Rosita: Towards Automatic Elimination of Power-Analysis Leakage in Ciphers

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

Niels Samwel
Radboud University
nsamwel@cs.ru.nl

Lejla Batina
Radboud University
lejla@cs.ru.nl

Francesco Regazzoni
University of Amsterdam and ALaRI – USI
f.regazzoni@uva.nl, regazzoni@alari.ch

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

Yuval Yarom
University of Adelaide and Data61
yval@cs.adelaide.edu.au

Abstract—**Since their introduction over two decades ago, side-channel attacks have presented a serious security threat. While many ciphers' implementations employ masking techniques to protect against such attacks, they often leak secret information due to unintended interactions in the hardware. We present Rosita, a code rewrite engine that uses a leakage emulator which we amend to correctly emulate the micro-architecture of a target system. We use Rosita to automatically protect masked implementations of AES, ChaCha, and Xoodoo. For AES and Xoodoo, we show the absence of observable leakage at 1 000 000 traces with less than 21% penalty to the performance. For ChaCha, which has significantly more leakage, Rosita eliminates over 99% of the leakage, at a performance cost of 64%.**

I. Introduction

The seminal work of Kocher [55] demonstrated that interactions of cryptographic implementations with their environment can result in *side channels*, which leak information on the internal state of ciphers. Since then multiple side channels have been demonstrated, exploiting various effects, such as timing [1, 11, 20, 21], power consumption [56], electromagnetic (EM) emanations [22, 39, 69], shared micro-architectural components [43, 78], and even acoustic and photonic emanations [4, 44, 58, 73]. These side channels pose a severe risk to the security of systems, and in particular to cryptographic implementations, and effective side-channel attacks have been demonstrated against block and stream ciphers [47, 70], public-key systems, both traditional [30, 65] and post quantum [68], cryptographic primitives implemented in real-world devices [5, 35], and even non-cryptographic algorithms [8].

Many approaches to protect devices have been suggested, in particular against power and EM attacks. These range from special logic styles that are designed to make leakage data-independent [24, 37, 77], through noise generation to hide the signal [64], to algorithmic changes designed to prevent certain leakage [63]. In particular, *masking* is a common algorithmic countermeasure in which all intermediate (secret-dependent) values in the ciphers are combined with random masks, so that the leakage of one or even a few values does not provide the attacker with enough information to recover the secrets.

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

	Cycles		**Leakage Points**	
Function	**Original**	**Fixed**	**Original**	**Remaining**
AES	1285	1479	31	0
ChaCha	1322	2162	238	1
Xoodoo	637	769	38	0

The protection afforded by masking is, however, only theoretical. In practice, masked implementations often fail to achieve the promised level of security. One of the most common reasons for leakage from masked implementations is unintended interactions between values in the micro-architecture. For example, the power consumption of updating the contents of a register may depend on a relationship between the values prior to and after the update.

To achieve secure cryptography in the presence of side-channel attacks, cryptographic implementations often go through multiple cycles of leakage evaluation, e.g. as specified in International Standard ISO/IEC 17825:2016(E) [49]. Such a process is costly because it requires a high level of expertise and significant manual labor, especially taking into account state-of-the-art side-channel adversaries.

Recently, several works have experimented with tools that provide a high resolution emulation of the power consumption [81]. The results of such emulations are combined with standard statistical tests [10] to perform leakage assessment of software without executing it on the actual hardware [67]. Observing that these tools eliminate the hardware from the leakage evaluation process, we ask the following question:

Can leakage emulators be used for automatic mitigation of side channel leakage from software implementations?

Network and Distributed Systems Security (NDSS) Symposium 2021
21-24 February 2021, Virtual Conference
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.23137
www.ndss-symposium.org

In this work, we answer this question in the affirmative (see Table I for results), albeit with some caveats. Specifically, we develop an automatic tool, ROSITA[1], that uses an emulator to detect leakage due to unintended interactions between values and then rewrites the code to eliminate the leakage. Automating leakage elimination reduces the amount of work required to ensure adherence with the ISO 17825 standard, and to develop secure cryptographic implementations.

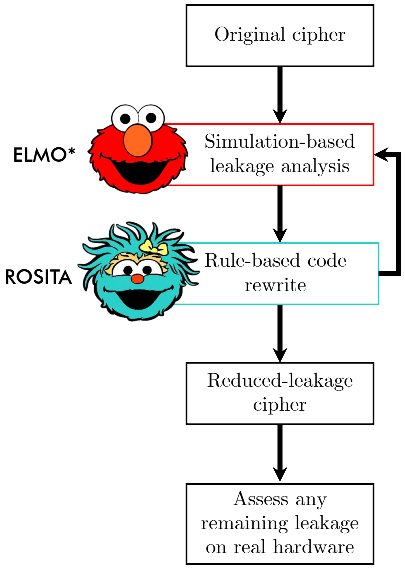

Original cipher
ELMO*
Simulation-based leakage analysis
ROSITA
Rule-based code rewrite
Reduced-leakage cipher
Assess any remaining leakage on real hardware

Figure 1: Leakage elimination workflow with extended ELMO (ELMO*) and ROSITA.

To emulate leakage, we develop ELMO*, a leakage emulator that uses the execution engine of the ELMO leakage emulator [62], but improves ELMO in three different directions. We first note that while ELMO detects leakage between consecutive instructions, it fails to detect leakage between instructions that are further apart. To improve the emulation accuracy, we first design and develop novel procedures for detecting leakage between non-consecutive instructions and for identifying the storage components involved in this leakage. We apply these procedures to the Cortex M0 processor, identifying several storage elements. We then modify ELMO to model these elements, achieving an accurate leakage model. Second, we modify ELMO to not only output the instruction that causes the leakage, but also to identify the cause of the leakage. Last, we modify the workflow in ELMO to perform on-line statistical analysis of the generated traces to detect leakage.

In its core, ROSITA is a rule-driven code rewrite engine. It uses the output from ELMO* to select rewrite rules and apply them at leaky points. To eliminate leakage, we follow the workflow in Figure 1. We repeatedly execute ELMO* to identify code locations that leak and then invoke ROSITA to rewrite the code. Finally, we test the code produced by ROSITA on the physical device to assess the level of remaining leakage.

We experiment with masked implementations of three ciphers with very different round functions: the AES block cipher [27], the ChaCha stream cipher [12], and the Xoodoo permutation [28], running up to 1 000 000 traces on each. We use the table-lookup based masked implementation of AES-128 by Yao et al. [87]. AES implemented with table lookups tends to be vulnerable to cache-based timing attacks. Recent ciphers however, mostly permutations, can be implemented efficiently with only bitwise Boolean instructions and (cyclic) shifts, e.g., Keccak-p, the permutation underlying SHA-3, [17, 34], Ascon [33], Gimli [14] and Xoodoo [29]. They all have a nonlinear layer of algebraic degree 2 and hence allow very efficient masking. Among those, we chose Xoodoo[12] because it is the simplest of all and it lends itself to efficient implementations for 32-bit architectures. Finally, we consider ChaCha20 as a symmetric-key algorithm that is very different than the other two as a representative of addition-rotation-xor (ARX) ciphers.

As Table I shows, ROSITA successfully eliminates leakage from the AES and Xoodoo implementations, at a moderate performance impact of less than 21%. For ChaCha, ROSITA is also successful, eliminating all but one leakage points, with a higher performance cost of 64%.

To sum up, the contributions of this work are:

- We propose a framework for generating first-order leakage-resilient implementations of masked ciphers by iteratively rewriting the code at leakage points. (Section III)
- We design and implement systematic approaches for identifying leakage through microarchitectural storage elements. We use these approaches to detect multiple sources of leakage in the Cortex M0 processor that ELMO fails to model. We extend ELMO and augment it to model these sources of leakage, achieving an accurate model of leakage. We further augment ELMO to report instructions that leak secret information and the specific cause of leakage for each. (Section IV)
- We develop ROSITA, a code rewrite engine that uses the output of ELMO*, our augmented ELMO, to rewrite leaking instructions and eliminate leakage. (Section V)
- We use ROSITA to rewrite masked implementations of AES, ChaCha, and Xoodoo. We test the code ROSITA produces and show that ROSITA eliminates almost all the leakage at up to 1 000 000 traces, with an acceptable performance loss. (Section VI)

II. BACKGROUND

A. Side-Channel Attacks

When software runs on hardware, it affects the environment it executes in. This effect can be manifested as variations in power consumption, electromagnetic emanations, temperature, and state of various CPU components. As these variations correlate with the operation of the algorithm, monitoring these variations discloses information about the internal state, and as such provides a ‘communication’ channel that transfers information from the software being observed to the observer. These unintended communication channels are often known as *side channels*.

In 1996, Kocher [55] noted that the information acquired through a timing side channel may reveal secret information processed by cryptographic algorithms. Since then a significant

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

effort has been dedicated to analyzing and eliminating side-channel leakage, particularly in the context of cryptographic implementations [9, 39, 56, 61, 69].

Protection against side-channel attacks depends on both the channel and the attacker capabilities. The *constant-time* programming style [13, 52], which avoids secret-dependent branches and table lookups, has proven effective against attacks that depend on the cryptographic operation timing or on its effect on micro-architectural components [11, 20, 43]. However, when the leakage correlates with the data values being processed, constant-time programming is not sufficient to protect the implementation.

One of the main approaches to protect cryptographic implementations against side channels that leak information on data values is *masking* [23, 63]. In a nutshell, t-order masking represents each internal value v using $t+1$ values $v_0, v_1, \ldots, v_t$ such that the *masks* $v_1, \ldots, v_t$ are chosen uniformly at random, and v_0 is set such that $v = v_0 \oplus v_1 \oplus \cdots \oplus v_t$. Consequently the leakage of up to t values does not disclose any information to the attacker, and the implementation is secure in the t-probing model [50]. In practice, due to the complexities involved in higher-order masking, most masked implementation are first order, where each value is represented by a mask and a masked value.

Although theoretically secure, naive masking often fails to provide the required protection. The main cause is that side-channel leakage correlates not only with the values being processed, but also with the changes in the logical values of internal components, resulting in unintended interactions between values in the processor. Past research has identified two main sources of such leakage: transitional effects and glitches.

Transitional effects are caused when the logical value of a register or even of a long wire, such as a bus, changes from one to zero or vice versa. Changing the value draws more power than maintaining the value unchanged. Consequently, when changing the value of a register, the power consumption corresponds with the Hamming distance between the old and new values [6].

