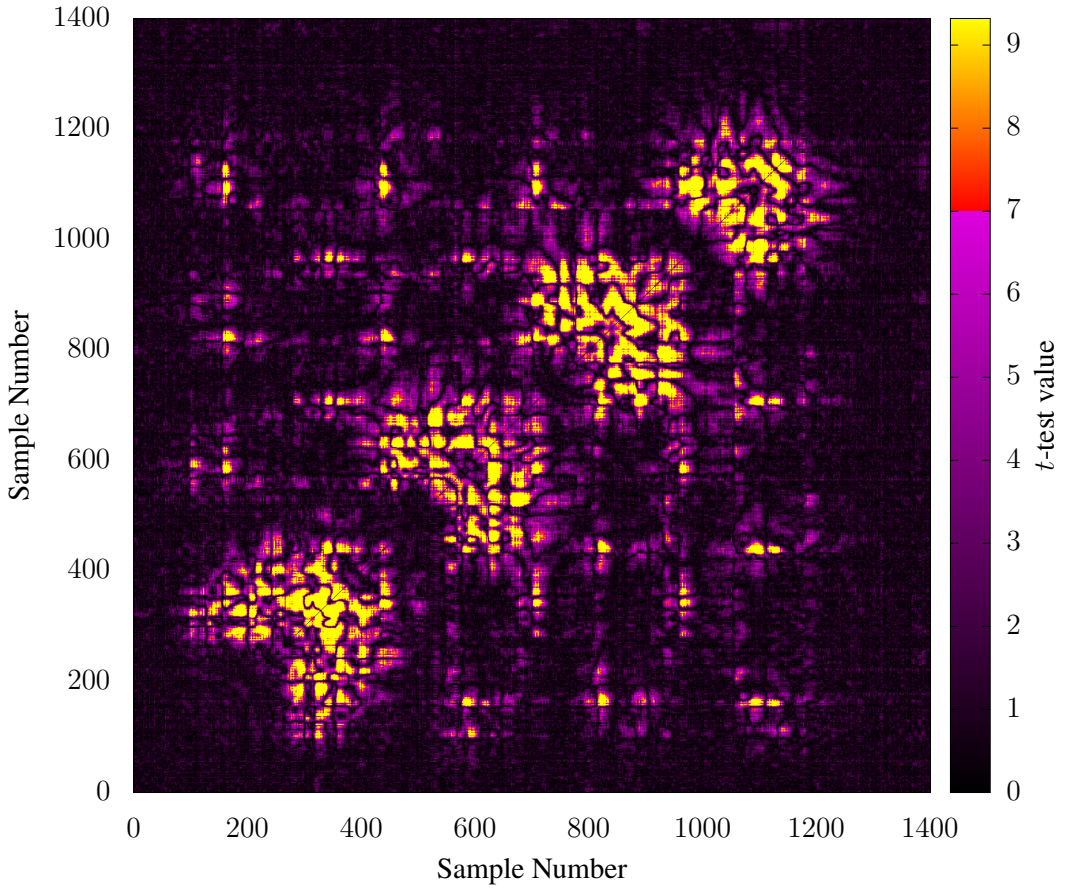


Figure 6.8: t-test values for PRESENT before applying code fixes, peak t-test value: 55.13.

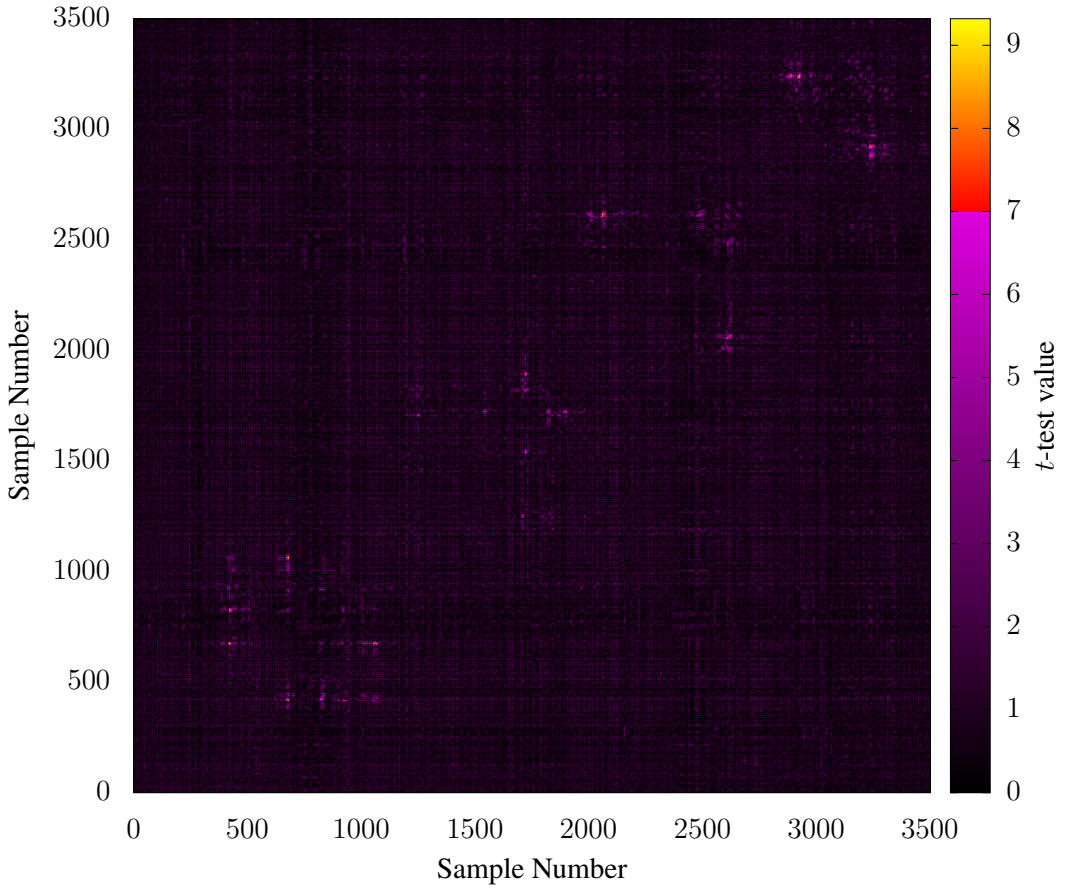


Figure 6.9: t-test values for PRESENT after applying code fixes, peak t-test value: 12.38.

