Project ID: 731591, Horizon 2020
http://www.reassure.eu

REASSURE

Deliverable D2.6

White Paper: Security Testing Via Simulators

Editor:	E. Oswald (UNI-KLU)
Deliverable nature:	R
Dissemination level: (Confidentiality)	PU
Delivery date:	March 31, 2020
Version:	1.0
Total number of pages:	25
Keywords:	evaluation, automation, profiling, side-channels

Horizon 2020
European Union funding
for Research & Innovation

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 731591.

Executive summary

This white paper guides users of the trace simulator (ELMO) released by the REASSURE consortium in the particular task of security testing code during the design phase. It is intended for software developers who need to write or integrate cryptography on ARM CORTEX-M devices, in particular the M0. We assume familiarity with symmetric encryption (AES), an awareness of side channel analysis (in particular power analysis and masking as a countermeasure), and ARM Thumb assembly.

We begin by explaining the different options and challenges for leakage simulation. Broadly speaking, there are two key aspects to simulation: the accurate tracking of data flow (at some appropriate level) and the mapping of the flow to some meaningful prediction of the power consumption (or other side-channel). Unlike most of the other existing tools, which focus more on one or other of these ends, ELMO (as we describe) makes a combined effort towards both, and was therefore chosen as the starting point for development within the project.

We next show how to set up and configure ELMO – for simulation generally, and in the context of 'fixed-versus-random' leakage detection. We provide some non-technical intuition for choosing the desired error rates of the tests and setting an appropriate sample size.

Finally, we give some case-study examples that show ELMO's leakage detection capabilities in practice and illustrate the types of problems it is able to flag up. We suggest ways to address the particular problems in question, thus demonstrating a workflow of testing, adjusting, and retesting that we recommend to developers in the code development phase of the design process.

The appendix of this whitepaper listes and reviews a number of existing leakage simulators.

List of authors

Company	Author
Riscure	Valentina Banciu
UNI-KLU	S. Gao
UNI-KLU	E. Oswald
Riscure	Christos Tzotzadinis
UNIVBRIS	C. Whitnall

Revision history

Revision number	Date	Comment
1.0	March 2020	Final Public Version

Contents

1 Introduction

This white paper is a deliverable of the Horizon 2020 project ‘REASSURE: Robust and Efficient Approaches to Evaluating Side Channel and Fault Attack Resilience’. It is aimed at equipping IoT software engineers with the knowledge to utilise REASSURE tools to test and improve the security of cryptographic implementations.

Cryptography is an essential feature of many products and applications in the IoT domain, which is vast in terms of implementation options. For those products that are on the ‘low end’, i.e. that have comparatively small processors (the ARM Cortex-M family, for example), implementations of cryptography are particularly vulnerable to power analysis attacks.

REASSURE developed a leakage simulator, ELMO, which enables a developer to run their code and find potential side channel leaks, without having to invest in an expensive lab setup. ELMO can also help to fix leaks, such that, step by step, developers can improve their code and make it leakage free.

This white paper explains how ELMO works (and some of its limitations), and then demonstrates how to use ELMO to find leaks in a masked AES implementation. It is aimed at IoT software developers who have a basic understanding of side channel attacks (that is, how a power analysis attack works in principle), and are working with ARM Cortex-M processors.

1.1 Structure of this white paper

We first overview the different options for simulation and survey existing simulators, before introducing ELMO. We describe its particular capabilities and limitations, before explaining how to compile and operate it, and especially how to configure it for the task of leakage detection. (Detailed documentation is provided along with the code [18]). We then present some examples of subtle leakages from imperfectly protected implementations that can be detected from ELM0 simulations, and show how to respond to problems once they have been identified.

2 Leakage Simulation Approaches

There are two component parts to any power (or other side channel) simulation: *emulating* what the device is doing internally, and *modelling* the externally observable behaviour of the device in order to accurately map the emulated processes onto predicted consumption. Simulations can be broadly categorised according to the architectural level at which they attempt to characterise the power/energy:

Transistor level simulation. Given sufficient information about the technology that a chip will be built in, a circuit can be mapped to a network of transistors, the power consumptions of which are modelled via known differential equations. Most modern commercial tools derive from an open source analogue electronic circuit simulator called SPICE (Simulation Program with Integrated Circuit Emphasis) [20]. SPICE takes a back annotated netlist of transistors as input, where back annotation refers to the process of “augmenting” a netlist with more accurate information, such as delays of various types. This approach accurately describes the aggregated power behaviour of the individual transistors, but is unable to take into full account more complex differential effects, occurring (for example) due to the placement of components on a circuit.

Gate level simulation. Gate level simulations are also based on (back annotated) netlists. For the simulation, the number of transitions in each gate is counted and weighted, according to the information in the netlist. The sum over all weighted transitions is then an approximation for the instantaneous power of the circuit. If there is no information about how to “translate” a transition to the power consumption, then one can simply count the number of transitions. This is often referred to as a “toggle count model”.

Behavioural level simulation. On this next level up, there is no information about the placement of circuit elements and the routing of signals between them. One only has access to the behavioural description of the components – for example, in the form of low level machine code or micro code, instruction/assembly code, or higher-level code (such as C). Developing accurate models on this level requires access to actual devices (and a laboratory setup) from which the average power of different (sequences of) instructions is estimated. They are well suited for small (and thus low complexity) devices, on which assembly instructions map directly to machine instructions without further decoding into micro instructions.

In the context of side channel analysis researchers recognised the value of simulations early on and many papers produce results based on simulated traces. Most fall into the 'behavioural level' category, but differ from more general purpose methods of this type as these typically *average over data inputs*, whereas it is precisely the *data-dependent variation* in power consumption that constitutes side-channel 'leakage'.

Because there are two components to simulation, there are two opportunities to 'get it wrong'. The tendency of previous work has been to focus attention on *either* emulation *or* modelling and to neglect the other, to the detriment of the overall achievement.

ELMO, in contrast, is based on an architecture-specific emulator with empirically estimated models for the data-dependent power consumption.

An overview of other leakage simulators developed and used within industry and academia can be found in Appendix A.

3 ELMO

ELMO has been designed with the following technical requirements in mind:

1. it must accept inputs (i.e. source code) that are suitably close to what is actually executed on the target platform;
2. it must include some standard analysis techniques that require minimal user interaction;
3. it must be capable of relating identified leaks to the inputs (i.e. the source code);
4. it must be modular (different emulators, leakage models, trace formats, etc).

Detailed explanations about our design process, alongside precise information for all aspects that we discuss in the following, can be found in [19].

3.1 Emulation

There are a number of device emulators available (both commercial as well as open source); the original and current versions of ELMO are supported by Thumbulator, an open source instruction set emulator for the ARM M0 [27]. Like other medium complexity processors, the M0 translates assembly code more or less directly into machine instructions. Thus it is possible, given arbitrary sequences of assembly instructions as input, to reproduce the behaviour (in particular, the data flow) of the core microprocessor with reasonable accuracy. The Thumbulator, as an instruction level emulator, is naturally agnostic to peripherals and in particular the memory subsystem.

Leakage from the memory subsystem is however relevant in the context of implementing cryptography with countermeasures against side channel leakage. Therefore we added a simplified model that represents the implementation of the bus architecture connecting the memory with the CPU as follows: we conjecture (based on our lab experiments) that there are two 32-bit buses, one for read and one for write, and that the bus value (or connected buffer) only changes when there is a new write/read operation happening. These buses are represented in our model by two 32-bit variables, representing the current state on the read and write buses respectively. Only read/write instructions update these two variables: for any other instruction, the values on the buses are preserved.

As we shall see in the next section, this add-on enables us to incorporate the memory subsystem into the power model. However, we stress that it is a targeted fix based on observations particular to a single manufacturer's product; it should not be presumed to be generally applicable, nor to cover other possible fluctuations across boards.

3.2 Modelling

The second component task of trace simulation is to map the instruction-level data flow generated by the emulator to power consumption predictions. An aim of ELMO, and of REASSURE, is to improve on the standard models resorted to by many previous simulators (for example, Hamming weight or Hamming distance assumptions) with custom-built models that take potentially complex data-dependencies into account, including those associated with instruction pipelining or interactions between adjacent wires in a circuit.