In contrast with transitional effects, which correspond to changes in logical values, *glitches* are temporary changes in electrical signals caused by signal timing differences. Because signals take time to propagate through the circuit, it takes time for the logical values to stabilize during a cycle. Until the signals stabilize, they may fluctuate between signal levels, leaking information that does not correspond just to the logical function computed [25, 60].

For masked implementations, unintended interactions, such as transitional effects and glitches can be disastrous. For example, suppose that a program that processes a secret value $v = v_0 \oplus v_1$ contains two consecutive instructions, where the first instruction uses v_0 and the second uses v_1. Internally, executing the first instruction would place v_0 on the bus, and executing the second instruction would change the contents of the bus to v_1. The transitional effect of changing the contents of the bus draws power which corresponds to the Hamming distance between v_0 and v_1, which is the Hamming weight of the secret value v.

Balasch et al. [6] demonstrate that unintended interactions due to transitions halve the number of intermediate values the adversary needs to acquire. However, as mentioned above, algorithms that use high-order masking are significantly more complex in terms of resources they require, than simple first-order masking. Moreover, Gao et al. [42] demonstrate that glitches may further reduce the security level of masked implementations.

Thus, the common practice, from the practitioners' point of view, is to use first-order masking, and to combine it with ad-hoc countermeasures for unintended interactions. Then, the implementation of the cryptographic software typically undergoes leakage assessment to find code locations that leak information. If leakage is detected, the operator applies manual modifications to the code to eliminate the leakage, this process repeats until no further leakage is evident.

We now turn our attention to assessing leakage in cryptographic implementations.

B. *Side Channel Leakage Assessment*

Leakage assessment of a device is very important for both the semiconductor and the security evaluation industries, and has accordingly received a lot of attention in past years. Depending on the attacker model, many attack vectors are possible and exhaustive evaluation (by trying all possible attacks) is simply not feasible. As an alternative, a leakage evaluation methodology called Test Vector Leakage Assessment (TVLA) was proposed [26, 79]. The question that it answers relates to the presence of any sort of leakage (from side channel measurements) of the targeted implementation running on the device of interest. TVLA does have some limitations. Specifically, a negative answer does not mean that the device is secure. However, the confidence level can be increased by testing multiple times with different inputs. Similarly, a positive result, i.e. indication of leakage, does not tell much on the exploitability of the leakage [74]. Nevertheless, due to its simplicity and efficacy, the TVLA method is considered useful first diagnostics tool for side-channel leakage assessment, and it has become the popular tool for security analysts. The core idea of TVLA is to use Welch's t-test [84] to differentiate between two sets of measurements, one with fixed inputs and the other with random inputs. If the test finds sufficiently strong evidence that the measurements leak, this implies that the device leaks some data-dependent information through a side channel. The main limitation is in evaluating each point in time independently, so the leakage from combining multiple points is not detected. To overcome this limitation, Schneider and Moradi [74] extend the t-test to handle multiple points. In addition, to address leakages distributed over multiple orders they propose the use of the χ^2-test as a natural complement to the Welch's t-test.

As a further guideline for analysts, the International Standard ISO/IEC 17825:2016(E) [49] suggests a specific procedure for assessing the security of devices. Specifically, the procedure requires selecting two fixed inputs and performing TVLA measurements, comparing the traces with these fixed inputs and those of random inputs. The number of traces in each assessment depends on the desired security level and ranges between 10 000 for level 3 and 100 000 for level 4.

C. Leakage Emulators

Because conducting real experiments for leakage detection is costly, leakage emulation has been adapted as an alternative. To the best of our knowledge, PINPAS [32], which detects leakage in Java-based smart cards, is the first such emulator. Since then, various other methods of emulating leakage have been suggested. Among the most accurate use SPICE [66] to simulate the internal circuits of a CPU down to a transistor level [3]. Its drawback is that transistor-level simulators tend to be very slow. Alternatively, researchers have looked at emulating at the source code level [80] and at machine instruction level [67, 81]. In source code level emulation, the emulator does not have any information about a specific CPU that will be used to run the compiled machine code of a given source code. It emulates leakage having source code as its only input. In instruction level emulation, the emulation is based on the machine code that will be executed on a certain CPU or more generally a specific CPU kind. Recently, advanced instruction level emulators have been introduced that use power and electromagnetic traces from real experiments to make better estimates [62, 75]. Similarly, advanced characteristics of CPUs such as instruction pipelining have found their way into recent leakage emulators [59].

COCO [46] suggests reformulating software testing for leakage as a hardware verification problem. Specifically, COCO uses a cycle-accurate simulator of masked software execution to acquire traces of execution on a target CPU. It then uses REBBECA [18] to verify the absence of leakage from the underlying hardware. An advantage of the approach is that leakage can be eliminated at both the software and the hardware levels. However, it requires access to the full netlist of the target processor and relies on manual tagging of masked values.

D. Automatic Approaches to Handling Side-Channel Leakage

Due to the numerous problems and pitfalls with countermeasures against side-channel attacks as previously discussed, researchers developed several automated approaches for handling side-channel leakage. The approaches can be grouped into three categories, simulation-based, code analysis, and hardware-assisted.

1) Simulation-based Approaches: Veshchikov [80] presents the SILK simulator, which simulates a high level abstraction of the source code of an algorithm that generates traces. Another simulator, MAPS [59] targets the Cortex-M3 and bases its leakage properties on the Hardware Description Language (HDL) source code. The simulator mainly focuses on leakage caused in the pipeline.

These two simulators only automate the generation of traces. Hence, they are basically assist the leakage evaluation process and speed it up.

2) Code Analysis: Barthe et al. [7] describe how to automatically verify higher-order masking schemes and present a new method based on program verification techniques. The work of Wang and Schaumont [82] explains how formal verification and program synthesis can be used to detect side-channel leakage, prove the absence of such leakage and modify software to prevent such leaks. However, both of these works remain limited in the ways they model the hardware and actual implementations.

Closer to ours are works that, although sacrificing generality, address the problem of “fixing” the leakage from a specific device. Papagiannopoulos and Veshchikov [67] perform an in-depth investigation of device specific effects that violate the *independent leakage assumption* (ILA) [71]. They also provide an automated tool that can detect such violations in AVR assembly code.

Another method to eliminate timing side channels in software was proposed by Wu et al. [86]. Their method requires a list with secret variables as input and produces code that is functionally equivalent to the original code but without timing side channels. In a recently published work Wang et al. [83] describe a type-based method for detecting leaks in source code. They implemented their mitigations in a compiler and evaluated their method. Eldib and Wang [36] propose a method to add countermeasures to source code that masks all intermediate computation results such that all intermediate results are statistically independent.

Agosta et al. [2] introduce a framework to automate the application of countermeasures against Differential Power Analysis (DPA). Their approach adds multiple versions of the code preventing an attacker from recognizing the exact point of leakage.

3) Hardware-Assisted Masking: Implementing masking within the processor promises a way of avoiding unintended interaction between masked values. Masking apply to the processor as a whole [31, 48], or only to a part of the instruction set [41, 54].

III. ROSITA OVERVIEW

ROSITA aims to automate the process of producing leakage-resilient software. Specifically, we focus on reducing the manual effort required for ensuring conformance with the ISO 17825 standard. We assume that the underlying algorithm employs a protection technique, such as masking. However, unintended interactions between data, introduced in the execution of the software, can break the independent leakage assumption [71] and leak secret information through a physical side channel, such as the power channel.

We consider two sources of interactions. In architectural interaction, the program overwrites a register with a related value, leaking information through transitional effects. Microarchitectural interactions occur due to transitional effects and glitches within the microarchitecture. For example, when a value of a pipeline stage register is overwritten. We do not handle cases where the application of masking is incorrect, either due to a programmer error or due to compiler optimizations. Similarly, we do not protect against attacks that expose the full state of the cipher [57].

To fix unintended interactions, implementers typically go through a manual, iterative process whereby the software is installed on the target device, the leakage is measured, and fixes are applied to the machine code, until the leakage is reduced to an acceptable level for the target use case.

This process, naturally, requires a significant level of expertise both in setting up and conducting the experiment to assess the leakage and in fixing the software to reduce the leakage. Moreover, because the assessment requires a large number of encryption rounds on relatively low-performing devices, and a number of repetitions in repairing the leakage and evaluating, the process is time consuming.

ROSITA automates this process as shown in Figure 1. To produce leakage-resilient cryptographic software, we start with a (masked) implementation of the cryptographic primitive. We use cross-compilation to produce both the assembly code (if the original code is in a high-level language) and the binary executable for the target device. The binary executable is then passed to a leakage emulator, in our case ELMO*, a modified version of ELMO [62], to perform leakage assessment. This assessment identifies the leakage and the machine instructions that cause it. ROSITA processes the output of ELMO*, together with the assembly code. It applies a set of rules that replace leaky assembly instructions with functionally-equivalent sequences of instructions that do not leak. Afterwards, the produced assembly program is assembled and fed back to ELMO* and the process repeats until no further fixes can be applied, at which time ROSITA produces a report indicating the remaining leakage, if any. In all of our experiments, ROSITA terminates within a small number of rounds when it detects no further leakage. The rules that ROSITA applies are incremental. Hence, ROSITA is guaranteed to converge within a finite number of rounds, when all rules are applied in all program positions.

Note that our approach makes use of a leakage emulator. Prior static-analysis-based solutions, such as [59, 67, 80, 82, 86], rely on tags that identify the nature of values within the program. For example, in ASCOLD [67] the programmer needs to assign tags to values, e.g. identifying them as random or masked. The main downside of the tagging approach is that any mistake the programmer makes in tagging values can be translated to missed leakage. In contrast, ROSITA applies TVLA, using a procedure that extends ISO 17825, to the emulated power trace. As such, ROSITA depends neither on the programmer's proficiency nor on specific properties of the masking scheme to detect leakage. Subject to the accuracy of the emulator and the strength of the statistical tools applied, ROSITA will detect leakage in the implementation (up to the level which the masking scheme used is meant to protect).

IV. LEAKAGE EMULATION

Due to ROSITA's reliance on a leakage emulator, care should be taken when selecting one. For this work we select ELMO [62] as a basis because, unlike instruction-level emulators, it is tailored to a specific processor model, while at the same time it does not require detailed design information to build its model. We now describe how ELMO models the device it emulates and the leakage. We then identify limitations for using ELMO with ROSITA and describe how we address these and develop ELMO*.

A. *The* ELMO *Leakage Model*

Emulating the hardware at the transistor level would produce the most accurate leakage estimate. However, this is often infeasible, both due to the complexity of such analysis and because the hardware implementation details are not available to the security evaluators and software developers.