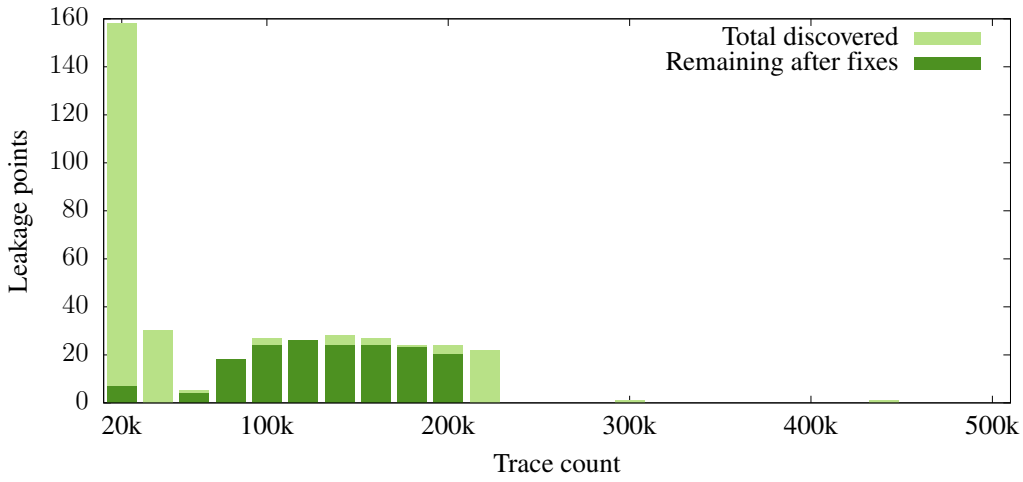


Figure 6.10: Number of leaky instructions before and after fixes for PRESENT.

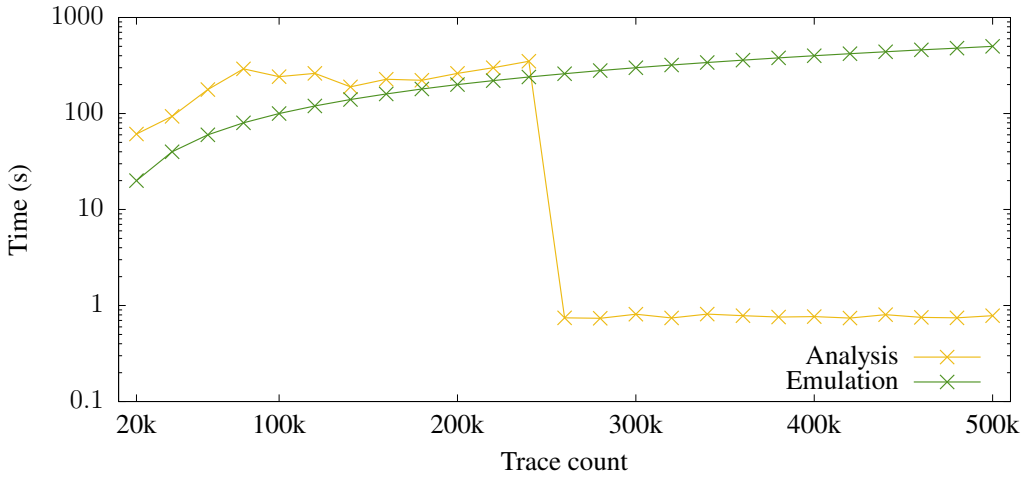


Figure 6.11: Time taken to emulate PRESENT by `elmo`.

We noticed several locations with significant remaining leakage in Figure 6.9 after evaluating the fixed version of PRESENT. Listing 6.4 shows the first leaky segment of the code corresponding to sample point (700, 440). After investigating the leaky points further we found out that this is due to the leakage in S-box of PRESENT in addresses. We confirmed the exact values that participate in the leakage through conducting CPA attack against the values of combined shares.

```
1 ldrb  r2, [r4, #16]
```

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

Listing 6.4: Leaky code segment of fixed PRESENT.

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 (not shown in listing). Both points showed high correlation to the corresponding share values. We confirmed this leakage pattern by reproducing the same effect in a separate fixed vs. random experiment which had only two shares used in a `ldrb` instruction for addresses. It showed significant first order leakage at 200,000 traces. This meant that some significant leakage was missed by Rosita++ as it does not support fixing leakage that happens on the address-bus. However, when comparing the leakage detected in the physical experiment against the ones that were fixed by Rosita++, it had eliminated 99.03% of leakage.

χ function of Xoodoo. Figure 6.12 shows the leakage that was detected from the power traces from our test device for the χ function of three share Xoodoo. The highest value that was detected from this t-test was 70.32. Similar to before, we used Rosita++ on this implementation. The time spent on emulation and analysis is depicted in Figure 6.15 and the discovered and fixed number of leaks are shown in Figure 6.14. The variations of time shown in Figure 6.15 follow the amount of leaky points that were analysed at each trace count level. A single leak was left over due to Rosita++ not having enough information for the root cause detection at 500,000 power traces. However, the heatmap from the fixed version of the code shows that all practically observed leakage has been eliminated.

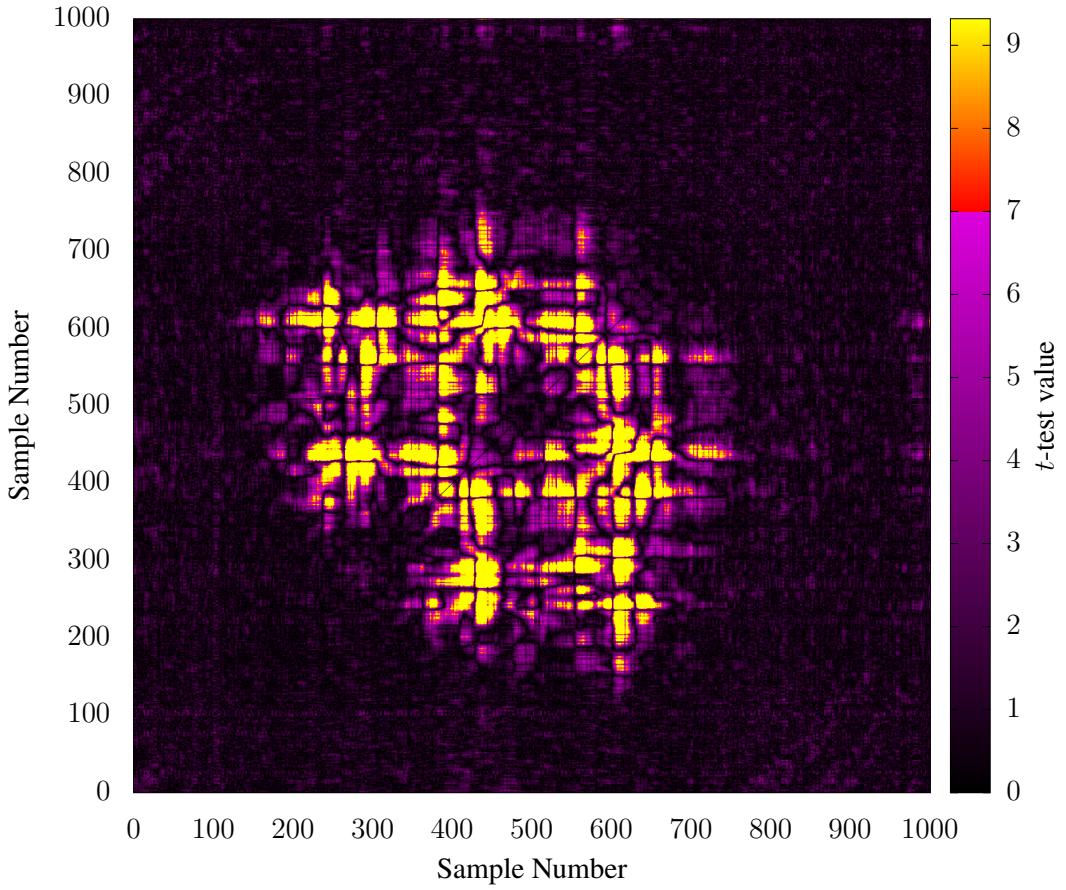


Figure 6.12: t-test values for Xoodoo before applying code fixes, peak t-test value: 70.32.