The models integrated into ELMO are built according to a 'grey box' approach – that is, they do not rely on detailed knowledge of the implementation of the architecture (which, as is the case for Cortex-M devices, is usually not easy to obtain) but they do take advantage of basic features that are publicly available. For example, Figure 1, redrawn from [11], depicts the architectural components of an ARM M0 core in simplified form. The CPU comprises an arithmetic-logic unit (ALU), a hardware multiplier, and a (barrel) shifter. Two buses feed from the register banks into the ALU, one of which also connects to some data in/out registers, while a third connects the ALU output back to the register banks. ELMO provides models for 23 Thumb instructions, which were selected because they feature in implementations of symmetric cryptography, which is most likely to feature on low end devices such as the Cortex M0.

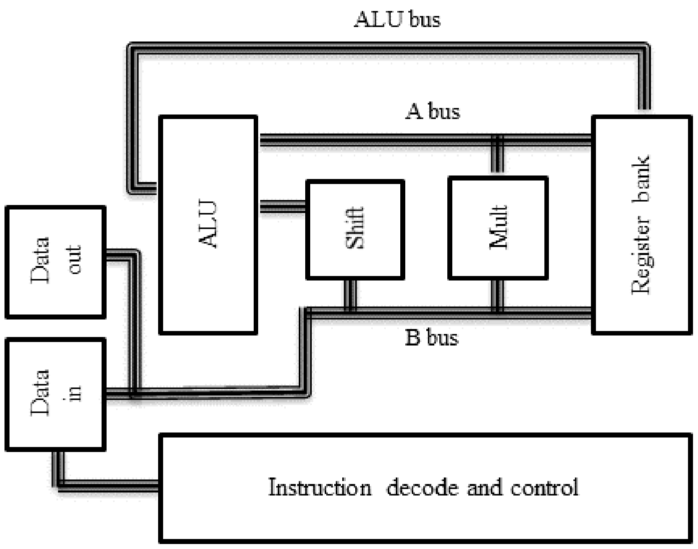


Figure 1: Simplified ARM CPU architecture (redrawn from [11]; re-used from [19] with permission) for a 3-stage pipeline architecture.

The ELMO models were built, tested and modified in an incremental manner using the statistical techniques of linear regression and the F-test, which we now (very) briefly review:

Linear regression is a method for learning relationships between observable processes from a data sample. A model is written down describing one **dependent variable** as a function of several **explanatory variables** with unknown coefficients. The coefficients are then estimated from the dataset, for example by choosing values that minimise the sum of squares of the differences between the raw data and the model predictions. It is important to bear in mind that the term linear refers to the fitted coefficients rather than the nature of the explanatory variables. In particular, ELMO models have several second order terms included.

The F-test is a 'statistical hypothesis test' which can be used to decide whether (subsets of) the explanatory variables in the model equation really *do* influence the dependent variable on the left hand side enough to be worth including. This gives us a tool to incrementally adapt our power consumption models whilst keeping estimation costs to a minimum (noting that every additional explanatory variable increases the number of observations needed for precise parameter estimation).

Model building always entails a trade between *model complexity*: the inclusion of as many explanatory variables as possible, in order to predict the dependent variable with sensitivity; and *data efficiency*: minimising the amount of data needed to get precise parameter estimates, which increases with model complexity. The priority for ELMO is to efficiently capture as much meaningful *data dependent* variation as possible, since this is what constitutes side-channel 'leakage'. We therefore focus on explanatory variables that relate directly to, or that interact with, the data inputs, and look for ways to sensibly reduce redundant complexity. We confirmed through a cluster analysis of model coefficients that the 23 Thumb instructions most essential for cryptographic applications fall naturally into 5 categories, corresponding to the separate components of the core as depicted in Figure 1, each of which can be represented in the models by a single instruction: ALU operations (represented by `eors`), shifts (`lsls`), loads (`ldr`), stores (`str`) and multiply (`muls`). Reducing the number of instructions to profile makes it feasible to collect traces for every possible sequence of three, and to incorporate the impact of previous and subsequent instructions on the target instruction. Model building thus proceeds as follows:

1. The power consumption of the device is measured as different sequences of assembly code are executed on randomly generated input data.
2. The clock cycle corresponding to the target instruction is identified by inspection and compressed to a univariate dependent variable (we choose the peak; another option might be to sum all the points in the cycle).
3. Candidate explanatory variables are derived, including the input bits, the transitions between consecutive inputs, the previous and subsequent instructions in the sequence, and various interactions between them.
4. For each of the 5 representative instructions, linear regression models are iteratively built by adding groups of variables (guided by intuition about the architecture), each time retaining or discarding them according to the outcome of an F-test for joint significance.
5. For the NXP board, the models are subsequently extended to incorporate the memory subsystem via the Hamming distance between the current and previous states on each bus, as tracked in the augmented data-flow described in Section 3.1 above.

Note that ELMO is modular with respect to the model: users may swap the model coefficients for a new set estimated from a board of their choice, provided they correspond to a format interpretable by ELMO.

It is important to understand the limitations of the existing ELMO models:

- They are based on measurements which reflect the quality of our lab setup, and signal processing capabilities.
- They account for data-dependent activity of the CPU core for the selected (symmetric cryptography relevant) subset of Thumb instructions only.
- The current workaround for the memory subsystem is effective for the M0 implementations that we utilise but may not translate across different manufacturers.

- They produce idealised predictions, free from electronic or environmental noise.
- They do not aim at a direct numerical estimation of the total power consumption, rather a *proportional* approximation of the *data-dependent* power consumption. Thus they at least preserve the *presence* of the relevant effects, which can be detected via leakage detection tests.
- They cannot be analysed to obtain the magnitude of an effect, the 'true' correlation coefficients, nor the sample size required for an attack. This is in part due to the previous item, but is also a consequence of a) the difference in data complexity between (deliberately generic) detection and (more targeted) attack strategies and b) the nature of statistical hypothesis testing, which aims to provide defensible *decisions* ('evidence of leakage' versus 'no evidence of leakage') rather than directly interpretable test statistics.

4 Getting Started With ELMO

ELMO is maintained at `github.com/sca-research/ELMO`. The repository contains a pre-compiled binary (for Linux), as well as the source files, documentation, examples, etc.

ELMO itself does not require any special libraries (everything is integrated) and it should compile with any standard compiler. There even is a pre-complied binary program for ELMO in the repository, which should work for many Linux-based systems. Because ELMO is dedicated for ARM M0 processors, it requires an ARM binary as input and therefore we an ARM compiler. We have tested ELMO on Linux and Mac based systems.

4.1 C Compiler

To compile ELMO, one needs the GCC compiler collection (tested version 7.3.0 on Ubuntu) and the *make* command :

- *Ubuntu*: Install GCC and *make* at the same time with "*sudo apt install build-essential*".
- *Mac*: Download and install "*Command Line Tools for Xcode*", which is available on Apple's developer page. After installation is completed, run "*gcc -v*" in a terminal to check that you can see the correct version information.

When the compiler toolchain is ready, enter the ELMO project directory and simply type "*make*" at the command line. Once the process has completed, a binary program with the name "*elmo*" should appear in your directory.

4.1.1 ARM Toolchain

In order for a user to compile his/her own code, a compiler that can output ARM code is required (we have only tested the GNU ARM Embedded Toolchain). The main tested version is *arm-none-eabi-gcc v7.3.1 20180622*, although we have found that a few other versions work for ELMO as well. The GNU ARM Embedded Toolchain can be downloaded from [4]. Follow ARM's installation guide:

- *Ubuntu*: Unpack the tarball to the install directory: *$ cd $install_dir && tar xjf gcc-arm-none-eabi–yyyymmdd-linux.tar.bz*; then invoke ARM toolchain as *$ export PATH=$PATH:$install_dir/gcc-arm-none-eabi-/bin; $ arm-none-eabi-gcc* (an older version is available through *apt-get*).
- *Mac*: Unpack the tarball to the install directory: *$ cd $install_dir && tar xjf gcc-arm-none-eabi–yyyymmdd-mac.tar.bz2*; then invoke ARM toolchain as *$ export PATH=$PATH:$install_dir/gcc-arm-none-eabi-/bin; $ arm-none-eabi-gcc*

4.2 Setting up ELMO

As a stand-alone tool written in C, ELMO should be able to run on various platforms, as long as all dependent libraries are correctly included. ELMO's behaviour can be altered based on a number of macros in "*elmodefines.h*". Details of all available macros can be found in "*ELMODocumentation.pdf*" on our Github repository [18]. Of course, this step is completely optional: if the users' goal is merely to generate traces without tuning the configuration, the default settings are a safe fall-back.