Instead, leakage emulators use an abstract model of the device and of its power consumption. The abstract model is significantly simpler than emulating at the transistor level. At the same time, using an abstract model reduces accuracy and may result in missing some leakage. Thus, the *leakage model* presents a trade-off between modeling cost and accuracy.

ELMO's model of the hardware considers bit values and changes in bit values over the Arithmetic Logic Unit (ALU) inputs and outputs and memory instructions. Specifically, each operand is compared to the corresponding operand of the preceding instruction. Power consumption is modeled as linear combinations of bit values or bit changes.

ELMO models 21 instructions that its authors claim cover typical use in cryptography. These 21 instructions are divided into five groups, each modeled separately. To generate the model, power traces are collected while the processor executes sequences of three instructions. Each trace is processed to select a point-of-interest to be used as a representative of the trace. ELMO then performs a linear regression on the data collected in the traces to find the coefficients for the model.

The model itself consists of 24 main components, each modeling a specific part of the architecture. These cover:

- A linear combination of the bit flips between each operand of the current instruction and the corresponding operand of the previous and the subsequent instructions.
- A linear combination of the bit values of the operands of the current instruction.
- The instruction groups of the previous and subsequent instructions.

ELMO provides a pre-computed model of the STM32F030[2] evaluation board which features an ARM Cortex-M0 based STM32F030R8T6 System-on-Chip (SoC).[3]

B. *Evaluation Setup*

To evaluate ELMO, we compare its output with leakage assessment of the code on the real hardware. Our evaluation setup is shown in Figure 2.

We evaluate ELMO with the same STM32F030 Discovery evaluation board used in McCann et al. [62]. Following the instructions of McCann et al., we disconnect one of the two power inputs of the System on Chip (SoC) and attach a 330 Ω shunt resistor to the second power input. To avoid switching noise, we use a battery to power the evaluation board.

We use a PicoScope 6404D 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. See circuit diagram in Figure 3.

We sample 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 2 giga samples before running out of memory.

[2] https://www.st.com/en/evaluation-tools/32f0308discovery.html

[3] ELMO also provides a model for the Cortex-M4-based STM32F4 Discovery board, which we do not use in this work.

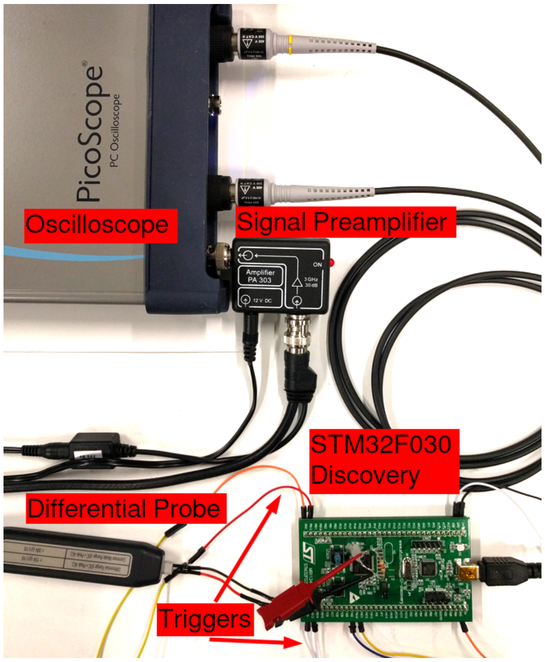


Figure 2: Evaluation Setup.

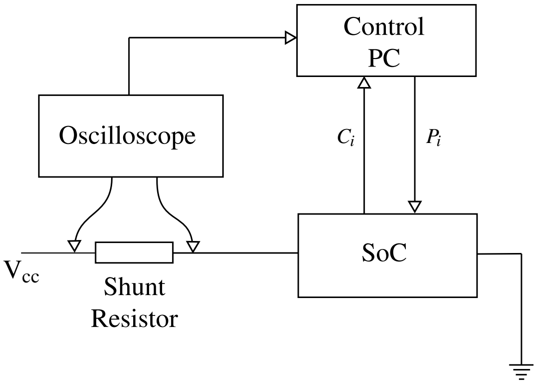


Figure 3: Evaluation Setup — Circuit Diagram

We use a control PC to orchestrate the experiments. The PC controls the oscilloscope and the STM32F030 Discovery evaluation board. It sends the software to be tested and the data to be used to the evaluation board, and collects the trace data from the oscilloscope. The control PC also generates all of the randomness required for the experiments. As a source we use `/dev/urandom`, which is considered cryptographically secure. The control PC generates the random inputs for the fixed vs. random tests. It further sends a stream of random values to be used for masks by the evaluation board.

Each experiment collects multiple power traces from running the software on the evaluation board. The execution of the software alternates between the fixed and the random cases. Thus, half of the collected traces are for the fixed case and the other half is for the random tests. Fixed and random tests are run randomly interleaved to make sure that the internal state of the device is non-deterministic at the start of each test [74]. To identify the start and end of the segment that we monitor, we use the output pins of the device to trigger the trace collection and to mark the end of the points of interest. Later, we use these trigger points to filter out traces with clock jitter.

To detect leakage, we employ non-specific TVLA. That is, we check the distribution of the values at each trace point and use the Welch t-test to check if the samples in the fixed and in the random traces are drawn from the same distribution. Following the common practice in the domain, we use a t-test value above 4.5 or below -4.5 as an indication of leakage. We validate the setup using the example from McCann et al. [62], getting results similar to theirs. See Appendix A.

C. *Storage Elements and the* ELMO *Model*

The ELMO model of the hardware only looks at interactions between arguments and outputs of successive instructions. However, it overlooks interactions that span multiple cycles. These interactions happen between instruction arguments and values that are stored in storage elements such as registers, memory, or latches.

To evaluate interactions overlooked by ELMO, we design a systematic battery of small sequences of code that aim at highlighting interactions via storage elements between instructions. An example of such code is shown in Listing 1. The code aims to check if there is an interaction between the value stored in Line 1 and the value used as the second argument of the `eors` instruction in Line 11. The purpose of the `movs` instructions between the two tested instructions is to eliminate leakage between pipeline stages. The sequence of nine `movs` instructions ensures that the `str` instruction is completed by the time the `eors` instruction enters the pipeline.

```
 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  movs  r7, r7
 8  movs  r7, r7
 9  movs  r7, r7
10  movs  r7, r7
11  eors  r3, r4
```

Listing 1: Evaluating interactions between the `str` and the `eors` instructions.

For the test, we collect 10 000 power traces of running the code segment, each run using different random values for the data the code processes. (For example, in Listing 1 we randomize `r1`, `r3`, `r4`, `r7`, and the contents of the memory address pointed to by `r2`.) For each run we also record the Hamming distance between the two values we investigate. (In this example, the values of `r1` and `r4`.) We then calculate the Pearson correlation coefficient between the Hamming distance and the values in each point of the trace. A high correlation coefficient indicates that the Hamming distance between the values leaks through the power trace, implying that the first instruction keeps the value it processes in some storage element that interacts with the data processed by the last instruction.

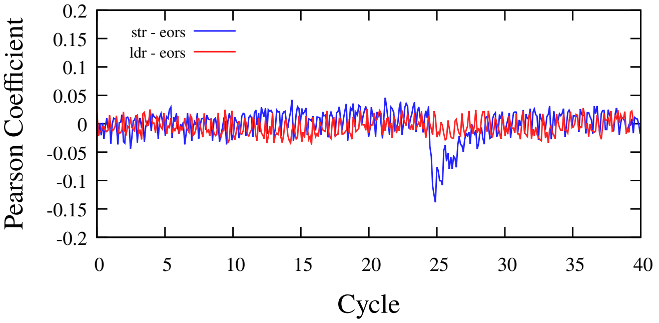


Figure 4: Pearson correlation coefficient of interference test.

Figure 4 shows the Pearson correlation coefficients for two code sequences. One from Listing 1, testing leakage from `str` to `eors`, and the other testing leakage from `ldr` to `eors`. As we can see, the code in Listing 1 show a pronounced dip in the correlation coefficient around cycle 25, indicating interaction between the values. Conversely, the correlation coefficient when replacing the `str` with `ldr` remains close to zero, indicating no leakage.

D. Dominating Instructions

The methodology we discuss in Section IV-C allows us to find instruction pairs that interact via hidden storage within the processor. However, each such instruction may affect multiple storage elements. To correctly model leakage through these elements, we need to know which instructions affect which storage elements. Because the design details of the processor are not public, we cannot positively identify the storage elements used by an instruction. Instead, we search for *dominating* instructions in pairs, i.e. instructions that set more storage elements than others. For that, we use code sequences similar to Listing 2, which checks if `str` dominates `eors`. Specifically, we pick a pair of instructions with interacting storage. In the test code we use two instances of the first (Lines 1 and 9), with the second instruction separating these two instances (Line 5). If the first instruction dominates the second, leakage will be visible at the second instance of the first instruction (Line 9).

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

Listing 2: Checking for a dominating instruction

Theoretically, it is possible to have a pair of instructions that each only affects part of the state set by the other. However, we did not find any such pair.

E. Findings

We run a broad range of experiments, with (1) some focusing on architecturally known storage elements, such as registers and memory, and (2) others aiming to find micro-architectural storage elements by testing interactions between pairs of instructions. We find several sources of leakage that ELMO does not identify. We note that Gao [40] also identifies many of the issues we find; however, their identification was driven by the iterative tweaking of a cipher, whereas our systematic approach is cipher-agnostic. Of the leakage we find, the first is an architectural issue, whereas the others are microarchitectural. Except where mentioned otherwise, we believe that the cause of all microarchitectural leakage is transition effects, because the leakage corresponds to a change in the logical state. However, without access to the processor implementation details, we cannot rule out the possibility that the cause is glitches. When the leakage does not correspond to a change in the logical state of the processor we assume that the leakage is due to glitches.

1) Registers: We find that overwriting a register leaks the (weighted) Hamming distance between the previous value and the new value. This is a significant leakage source, because reusing a register that contains a masked value for another value with the same mask leaks secret information. Unlike Papagiannopoulos and Veshchikov [67], we do not find leakage across different registers.

2) Memory: Writing data to memory interacts with data already stored in the same location. Hence, overwriting one masked value with another may remove the mask, leaking the values.

3) Instruction Pairs: We analyzed all pairs of instructions for leakage from both arguments. The results for the second argument are summarized in Table II. We see that all instructions set some state, and that most pairs do interact with this state. We now discuss some of our observations about the storage elements used.

Table II: State interactions between the second operands of instruction pairs. Triangles point to the dominating instruction. Circles indicate interactions on the same storage.

