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  
ALaRI – USI  
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, physical 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 amended to correctly emulate the micro-architecture of a target system. We use ROSITA to automatically protect masked implementations of AES and Xoodoo and show the absence of observable leakage at only a 25% penalty to the performance.

I. INTRODUCTION

The seminal work of Kocher [44] 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, 18, 19], power consumption [45], electromagnetic (EM) emanations [20, 35, 59], shared micro-architectural components [37, 66], and even acoustic and photonic emanations [4, 38, 46, 62]. 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 [40, 60], public-key systems, both traditional [27, 55] and post quantum [58], cryptographic primitives implemented in real-world devices [5, 31], 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 [22, 33, 65], through noise generation to hide the signal [52], to algorithmic changes designed to prevent certain leakage [51]. 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.

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) [41]. 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 [70]. 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 [57]. 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?

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

\(^1\)We plan to open-source ROSITA so that others can utilize it and expand on it.
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.

![ Leakage elimination workflow with extended ELMO (ELMO*) and ROSITA.](image)

To emulate leakage, we develop ELMO*, an extended version of the ELMO leakage emulator [50]. We find that, while ELMO correctly emulates leakage between consecutive instructions, it fails to emulate leakage caused by sharing micro-architectural storage elements by instructions further apart. We augment the ELMO model to account for storage elements and develop a procedure to detect architecturally-hidden elements. We identify several hidden elements in the ARM Cortex-M0 we experiment with. We further modify ELMO to not only output the instruction that causes the leakage, but also to identify the cause of the 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 two ciphers with very different round functions: the AES block cipher [24] and the Xoodoo permutation [25], running up to 200,000 traces on each. For AES we use the table-lookup based masked implementation by Yao et al. [76]². 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, [16, 30], Ascon [29], Gimli [13] and Xoodoo [26]. 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. Basically, the two ciphers used are very different and together make our analysis more comprehensive.

As Table I shows, ROSITA successfully eliminates leakage from both AES and Xoodoo. The performance impact of ROSITA is moderate 25%.

To sum up, the contributions of this work are:

- We propose a framework for generating leakage-resilient implementations of masked ciphers by iteratively rewriting the code at leakage points. (Section III)
- We develop techniques to identify multiple sources of leakage that the ELMO leakage model fails to detect. We extend ELMO and augment it to identify these sources 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 and Xoodoo. We test the code ROSITA produces and show the absence of observable leakage in up to 200,000 traces for each cipher. (Section VI)

II. BACKGROUND

A. Side-Channel Attacks

When software runs on some 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 disclose 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 [44] noted that the information acquired through a timing side channel may reveal secret information processed by a cryptographic algorithms. Since then a significant effort has been dedicated to analyzing and eliminating side-channel leakage, particularly in the context of cryptographic implementations [9, 35, 45, 49, 59].

Protection against side-channel attacks depends on both the channel and the attacker capabilities. The constant-time programming style [12, 42], 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, 18, 37].
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 [21, 54]. In a nutshell, masking combines internal values processed by the cryptographic algorithm with random masks, typically by xoring a mask and the internal value. The masked internal values are uniformly distributed, thus the leakage of a masked value does not reveal any information to the attacker.

Although theoretically secure, naive masking often fails to provide the required protection. The main cause is that side-channel leakage often correlates not only with the value being processed, but also with interactions with other values stored within the processor. Transitional effects are caused by the differing power requirements between changing the value of a stored bit and keeping its value unchanged. Thus, the power required for changing the contents of a storage element correlates not only with the new value stored in the element, but also with the difference between the old and the new values. Similarly, when multiple signals arrive at the input of an electronic gate with different timing, a glitch may occur, resulting in power consumption that correlates with the change of the output signal [48].

For masked implementations, unintended interactions, such as transitional effects and glitches can be disastrous. For example, let us assume that the software processes two values, \( v_1 \oplus m \) and \( v_2 \oplus m \), where \( v_1 \) and \( v_2 \) are two internal values that need to be kept secret, and \( m \) is a random mask. Leaking either \( v_1 \oplus m \) or \( v_2 \oplus m \) provides the attacker with no information, because these values are uniformly distributed. However, if the software writes \( v_1 \oplus m \) to a register that contains \( v_2 \oplus m \), the power consumption will correlate with \( (v_1 \oplus m) \oplus (v_2 \oplus m) = v_1 \oplus v_2 \), leaking information on the relationship between the two unmasked values.

One approach for fixing this issue is to go to higher order masking [53], where multiple masks are applied such that secret values cannot be reconstructed with less than \( n > 1 \) intermediate values. Specifically, Balasch et al. [6] demonstrate that unintended interactions due to transitions half the number of intermediate values the adversary needs to acquire. However, algorithms that use high-order masking are significantly more complex than simple first-order masking, and require much more resources and randomness.