4.2.1 ELMO Macros

We briefly explain the two most relevant macros:

```
#define FIXEDVSRANDOM
```

ELMO will perform a fixed vs. random test at the end of trace generation. The first half of the generated traces will always be treated as the fixed input group, whereas the second half is taken to be the random group. Unless this macro is set, ELMO will not do the test (this is the default setting).

```
#define CYCLEACCURATE 1
```

ELMO will generate traces that are cycle-accurate (i.e. multiple copies of the predicted power consumption per target instruction in the case of multi-cycle instructions); otherwise, the generated traces always contain one sample for each instruction.

4.3 Compiling Your Own Design with ELMO

As ELMO takes binary code as its input, before any simulation, users must (cross-)compile their own source code to an ARM binary program.

As ELMO is a "substitute" measurement setup, it has to be instructed "what" to measure. For this purpose, there are a few library functions that need to be included into the code that is analysed, which tell ELMO when to start and when to stop "measuring power".

4.3.1 Code Framework

Figure 2 demonstrates how this is done in the MbedAES [5] implementation. Readers can find this project in ELMO's repository *"Examples/DPATraces/MBedAES"* [18]. Lines 9/11 mark the beginning/end of the simulation trace with the ELMO library function *starttrigger()* and *endtrigger()*. The header file for these functions must be included as

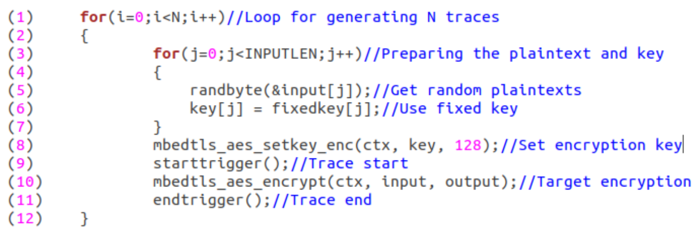
```
#include "${PATH_TO_HEADER}/elmoasmfunctionsdef.h"
```

For most users, it would be sufficient to modify line 10 to call their own code.

4.4 Makefiles

We recommend taking one of the example projects on ELMO's Github repository [18] as a starting point and revising as appropriate. In the case that the code hierarchy is more complicated, users may need to write their own *Makefile*. This should follow exactly the same rules as the supplied *Makefiles*, bearing some points in mind:

- *Compiler.* This needs to be ARM-GCC rather than general GCC.



```
(1)     for(i=0;i<N;i++)//Loop for generating N traces
(2)     {
(3)             for(j=0;j<INPUTLEN;j++)//Preparing the plaintext and key
(4)             {
(5)                 randbyte(&input[j]);//Get random plaintexts
(6)                 key[j] = fixedkey[j];//Use fixed key
(7)             }
(8)             mbedtls_aes_setkey_enc(ctx, key, 128);//Set encryption key
(9)             starttrigger();//Trace start
(10)            mbedtls_aes_encrypt(ctx, input, output);//Target encryption
(11)            endtrigger();//Trace end
(12)    }
```

Figure 2: Code framework for trace generation: change only the target encryption part (line 10).

- *ELMO library related.* The so-called ELMO library functions have their header in *elmoasmfunctions.h* and binary code in *elmoasmfunctions.o*. The make system must be able to find those files.

- *Device description file.* A **.ld* file describes the target device. This is part of the standard ARM development flow: it could exist in the project directory or wherever as stated in the *Makefile*. In most cases, it does not affect ELMO, although one has to make sure that the RAM/ROM is large enough to host the target encryption. Simply copying this file from the ELMO example project and renaming it would work in most cases.

- *A startup file.* An assembler startup file for the target device: this is also part of the standard development flow. As ELMO is simply a simulator, our "*vector.o*" in the example is good enough. This file could be in the project directory or wherever as stated in the *Makefile*. Copying this file to your own project is sufficient in most cases.

4.5 Finally: Generating Traces With ELMO

Having generated the binary code, the subsequent simulation is rather trivial: if the number of traces is defined within the ARM source code, the command

```
./elmo ${PATH_TO_BINARY}/${BINARY}.bin
```

will start the trace generation as follows:

```
GENERATING TRACES...
TRACE NO: 0000000001
TRACE NO: 0000000002
TRACE NO: 0000000003
...
SUMMARY:
cycle accurate model
instructions/cylces 2274
```

Alternatively, one can also set the total number of traces in the command line arguments. In order to do so, in the ARM source code, the number of traces N must be loaded from the ELMO library function:

```
LoadN(&N)
```

Then ELMO can be called as follows:

```
./elmo ${PATH_TO_BINARY}/${BINARY}.bin -NTrace ${NUM_OF_TRACES}
```

By the end of each simulation, all traces will be printed to separate files in *output/traces/*, while the directory *output/asmoutput* contains the disassembly code for at least one of the traces. If a leakage detection procedure is selected, the t-statistic trace is saved to *output/fixedvsrandomtstatistics.txt*.

5 Configuring Leakage Detection

Whilst ELMO could be used to only generate power traces (which can be analysed using other tools), we supply a built-in a generic leakage detection facility that enables developers to simply identify leaks in their code.

Generic leakage detection differs from a leakage attack in that it will show many leaks (i.e. any instruction that operates on data which leaks information about secret key material), whereas a specific leakage attack will only show leaks that correspond to that particular attack vector. Generic leakage detection thus potentially provides better code coverage (in comparison to specific leakage attacks), but it requires more leakage traces (in comparison to leakage attacks).

ELMO's leakage detection facility is based on statistical hypothesis testing of the type taken up in the side-channel community for the purposes of (real) device evaluation. Unlike evaluations performed on real trace measurements, where locating the source of the leakage within the algorithm relies on guesswork and trial-and-error, simulation-based evaluations have the advantage that measurement points (and thus detected vulnerabilities) are more identifiably tied to their instructions of origin.

The in-built procedure derives from the so-called 'fixed-versus-random' test introduced by Cryptography, Inc. as part of their widely-used Test Vector Leakage Assessment (TVLA) framework [14]. The intuition behind the tests is that, for a physically secure implementation, traces generated (in our case, simulated) by repeated operation on fixed data inputs should 'look like' traces generated (simulated) during the same operations on random inputs. Therefore, if a statistical test (TVLA uses Welch's t-test) concludes that there is a difference between the two sets of traces at one or more point in time, the implementation should be considered vulnerable.

5.1 A Cautionary Note

Research carried out within REASSURE uncovered limitations of the TVLA framework and (more specifically) of a public ISO standard adapted from it (ISO/IEC 17825:2016 [15]). Subsequent work published by Whitnall et al. at Asiacrypt 2019 [28] addresses the question of how best to configure tests in order to reliably achieve the related but distinct goals of certifying vulnerability (demonstrating convincingly the presence of a leak) and certifying security (demonstrating convincingly the absence of a leak). In both cases responsible reporting is also called for. The following recommendations on how to configure ELMO for leakage detection are consistent with the findings of these works for the general case where the user has no *a priori* knowledge about the expected form and density of the leakage, though a user who is willing and able to make some such assumptions may be interested in improving upon the configuration as guided by [28].

5.2 Configuring Sound Leakage Detection

The reason test configuration matters is that it impacts on the reliability of the conclusions. The aim of ELMO's fixed-versus-random procedure is to detect arbitrary leaks: we therefore want to keep the rate of false negatives low. But we also want to be confident that the leaks we *do* find are really there, that is, we want to avoid false positives too. The relevant parameters can be understood as follows:

- α controls the per-test false positive rate – the probability of concluding that there is a leak when there isn't. We fix this within ELMO (with the general case in mind) to the level implied by the TVLA specification, which is $\alpha = 0.00001$. (Choosing it to be very small ensures that the overall false positive rate is still reasonable; more sophisticated methods of controlling the overall rate exist but are scenario dependent [28]).
- d is called the standardised effect size and describes the types of vulnerability we want the test to be able to detect. ELMO fixes this to $d = 0.2$ which is 'small' according to a widely-recognised categorisation by Cohen [6] and is consistent with the standardised size of many of the effects that we observed for real M0 devices. However, we also found effects as small as $d \approx 0.04$ and recognise that these may pose a

risk in the presence of particularly well-resourced adversaries. Evaluators concerned with such a threat may select a smaller value for d by editing the final row of the *coeffs.txt* file provided as input to ELMO.

- The desired per-test false negative rate, β, is for the user to decide. Two options might be $\beta = \alpha$, to represent the case where both types of errors are equally concerning, or $\beta = 0.05$, to represent the case where false positives are more of a concern than false negatives. (As the value of d decreases such a trade-off may become necessary). β is not an input into ELMO but rather is used to determine the sample size, as follows...