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

4) Memory Bus: The memory bus seems to have a storage element that stores the most recent value stored to or loaded from the memory. When loading from or storing to memory, the value of the storage element is overwritten, leaking the Hamming distance between the previous and the new value. This leakage differs from the two described above, and happens irrespective of the registers and the memory addresses used. Consequently, when writing to or reading from memory, care should be taken to only access non-secret values or values masked with different masks. We 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 3. In this example we assume that memory locations 0x300 and 0x400 both contain one secret byte each, both masked with the same mask. The code in this example performs two memory operations, the first stores a byte into 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 data stored in these locations.

```
movs r3, 0x303
movs r4, 0x402
movs r7, r7
movs r7, r7
movs r7, r7
strb r5, [r3]
movs r7, r7
movs r7, r7
movs r7, r7
ldrb r6, [r4]
movs r7, r7
movs r7, r7
movs r7, r7
```

Listing 3: Example of word interaction

A further issue in the memory bus is an 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.

5) Store Latch: We find that storing 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 r5, r2
6 movs r7, r7
7 movs r7, r7
8 movs r7, r7
9 eors r1, r4
```

Listing 4: Store latch example.

For example, in the example in Listing 4, the code stores the value of `r5` to memory (Line 1). It then updates the value of `r5`, moving the contents of `r2` to it (Line 5). Finally, it calculates the exclusive-or of `r1` and `r4`. Our experiments show leakage in Line 9, 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 6–8 in Listing 4 removes the interaction between the original values of `r2` and `r4`, but leaves an interaction between the original values of `r5` and `r4`.

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. Implementing the `str` instruction requires two cycles [38, Figure 4.6]. In the first, the processor calculates the store address and in the second it performs the store. We believe that, to avoid locking the register file, in the first cycle the processor copies the contents of the register to an intermediate latch, from which it is retrieved in the second cycle. We believe that a glitch on the bus causes interference between the contents of the latch and the second argument of subsequent instructions, explaining the leakage we observe.

F. Extending the ELMO *Model*

Recall (Section IV-A) that ELMO builds its model using a linear regression from traces collected from sequences of three instructions. To account for the effects of storage elements we identified, we update the model to include a few more components. We call out extension ELMO*.

Whereas the ELMO model treats each operand separately, we also look at combinations of bits across the two operands of the instruction. Because the first operand is typically the destination register, correlating the two operands captures the effect of calculating the result of the operation and overwriting the destination register.

To capture interactions via memory and internal storage elements, we track the contents of these elements, and add model components that correlate with them.

In total, our model consists of 25 components. We validate our model by repeating the test cases used for identifying the

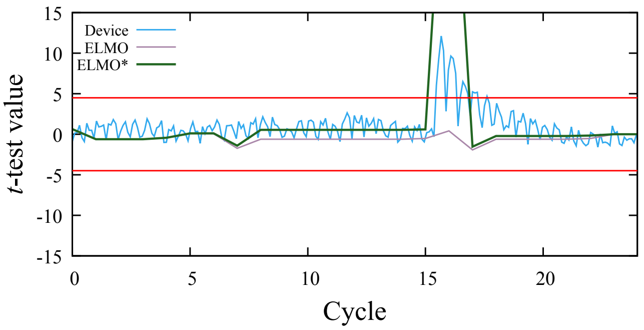


Figure 5: Leakage from `str` to `eors`.

storage elements. For example, Figure 5 shows the real and the emulated leakage from running the code in Listing 1 on the real hardware, ELMO, and in ELMO*. We see that our model identifies the leakage that the original ELMO model misses. We note that the leakage in ELMO* has a higher *t*-test value than the actual leakage. We speculate that the reason is that the model does not suffer from physical noise. As a result, ELMO* requires less traces than a hardware measurement for leakage detection.

Unlike ELMO, ELMO* not only emulates the leaked signal, but also finds which of the components of the model is causing the leak. We, therefore calculate the *t*-test value not only for the combined emulated signal, but also to separate components of the signal.[4] Thus, for example, we keep track of the *t*-test value of the part of the signal contributed by each of the instruction operands, by interactions between the instruction result and its operands, and by interactions between the instruction results or operands with values previously stored to or loaded from memory. Using this information, when ELMO* reports that an instruction leaks, we can inspect the components and identify the leakage cause.

V. Code Rewrite in Rosita

As Section III describes, the core of Rosita is a rewrite engine that uses the output of the ELMO* emulator to drive code fixes for leakage. We assume that the original code is masked, i.e. it does not leak at the algorithm level. However, the translation of the algorithm into machine code and the execution of this machine code can result in unintended and unexpected leakage. In this section, we review the causes of leakage we identify, and describe the fixes the Rosita applies for each. We begin with a high-level description of Rosita and proceed with the details of the rewrite rules it applies.

A. Rosita *Design*

Rosita is a rewrite engine that takes the code and the output of ELMO*, and rewrites the code to avoid leakage. To decide which rewrite rule to apply, Rosita relies on ELMO* to identify the leaking component. (See Section IV-F.)

The main strategy Rosita uses to fix the leakage is to wipe stored state with a random mask. For that, Rosita dedicates a *mask register* (Rosita uses register `r7`), which is initialized with a random 32-bit mask. When compiling the software, we use the flag `-ffixed-r7` to direct the compiler not to use the mask register, ensuring that its contents are not modified except by Rosita. Similarly, we require assembly implementation to not use `r7`.

It is important to note that all of the rewrite rules do not eliminate values used by the program. Thus, if the implementation is not fully masked or uses incorrect randomness, Rosita will be unable to remove the leakage. Conversely, a success in eliminating a leak is a demonstration that the leak originated from unintended interaction.

B. *Operand Interaction*

One of the common forms of unintended interaction is between the operands of successive instructions. Technically, as McCann et al. [62] note, loading an operand to the bus leaks the Hamming distance between the value previously held in the bus and the new value. If both values use the same mask, the Hamming distance between the masked values is the same as that between the original values.

Rosita identifies such leakage by checking the various *t*-test values calculated for the operands and their relationship with those of prior instructions. In the case that the leakage is caused by such interaction, Rosita inserts an access to the mask register, using `movs r7, r7` The instruction moves the contents of the mask register into the mask register, and is therefore functionally a no-op. However, because the value of the mask register goes through the bus, the previous contents of the bus is wiped, removing the interaction between the two masked values.

C. *Register Reuse*

Due to the limited number of registers, compilers and programmers often reuse those, e.g. when the old contents are either consumed or stored in memory. Reusing a register rarely removes the old contents from it. Consequently, when new data is loaded into a register, it interacts with algorithmically unrelated data that remains from prior uses of the register.

Papagiannopoulos and Veshchikov [67] note that if the old contents and the new contents are both masked using the same mask value, the difference between the masked contents, i.e. their exclusive or, is the same as the difference between the unmasked contents. Consequently, when a register is used consecutively for two values with the same mask, it leaks the difference between the values.

To identify this form of leakage, Rosita checks the *t*-test value of overwriting register value. Once identified, Rosita wipes the old contents of the register by copying the contents of the mask register to the destination register of the leaking instruction, as Papagiannopoulos and Veshchikov [67] suggest. For example, suppose that the instruction `movs r3, r4` leaks because both `r3` and `r4` contain values masked with the same mask. To eliminate the leak, Rosita inserts `movs r3, r7` before the leaking instruction.

[4] Implementation note: to reduce memory usage, we calculate the *t*-test values incrementally, using Welford's algorithm [85].

D. Rotation Operations

Rotation operations show interaction between the value pre and post rotation. When a single masked value is rotated, this interaction is unlikely to leak secret data because the mask hides the contents. However, when rotating a word comprised of multiple masked values that all use the same mask, the result of the rotation may align the masked values, effectively nullifying the mask, leaking the difference of the unmasked values.

We propose two approaches to remove this leakage. As an example, suppose that we would like to rotate the register `r2`, whose value is a concatenation of four masked bytes: $(b_1 \oplus m)||(b_2 \oplus m)||(b_3 \oplus m)||(b_4 \oplus m)$. Rotation of `r2` by a multiple of 8 bits would result in leakage of information on the value of the b_i's. For example, assuming `r3` contains the value 8, the instruction `ror r2, r3` would set the value of `r2` to $(b_2 \oplus m)||(b_3 \oplus m)||(b_4 \oplus m)||(b_1 \oplus m)$, and the interaction between the original and the rotated values of `r2` would leak the Hamming weight of $(b_1 \oplus b_2)||(b_2 \oplus b_3)||(b_3 \oplus b_4)||(b_4 \oplus b_1)$.

Word Mask. A straightforward approach for preventing such leakage is to mask the word with our mask register (`r7`), rotate both the word and the mask register and then use the rotated mask to unmask the word. Thus, instead of rotating `r2`, we rotate `r2` $\oplus$ `r7`. As an example, Figure 6 shows how Rosita fixes a `rors r2, r3` instruction that Elmo* indicates is leaking.

```
rors r2, r3          eors r2, r7
                     rors r2, r3
                     rors r7, r3
                     eors r2, r7
```

Figure 6: Masking rotation operations. The leaking `ror` operation on the left is replaced with a masking sequence on the right.

We note that this sequence modifies the contents of our mask register. However, this has no effect on the functionality because the mask register is assumed to be random and there is no long-term dependency on its exact contents.

Partial Rotations. An alternative approach is to combine multiple shifts to avoid rotations of multiples of the data size. For example, a rotation by 8 bits can be replaced with a rotation by 3 bits followed by a rotation by 5 bits.

Rosita employs the word mask approach both because it is more general, i.e. does not depend on the size of the rotation, and because it already has the mask register, which it uses for the other fixes.

E. Memory Operations

As discussed in Section IV-E, there are several effects that can cause interactions between values used in memory operations. These include a storage element in the memory bus that remembers recently accessed memory value and consequently leaks the Hamming distance between the remembered value and the current one on memory access operations, interaction between loaded and stored values and the previous contents they overwrite, and an interaction between bytes in stored words.

When Elmo* indicates that a load instruction leaks due to interaction with the memory bus storage element, Rosita wipes the contents of the bus by pushing the mask register to the stack and popping from the stack to the destination register of the load instruction. Figure 7 shows an example of an `ldr` instruction (left) that leaks through interaction of the loaded value with a previously loaded value. To fix this, Rosita inserts a `push` and a `pop` instructions before the load, yielding the code fragment in the right. Popping the mask to the destination of the load instruction also protects against leakage through interaction with the previous value of the destination register.