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 implementers 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 [23, 67]. 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, indication leakage, does not tell much on the exploitability of the leakage [63]. 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 [73] to differentiate between two sets of measurements, one with fixed inputs and the other with random inputs. If the test finds a sufficiently strong evidence that the measurements 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 [63] extend the \( t \)-test to handle multiple points. In addition, to address leakages distributed over multiple orders they propose the use of the \( \chi^2 \)-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) [41] 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 [28], 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 [56] to simulate the internal circuits of a CPU down to a transistor level Aigner et al. [3]. Its drawback is that transistor-level simulators tend to be very slow. Alternatively, researchers have looked at emulating at the source code level [68] and at machine instruction level [57, 69]. 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 traces from real experiments to make better estimates [50]. Similarly, advanced characteristics of CPUs such as instruction pipelining have found their way into recent leakage emulators [47].

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 two categories, simulation-based and code analysis.

1) Simulation-based Approaches: Veshchikov [68] presents the SILK simulator, which simulates a high level abstraction of the source code of an algorithm that generates traces. Another simulator, MAPS [47] 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 generations of traces. Hence, they are basically assisting the leakage evaluation process in terms of speeding 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 [71] 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 the works that, although sacrificing generality, address the problem of “fixing” the leakage from a specific device. Papagiannopoulos and Veshchikov [57] performed an in-depth investigation of device specific effects that violate the independent leakage assumption (ILA) [61]. 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. [75]. 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. [72] 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 [32] 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.

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, e.g. when overwriting one value with another where both values are masked with the same mask, can break the independent leakage assumption [61] and leak secret information through a physical side channel, such as the power channel. Thus, to fix such leaks, 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 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 [50], 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.

Note that our approach makes use of a leakage emulator. Prior static-analysis-based solutions, such as [47, 57, 68, 71, 75], rely on tags that identify the nature of values within the program. For example, in ASCOLD [57] 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 [50] 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 19 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 evaluation board which features an ARM Cortex-M0 based STM32F030R8T6 System-on-Chip (SoC).

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. [50]. 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.

We use a PicoScope 6404D with a Pico Technology TA046 differential probe connected to the oscilloscope via a Langer

---


4ELMO also provides a model for the Cortex-M4-based STM32F4 Discovery board, which we do not use in this work.
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.

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.

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 [63]. 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. In order to align the traces, we use the eShard SCAred library.

To detect leakage, we employ non-specific TVLA. That is, we check the distribution of the values at each trace point (after alignment) 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. [50], 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 five movs instructions ensures that the str instruction is completed by the time the eors instruction enters the pipeline.

```plaintext
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 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 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 \texttt{str} dominates \texttt{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. \texttt{str r1, [r2]}
2. \texttt{mova r7, r7}
3. \texttt{mova r7, r7}
4. \texttt{eors r3, r7}
5. \texttt{mova r7, r7}
6. \texttt{mova r7, r7}
7. \texttt{mova r7, r7}
8. \texttt{mova r7, r7}
9. \texttt{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 [36] 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.

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 [57], 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.

<table>
<thead>
<tr>
<th>Instruction Pair</th>
<th>Storage Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>str r1, [r2]</strong></td>
<td>str, eors, movs, adds, bics, ldsr, orrs, lsls, strb, strh, strm, strp, push, pop</td>
</tr>
<tr>
<td><strong>mova r7, r7</strong></td>
<td>movs, adds, subs, strb, strh, strm, strp, push, pop</td>
</tr>
</tbody>
</table>

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 content 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.

1. \texttt{mova r3, 0x303}
2. \texttt{mova r4, 0x402}
3. \texttt{mova r7, r7}
4. \texttt{mova r7, r7}
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.

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 [34, 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.

Listing 3: Example of word interaction

```
 5  movs r7, r7
 6  strb r5, [r3]
 7  movs r7, r7
 8  movs r7, r7
 9  movs r7, r7
10  ldrb r6, [r4]
11  movs r7, r7
12  movs r7, r7
13  movs r7, r7
Listing 3: Example of word interaction
```

Listing 4: Store latch example.