- The sample size N is set by the user and given as input to ELMO (the acquisition step must be set-up to collect $N/2$ traces according to the fixed and $N/2$ according to the random data inputs). N can be chosen by a known formula[1] in such a way as to achieve the desired β for the fixed values of α and d. For $d = 0.2$ and $\alpha = 0.00001$ as above, a sample size of 8,000 (4,000 fixed and 4,000 random) will achieve balanced error rates so that β is no more than α, while a sample size of half that will keep β below 0.05. Guidance for some alternative choices of d can be found in Table 1. Note that the maximum sample size ELMO can handle depends on the length of the code sequence and the memory resources of a user's PC; we have performed successful experiments with sample sizes of 100k but users interested in very small effect sizes may sometimes need to settle for a higher rate of false negatives than positives (see for example the bottom row of the table).

Standardised effect size	$\beta = \alpha = 0.00001$	$\beta = 0.05$	$\beta = 0.2$
$d = 0.2$ (Cohen's 'small')	8,000	4,000	3,000
$d = 0.1$	31,000	15,000	12,000
$d = 0.04$ (Smallest observed)	189,000	92,000	70,000

Table 1: Suitable sample sizes (rounded up to the nearest 1,000) to achieve desired false negative rates for a given standardised effect size and a TVLA-inspired false positive rate of 0.00001.

ELMO does not enforce an appropriate choice of N, but it does compute the actual associated β and report it as part of the test outcome. The user should pay attention to this information, at the very least including it in the summary of their findings and (if it is lower than desired) ideally repeating the test procedure with a more adequate sample size.

6 Getting Hands On

We now illustrate how to find and fix some leaks in software implementations of the Advanced Encryption Standard (AES), which is a likely use case. We start with a simple first-order masking scheme implemented in C, which may be the "default" starting point for many developers. In principle, such a scheme should already (provably) prevent the most straightforward form of power analysis (Differential Power Analysis, DPA), but as we will demonstrate, this is rarely true in practice.

6.1 Byte-wise First Order Masked AES

Let us analyse a first order byte-wise masking scheme applied to AES. A very simple version of such a scheme is provided via an open source implementation suitable for our target platform by the Secure Embedded Systems Research group at Virginia Tech [29]: the whole scheme is written in C, which forms a fairly comprehensible starting point. The ELMO-adapted version can be found on the ELMO Github repository (*"Examples/Whitepaper_examples/Bytewise_MaskedAES_C/Original"*) [18].

[1]Precisely, $N = 4 \cdot \frac{(z_{\alpha/2}+z_\beta)^2}{d^2}$, where z_γ is the critical value of the standard normal distribution at significance level γ (that is, the probability of an observation greater than z_γ is γ). See the Asiacrypt 2019 paper resulting from this project for derivation and further details [28].

6.1.1 Code hierarchy

To set up the code to perform a leakage detection experiment for this implementation, we need to ensure that the correct ELMO function calls are in the right place. The code hierarchy of the analysed implementation is rather simple: the whole scheme is written in one *.c* file (*byte_mask_aes.c*) with just a single header file (*byte_mask_aes.h*) stating the function definitions. The main encryption function is defined as

```
void aes128(uint8_t* state)
```

where *state* contains the AES-128 input plaintext. The key expansion should be called beforehand, as

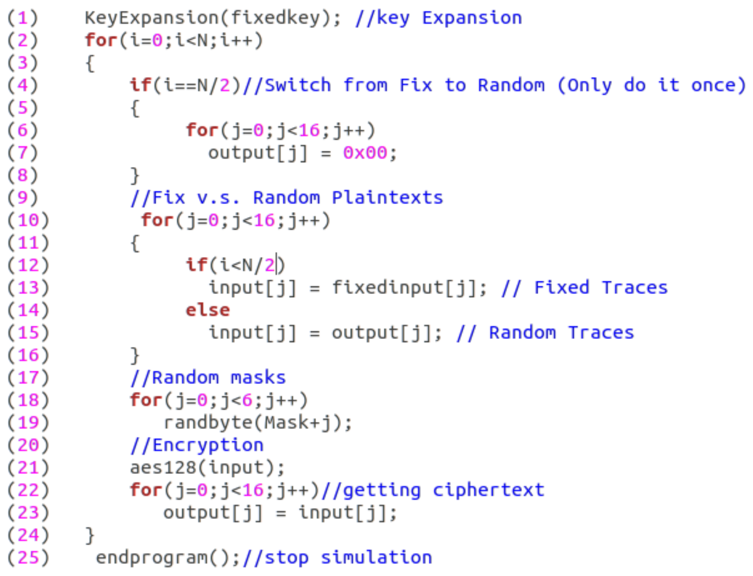
```
void KeyExpansion(uint8_t* Key)
```

The random masks (which are required for the masking scheme) are defined as a 10-byte global array:

```
uint8_t Mask[10];
```

In the following, we denote these masks as m_0 to m_9. The first six masks $(m_0, ..., m_5)$ are generated at random, and the other four masks are derived from $m_0, ...m_3$ via the MixColumns transformation: $(m_6, m_7, m_8, m_9) = MixColumn(m_0, m_1, m_2, m_3)$. To read up on details about first order masking, we recommend to consult the "DPA book" [17].

Compiling target code As per the discussion in Section 4.2, in order to conduct leakage detection in ELMO, users must first compile this scheme to binary code and insert the ELMO trigger functions. The code framework for this is in Figure 3. Note that ELMO does NOT initialise a fully-automated leakage detection; rather, it simply performs a t-test on the first and second half of the simulated traces. This means users must instruct ELMO, in their own code, to generate suitable fixed/random traces; this is illustrated in line 4–16.

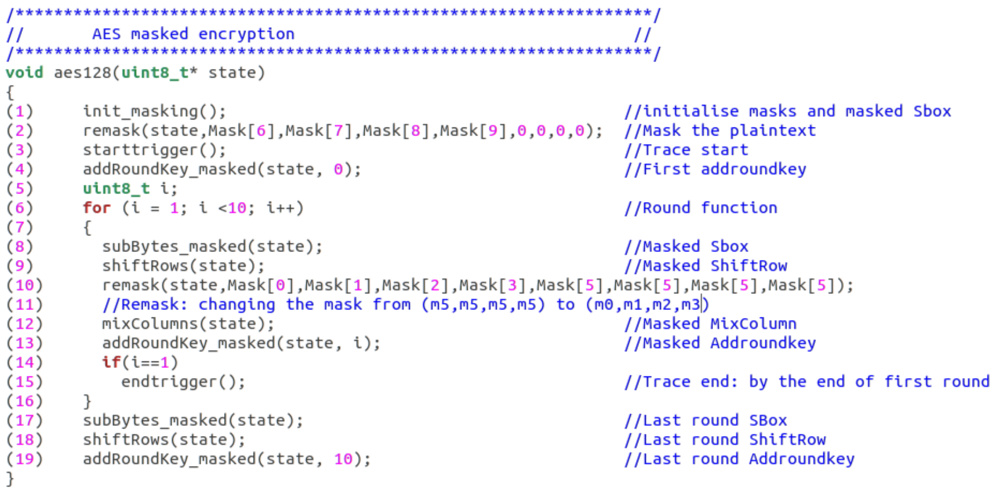


```
(1)    KeyExpansion(fixedkey); //key Expansion
(2)    for(i=0;i<N;i++)
(3)    {
(4)        if(i==N/2)//Switch from Fix to Random (Only do it once)
(5)        {
(6)             for(j=0;j<16;j++)
(7)               output[j] = 0x00;
(8)        }
(9)        //Fix v.s. Random Plaintexts
(10)        for(j=0;j<16;j++)
(11)       {
(12)            if(i<N/2)
(13)              input[j] = fixedinput[j]; // Fixed Traces
(14)            else
(15)              input[j] = output[j]; // Random Traces
(16)       }
(17)       //Random masks
(18)       for(j=0;j<6;j++)
(19)           randbyte(Mask+j);
(20)       //Encryption
(21)       aes128(input);
(22)       for(j=0;j<16;j++)//getting ciphertext
(23)           output[j] = input[j];
(24)   }
(25)    endprogram();//stop simulation
```

Figure 3: Detection framework for the target masked encryption.

To implement masking some random numbers are required, which are stored in the byte array `Mask`. Producing them via native C functions would cause ELMO to emulate also the randomness generation, which is unnecessary. Consequently, ELMO provides a workaround in the form of a library function *randbyte()*, which generates a random byte and writes it to the targeted address, so that the random generator does not have to be emulated.