```
ldr r2, [r3]          push r7
                      pop r2
                      ldr r2, [r3]
```

Figure 7: A leaking load instruction (left) and the fixed sequence (right).

Due to the more intricate potential interactions, the picture with store instructions is a bit more complex. To overcome interactions with the previous value used on the memory bus and to address possible interactions with the previous contents of memory, Rosita first stores the mask register into the destination location and then performs the required store (See Figure 8).

```
str r2, [r3]          str r7, [r3]
                      str r2, [r3]
```

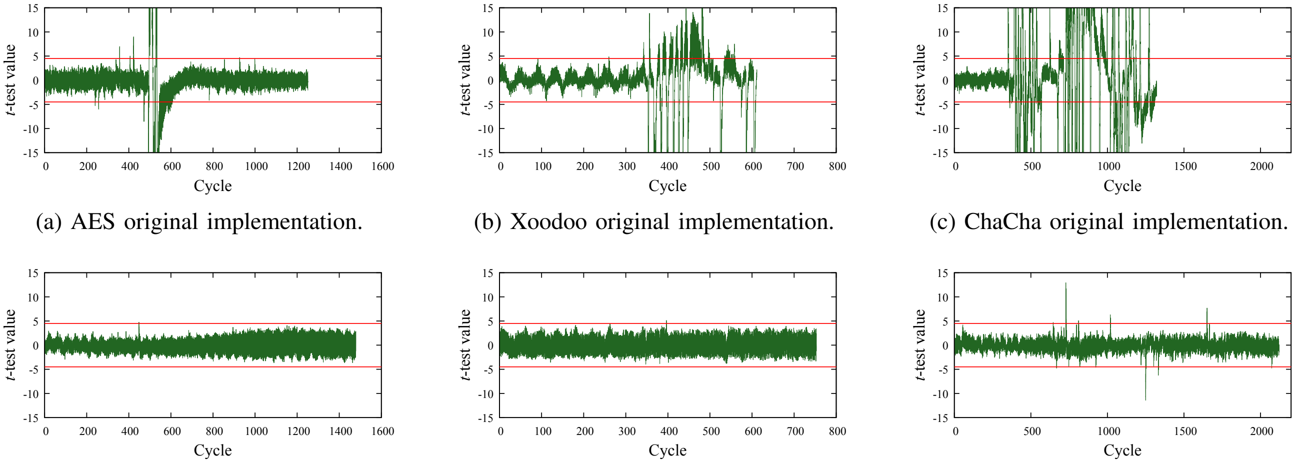
Figure 8: A leaking store instruction (left) and the fixed sequence (right).

When byte interaction within the stored data leaks, Rosita stores one byte at a time. In such a case, care should be taken to ensure that these bytes and the operations required for their storage do not create unintended interactions, leading to a relatively long code segment in Figure 14 in Appendix B. While this rewrite rule eliminates the leakage, the performance cost of using it is significant. As such, it may be better to avoid stores of words that contain multiple values masked with the same mask. Changing the logic of the cipher is outside the scope of Rosita.

VI. Evaluation

We evaluate Rosita with masked implementations of three cryptographic primitives. AES [27] is one of the most commonly used ciphers, having been an international standard since 2001. We use the byte-masked implementation of AES-128 by Yao et al. [87].[5] To perform the ShiftRows operation of AES, which permutes bytes in the data being encrypted, the implementation uses byte loads and stores. Following the suggestion of Gao [40], we use different masks for each row to avoid leakage through interactions between bytes in memory words.

[5] https://github.com/Secure-Embedded-Systems/Masked-AES-Implementation/tree/master/Byte-Masked-AES



(a) AES original implementation.

(b) Xoodoo original implementation.

(c) ChaCha original implementation.

(d) AES fixed with ROSITA.

(e) Xoodoo fixed with ROSITA.

(f) ChaCha fixed with ROSITA.

Figure 9: Fixed vs. random tests for the three cipher (one fixed input, 1M traces).

Efficient software AES implementations on CPUs (without dedicated AES instruction) use table lookups, which makes them vulnerable to cache-based attacks. As an example of a modern primitive, the second cipher we use is the cryptographic permutation Xoodoo. Xoodoo was proposed recently by Daemen et al. in [28] for use in authenticated encryption modes [29]. The optimized and non-masked implementation of Xoodoo we took from [16] and we implemented the 2-share boolean masking scheme of the non-linear layer χ as in Bertoni et al. [15] ourselves. Implementing the 2-share boolean masking of the linear layer was trivial. In contrast to what Bertoni et al. [15] mention, we initialize the state with fresh randomness for each trace to keep it consistent with the implementation of AES, even though this is not required.

The third cipher we consider is ChaCha as a prominent example of an ARX cipher. ChaCha is very efficient in software and widely used in TLS implementations. The challenge is to mask it at low cost and the best result for ARM Cortex-M3 and Cortex-M4 processors was recently published by Jungk et al. [51]. We use their implementation in our experiments.

A. Fixing Leakage

We first show ROSITA's success in fixing the leakage it detects. Figure 9a shows the results of a non-specific fixed vs. random experiments with 1 000 000 traces of executing the first round of the AES implementation. The figure shows leakage (t-value above the threshold of 4.5) around cycles 500–550, which correspond to the AES SHIFTROWS operation. As Figure 9d shows, ROSITA detects the leaks and fixes them. This fix does not, however, come for free. The first round now takes 1 479 cycles, compared with 1 285 for the original implementation—a slowdown of 15%. Figures 9b and 9e show similar results for ChaCha and Xoodoo with 61% (1322 vs. 2122 cycles) and 18% (637 vs. 753 cycles) slowdowns respectively.

To determine the trend of leakage, we perform the fixed vs. random test on the hardware with a varying number of traces. Figure 10 shows the results for both the original and the fixed implementations. The horizontal axis shows the number of traces used for the fixed vs. random test, and the vertical is the maximum absolute value of the t-test for each of the implementations. As we can see, the original implementations show increasing leakage, significant leakage is visible even with as little as 1 000 traces, and the confidence increases as traces are added. Our fixed implementations show significantly less leakage up to 1 000 000 traces. To remove the remaining leakage, we need to use more than one input. We discuss this issue next.

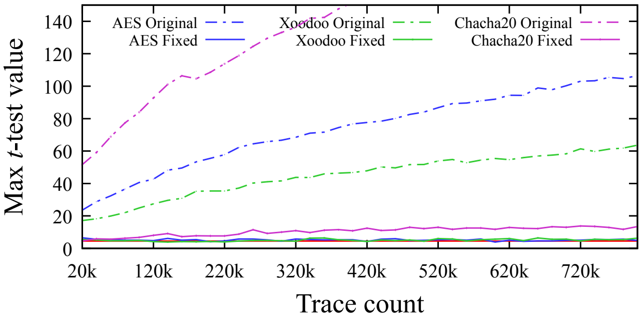
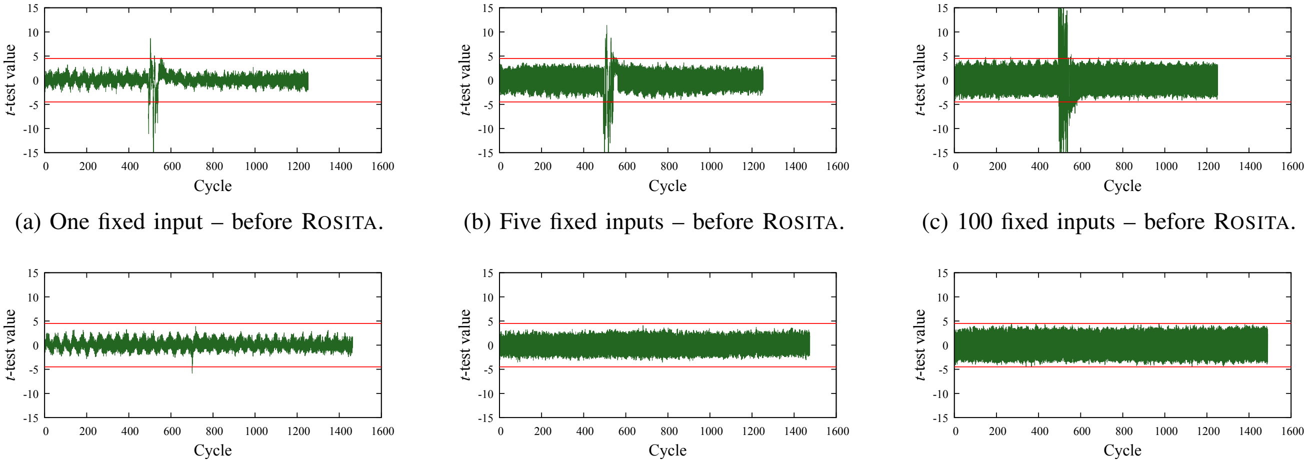


Figure 10: t-test value trend.

B. Multiple Fixed Inputs

One of the known limitations of TVLA is that it may miss some leakage if used with only one input [76]. Thus, common operating procedures require running multiple fixed vs. random tests, each with a different fixed input. For example, the ISO 17825 standard requires two fixed inputs. To test the impact of multiple fixed inputs, we perform multiple fixed vs. random tests, each with a different randomly chosen fixed input. To combine the results of multiple experiments, we use the largest absolute t-value calculated for a sample point as a representative for leakage at that point. Thus, if any of the fixed vs. random tests indicates leakage at a point, the combined result will also indicate leakage there. The top row of Figure 11



(a) One fixed input – before ROSITA.

(b) Five fixed inputs – before ROSITA.

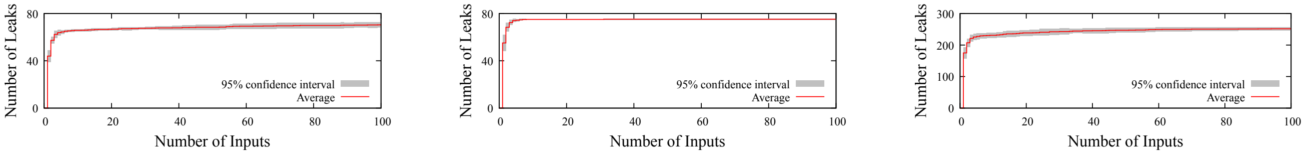
(c) 100 fixed inputs – before ROSITA.

(d) One fixed input – after ROSITA.

(e) Five fixed inputs – after ROSITA.

(f) 100 fixed inputs – after ROSITA.

Figure 11: t-test of masked AES implementation before and after ROSITA, varying the number of fixed vs. random pairs.



(a) AES

(b) Xoodoo

(c) ChaCha

Figure 12: Average number of leaks per fixed inputs.

shows the combined results from 1, 5, and 100 fixed inputs for AES. As we can observe, when we increase the number of fixed points, the number of locations that show leakage increases. Figures 15 and 16 in Appendix C show the results for Xoodoo and ChaCha.

Running ROSITA with these inputs allows us to fix most of these leakages. Repeating the experiment with the code that ROSITA produces and the same fixed inputs we get the t-test values in the bottom rows of the figures. We can observe that ROSITA fixes the leakage, at a cost to the performance. Also, when there is higher leakage points, it is required to have more fixed inputs to cover all of them. For ChaCha, 100 fixed inputs were not enough to cover all leakage points as the final result shows 1 leaky point. Table III compares the performance of the code before and after running ROSITA, showing a maximum overhead of 64% for ChaCha.

Table III: Encryption length (cycles) after fixing with ROSITA with varying number of fixed inputs.

cipher	Original	1 Fixed	5 Fixed	100 Fixed
AES	1 285	1 464	1 475	1 479
ChaCha	1 322	2 066	2 114	2 162
Xoodoo	637	735	765	769

The number of leakage points identified depends on the fixed inputs chosen for the fixed vs. random test. To better understand the relationship, we want to find out how many leakage points we expect to find for a given number of fixed vs. random inputs. Figure 12 shows this for AES, ChaCha and Xoodoo. For a given number of inputs, the plots display the average number of leakage points that ELMO* discovers over 10 selections of inputs. The figure also displays the 95% confidence interval. We see that 10 fixed inputs are enough to find 93% of the leakage points in AES, 92% in ChaCha and 99% in Xoodoo. When we compare the identified leakage points against the ground truth, we find that many of the discovered leakage points are false positives, explaining the discrepancy between the figures and Table I. To verify that we discover all of the real leakage points, we use ROSITA with 100 fixed inputs to fix AES and Xoodoo. We then use the produced code on the hardware with a new set of 100 fixed inputs. We found no evidence for leakage with either AES or Xoodoo but for ChaCha we found one leaky point that had a t-test value of 5.2.

We note that ROSITA's success in eliminating leakage demonstrates that the original programs are, indeed, first-order secure, at least semantically. As we mention in Section V-A, the rewrite rules can only fix leakage that stems from unintended interactions.

C. Performance

The performance bottleneck for ROSITA is running ELMO* to generate the simulated traces. We can collect 10 000 traces of AES in 26 seconds, ChaCha in 45 and Xoodoo in 21. In comparison, the code rewrite phase of ROSITA takes around 0.1 seconds.

Collecting the same number of 1-round traces from the hardware takes 117 seconds for AES, 220 for ChaCha and

147 for Xoodoo. The collection of traces for Xoodoo from the hardware is slower than for AES because we provide fresh shares for every trace as mentioned above. Hence, the communication dominates the execution time. Thus, ELMO* is 4.5–7 times faster than the real hardware.

We note that the task of collecting traces is ridiculously parallelizable. Hence, on a typical desktop, we can collect traces eight times faster, and with an investment of \$1 000 we can double the rate again. In contrast, to parallelize trace collection from the hardware, we would need to replicate the setup, at a cost of over \$10 000 per node. Thus, the effective speed of ROSITA is about two order of magnitude faster than the hardware.

VII. LIMITATIONS AND FUTURE WORK

Possibly the main limitation of ROSITA is that, while we have found no evidence for leakage, there is no guarantee that it fixes all leakage. There two main reasons for that:

- **Methodology:** While popular and standardized, non-specific fixed vs. random tests are not a panacea for leakage. Leakage they detect is not necessarily exploitable and the absence of detection does not necessarily mean that no leakage exists. In particular, the method only detects first-order leakage. We note, however, that this does not detract from ROSITA achieving its aim of assisting in assessing compliance with the standard.
- **Model Limitations:** ROSITA relies on ELMO* which is only a model of the hardware. Gaps between the model and the real hardware, both in terms of capturing the hardware behavior and in terms of accuracy of the model can result in missed leakage. Moreover, hardware behavior may change over time, e.g. due to a microcode update [72]. For these reasons we recommend that operators do not rely solely on ROSITA. It can be used to achieve a high degree of assurance, and is likely to automate a significant part of the work required for compliance, but testing with the real hardware is essential.

A further limitation of ROSITA, which, like others, stem from the ELMO* model is its suitability for other processors. ELMO* uses a very simple model of the processor. It is suitable for small, in-order, cacheless microcontrollers, such as the Cortex-M0, AVR processors such as the ATMEGA328p, or small RISC-V processors. However, the model is unlikely to be suitable for more advanced processors. Nonetheless, we believe that the ROSITA is important, because these microcontrollers it targets are extremely popular in embedded devices, where they often implement cryptographic functionalities. At the same time, there is very little control of the physical environment of such devices, allowing the attacker unfettered access and enabling the type of attacks we defend against. We leave porting ELMO* and ROSITA to other microcontrollers to future work.

Another direction to which ROSITA could be extended is using other statistical tests. While the ISO 17825 does not require them, tests such as mutual information [45] and χ^2 [53], have been proven useful as statistical distinguishers. Moreover, extending ROSITA to support second- and higher-order attacks would allow supporting more secure implementations. It may also be possible to extend ROSITA to use correlation power analysis (CPA) [19]. This would, however, require the operator to provide ROSITA with possible values to search for correlation. It is not clear that this can be done in a generic fashion.

VIII. CONCLUSIONS

Since their introduction over two decades ago, physical side-channel attacks have presented a serious security threat, particularly to small computational devices that need to maintain secrets under the physical control of the adversary. To protect against such attacks, many ciphers' implementations employ masking techniques that combine intermediate values with randomly selected masks. As a consequence, due to the mask being uniformly distributed, leakage of a masked value does not reveal information to the adversary. While proven secure against certain attacks, in practice masked implementations often leak secret information due to unintended interactions between masked values involving hardware they are loaded and stored to. To fix these leaks, the common practice is to repeatedly "tweak the code until it stops leaking".

In this work, we have set out to explore if leakage emulators can be used for the *automatic* elimination of side channel leakage from software implementations. To achieve this, we have created a code rewrite engine called ROSITA and combined it with the extended leakage emulator ELMO*:

- ROSITA incorporates rules to mitigate leakage arising from operand interactions, register reuse, rotation operations, and memory operations.
- ELMO has undergone a major upgrade to ELMO* for two reasons: firstly, it had to be able to tell ROSITA the cause of the leakage, and secondly, we have added support by including the values that instructions store in various micro-architectural storage elements, which hold state that can leak information.