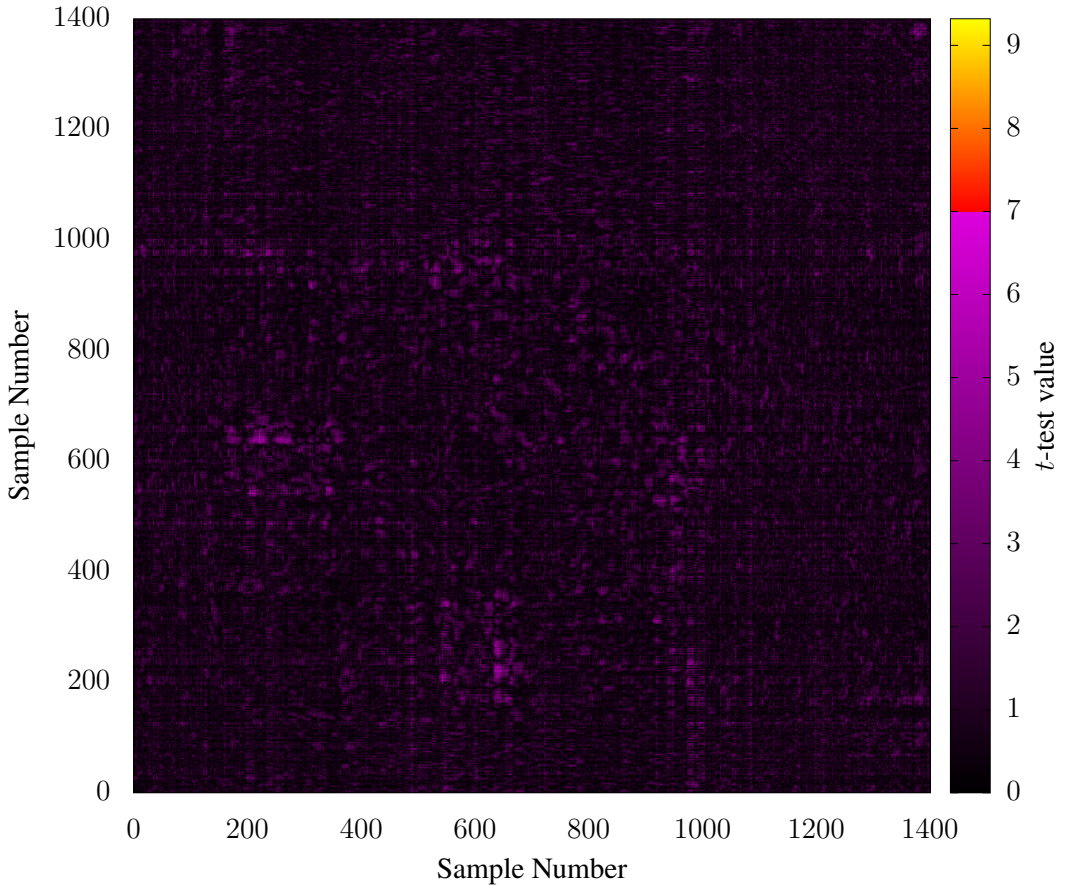


Figure 6.13: t-test values for Xoodoo after applying code fixes, peak t-test value: 6.44.

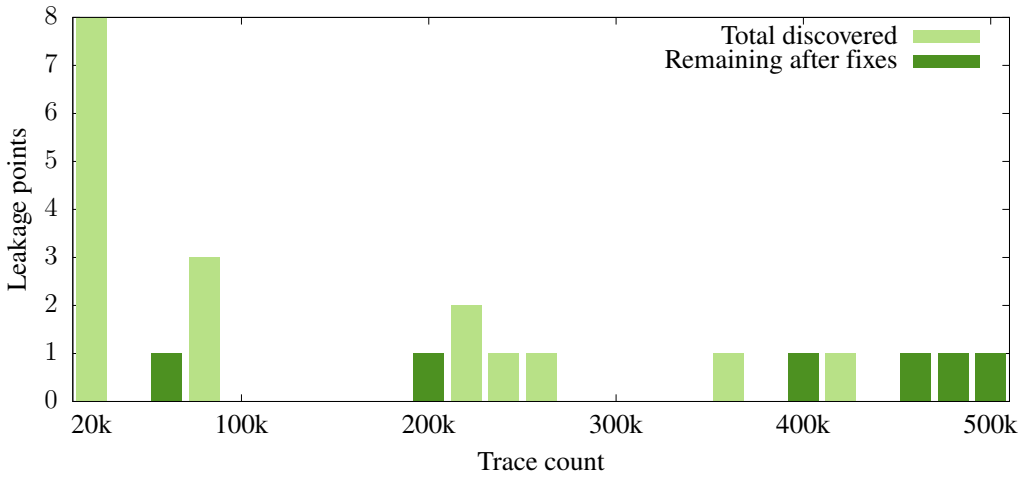


Figure 6.14: Number of leaky instructions before and after fixes for Xoodoo.

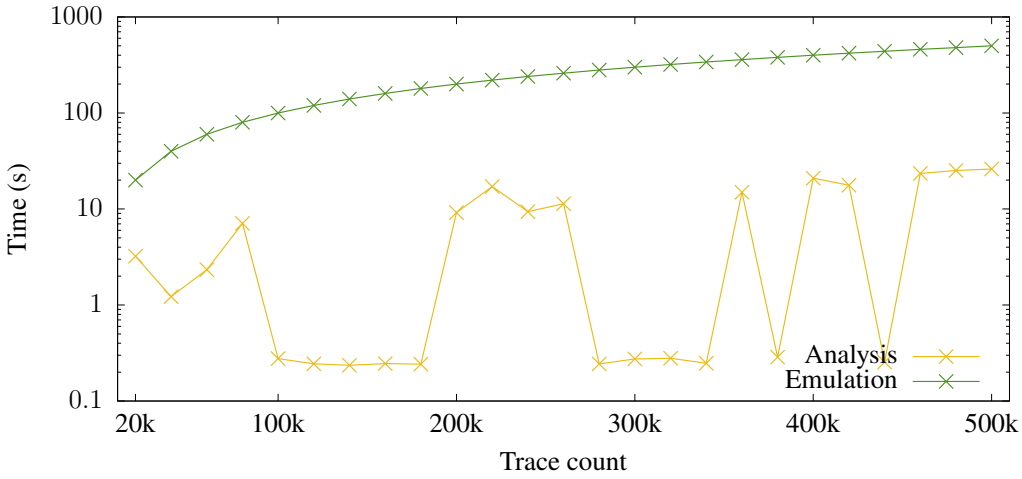


Figure 6.15: Time taken to emulate Xoodoo by `elmo`.