Readers might notice that the *starttrigger()* and *endtrigger()* calls are not in this part of the code: in order to better locate the leakage, we have moved the start/end point of the trace to the start/end of the first encryption round. Figure 4 shows the detailed implementation of *void aes128(uint8_t* state)*. Because we are only testing the first round, line 15 marks the end of trace that ELMO will record. Despite this, the emulation will keep executing for full 10 rounds. If this is not desirable, then simply the round counter would need to be changed to limit the AES implementation to a single round during testing.



```
/*******************************************************************/
//      AES masked encryption                                     //
/*******************************************************************/
void aes128(uint8_t* state)
{
(1)     init_masking();                                         //initialise masks and masked Sbox
(2)     remask(state,Mask[6],Mask[7],Mask[8],Mask[9],0,0,0,0);  //Mask the plaintext
(3)     starttrigger();                                         //Trace start
(4)     addRoundKey_masked(state, 0);                           //First addroundkey
(5)     uint8_t i;
(6)     for (i = 1; i <10; i++)                                 //Round function
(7)     {
(8)       subBytes_masked(state);                               //Masked Sbox
(9)       shiftRows(state);                                     //Masked ShiftRow
(10)      remask(state,Mask[0],Mask[1],Mask[2],Mask[3],Mask[5],Mask[5],Mask[5],Mask[5]);
(11)      //Remask: changing the mask from (m5,m5,m5,m5) to (m0,m1,m2,m3)
(12)      mixColumns(state);                                    //Masked MixColumn
(13)      addRoundKey_masked(state, i);                         //Masked Addroundkey
(14)      if(i==1)
(15)        endtrigger();                                       //Trace end: by the end of first round
(16)    }
(17)    subBytes_masked(state);                                 //Last round SBox
(18)    shiftRows(state);                                       //Last round ShiftRow
(19)    addRoundKey_masked(state, 10);                          //Last round Addroundkey
}
```

Figure 4: The original implementation from Virginia Tech (with ELMO adaption)

As line 3 and 15 in Figure 4 called the ELMO library functions, the header file for those functions must be included as follows:

```
#include "elmoasmfunctionsdef.h"
```

Since the code hierarchy is rather simple, we can use the existing *Makefile* in ELMO's repository [18] (i.e. we simply use *make* in the respective project folder) to generate a *byte_mask_aes.bin* file. Remember that to perform leakage detection, the macro *FIXEDVSRANDOM* needs to be defined (ensure it is correctly set in *elmodefines.h* and if necessary re-make ELMO).

Running ELMO. When successfully compiled, a binary file *byte_mask_aes.bin* should exist in the targeted directory. Users can now run the following

```
./elmo ${PATH_TO_BINARY}/byte_mask_aes.bin
```

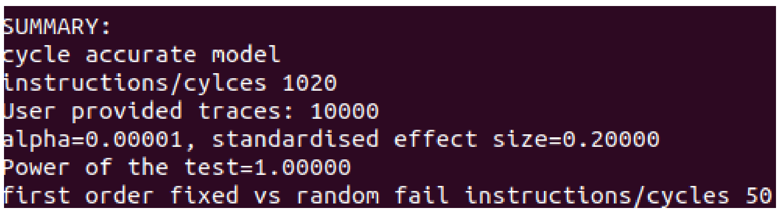
on the command line to start the leakage simulation and associated detection.

6.2 Leakage Analysis

Figure 5 shows ELMO's output. In a cycle accurate model, the first round of encryption takes 1020 cycles to proceed. With 10k traces, if the false positive rate is set to 10^{-5} (close to TVLA's ± 4.5), the leakage detection achieves full statistical power to detect a "small" difference (standardised effect size $d = 0.2$). According to the performed leakage detection, 50 of 1020 cycles (corresponding to a single round of AES) produce significant leakage: this means that leakage with an effect size of 0.2 is identified with near certainty. Such leakage would definitely enable DPA style attacks and is therefore of practical concern.
Figure 6 plots the trace resulting from the leakage detection. Because the simulation was cycle accurate, it is possible to attribute portions of the trace to the respective C code. The trace visualises where leakage occurs: any large peak (a peak that is outside ± 4.5) indicates data dependency. Therefore there appears to be a problem towards the beginning of the SubBytes (S-box) lookups, there appears to be some severe issues with ShiftRows (SR), and there are some issues in the remasking step.

6.2.1 ShiftRows

We shall consider ShiftRows first. This step is a likely cause for problems on the M0 architecture and it has been the demonstrator for ELMO's usefulness already in [19]. The root of the problem is that the ShiftRows



```
SUMMARY:
cycle accurate model
instructions/cylces 1020
User provided traces: 10000
alpha=0.00001, standardised effect size=0.20000
Power of the test=1.00000
first order fixed vs random fail instructions/cycles 50
```

Figure 5: Detection outputs for the tested masked AES.

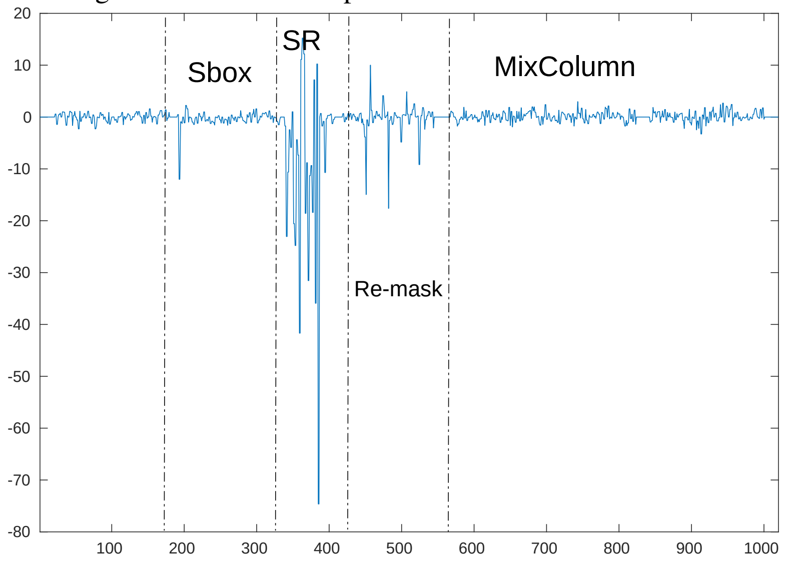


Figure 6: ELMO Leakage Detection, First Round of Masked AES in C

implementation will likely be translated into assembly instructions involving `ROR` (rotate right). This instruction leaks the Hamming distance to its predecessor, which will be masked with the same value. Therefore, this is likely to “strip off” the mask and expose the actual value. A simple injection of instructions with random values can solve this problem, but this is not necessarily straigthforward when working in C. Another option would be to utilise more masks to begin with, such that masks cannot be stripped off by instructions that exhibit Hamming distance leakage.

6.2.2 *remask* function

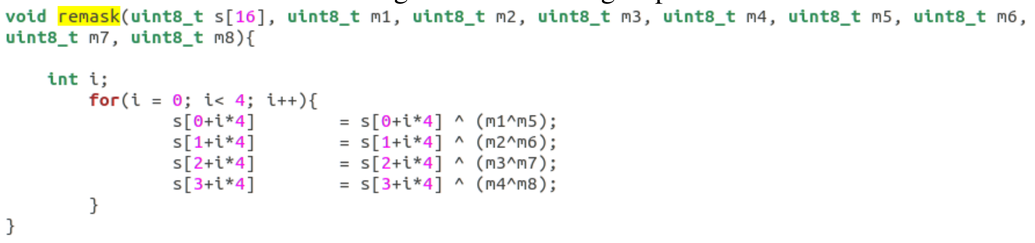
Let us now focus on the remasking step in line 10 of Figure 4. To understand the purpose of this function we briefly review the use of masks to secure the implementation of AES. At the beginning of the encryption, the masking scheme applies a random value (which we call the mask) to each byte of the AES state. Then the implementation has to manage, and in particular maintain, this masked state. This requires to switch masks over from time to time as follows:

- *Sbox*: m_4 and m_5 protect the Sbox input and output respectively. The masked *Sbox* transfers each byte from $x \oplus m_4$ to $S(x) \oplus m_5$.
- *ShiftRow*: Each byte has the same mask m_5, *ShiftRow* does not change the masks.
- *MixColumn*: As pointed out in the DPA book [17], protecting *MixColumn* with a single byte-wise mask is risky. The most common solution is using four masks for *MixColumn* (m_0, m_1, m_2, m_3).