In our proof-of-concept, we used ROSITA with ELMO* to automatically protect masked implementations of AES, Xoodoo and ChaCha. Our experiments using the actual hardware show the absence of exploitable leakage at only a 64% penalty to the performance, which is the worst case i.e. ChaCha. As ChaCha is the most complex cipher of the three to mask (in terms of the overhead in performance and resources) this is also reflected in the penalty in the tools' performance.

ACKNOWLEDGEMENTS

Francesco Regazzoni received support from the European Union Horizon 2020 research and innovation program under CERBERO project (grant agreement number 732105).

This research was supported by the Australian Research Council project numbers DE200101577, DP200102364, and DP210102670; the Research Center for Cyber Security at Tel-Aviv University established by the State of Israel, the Prime Minister's Office and Tel-Aviv University; and by a gift from Intel.

REFERENCES

[1] O. Acıiçmez, W. Schindler, and Ç. K. Koç, "Improving Brumley and Boneh timing attack on unprotected SSL implementations," in *CCS*, 2005.

[2] G. Agosta, A. Barenghi, and G. Pelosi, "A code morphing methodology to automate power analysis countermeasures," in *DAC*, 2012, pp. 77–82.

[3] M. J. Aigner, S. Mangard, F. Menichelli, R. Menicocci, M. Olivieri, T. Popp, G. Scotti, and A. Trifiletti, "Side channel analysis resistant design flow," in *ISCAS*, 2006.

[4] M. Backes, M. Dürmuth, S. Gerling, M. Pinkal, and C. Sporleder, "Acoustic side-channel attacks on printers," in *USENIX Security*, 2010, pp. 307–322.

[5] J. Balasch, B. Gierlichs, R. Verdult, L. Batina, and I. Verbauwhede, "Power analysis of Atmel CryptoMemory - recovering keys from secure EEPROMs," in *CT-RSA*, 2012, pp. 9–34.

[6] J. Balasch, B. Gierlichs, V. Grosso, O. Reparaz, and F. Standaert, "On the cost of lazy engineering for masked software implementations," in *CARDIS*, 2014, pp. 64–81.

[7] G. Barthe, S. Belaïd, F. Dupressoir, P. Fouque, B. Grégoire, and P. Strub, "Verified proofs of higher-order masking," in *EUROCRYPT (1)*, 2015, pp. 457–485.

[8] L. Batina, S. Bhasin, D. Jap, and S. Picek, "CSI NN: reverse engineering of neural network architectures through electromagnetic side channel," in *USENIX Security*, 2019, pp. 515–532.

[9] A. G. Bayrak, F. Regazzoni, D. Novo, P. Brisk, F.-X. Standaert, and P. Ienne, "Automatic application of power analysis countermeasures," *IEEE Trans. Computers*, vol. 64, no. 2, pp. 329–341, 2015.

[10] G. Becker, J. Cooper, E. DeMulder, G. Goodwill, J. Jaffe, G. Kenworthy, T. Kouzminov, A. Leiserson, M. Marson, P. Rohatgi, and S. Saab, "Test vector leakage assessment (TVLA) methodology in practice," http://icmc-2013.org/wp/wp-content/uploads/2013/09/Rohatgi_Test-Vector-Leakage-Assessment.pdf, 2013.

[11] D. J. Bernstein, "Cache-timing attacks on AES," 2005, preprint available at http://cr.yp.to/papers.html#cachetiming.

[12] ——, "ChaCha, a variant of Salsa20," https://cr.yp.to/chacha/chacha-20080128.pdf, 2008.

[13] D. J. Bernstein, T. Lange, and P. Schwabe, "The security impact of a new cryptographic library," in *LATINCRYPT*, 2012, pp. 159–176.

[14] D. J. Bernstein, S. Kölbl, S. Lucks, P. M. C. Massolino, F. Mendel, K. Nawaz, T. Schneider, P. Schwabe, F.-X. Standaert, Y. Todo *et al.*, "Gimli: a cross-platform permutation," in *CHES*, 2017, pp. 299–320.

[15] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, and R. V. Keer, "KECCAK implementation overview," May 2012, https://keccak.team/papers.html.

[16] G. Bertoni, J. Daemen, S. Hoffert, M. Peeters, G. Van Assche, and R. Van Keer, "The extended Keccak code package (XKCP)." [Online]. Available: https://github.com/XKCP/XKCP

[17] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, "Keccak," in *Eurocrypt*, T. Johansson and P. Q. Nguyen, Eds., 2013, pp. 313–314.

[18] R. Bloem, H. Groß, R. Iusupov, B. Könighofer, S. Mangard, and J. Winter, "Formal verification of masked hardware implementations in the presence of glitches," in *EUROCRYPT (2)*, 2018, pp. 321–353.

[19] E. Brier, C. Clavier, and F. Olivier, "Correlation power analysis with a leakage model," in *CHES*, 2004, pp. 16–29.

[20] B. B. Brumley and N. Tuveri, "Remote timing attacks are still practical," in *ESORICS*, 2011, pp. 355–371.

[21] D. Brumley and D. Boneh, "Remote timing attacks are practical," in *USENIX Security*, 2003, pp. 1–14.

[22] V. Carlier, H. Chabanne, E. Dottax, and H. Pelletier, "Electromagnetic side channels of an FPGA implementation of AES," IACR Cryptology ePrint Archive report 2004/145, 2004.

[23] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi, "Towards sound approaches to counteract power-analysis attacks," in *CRYPTO*, 1999, pp. 398–412.

[24] Z. Chen and Y. Zhou, "Dual-rail random switching logic: A countermeasure to reduce side channel leakage," in *CHES*, 2006, pp. 242–254.

[25] Z. Chen, S. Haider, and P. Schaumont, "Side-channel leakage in masked circuits caused by higher-order circuit effects," in *ISA*, 2009, pp. 327–336.

[26] J. Cooper, E. DeMulder, G. Goodwill, J. Jaffe, and G. Kenworthy, "Test vector leakage assessment methodology in practice," *International Cryptographic Module Conference*, 2013.

[27] J. Daemen and V. Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard*, ser. Information Security and Cryptography. Springer, 2002.

[28] J. Daemen, S. Hoffert, G. Van Assche, and R. Van Keer, "The design of Xoodoo and Xoofff," *IACR Trans. Symmetric Cryptol.*, vol. 2018, no. 4, pp. 1–38, 2018.

[29] ——, "Xoodoo cookbook," IACR Cryptology ePrint Archive report 2018/767, 2018.

[30] E. De Mulder, S. B. Örs, B. Preneel, and I. Verbauwhede, "Differential power and electromagnetic attacks on a FPGA implementation of elliptic curve cryptosystems," *Comput. Electr. Eng.*, vol. 33, no. 5-6, pp. 367–382, 2007.

[31] E. De Mulder, S. Gummalla, and M. Hutter, "Protecting RISC-V against side-channel attacks," in *DAC*, 2019, p. 45.

[32] J. den Hartog, J. Verschuren, E. P. de Vink, J. de Vos, and W. Wiersma, "PINPAS: a tool for power analysis of smartcards," in *SEC*, 2003, pp. 453–457.

[33] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, "Ascon v1. 2," *Submission to the CAESAR Competition*, 2016.

[34] M. Dworkin, "Sha-3 standard: Permutation-based hash and extendable-output functions, nist fips-202. august 2015."

[35] T. Eisenbarth, T. Kasper, A. Moradi, C. Paar, M. Salmasizadeh, and M. T. M. Shalmani, "On the power of power analysis in the real world: A complete break of the KeeLoq code hopping scheme," in *CRYPTO*, 2008, pp. 203–220.

[36] H. Eldib and C. Wang, "Synthesis of masking countermeasures against side channel attacks," in *CAV*, 2014, pp. 114–130.

[37] W. Fischer and B. M. Gammel, "Masking at gate level in the presence of glitches," in *CHES*, 2005, pp. 187–200.

[38] S. Furber, *ARM System-on-Chip Architecture*, 2nd ed. Addison-Wesley, 2000.

[39] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic analysis: Concrete results," in *CHES*, 2001, pp. 251–261.

[40] S. Gao, "A Thumb assembly based byte-wise masked AES implementation," https://github.com/bristol-sca/ASM_MaskedAES, 2019.

[41] S. Gao, J. Großschädl, B. Marshall, D. Page, T. Pham, and F. Regazzoni, "An instruction set extension to support software-based masking," Cryptology ePrint Archive, Report 2020/773, 2020.

[42] S. Gao, B. Marshall, D. Page, and E. Oswald, "Share-slicing: Friend or foe?" *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2020, no. 1, pp. 152–174, 2020.

[43] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *J. Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, 2018.

[44] D. Genkin, A. Shamir, and E. Tromer, "RSA key extraction via low-bandwidth acoustic cryptanalysis," in *CRYPTO (1)*, 2014, pp. 444–461.

[45] B. Gierlichs, L. Batina, P. Tuyls, and B. Preneel, "Mutual information analysis," in *CHES*, 2008, pp. 426–442.

[46] B. Gigerl, V. Hadzic, R. Primas, S. Mangard, and R. Bloem, "Coco: Co-design and co-verification of masked software implementations on CPUs," IACR Cryptology ePrint Archive report 2020/1294, 2020.

[47] L. Goubin and J. Patarin, "DES and differential power analysis. the "duplication" method," in *CHES*, Aug 1999, pp. 158–172.

[48] H. Groß, M. Jelinek, S. Mangard, T. Unterluggauer, and M. Werner, "Concealing secrets in embedded processors designs," in *CARDIS*, 2016, pp. 89–104.

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

[50] Y. Ishai, A. Sahai, and D. A. Wagner, "Private circuits: Securing hardware against probing attacks," in *CRYPTO*, vol. 2729, 2003, pp. 463–481.

[51] B. Jungk, R. Petri, and M. Stöttinger, "Efficient side-channel protections of ARX ciphers," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, pp. 627–653, 2018.

[52] E. Käsper and P. Schwabe, "Faster and timing-attack resistant AES-GCM," in *CHES*, 2009, pp. 1–17.

[53] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side channel cryptanalysis of product ciphers," in *ESORICS*, 1998, pp. 97–110.

[54] P. Kiaei and P. Schaumont, "Domain-oriented masked instruction set architecture for RISC-V," in *SECRISC-V*, 2020.

[55] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *CRYPTO*, 1996, pp. 104–113.

[56] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *CRYPTO*, 1999, pp. 388–397.

[57] T. Krachenfels, F. Ganji†, A. Moradi, S. Tajik†, and J.-P. Seifert, "Real-world snapshots vs. theory:questioning the t-probing security model," in *IEEE SP*, 2021.

[58] J. Krämer, D. Nedospasov, A. Schlösser, and J.-P. Seifert, "Differential photonic emission analysis," in *COSADE*, 2013, pp. 1–16.

[59] Y. Le Corre, J. Großschädl, and D. Dinu, "Micro-architectural power simulator for leakage assessment of cryptographic software on ARM

Cortex-M3 processors," in *COSADE*, 2018, pp. 82–98.

[60] S. Mangard, T. Popp, and B. M. Gammel, "Side-channel leakage of masked CMOS gates," in *CT-RSA*, 2005, pp. 351–365.

[61] S. Mangard, E. Oswald, and T. Popp, *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007.

[62] D. McCann, E. Oswald, and C. Whitnall, "Towards practical tools for side channel aware software engineering: 'grey box' modelling for instruction leakages," in *USENIX Security*, 2017, pp. 199–216.

[63] T. S. Messerges, "Securing the AES finalists against power analysis attacks," in *FSE*, April 2000, pp. 150–164.

[64] ——, "Power analysis attacks and countermeasures for cryptographic algorithms," Ph.D. dissertation, University of Illinois at Chicago, USA, 2000.

[65] T. S. Messerges, E. A. Dabbish, and R. H. Sloan, "Investigations of power analysis attacks on smartcards," in *USENIX — Smartcard'99*, 1999, pp. 151–162.

[66] L. W. Nagel and D. O. Pederson, "Simulation program with integrated circuit emphasis (SPICE)," in *Midwest Symposium on Circuit Theory*, 1973.

[67] K. Papagiannopoulos and N. Veshchikov, "Mind the gap: Towards secure 1st-order masking in software," in *COSADE*, 2017, pp. 282–297.

[68] A. Park, K. Shim, N. Koo, and D. Han, "Side-channel attacks on post-quantum signature schemes based on multivariate quadratic equations," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2018, no. 3, pp. 500–523, 2018.

[69] J.-J. Quisquater and D. Samyde, "Electromagnetic analysis (EMA): Measures and counter-measures for smart cards," in *Smart Card Programming and Security*, 2001, pp. 200–210.

[70] C. Rechberger and E. Oswald, "Practical template attacks," in *WISA*, 2004, pp. 443–457.

[71] M. Renauld, F. Standaert, N. Veyrat-Charvillon, D. Kamel, and D. Flandre, "A formal study of power variability issues and side-channel attacks for nanoscale devices," in *EUROCRYPT*, 2011, pp. 109–128.

[72] O. Reparaz, J. Balasch, and I. Verbauwhede, "Dude, is my code constant time?" in *DATE*, 2017, pp. 1697–1702.

[73] A. Schlösser, D. Nedospasov, J. Krämer, S. Orlic, and J.-P. Seifert, "Simple photonic emission analysis of AES," *J. Cryptographic Engineering*, vol. 3, no. 1, pp. 3–15, 2013.

[74] T. Schneider and A. Moradi, "Leakage assessment methodology - a clear roadmap for side-channel evaluations," in *CHES*, 2015, pp. 495–513.

[75] N. Sehatbakhsh, B. B. Yilmaz, A. G. Zajic, and M. Prvulovic, "EMSim: A microarchitecture-level simulation tool for modeling electromagnetic side-channel signals," in *HPCA*, 2020, pp. 71–85.

[76] F.-X. Standaert, "How (not) to use Welch's t-test in side-channel security evaluations," in *COSADE*, 2018, pp. 65–79.

[77] K. Tiri and I. Verbauwhede, "A digital design flow for secure integrated circuits," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 25, no. 7, pp. 1197–1208, 2006.

[78] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on AES, and countermeasures," *J. Cryptology*, vol. 23, no. 1, pp. 37–71, 2010.

[79] M. Tunstall and G. Goodwill, "Applying TVLA to public key cryptographic algorithms," IACR Cryptology ePrint Archive report 2016/513, 2016.

[80] N. Veshchikov, "SILK: high level of abstraction leakage simulator for side channel analysis," in *PPREW@ACSAC*, 2014, pp. 3:1–3:11.

[81] N. Veshchikov and S. Guilley, "Use of simulators for side-channel analysis," in *EuroS&P*, 2017, pp. 51–59.

[82] C. Wang and P. Schaumont, "Security by compilation: An automated approach to comprehensive side-channel resistance," *SIGLOG News*, vol. 4, no. 2, pp. 76–89, 2017.

[83] J. Wang, C. Sung, and C. Wang, "Mitigating power side channels during compilation," in *ESEC/SIGSOFT FSE*, 2019, pp. 590–601.

[84] B. L. Welch, "The generalization of 'Student's' problem when several different population varlances are involved," *Biometrika*, vol. 34, no. 1–2, pp. 28–35, Jan. 1947.

[85] B. P. Welford, "Note on a method for calculating corrected sums of squares and products," *Technometrics*, vol. 4, no. 3, pp. 419–420, 1962.

[86] M. Wu, S. Guo, P. Schaumont, and C. Wang, "Eliminating timing side-channel leaks using program repair," in *ISSTA*, 2018, pp. 15–26.

[87] Y. Yao, M. Yang, C. Patrick, B. Yuce, and P. Schaumont, "Fault-assisted side-channel analysis of masked implementations," in *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2018, pp. 57–64.

Appendix A
Validating the Setup

To validate our setup, we reproduce the results of McCann et al. [62]. Specifically, we perform a fixed vs. random test on the code in Listing 5, which contains an implementation of one of the steps in the AES encryption known as the ShiftRows operation. Specifically, register `r1` points to the 16 bytes that represent the state of the AES encryption. ShiftRows performs a fixed permutation of these bytes. The implementation loads three four-byte words and uses the `rors` instruction to rotate the bytes, before storing them back to the state.