6.5.3.2 Trivariate evaluation

In trivariate evaluation, we limited ourselves to only evaluate the four share synthetic example from Section 6.5.1 due to the high computational overhead of trivariate analysis. We had to increase the trace count to 30 million traces so that it was possible for us to detect significant leakage from the power traces from the test device. We visualised the t-test results by a cube shown in Figure 6.16. This cube was drawn by combining all samples from 50 samples ranges near the Lines 1, 5 and 10 in Listing 6.3 and by using the combined samples in an univariate t-test. Then we fixed the four share synthetic example using Rosita++ it only took two million emulated traces to detect and fix the leakage. We then ran the fixed version of code on our device and drew Figure 6.17. The only difference between the procedure used to draw the final cube was that we used 50 samples each from sample points that corresponded to Lines 1, 5 and 13 from the fixed code listed in Listing 6.5. We observed that none of the t-test values in Figure 6.17 were significant meaning that Rosita++ successfully eliminated third order leakage that was detected at 30 million power traces by emulating only two million traces.

```
1  ldr r3, [r1, #0]
2  push {r7}
3  pop {r7}
4  ; nop padding
5  ldr r4, [r1, #4]
6  push {r7}
7  pop {r7}
8  ; nop padding
9  ldr r5, [r1, #16]
10 push {r7}
11 pop {r7}
12 movs r7, r7
13 ldr r6, [r1, #20]
14 ; nop padding
```

Listing 6.5: Synthetic example.

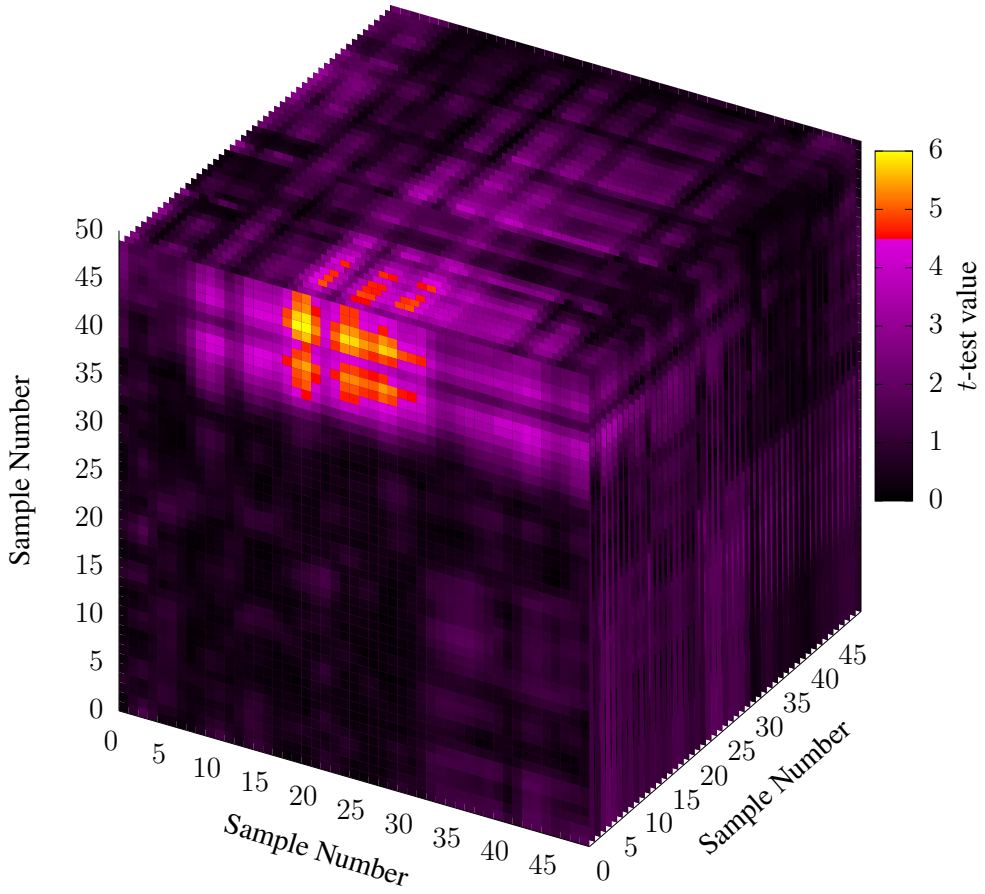


Figure 6.16: t-test results for Listing 6.3 before applying fixes.

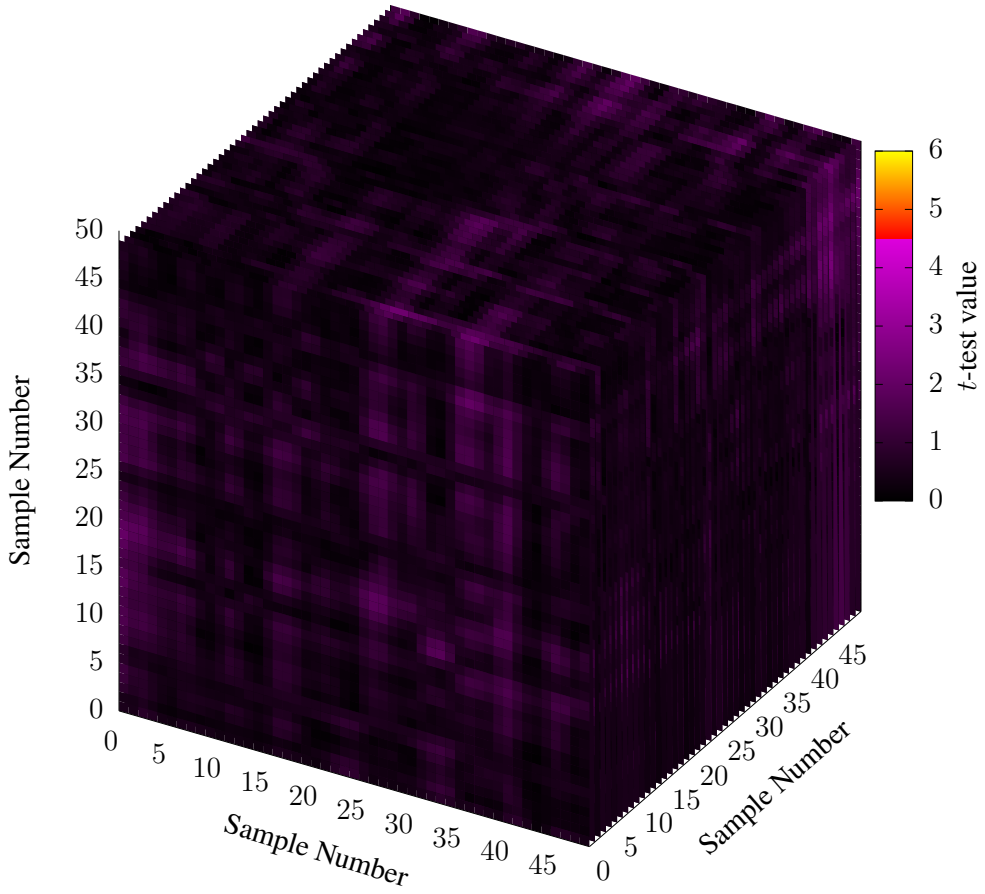


Figure 6.17: t-test results for Listing 6.3 after applying fixes.