Consequently, to get from the single mask m_5 that protects the state in ShiftRows to the four masks that are needed to protect MixColumns, one needs to remask the state. This is done in the `remask()`, see Figure 7 for a simple graphical representation and Figure 8 for the corresponding program code. In the program code we can see that several input masks have to be specified. In the context of remasking the state in order to have the correct input for MixColumns, the first four input masks would be m_5 and the second four input masks would be (m_0, m_1, m_2, m_3). The remask function then runs through state column by column and changes the mask by an exclusive-or between the old mask (m_5) and the respective new mask (one of (m_0, m_1, m_2, m_3)).

$$\begin{array}{|c|c|c|c|}\hline m_5 & m_5 & m_5 & m_5 \\ \hline m_5 & m_5 & m_5 & m_5 \\ \hline m_5 & m_5 & m_5 & m_5 \\ \hline m_5 & m_5 & m_5 & m_5 \\ \hline \end{array} \xrightarrow{\text{remask()}} \begin{array}{|c|c|c|c|}\hline m_0 & m_0 & m_0 & m_0 \\ \hline m_1 & m_1 & m_1 & m_1 \\ \hline m_2 & m_2 & m_2 & m_2 \\ \hline m_3 & m_3 & m_3 & m_3 \\ \hline \end{array}$$

Figure 7: Remasking step



```
void remask(uint8_t s[16], uint8_t m1, uint8_t m2, uint8_t m3, uint8_t m4, uint8_t m5, uint8_t m6,
uint8_t m7, uint8_t m8){

    int i;
        for(i = 0; i< 4; i++){
                s[0+i*4]        = s[0+i*4] ^ (m1^m5);
                s[1+i*4]        = s[1+i*4] ^ (m2^m6);
                s[2+i*4]        = s[2+i*4] ^ (m3^m7);
                s[3+i*4]        = s[3+i*4] ^ (m4^m8);
        }
}
```

Figure 8: Code of the *remask* function

Because we are working with C code rather than on assembly level, it is now difficult to exactly pinpoint the source of the detected leakage. By mapping the leakage from its precise cycle to the assembly code (which the compiler does produce as well) one can guess that perhaps the moving of the masked state byte followed by loading the same mask may lead to this leakage.

Remasking Pitfalls Before we dig further into the identified issue, we want to draw some attention to the potential pitfalls that there are when working with masks. Some of them are due to a misunderstanding of how remasking works, and others can be due to compiler optimisations.

We begin with a fundamental observation for remasking: it constitutes changing a mask from one random value to another. This implies that remasking must be implemented in such a way that at **no** time the state byte itself is present in the processor in an unmasked form. Therefore it is vital that the "new mask" is added to the state byte **before** the old mask is removed; see Figure 9 for an example of how to get it wrong, and Figure 10 for the corresponding leakage detection result.

Whilst it is possible that such an unmasking happens because of developer error, it is also possible that aggressive optimisation options in the compiler can re-order instructions on assembly level. This is particularly dangerous in the context of masking.

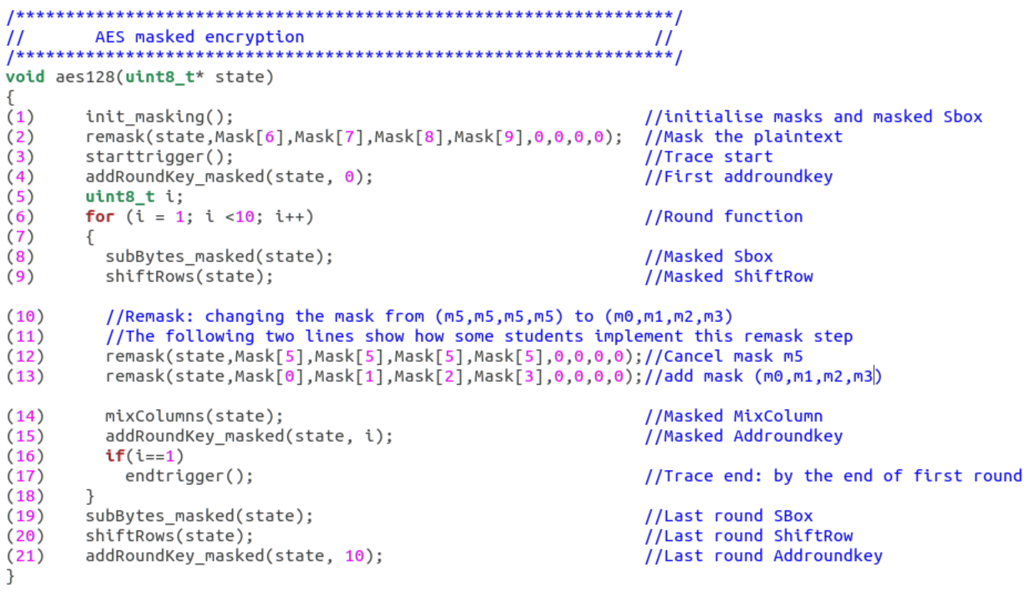
6.3 Leakage Analysis: Assembly Level and Tackling Memory Related Leaks

We argued before that the leakage during ShiftRows is likely due to the use of the ROR instruction. When working with code written in C, we give significant control to the compiler as we allow the compiler to produce the code that actually runs on the microprocessor. Until there are compilers which are side channel aware, serious crypto implementations are better written in Assembly directly.

Therefore, we now continue with our analysis using an ARM assembly implementation of the code that we analysed before; implementation details can be found at [13].

Our simulation up until now used models which were specifically derived to describe the power leakage of the CPU core. However in the case of many implementations, memory transfers are frequently necessary, and the processor core therefor transfers potentially sensitive data over the memory bus.

Based on ARM's AHB/APB protocol [3, 2], ELMO has an extension that provides a very simple model for the leakage of this protocol. It assumes the memory system has two 32-bit separate buses: one for read and one for write. The bus value (or connected buffer) only changes when there is a new read/write operation happening. The bit-flip on the bus produces exploitable leakage, which will be reflected by the power consumption in that



```
/*********************************************************************/
//        AES masked encryption                                     //
/*********************************************************************/
void aes128(uint8_t* state)
{
(1)     init_masking();                                    //initialise masks and masked Sbox
(2)     remask(state,Mask[6],Mask[7],Mask[8],Mask[9],0,0,0,0);  //Mask the plaintext
(3)     starttrigger();                                    //Trace start
(4)     addRoundKey_masked(state, 0);                      //First addroundkey
(5)     uint8_t i;
(6)     for (i = 1; i <10; i++)                            //Round function
(7)     {
(8)       subBytes_masked(state);                          //Masked Sbox
(9)       shiftRows(state);                                //Masked ShiftRow

(10)      //Remask: changing the mask from (m5,m5,m5,m5) to (m0,m1,m2,m3)
(11)      //The following two lines show how some students implement this remask step
(12)      remask(state,Mask[5],Mask[5],Mask[5],Mask[5],0,0,0,0);//Cancel mask m5
(13)      remask(state,Mask[0],Mask[1],Mask[2],Mask[3],0,0,0,0);//add mask (m0,m1,m2,m3)

(14)      mixColumns(state);                               //Masked MixColumn
(15)      addRoundKey_masked(state, i);                    //Masked Addroundkey
(16)      if(i==1)
(17)        endtrigger();                                  //Trace end: by the end of first round
(18)    }
(19)    subBytes_masked(state);                            //Last round SBox
(20)    shiftRows(state);                                  //Last round ShiftRow
(21)    addRoundKey_masked(state, 10);                     //Last round Addroundkey
}
```