```
ldr  r4, [ r1, #4 ]
rors r4, r5
str  r4, [ r1, #4 ]
ldr  r4, [ r1, #8 ]
rors r4, r6
str  r4, [ r1, #8 ]
ldr  r4, [ r1, #12 ]
rors r4, r3
str  r4, [ r1, #12 ]
```

Listing 5: ShiftRows from McCann et al. [62]

For the fixed vs. random test we collect 2500 traces where the state contains fixed data masked with the same mask value and 2500 traces where the state consists of random values masked with the same mask. A random value for the mask is chosen for each trace. We compare the distribution of the power reading in each sample point between the fixed and the random traces, and calculate the Welch t-test to check the likelihood that the two distributions are the same. As mentioned before, following common practice in side-channel analysis, we consider the distributions different enough to indicate leakage if the absolute value of the t-test value is above 4.5.

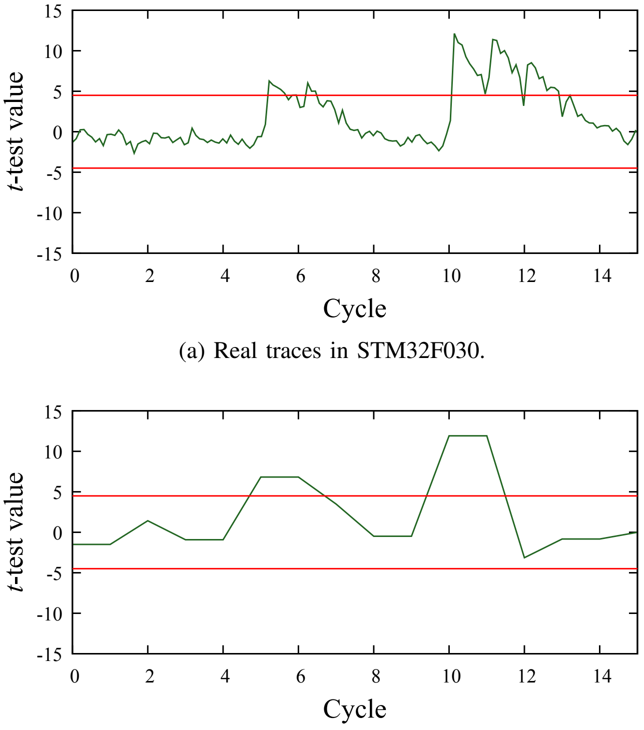
Figure 13a shows the result of the fixed vs. random test. The horizontal axis shows the time and the vertical axis shows the t-test value. We indicate instruction boundaries with vertical bars, and the t-test threshold of ±4.5 with horizontal red lines. Comparing the figure to the results of running Elmo on the same code, shown in Figure 13b, we see that Elmo produces a fairly accurate simulation of the leakage.

In particular, our figure resembles Figure 5 of McCann et al. [62], with only minor differences that reflect the different test environment.

Appendix B
Eliminating Byte Interaction in Stores

Figure 14 shows an example of the rewrite rule for eliminating interactions in byte stores. The code uses two registers chosen to not conflict with the store, `r0` and `r6` in the example in Figure 14. The first is used for selecting the byte to store, while the second is used for the byte. Rosita uses two stores for each byte to avoid interactions on the memory bus or in the DRAM.

Appendix C
Additional Figures



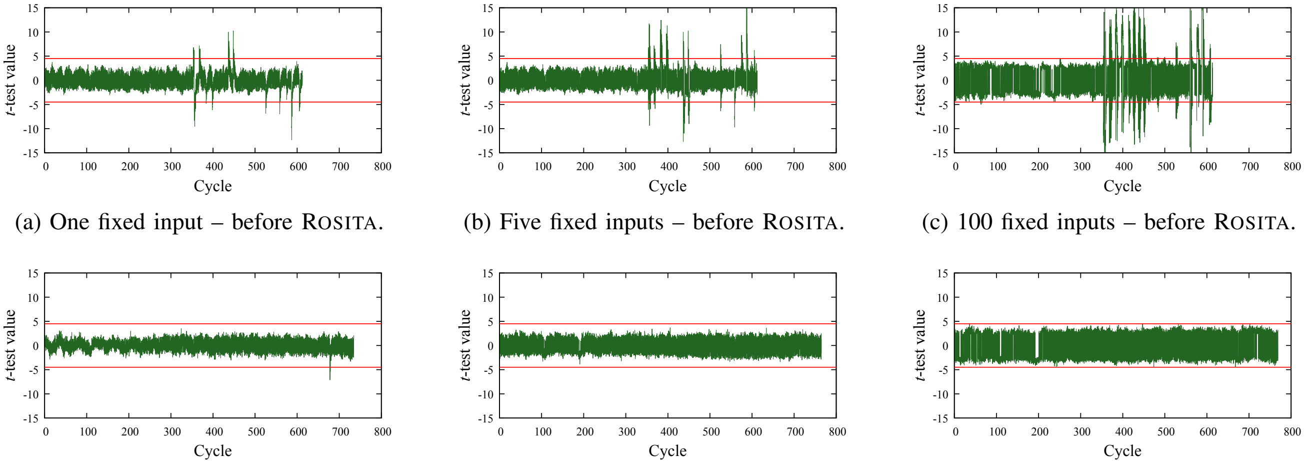
(a) Real traces in STM32F030.

(b) Simulated traces from Elmo.

Figure 13: Fixed vs. random of the AES ShiftRows operation.