6.5.4 Discussion

In all of the tests we have carried out to evaluate Rosita++ it eliminated more than 99% of all observed leakage on the physical experiment. Further, Rosita++ eliminated 100% of observed leakage in all but one implementation that we tested. Similar to the univariate case, we only had to emulate power for a fraction of the number of traces that were collected from the physical experiment. This was highlighted in the synthetic example where we gathered 30 million power traces from the real experiment, only two million emulated power traces were required by Rosita++ to detect and fix the leakage. Similarly, in other tested implementations we only emulated power for 500,000 traces where the we gathered two million power traces from the real experiments.

Rosita++ used Monte Carlo method in one out of 15 root cause detections in the Boolean-to-arithmetic conversion algorithm, 70 out of 262 in PRESENT and four out of 16 in Xoodoo. All other root cause detections were done using elimination of terms. The time spent by Rosita++ to emulate and detect root causes is listed in Table 6.5 and the overhead added by the addition of code fixes to an implementation is shown in Table 6.6. Finally, we conclude that Rosita++ was able to eliminate almost all of the multivariate leakage that it was presented with.

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

Table 6.5: Time taken for emulation and root cause detection for Rosita++.

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

Table 6.6: Performance overhead of fixes applied by Rosita++.

Chapter 7

Conclusions

This thesis explores automatic application of countermeasures to masked software implementations. These countermeasures aim to fix the effects of unintended breaches of the Independent Leakage Assumption (ILA) that are detected using leakage emulation. These are caused by unintended interactions in hardware that end up combining the shares of masked implementations and reducing the effective security. These effects are unique to hardware configurations and therefore, it is not possible to define characteristics of such interactions without access to hardware descriptions. However, such interactions can be modelled after empirical observations [Papagiannopoulos and Veshchikov, 2017]. One of the main contributions of this work is to extend the capability of formulating such models without access to exact hardware descriptions. It is important as the hardware descriptions of many microprocessors are not available to evaluators due to commercial reasons. The methodology presented in this work has been tested empirically in Sections 5.6 and 6.5. These evaluations show that the methods used were effective in eliminating detected leakage. The contributions presented in this thesis were published in the Network and Distributed System Security Symposium (NDSS) 2021 and in ACM SIGAC Conference on Computer and Communications Security (CCS) 2021.

7.1 Contributions

We discovered a gap in research where investigation in to root cause detection methods for power analysis based on emulated power traces was lacking. The obvious benefits are the cost savings and the increases in efficiency for finding and fixing leakage.

Many other methods of root cause detection existed before our work. Notably, solving satisfiability equations [Wang et al., 2019] and static analysis [Bayrak et al., 2015] were competing methods of root cause detection. However, they apply fixes at a general level where specific leakages in different devices have not been taken into consideration. Profiling based leakage emulation solves this by adapting a power model to a certain device's leakage in the profiling phase. After profiling, it approximates the leakage of the device accurately. In this work, a profiling based leakage emulator ELMO [McCann et al., 2017] was adapted for this purpose by adding new features to its power model. These include leakage from memory bus interactions, inter-bus effects and internal state interactions. The modifications that were done to ELMO resulted in ELMO* which support a wider range of leakages. A general methodology was developed in Section 4.1.2 to find internal state related leakage in a given device.

Rosita focuses on applying countermeasures to fix unintentional ILA breaches that result in univariate leakage with respect to first-order masked implementations. Even-though memory bus related transition leakages have been observed in past literature [Papagiannopoulos and Veshchikov, 2017] it has not been incorporated to a profiling based emulator. Compared to ASCOLD [Papagiannopoulos and Veshchikov, 2017], our approach of root cause detection reduces work for an evaluator by only tasking them with the profiling of a new device. Additionally, our system automatically applies fixes and keeps evaluating the new implementation until all detected leakage from emulated traces have been fixed. Rosita was evaluated by using it to fix detected leakage in several leaky code segments from AES, ChaCha and Xoodoo. A discussion of these evaluations is in Section 5.6. The source code of Rosita was published under an open source license[1].

Next, we focused on solving the problem at a more general level by handling multivariate leakage in Rosita++. This work is discussed in detail in Chapter 6. This method employs a model agnostic approach to root cause detection. A discussion of the evaluation of Rosita++ is in Section 6.5, where leaky code segments from second order masked implementations of Xoodoo and PRESENT were protected using Rosita++. Rosita++ was also able to fix third-order leakage detected from a synthetic example.

[1] https://github.com/0xADE1A1DE/Rosita

Additionally, this work contributes a high-performance trace analysis framework written in C++ that is fast enough to evaluate millions of power traces for second and third order leakage in a few hours. This framework has been open sourced and is available publicly[2]. Rosita++ was published in ACM SIGAC Conference on Computer and Communications Security (CCS) 2021.

7.2 Limitations

Although Rosita and Rosita++ are effective in removing detected leakage in a given masked cipher implementation, they do have some limitations, which we discuss below.

Address based leakage. ELMO and by extension ELMO* do not regard the value of address operands of instructions as part of their models. Instead, they rely only on the value of the content that is pointed to by an address. For example, when modelling `str r0, [r2]` the content that is at the memory location given by value of `r2` is considered instead of the value of `r2`. We found out in evaluations of the AES and PRESENT ciphers in Sections 5.6 and 6.5 that leakage from addresses are prominent in substitution boxes implemented using look up tables. The leakage left unfixed in AES was due to state that was preserved in the address-bus. The state of the address-bus requires to be refreshed between the loading or storing from addresses that are dependent on values of shares. An automatic fix for such leakage would need to have higher level knowledge of the range of the buffer that is accessed by `ldr` and `str` instructions. Due to the current design of Rosita, this level of information is not available. The code segment in PRESENT that was responsible for the leakage cannot be fixed by using the countermeasures that we use due to it combining shares when the address is calculated. This is value-based leakage and is clearly not fixable by countermeasures designed to fix unintentional ILA breaches based leakage. Rosita and Rosita++ uses a library of manually selected code patterns that overwrite internal states to fix code leaky code segments. These never affect the functionality of the code they are supposed to fix, all they do is clear the internal states set by previous instructions so that they do not propagate and cause leakage. Value-based leakage on the other hand, can only be fixed through

[2] https://github.com/0xADE1A1DE/tracetools

implementing masking properly. A single intermediate value must not include more than one share.

Detecting combination of shares. The methods this work proposes to use for root cause detection only detect leakage if all shares combine. Therefore, the combinations of shares that do not expose a secret value are not detected. In other words, these are partial combinations where the number of shares that are combined are less than or equal to d in a dth-order masked implementation. Even-though these do not exhibit leakage, they do reduce the level of security of a masked implementation.

Micro-architectures with longer pipelines. As discussed in Chapter 4, the profiling stages of both ELMO and ELMO* depend on measurement traces that are gathered from a target physical device. The amount of such traces required grow exponentially when the number of stages in the CPU pipeline increases. The subset of emulated instructions (21) is approximated by five representative instructions. Therefore, the total number of combinations for the ARM Cortex-M0 is $5^3 = 125$. This is also not trivially changeable due to it being a fundamental building block of the Multiple Linear Regression model that both ELMO and ELMO* depend on.

Input dependency of TVLA. The input dependence on the results of Test Vector Leakage Assessment (TVLA) is a limiting factor with respect to detection of leakage. The benefit of using TVLA for evaluation is that it can detect leakage only from the measurement traces. However, when countermeasures are applied based a certain set of fixed inputs, it does not mean that there would not be any leakage detected for any other different input. Therefore, the leakage detection of Rosita and Rosita++ will benefit from a more generic leakage assessment that is not dependent on the exact fixed input values used.