```
 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.
```

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

Figure 5: Leakage from str to eors.

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 needs more information than the original ELMO provides. Specifically, when leakage is identified, ROSITA needs to know the cause of the leakage.

To find the cause of the leakage, 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.\(^6\) 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.

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.

B. Operand Interaction

One of the common forms of unintended interaction is between the operands of successive instructions. Technically, as McCann et al. [50] 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 \(\text{movs } r7, r7\). 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.

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 \(\text{ror } r2, r2\) would set the value of \(r2\) to \((b_2 \oplus m)[(b_1 \oplus m)][(b_4 \oplus m)](b_1 \oplus m)\). 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 \(\text{rors } r2\), \(r3\) instruction that ELMO* indicates is leaking.

\[
\begin{align*}
\text{rors } r2, r3 &\quad \text{eors } r2, r7 \\
\text{rors } r2, r3 &\quad \text{rors } r7, r3 \\
\text{eors } r2, r7
\end{align*}
\]

Figure 6: Masking rotation operations. The leaking \(\text{ror}\) 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.

\(^6\)Implementation note: to reduce memory usage, we calculate the t-test values incrementally, using Welford’s algorithm [74].
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 9. The code uses two registers chosen to not conflict with the store, r0 and r6 in the example in Figure 9. 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. 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.

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

VI. Evaluation

We evaluate ROSITA with masked implementations of two cryptographic primitives. AES [24] is one of the most commonly used ciphers, having been an international standard since 2001. We use the byte-masked implementation by Yao et al. [76]. 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 [36], we use different masks for each row to avoid leakage through interactions between bytes in memory words.

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

1https://github.com/Secure-Embedded-Systems/Masked-AES-Implementation/tree/master/Byte-Masked-AES
modern primitive, the second cipher we use is the cryptographic permutation Xoodoo. Xoodoo was proposed recently by Daemen et al. in [25] for use in authenticated encryption modes [26]. The optimized and non-masked implementation of Xoodoo we took from [15] and we implemented 2-share boolean masking scheme of Bertoni et al. [14] ourselves.

A. Fixing Leakage

![Graph](image1)

We first show ROSITA’s success in fixing the leakage it detects. Figure 10a shows the results of a non-specific fixed vs. random experiments with 200,000 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 10b shows, ROSITA detects the leaks and fixes them. This fix does not, however, come for free. The first round now takes 1520 cycles, compared with 1285 for the original implementation—a slowdown of 18%. Figures 11a and 11b show similar results for Xoodoo, with a 25% slowdown (637 vs. 795 cycles).

To determine the trend of leakage, we perform the fixed vs. random test on the hardware with a varying number of traces. Figure 12 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. In contrast to this, our fixed implementation does not show any significant leakage at up to 200,000 traces.

B. Multiple Fixed Inputs

![Graph](image2)

One of the known limitations of TVLA is that it may miss some leakage if used with only one input [64]. 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 Figures 14 and 15 shows the combined results from 1, 5, and 100 fixed inputs for AES and Xoodoo, respectively. As we can observe, when we increase the number of fixed points, the number of locations that show leakage increases.

Running ROSITA with these inputs allows us to fix all of these leakages. Repeating the experiment with the code that ROSITA produces and the same fixed inputs we get the test values in the bottom rows of these figures. We can observe that ROSITA fixes the leakage, at a cost to the performance. Table III compares the performance of the code before and after running ROSITA, showing a maximum overhead of 25%.

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

<table>
<thead>
<tr>
<th>cipher</th>
<th>Original</th>
<th>1 Fixed</th>
<th>5 Fixed</th>
<th>100 Fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>1 285</td>
<td>1 519</td>
<td>1 520</td>
<td>1 526</td>
</tr>
<tr>
<td>Xoodoo</td>
<td>637</td>
<td>765</td>
<td>771</td>
<td>794</td>
</tr>
</tbody>
</table>

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 13 shows this for AES 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 and 98% 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 1. 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.

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, and Xoodoo in 21. In comparison, the code rewrite phase of ROSITA takes around 0.1 seconds.

Collecting the same amount of traces from the hardware takes 117 seconds for AES, and 147 for Xoodoo. We note that although Xoodoo is faster than AES, collection is slower, because we need to provide the masked implementation of Xoodoo much more randomness, and 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 are 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. For this reason 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 [39] and $\chi^2$ [43], have been proven useful as statistical distinguishers. It may also be possible to extend ROSITA to use correlation power analysis (CPA) [17]. 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 and Xoodoo. Our experiments using the actual hardware show the absence of exploitable leakage at only a 25% penalty to the 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 an Australian Research Council Discovery Early Career Researcher Award (project number DE200101577) and by a gift from Intel.

**REFERENCES**


**NEW REFERENCES**

To validate our setup, we reproduce the results of McCann et al. [50]. 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.

Listing 5: ShiftRows from McCann et al. [50]

<table>
<thead>
<tr>
<th>ldr r4, [ r1, #4 ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>rors r4, r5</td>
</tr>
<tr>
<td>str r4, [ r1, #4 ]</td>
</tr>
<tr>
<td>ldr r4, [ r1, #8 ]</td>
</tr>
<tr>
<td>rors r4, r6</td>
</tr>
<tr>
<td>str r4, [ r1, #8 ]</td>
</tr>
<tr>
<td>ldr r4, [ r1, #12 ]</td>
</tr>
<tr>
<td>rors r4, r3</td>
</tr>
<tr>
<td>str r4, [ r1, #12 ]</td>
</tr>
</tbody>
</table>

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.

?? 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 ??, we see that ELMO produces a fairly accurate simulation of the leakage.

In particular, our figure resembles Figure 5 of McCann et al. [50], with only minor differences that reflect the different test environment.