Figure 9: How not to remask: line 10-13

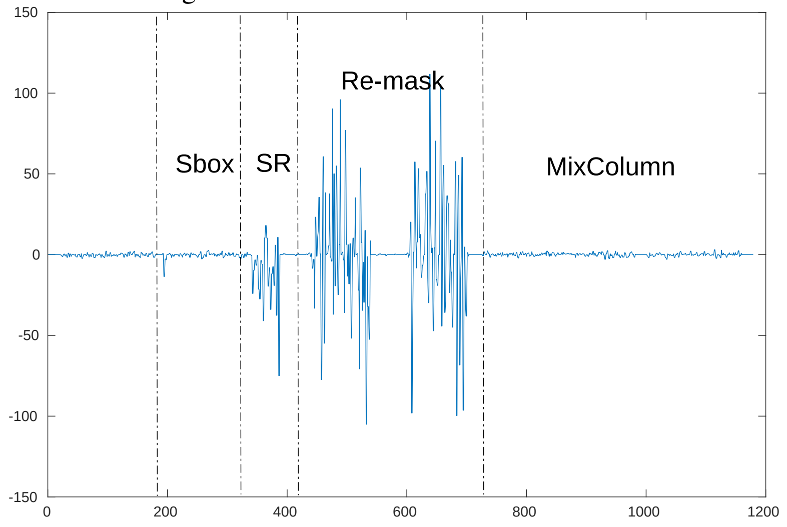
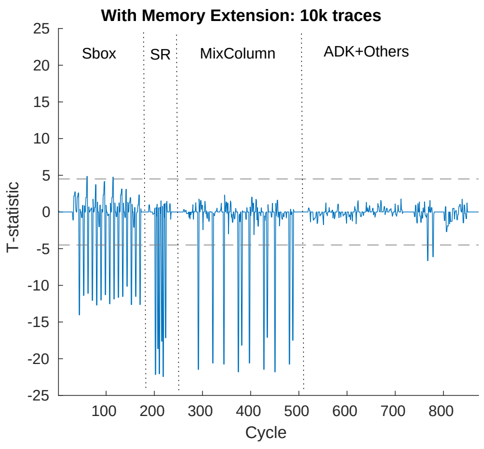


Figure 10: ELMO leakage detection results for Fig. 9



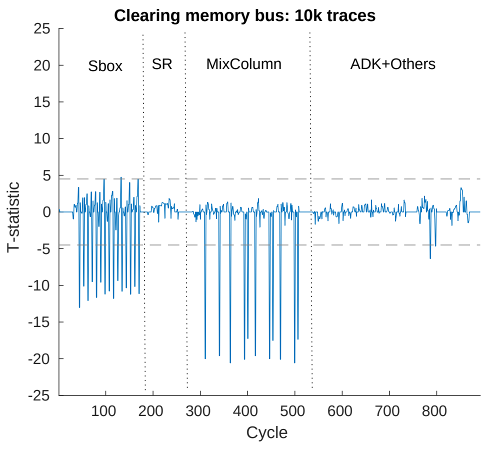


Figure 11: ELMO with the memory extension: first round of ASM-based AES, 10k traces.

specific cycle. Two 32-bit variables have been added to the ELMO data flow, which represent the current state on the read/write buses. Only read/write instructions update those two variables: for any other instruction, the values on the buses are preserved. In each cycle, an additional term is added to the power consumption sample: the Hamming distance between the current and previous states on the read/write bus.

Although this modelling is not necessarily representative for all implementations of the AHB/APB protocol, it represents a minimal base line modelling and we recommend to turn on the memory extension adding the respective definition (`#define MEMORY_EXTENSION`) in *elmodefine.h* and recompiling ELMO.

With the memory extension, ELMO now detects many more leakage samples than before, see the left hand side of Figure 11. As we can see, almost all AES components still show exploitable leakage if the memory system leaks this way. Because we know that all these leakages originate from the memory bus, producing a security patch is quite straightforward.

We look at *ShiftRows* as an example. As we know that the leakage comes from consecutive load and store instructions (the memory transfers), any instruction with an independent (random or constant) value that is placed between loads/stores will clear that memory bus and therefore remove the leakage. The code fragment below illustrates this principle:

```
ldr r4, [ r0, #4 ] //Load a row=x
eors r1, r1 //Remove HD: r1=0
str r1, [ r0, #4 ]  //Clear write bus
ldr r1, [ r0, #4 ]  //Clear read bus
rors r4, r5         //Rotate 8 bits=ROT8(x)
eors r1, r1 //Remove HD: r1=0
str r4, [ r0, #4 ]  //Store the result
```

In this code fragment, the `ldr`/`str` instructions are loading/storing 32-bit word with exactly the same mask (m_5, m_5, m_5, m_5). Thus, any Hamming distance between them would be a leakage. The extra `eors r1, r1` above clears the ALU buses, but not the memory system. Meanwhile, if we execute a (redundant) `ldr`/`str` in-between (the `str r1, [ r0,#4 ]` and `ldr r1, [ r0, #4 ]`), it will clear the read/write bus to zero before next read or write. The improvement in Figure 11 is quite significant: on the right hand, although other parts are still leaking, the *ShiftRow* part is now fine in the simulation.

6.4 Towards a First Order Secure Scheme

Given the difficulty of modelling and dealing with leaks that are due to the memory subsystem, it should be obvious that using the very minimum of masks in a scheme is potentially very dangerous. Hence it is sensible to abandon the attempt to rely on the simple security "patches" that we described, and instead look at low-cost revisions to strengthen the masking scheme itself.

The core idea is to take full advantage of the 4 byte masks required for *MixColumns* and utilise them throughout the entire encryption round. The resulting scheme's description as well as implementation is available on the ELMO Github repository [13]. As both load/store and shift instructions are now protected by different masks, the first order leakage disappears from the ELMO simulations, see left hand of Figure 12. To verify that this implementation does not show leaks when executed on a real device, we ran experiments on an M0 device. The leakage detection results are shown on the right hand of Figure 12. These results are based on one million traces, and $\alpha = 0.00001$. With these parameters we known that it is possible to detect even the smallest observed effect sizes of $d = 0.04$ with power.

Whilst this is reassuring, it still could be that smaller leaks could be present. In addition that at best, this only means that the implementation achieves what the masking scheme claims — "first-order security". This is in fact a rather low guarantee because second order DPA attacks are definitely practical [21].

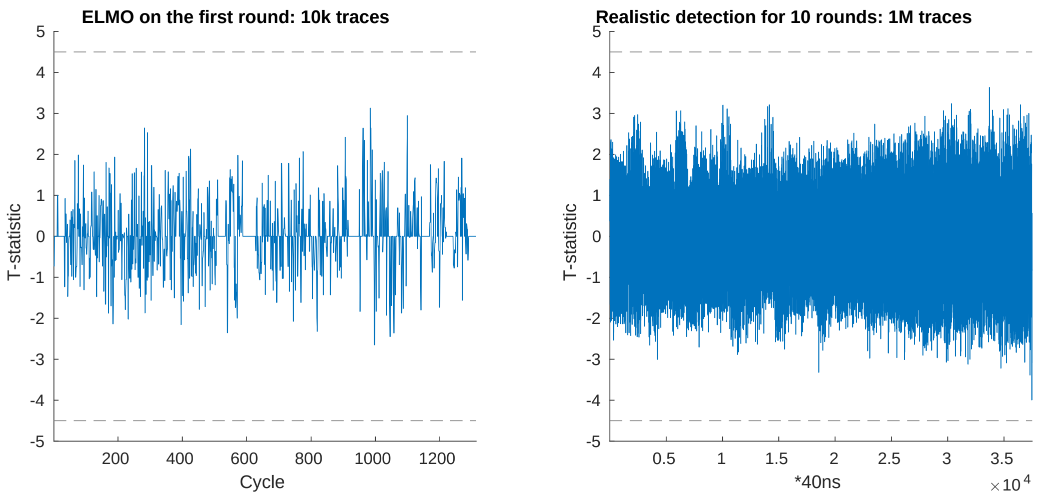


Figure 12: ELMO & Realistic tests on the final scheme

References

[1] 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 *International Symposium on Circuits and Systems (ISCAS 2006), 21-24 May 2006, Island of Kos, Greece*. IEEE, 2006.

[2] ARM. AMBA APB Protocol Version 2.0 Specification. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0024c/index.html`.

[3] ARM. ARM AMBA 5 AHB Protocol Specification (AHB5, AHB-Lite). `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0033/index.html`.

[4] ARM. GNU Arm Embedded Toolchain: Pre-built GNU toolchain for Arm Cortex-M and Cortex-R processors. `https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm`.

[5] ARM. Armmbedtls. `https://tls.mbed.org/`, 2008–2015.

[6] J. Cohen. *Statistical power analysis for the behavioral sciences*. Routledge, 1988.

[7] Y. L. Corre, J. Großschädl, and D. Dinu. Micro-architectural power simulator for leakage assessment of cryptographic software on ARM Cortex-M3 processors. In J. Fan and B. Gierlichs, editors, *Constructive Side-Channel Analysis and Secure Design – 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings*, volume 10815 of *LNCS*, pages 82–98. Springer, 2018.

[8] N. Debande, M. Berthier, Y. Bocktaels, and T.-H. Le. Profiled model based power simulator for side channel evaluation. Cryptology ePrint Archive, Report 2012/703, 2012.

[9] J. den Hartog, J. Verschuren, E. P. de Vink, J. de Vos, and W. Wiersma. PINPAS: A tool for power analysis of smartcards. In *International Conference on Information Security (SEC2003)*, volume 250 of *IFIP Conference Proceedings*, pages 453–457. Kluwer, 2003.