Manual creation of replacement code patterns. The code patterns that are used in both Rosita and Rosita++ are required to be synthesised manually by an evaluator after going through the table of dominating instructions shown in Table 5.1. Additionally, these code patterns are not portable to other ISAs.

7.3 Future work

We have recognised the following future works which would improve the effectiveness Rosita and Rosita++ significantly.

Machine learning based power models for root cause detection. Machine learning based models can be used for emulation of leakage instead of the Multiple Linear Regression based power model that ELMO* is based on. ABBY [Bazangani et al., 2021] is a recent work that has proven the effectiveness of this approach. In a machine learning based model, the terms of the model would not be as straightforward as is in a Multiple Linear Regression model. Because of this the elimination of terms in Rosita++'s root cause detection cannot continue its operation. The same issue affects the Monte-Carlo method as discussed in Section 6.3. This challenge can be overcome by replacing inputs with zeros in the prediction phase. Doing this will have the same effect as removing terms from the base model.

Improving speed of code rewrites. The root cause detection system of both Rosita and Rosita++ labels instructions that participate in leakage only considering local leakage. This means that a certain set of $d + 1$ power value samples that are leaky are only considered for a leakage detection of dth-order masked implementation within a single run. Due to this, there can be instances where the applied fixes are redundant. This can be mitigated by globally optimising the fixes again after they have been applied. A global optimisation will consider all leakage that is found in a certain code segment and the effects of applied code fixes. This will reduce the redundant application of code fixes and will improve the resulting code performance.

7.4 Summary

In summary, this work suggests improvements to leakage emulation, a general methodology to find new internal state related leakage and algorithms to apply countermeasures to masked implementations automatically. The evaluations carried out in this work show that this framework can adequately fix leakage in any order of masking. The thesis also introduces a high-performance trace processing tool-set that can be easily configured to run commonly used tasks in power traces. Finally, a discussion of the limitations and future work related to this work concludes this thesis.

Bibliography

Giovanni Agosta, Alessandro Barenghi, and Gerardo Pelosi. A code morphing methodology to automate power analysis countermeasures. In *DAC*, pages 77–82. ACM, 2012.

Antoine Amarilli, Sascha Müller, David Naccache, Dan Page, Pablo Rauzy, and Michael Tunstall. Can code polymorphism limit information leakage? In *WISTP*, volume 6633 of *Lecture Notes in Computer Science*, pages 1–21. Springer, 2011.

Victor Arribas, Svetla Nikova, and Vincent Rijmen. VerMI: Verification tool for masked implementations. In *ICECS*, pages 381–384. IEEE, 2018.

Jean-Philippe Aumasson, L. Henzen, W. Meier, and R. Phan. Sha-3 proposal blake, 2009.

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

Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the cost of lazy engineering for masked software implementations. In *CARDIS*, pages 64–81, 2015.

Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. CSI NN: reverse engineering of neural network architectures through electromagnetic side channel. In *USENIX Security Symposium*, pages 515–532, 2019.

Ali Galip Bayrak, Francesco Regazzoni, Philip Brisk, François-Xavier Standaert, and Paolo Ienne. A first step towards automatic application of power analysis countermeasures. In *DAC*, pages 230–235. ACM, 2011.

Ali Galip Bayrak, Francesco Regazzoni, David Novo, and Paolo Ienne. Sleuth: Automated verification of software power analysis countermeasures. In *CHES*, volume 8086 of *Lecture Notes in Computer Science*, pages 293–310. Springer, 2013.

Ali Galip Bayrak, Francesco Regazzoni, David Novo, Philip Brisk, François-Xavier Standaert, and Paolo Ienne. Automatic application of power analysis countermeasures. *IEEE Trans. Computers*, 64(2):329–341, 2015.

Omid Bazangani, Alexandre Iooss, Ileana Buhan, and Lejla Batina. ABBY: automating the creation of fine-grained leakage models. *IACR Cryptol. ePrint Arch.*, page 1569, 2021.

Daniel Bernstein. ChaCha, a variant of Salsa20, 2008.

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

Daniel J. Bernstein, Stefan Kölbl, Stefan Lucks, Pedro Maat Costa Massolino, Florian Mendel, Kashif Nawaz, Tobias Schneider, Peter Schwabe, François-Xavier Standaert, Yosuke Todo, et al. Gimli: a cross-platform permutation. In *CHES*, pages 299–320, 2017.

Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. The eXtended Keccak code package (XKCP). URL https://github.com/XKCP/XKCP.

Guido Bertoni, Vittorio Zaccaria, Luca Breveglieri, Matteo Monchiero, and Gianluca Palermo. AES power attack based on induced cache miss and countermeasure. In *ITCC*, pages 586–591, 2005.

Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The Keccak reference, january 2011, 2011.

Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. Formal verification of masked hardware implementations in the presence of glitches. In *EUROCRYPT (2)*, volume 10821 of *Lecture Notes in Computer Science*, pages 321–353. Springer, 2018.

Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: an ultra-lightweight block cipher. In *CHES*, pages 450–466, 2007.

Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *CRYPTO*, pages 398–412, 1999.

Zhimin Chen and Yujie Zhou. Dual-rail random switching logic: A countermeasure to reduce side channel leakage. In *CHES*, pages 242–254, 2006.

Hamid Choukri and Michael Tunstall. Round reduction using faults. *FDTC*, 5:13–24, 2005.

Thomas De Cnudde, Begül Bilgin, Oscar Reparaz, Ventzislav Nikov, and Svetla Nikova. Higher-order threshold implementation of the AES S-Box. In *CARDIS*, pages 259–272, 2015.

Thomas De Cnudde, Oscar Reparaz, Begül Bilgin, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Masking AES with d+1 shares in hardware. In *TIS@CCS*, page 43, 2016.

Yann Le Corre, Johann Großschädl, and Daniel Dinu. Micro-architectural power simulator for leakage assessment of cryptographic software on ARM Cortex-M3 processors. In *COSADE*, volume 10815 of *Lecture Notes in Computer Science*, pages 82–98. Springer, 2018.

Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

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

Joan Daemen, Seth Hoffert, Gilles Van Assche, and Ronny Van Keer. Xoodoo cookbook. IACR Cryptology ePrint Archive report 2018/767, 2018b.

Debayan Das and Shreyas Sen. Electromagnetic and power side-channel analysis: Advanced attacks and low-overhead generic countermeasures through white-box approach. *Cryptogr.*, 4(4):30, 2020.

Nicolas Debande, Maël Berthier, Yves Bocktaels, and Thanh-Ha Le. Profiled model based power simulator for side channel evaluation. *IACR Cryptol. ePrint Arch.*, page 703, 2012.

Jerry den Hartog, Jan Verschuren, Erik P. de Vink, Jaap de Vos, and W. Wiersma. PINPAS: A tool for power analysis of smartcards. In *SEC*, volume 250 of *IFIP Conference Proceedings*, pages 453–457. Kluwer, 2003.

A. Adam Ding, Liwei Zhang, François Durvaux, François-Xavier Standaert, and Yunsi Fei. Towards sound and optimal leakage detection procedure. In *CARDIS*, pages 105–122, 2017.

Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1. 2. *Submission to the CAESAR Competition*, 2016.

M. J. Dworkin. SHA-3 standard: Permutation-based hash and extendable-output functions, NIST FIPS-202. august 2015.

Hassan Eldib and Chao Wang. Synthesis of masking countermeasures against side channel attacks. In *CAV*, volume 8559 of *Lecture Notes in Computer Science*, pages 114–130. Springer, 2014.

Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The Skein hash function family, 2010. URL https://www.schneier.com/academic/skein/.

Steve Furber. *ARM System-on-Chip Architecture*. Addison-Wesley, 2nd edition, 2000.

Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In *CHES*, pages 251–261, 2001.

Si Gao. A Thumb assembly based byte-wise masked AES implementation. https://github.com/bristol-sca/ASM_MaskedAES, 2019.

Si Gao and Elisabeth Oswald. A novel completeness test and its application to side channel attacks and simulators. *IACR Cryptol. ePrint Arch.*, page 756, 2021.

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

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

Si Gao, Johann Großschädl, Ben Marshall, Dan Page, Thinh Pham, and Francesco Regazzoni. An instruction set extension to support software-based masking. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(4):283–325, 2021a.

Si Gao, Elisabeth Oswald, and Dan Page. Reverse engineering the micro-architectural leakage features of a commercial processor. *IACR Cryptol. ePrint Arch.*, page 794, 2021b.

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

Daniel Genkin, Adi Shamir, and Eran Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In *CRYPTO (1)*, pages 444–461, 2014.

Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. ECDSA key extraction from mobile devices via nonintrusive physical side channels. In *CCS*, pages 1626–1638, 2016.

Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. Coco: Co-design and co-verification of masked software implementations on CPUs. In *USENIX Security Symposium*, pages 1469–1468. USENIX Association, 2021.

Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A testing methodology for side-channel resistance validation. *NIST non-invasive attack testing workshop*, 2011.

Louis Goubin. A sound method for switching between Boolean and arithmetic masking. In *CHES*, pages 3–15, 2001. ISBN 978-3-540-44709-2.

Wolfgang Karl Hardle, Sigbert Klinke, and Bernd Ronz. *Introduction to Statistics: Using Interactive MM*Stat Elements*. Springer Publishing Company, Incorporated, 2015. doi: 10.1007/978-3-319-17704-5.

Michael Hutter and Michael Tunstall. Constant-time higher-order Boolean-to-arithmetic masking. *Journal of Cryptographic Engineering*, 9, 06 2019. doi: 10.1007/s13389-018-0191-z.

Yuval Ishai, Amit Sahai, and David A. Wagner. Private circuits: Securing hardware against probing attacks. In *CRYPTO*, volume 2729, pages 463–481, 2003.

Bernhard Jungk, Richard Petri, and Marc Stöttinger. Efficient side-channel protections of ARX ciphers. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, pages 627–653, 2018.

S.K. Kachigan. *Statistical Analysis: An Interdisciplinary Introduction to Univariate & Multivariate Methods*. Radius Press, 1986. ISBN 9780942154993. URL https://books.google.com.au/books?id=zqZpAAAAMAAJ.

David Kahn. *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet*. Scribner, 1996. URL http://amazon.com/o/ASIN/0684831309/.

Dina Kamel, Mathieu Renauld, Denis Flandre, and François-Xavier Standaert. Understanding the limitations and improving the relevance of SPICE simulations in side-channel security evaluations. *J. Cryptogr. Eng.*, 4(3):187–195, 2014.

Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*, pages 104–113, 1996.

Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, pages 388–397, 1999.

Juliane Krämer, Dmitry Nedospasov, Alexander Schlösser, and Jean-Pierre Seifert. Differential photonic emission analysis. In *COSADE*, volume 7864 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2013.

Dirk P. Kroese, Tim J. Brereton, Thomas Taimre, and Zdravko I. Botev. Why the Monte Carlo method is so important today. *Wiley Interdisciplinary Reviews: Computational Statistics*, 6:386–392, 2014.

Xuejia Lai and James L. Massey. A proposal for a new block encryption standard. In *Eurocrypt*, pages 389–404, 1991.

Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. A survey of microarchitectural side-channel vulnerabilities, attacks and defenses in cryptography. *CoRR*, abs/2103.14244, 2021.

Stefan Mangard, Thomas Popp, and Berndt M. Gammel. Side-channel leakage of masked CMOS gates. In *CT-RSA*, pages 351–365, 2005.

Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007. ISBN 978-0-387-30587-9.

David McCann, Carolyn Whitnall, and Elisabeth Oswald. ELMO: emulating leaks for the ARM Cortex-M0 without access to a side channel lab. IACR Cryptol. ePrint Arch., 2016.

David McCann, Elisabeth Oswald, and Carolyn Whitnall. Towards practical tools for side channel aware software engineering: 'grey box' modelling for instruction leakages. In *USENIX Security Symposium*, pages 199–216, 2017.

Robert P. McEvoy, Colin C. Murphy, William P. Marnane, and Michael Tunstall. Isolated WDDL: A hiding countermeasure for differential power analysis on FPGAs. *ACM Trans. Reconfigurable Technol. Syst.*, 2(1):3:1–3:23, 2009.

Thomas S. Messerges. *Power Analysis Attacks and Countermeasures for Cryptographic Algorithms*. PhD thesis, USA, 2000. AAI9978665.

Thomas S. Messerges, Ezzy A. Dabbish, and Robert H. Sloan. Power analysis attacks of modular exponentiation in smartcards. In *CHES*, volume 1717 of *Lecture Notes in Computer Science*, pages 144–157. Springer, 1999.

Lauren De Meyer, Elke De Mulder, and Michael Tunstall. On the effect of the (micro)architecture on the development of side-channel resistant software. *IACR Cryptol. ePrint Arch.*, page 1297, 2020.

Amir Moradi and Oliver Mischke. Comprehensive evaluation of AES dual ciphers as a side-channel countermeasure. In *ICICS*, pages 245–258, 2013.

Amir Moradi, Oliver Mischke, and Christof Paar. One attack to rule them all: Collision timing attack versus 42 AES ASIC cores. *IEEE Trans. Computers*, 62 (9):1786–1798, 2013.

Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. Compiler assisted masking. In *CHES*, volume 7428 of *Lecture Notes in Computer Science*, pages 58–75. Springer, 2012.

Laurence W. Nagel and D.O. Pederson. SPICE (simulation program with integrated circuit emphasis). Technical Report UCB/ERL M382, EECS Department, University of California, Berkeley, 1973. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/1973/22871.html.

Adib Nahiyan, Jungmin Park, Miao Tony He, Yousef Iskander, Farimah Farahmandi, Domenic Forte, and Mark M. Tehranipoor. SCRIPT: A CAD framework for power side-channel vulnerability assessment using information flow tracking and pattern generation. *ACM Trans. Design Autom. Electr. Syst.*, 25(3):26:1–26:27, 2020.

National Institute of Standards and Technology. 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., 2015a.

National Institute of Standards and Technology. Security requirements for cryptographic modules. Technical Report NIST Special Publication 800-57 Part 3, Revision 1, U.S. Department of Commerce, Washington, D.C., 2015b.

Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In *ICICS*, pages 529–545, 2006.

Kostas Papagiannopoulos and Nikita Veshchikov. Mind the gap: Towards secure 1st-order masking in software. In *COSADE*, volume 10348 of *Lecture Notes in Computer Science*, pages 282–297. Springer, 2017.

Scott Pardo. *Equivalence and Noninferiority Tests for Quality, Manufacturing and Test Engineers*. CRC Press LLC, Philadelphia, PA, 1 edition, 2013.

Philippe Pierre Pebay. Formulas for robust, one-pass parallel computation of covariances and arbitrary-order statistical moments. 9 2008. doi: 10.2172/1028931. URL https://www.osti.gov/biblio/1028931.

Emmanuel Prouff, Matthieu Rivain, and Régis Bevan. Statistical analysis of second order differential power analysis. *IACR Cryptol. ePrint Arch.*, 2010:646, 2010.

Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): measures and counter-measures for smart cards. In *E-smart*, pages 200–210, 2001.

Francesco Regazzoni, Thomas Eisenbarth, Axel Poschmann, Johann Großschädl, Frank K. Gürkaynak, Marco Macchetti, Zeynep Toprak Deniz, Laura Pozzi, Christof Paar, Yusuf Leblebici, and Paolo Ienne. Evaluating resistance of MCML technology to power analysis attacks using a simulation-based methodology. *Trans. Comput. Sci.*, 4:230–243, 2009.

Marian Rejewski. How polish mathematicians broke the enigma cipher. *IEEE Ann. Hist. Comput.*, 3(3):213–234, 1981.

Mathieu Renauld, François-Xavier Standaert, Nicolas Veyrat-Charvillon, Dina Kamel, and Denis Flandre. A formal study of power variability issues and side-channel attacks for nanoscale devices. In *EUROCRYPT*, pages 109–128, 2011.

Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In *CHES*, volume 6225 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2010.

Ronald L. Rivest, M. J. B. Robshaw, R. Sidney, and Y. L. Yin. The rc6 block cipher. In *in First Advanced Encryption Standard (AES) Conference*, page 16, 1998.

Pascal Sasdrich, René Bock, and Amir Moradi. Threshold implementation in software - case study of PRESENT. In *COSADE*, volume 10815 of *Lecture Notes in Computer Science*, pages 227–244. Springer, 2018.

Werner Schindler, Kerstin Lemke, and Christof Paar. A stochastic model for differential side channel cryptanalysis. In *CHES*, volume 3659 of *Lecture Notes in Computer Science*, pages 30–46. Springer, 2005.

Tobias Schneider and Amir Moradi. Leakage assessment methodology - A clear roadmap for side-channel evaluations. In *CHES*, volume 9293 of *Lecture Notes in Computer Science*, pages 495–513. Springer, 2015.

Donald J Schuirmann. 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):657–680, 1987.

Peter Schwabe and Ko Stoffelen. All the AES you need on cortex-m3 and M4. In *SAC*, volume 10532 of *Lecture Notes in Computer Science*, pages 180–194. Springer, 2016.

Hermann Seuschek and Stefan Rass. Side-channel leakage models for RISC instruction set architectures from empirical data. *Microprocess. Microsystems*, 47:74–81, 2016.

Hermann Seuschek, Fabrizio De Santis, and Oscar M. Guillen. Side-channel leakage aware instruction scheduling. In *CS2@HiPEAC*, pages 7–12. ACM, 2017.

Aria Shahverdi, Mahammad Shirinov, and Dana Dachman-Soled. Database reconstruction from noisy volumes: A cache side-channel attack on SQLite. *CoRR*, abs/2006.15007, 2020.

C. E. Shannon. Communication theory of secrecy systems. *The Bell System Technical Journal*, 28(4):656–715, 1949. doi: 10.1002/j.1538-7305.1949.tb00928.x.

Madura A. Shelton, Lukasz Chmielewski, Niels Samwel, Markus Wagner, Lejla Batina, and Yuval Yarom. Rosita++: Automatic higher-order leakage elimination from cryptographic code. In *CCS*, pages 685–699. ACM, 2021a.

Madura A. Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. Rosita: Towards automatic elimination of power-analysis leakage in ciphers. In *NDSS*. The Internet Society, 2021b.

Peter Williston Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994. doi: 10.1109/SFCS.1994.365700.

Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *USENIX Security Symposium*, pages 639–656, 2019.

Danilo Sijacic, Josep Balasch, Bohan Yang, Santosh Ghosh, and Ingrid Verbauwhede. Towards efficient and automated side-channel evaluations at design time. *J. Cryptogr. Eng.*, 10(4):305–319, 2020.

Wilson Snyder. Verilator. URL https://www.veripool.org/verilator/.

J Thiyagalingam, O Beckmann, and PHJ Kelly. An exhaustive evaluation of row-major, column-major and morton layouts for large two-dimensional arrays. pages 340–350. Warwick University, 2003.

Stefan Tillich and Johann Großschädl. Power analysis resistant AES implementation with instruction set extensions. In *CHES*, volume 4727 of *Lecture Notes in Computer Science*, pages 303–319. Springer, 2007.

K. Tiri. Side-channel resistant circuit styles and associated IC design flow. In *Secure Integrated Circuits and Systems*, Integrated Circuits and Systems, pages 145–157. Springer, 2010.

K. Tiri and I. Verbauwhede. Simulation models for side-channel information leaks. In *DAC*, pages 228–233. ACM, 2005.

K. Tiri, M. Akmal, and I. Verbauwhede. A dynamic and differential CMOS logic with signal independent power consumption to withstand differential power analysis on smart cards. In *Proceedings of the 28th European Solid-State Circuits Conference*, pages 403–406, 2002.

Alan M. Turing and Jack Copeland. The essential turing: Seminal writings in computing, logic, philosophy, artificial intelligence, and artificial life plus the secrets of enigma. 2004.

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

Nikita Veshchikov and Sylvain Guilley. Use of simulators for side-channel analysis. In *EuroS&P*, pages 51–59. IEEE, 2017.

Dan Walters, Andrew Hagen, and Eric Kedaigle. Sleak: A side-channel leakage evaluator and analysis kit. Technical report, MITRE Corporation, United States, 2014.

Jingbo Wang, Chungha Sung, and Chao Wang. Mitigating power side channels during compilation. In *ESEC/SIGSOFT FSE*, pages 590–601. ACM, 2019.

Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures. In *USENIX Security Symposium*, pages 2003–2020, 2020.

Yuan Yao, Mo Yang, Conor Patrick, Bilgiday Yuce, and Patrick Schaumont. Fault-assisted side-channel analysis of masked implementations. In *HOST*, pages 57–64. IEEE Computer Society, 2018.

Yuan Yao, Tarun Kathuria, Baris Ege, and Patrick Schaumont. Architecture correlation analysis (ACA): identifying the source of side-channel leakage at gate-level. In *HOST*, pages 188–196. IEEE, 2020.

Han-Sen Zhong, Hui Wang, Yu-Hao Deng, Ming-Cheng Chen, Li-Chao Peng, Yi-Han Luo, Jian Qin, Dian Wu, Xing Ding, Yi Hu, et al. Quantum computational advantage using photons. *Science*, 370(6523):1460–1463, 2020.