```
str r2, [r3]          push {r6}
                      push {r0}
                      movs r0, #0xff
                      movs r6, r2
                      ands r7, r7
                      ands r6, r0
                      lsls r0, #0
                      strb r0, [r3, #0]
                      strb r6, [r3, #0]
                      movs r6, r7
                      movs r6, r2
                      movs r0, #0xff
                      lsls r0, #8
                      ands r7, r7
                      ands r6, r0
                      lsrs r0, #8
                      lsrs r6, #8
                      strb r0, [r3, #1]
                      strb r6, [r3, #1]
                      .
                      .
                      .
                      pop {r0}
                      pop {r6}
```

Figure 14: Addressing byte interaction in stores. A leaking store instruction (left) and part of the fixed sequence (right).



(a) One fixed input – before ROSITA.

(b) Five fixed inputs – before ROSITA.

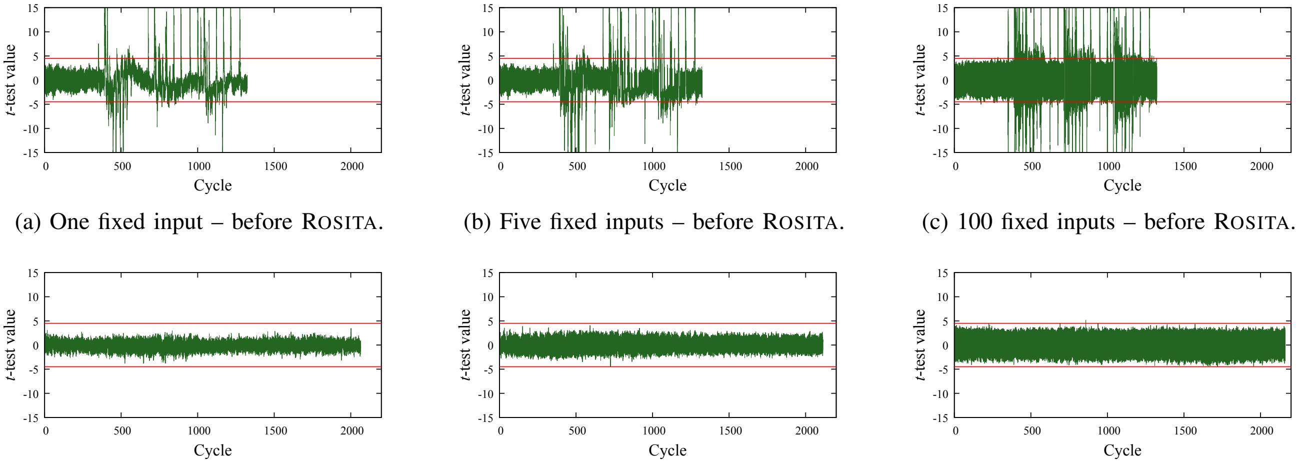
(c) 100 fixed inputs – before ROSITA.

(d) One fixed input – after ROSITA.

(e) Five fixed inputs – after ROSITA.

(f) 100 fixed inputs – after ROSITA.

Figure 15: t-test of masked Xoodoo implementation before and after ROSITA, varying the number of fixed vs. random pairs.



(a) One fixed input – before ROSITA.

(b) Five fixed inputs – before ROSITA.

(c) 100 fixed inputs – before ROSITA.

(d) One fixed input – after ROSITA.

(e) Five fixed inputs – after ROSITA.

(f) 100 fixed inputs – after ROSITA.

Figure 16: t-test of masked ChaCha implementation before and after ROSITA, varying the number of fixed vs. random pairs.