[10] eshard. esDynamic. `https://www.eshard.com/product/esdynamic/`, accessed June 2018.

[11] S. Furber. *ARM System-on-Chip Architecture*. Addison Wesley, 2000.

[12] G. Gagnerot. Étude des attaques et des contre-mesures assoccées sur composants embarqués. PhD thesis, Université de Limoges, 2013.

[13] S. Gao. A Thumb Assembly based Byte-wise Masked AES implementation. `https://github.com/sca-research/ASM_MaskedAES`.

[14] G. Goodwill, B. Jun, J. Jaffe, and P. Rohatgi. A testing methodology for side-channel resistance validation. In *NIST Non-invasive attack testing workshop*, 2011.

[15] Information technology – Security techniques – Testing methods for the mitigation of non-invasive attack classes against cryptographic modules. Standard, International Organization for Standardization, Geneva, CH, 2016.

[16] M. Kirschbaum and T. Popp. *Evaluation of Power Estimation Methods Based on Logic Simulations*, pages 45–51. Verlag der Technischen Universität Graz, 2007.

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

[18] D. McCann. ELMO. `https://github.com/bristol-sca/ELMO`, 2017.

[19] D. McCann, E. Oswald, and C. Whitnall. Towards practical tools for side channel aware software engineering: 'grey box' modelling for instruction leakages. In E. Kirda and T. Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, pages 199–216. USENIX Association, 2017.

[20] L. W. Nagel and D. Pederson. SPICE (Simulation Program with Integrated Circuit Emphasis). Technical Report UCB/ERL M382, EECS Department, University of California, Berkeley, Apr 1973.

[21] E. Oswald, S. Mangard, C. Herbst, and S. Tillich. Practical second-order DPA attacks for masked smart card implementations of block ciphers. In *CT-RSA 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 192–207. Springer, 2006.

[22] O. Reparaz. Detecting flawed masking schemes with leakage detection tests. In T. Peyrin, editor, *Fast Software Encryption*, pages 204–222, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[23] Riscure. Inspector SCA Security Tool. `https://www.riscure.com/security-tools/inspector-sca/`.

[24] Secure-IC. Virtualyzr. `http://www.secure-ic.com/solutions/virtualyzr/`, accessed June 2018.

[25] C. Thuillet, P. Andouard, and O. Ly. A smart card power analysis simulator. In *Proceedings of the 12th IEEE International Conference on Computational Science and Engineering, CSE 2009*, pages 847–852. IEEE Computer Society, 2009.

[26] N. Veshchikov. Use of Simulators for Side-Channel Analysis: Leakage Detection and Analysis of Cryptographic Systems in Early Stages of Development. PhD thesis, Université Libre de Bruxelles, 2017.

[27] D. Welch. Thumbulator. `https://github.com/dwelch67/thumbulator.git/`, 2014.

[28] C. Whitnall and E. Oswald. A critical analysis of ISO 17825 ('Testing methods for the mitigation of non-invasive attack classes against cryptographic modules'). In S. D. Galbraith and S. Moriai, editors, *Advances in Cryptology - ASIACRYPT 2019*, volume 11923 of *LNCS*, pages 256–284. Springer, 2019.

[29] Y. Yao. A byte-wise Masked AES Implementation. `https://github.com/Secure-Embedded-Systems/Masked-AES-Implementation/tree/master/Byte-Masked-AES`.

A Existing Leakage Simulators

The appeal of leakage simulation was evident already during the early days of side channel research: when proposing a novel analysis method it is necessary to demonstrate its efficiency across many different use cases. Instrumenting many different devices (and for a range of different signal-to-noise ratios) is cumbersome and time consuming, and thus researchers were motivated to look at alternatives such as running simulations on different (artificially generated leakage models) and by adding different amounts of Gaussian noise.

Another area of interest for simulations was the evaluation of hardware implementations. In the context of hardware it is even more crucial to make statements early on in the design, because of the cost implications of having to tape out multiple test devices for evaluations. Consequently, the development and use of tools to estimate instantaneous power at the transistor and gate level was of considerable interest; a previous EU funded project called SCARD devoted some time to this task [1].

We now run through a list of simulators that we were able to find information for and review their alignment with the aims of the REASSURE project.

A.1 Industrial Tools

PinPas. PinPas (Program Inferred Power Analysis Simulator) [9] was one of the earliest simulation tools designed with side-channel analysis in mind. It was written in Java and aimed to simulate simple leakage from small 8-bit microcontrollers. The leakage model itself was chosen *a priori* (specifically, it was the Hamming weight) rather than being derived from any concrete device. It was not open source and is no longer available.

Tools from the SCARD project. A number of hardware simulation approaches were investigated and implemented as part of the EU-funded project SCARD, including two prototype gate-level power estimators: one a stand-alone tool based on value change dumps (VCDd), the other embedded into Mentor's ModelSim via the PLI interface [1]. They also developed a methodology to model the effect of the IC package and bonding wires on the power consumption. A follow up paper [16] drew positive conclusions about the toggle-count models based on visual comparisons of the (simulated versus real) traces and DPA results, but the coverage and statistical rigour of the assessment was limited.

Riscure Inspector. Riscure's Inspector toolbox [23] contains some functionality for high level power simulation: given a high level piece of code, and a power model, a tool will produce traces based on applying the given leakage function to intermediates that are specified in the high level code. It does not take information about an actual target architecture into account.

RiscaSim. RiscaSim is a hybrid side channel analysis simulator, running on a Riscurino board designed by Riscure that enables the simulation of side channel analysis traces without dedicated measurement equipment. RiscaSim is sending "artificial" traces over the serial port UART to the computer. RiscaSim allows a user to select from a number of leakage models, including returning the "raw" register content to facilitate the analysis of "Whitebox" implementations. It is envisioned to extend its' functionality to include fault attacks. The documentation to RiscaSim was developed with REASSURE input and is available here `https://rise.articulate.com/share/rj1yOz06EtTY4dsACPM7D14t5QgpiLzS#/lessons/eZWsV9izVZnmvCBeOt2R-pEpzP6iQz1H`.

esDynamic. esDynamic is a set of tools to enable the analysis of cryptographic implementations [10]. Included is the ability to emulate binary code on a number of platforms and to perform some standard side channel analysis on these emulations. There does not seem to be any actual characterisation of the emulated platforms included.

Virtualyzr. The Virtualyzr tool from Secure-IC is a HDL simulator [24]. There is no detailed public information available for it, but from the website one can gather that it can be used with various types of descriptions that are available within a typical hardware design flow.

A.2 Academic Tools

Thuillet et al. [25] report on a simulator that is similar to PinPas and is designed for the analysis of smart cards. It takes in a program in a high level language and works using a standard power model.

Debande et al. [8] emphasise the importance of (and the complexities involved in) deriving realistic leakage models empirically. They fit linear models in function of the state bits and state transitions using the techniques of linear regression.

Gagnerot [12] reports on writing a simulator for side channel and fault attacks as part of his PhD thesis. The simulator was designed for a 16-bit RISC architecture under an NDA. It was based on using standard power models (HW and HD).

Reparaz [22] suggests to essentially use a power simulation with some leakage detection testing to spot problems in masked implementations as an alternative to more costly verification tools.

Bristol University Applied Security unit has been using an 8051 emulator together with a standard power model (HW) for some years. Every student receives a compiled binary that includes a 'secret key' and can then interact with the leakage simulator to generate side channel traces to perform attacks.

ELMO [19] is a simulator that takes in the Thumb assembly of an algorithm and maps the data flow via power models that have been provided especially and are intended to be specific to a processor architecture. The featured approach is based on model *building* rather than simply fitting (i.e. rather than estimating the parameters for one fixed model, a variety of candidate configurations are explored and decisions about the functional form made based on the statistical significance or otherwise of tested alternatives). The underlying M0 emulator is open source [27], and thus the tool is also available as open source [18].

Veshchikov's PhD thesis [26] proposes a number of tools: Silk, Ascold and Savrasca, each working on a different level of abstraction. Savrasca can take in custom models and is capable of dealing with an AVR architecture because it is essentially based on SimulAVR.

MAPS [7] is the most recent tool. It also focuses on an M class processor from ARM. Since the development of ELMO, ARM has enabled researchers to get access to a (semi) obfuscated RTL description of the M0 and M3 processor. MAPS uses this information to model the pipeline of an M3, which is an important source of HD leakage. The emulator is written from scratch in a mix of C and C++ and uses standard power models (HW, HD).