Micro-Architectural Power Simulator for Leakage Assessment of Cryptographic Software on ARM Cortex-M3 Processors

Yann Le Corre, Johann Großschädl, and Daniel Dinu

CSC and SnT, University of Luxembourg
6, Avenue de la Fonte, L–4364 Esch-sur-Alzette, Luxembourg
{yann.lecorre,johann.groszschaedl,dumitru-daniel.dinu}@uni.lu

Abstract. Masking is a common technique to protect software implementations of symmetric cryptographic algorithms against Differential Power Analysis (DPA) attacks. The development of a properly masked version of a block cipher is an incremental and time-consuming process since each iteration of the development cycle involves a costly leakage assessment. To achieve a high level of DPA resistance, the architecture-specific leakage properties of the target processor need to be taken into account. However, for most embedded processors, a detailed description of these leakage properties is lacking and often not even the HDL model of the micro-architecture is openly available. Recent research has shown that power simulators for leakage assessment can significantly speed up the development process. Unfortunately, few such simulators exist and even fewer take target-specific leakages into account. To fill this gap, we present *MAPS*, a micro-architectural power simulator for the M3 series of ARM Cortex processors, one of today's most widely-used embedded platforms. *MAPS* is fast, easy to use, and able to model the Cortex-M3 pipeline leakages, in particular the leakage introduced by the pipeline registers. The M3 leakage properties are inferred from its HDL source code, and therefore *MAPS* does not need a complicated and expensive profiling phase. Taking first-order masked Assembler implementations of the lightweight cipher Simon as example, we study how the pipeline leakages manifest and discuss some guidelines on how to avoid them.

Keywords: Leakage assessment, architecture-specific leakage, pipeline leakage, power simulator, Cortex-M3

1 Introduction

Side-channel attacks [14] pose a serious threat to the security of cryptographic primitives, in particular when they are executed on mobile or embedded devices that are physically accessible to an attacker. A typical example of such devices are wireless sensor nodes, which are often deployed in unattended areas and do not come with any measures or techniques to minimize the leakage of sensitive information through power or electromagnetic (EM) side channels. One of the most sophisticated forms of side-channel attack is Differential Power Analysis (DPA), first described in the open cryptographic literature almost 20 years ago by Kocher et al. [13]. A standard DPA attack involves two steps, namely (i) an acquisition step, in which the attacker measures the power consumption of the target device while it executes a cryptographic algorithm, and (ii) an analysis step, in which she uses advanced statistical techniques to recover the sensitive (i.e. key-dependent) data processed during the execution of the algorithm from the acquired power consumption traces. There exists a large body of literature demonstrating successful DPA attacks against (unprotected) implementations of both secret-key and public-key cryptographic primitives, see e.g. [15] and the references therein. In the case of block ciphers, it was shown that a few dozens of power traces can be sufficient to reveal the full secret key [7].

Due to the efficacy of DPA attacks, it is necessary to protect an implementation of a block cipher through the integration of countermeasures. One of the most well-known and widely used DPA countermeasures is *masking*, which can be realized in both hardware and software [8,11,22]. Masking aims to conceal every key-dependent variable with a random value, called mask, to de-correlate the sensitive data of the algorithm from the data that is actually processed on the device. The basic principle is related to the idea of secret sharing because every sensitive variable is split up into $n \geq 2$ "shares" so that any combination of up to $d = n - 1$ shares is statistically independent of any secret value. These n shares have to be processed separately during the execution of the algorithm (to ensure their leakages are independent of each other) and then recombined in the end to yield the correct result. What makes masking attractive is that its security can be proven in the framework of Isai, Sahai, and Wagner [12]. However, despite the theoretical security guarantees, it turned out that masking is challenging to implement in practice without introducing unintended (and often unobvious) leakage. For example, it was shown in [16] that a masked hardware implementation of the AES can be broken by exploiting glitches at the outputs of logic gates. On the other hand, software implementations of masked ciphers can also be vulnerable to DPA attacks because of unintended violations of the independent leakage requirement mentioned above, typically caused by certain micro-architectural effects and features [4,18,21]. Therefore, it is important to check whether a masked implementation meets its theoretical security promises also in practice (i.e. does not show any DPA-exploitable leakage), which can be done by performing a leakage assessment test [6] or a full DPA attack.

Developing a masked software implementation of a block cipher is a tedious and highly iterative endeavor. The developer tries to eliminate existing leakage and then performs a leakage assessment, and thereafter the same cycle starts again until no leakage can be detected anymore [4,7]. In order to decrease the development time, a power simulator like ELMO [17] can be used to generate the power traces. However, to get realistic traces, the simulator should be able to take certain micro-architectural effects into account, in particular the inter-instruction dependencies in the power consumption (and, hence, leakage) of the target processor. For example, due to pipelining effects, the power consumption caused by the execution of a certain instruction does not solely depend on the operands/results and from/to which register(s) they are read/written, but also on preceding and succeeding instructions that are in the pipeline at the same time. ELMO takes these effects into account by using measured power characteristics and by grouping instructions together. In the case of ARM Cortex M0 and M4 microcontrollers, which are currently supported by ELMO, up to three instructions need to be considered since the pipeline has three stages.

Even though ELMO is a undoubtedly a useful tool, it suffers from a couple of shortcomings. In particular, getting realistic instruction-level power models is a non-trivial task and requires a lot of measurements. Furthermore, in order to model differential data-dependent effects of “neighboring” instructions, ELMO uses power models for groups of instructions, whereby the size of the groups is determined by the number of instructions that can be in the processor pipeline at the same time (i.e. the number of pipeline stages). This approach achieves promising results, as demonstrated through several experiments by the authors of [17], but seems only viable for processors with few (e.g. up to three) pipeline stages. However, there exists a large number of embedded processors with five (e.g. ARM9), eight (e.g. ARM11), or even eleven (e.g. Cortex-R7/R8) pipeline stages, which makes it very costly to develop accurate power models for groups of instructions. Our simulator *MAPS* (Micro-Architectural Power Simulator) is based on a different approach and takes the inter-instruction dependency of the power consumption into account by utilizing a more refined micro-architectural model of the target processor. Specifically, *MAPS* models all pipeline registers and validates these models through simulations with an HDL description of the target micro-architecture. Thus, *MAPS* has two advantages over ELMO: (i) the power model does not require any measurements, especially no measurements of inter-instruction dependencies, and (ii) *MAPS* is also suitable for processors with deep(er) pipelines consisting of more than three stages.

Our Contributions. We present the basic concepts of *MAPS*, which is to the best of our knowledge the first open-source power simulator for leakage assessment targeting the Cortex-M3 architecture, one of the most popular platforms in the embedded domain. Besides being fast and easy to use, *MAPS* is capable to model (certain) architecture-specific leakages based on a structural analysis of an HDL description of the Cortex-M3 pipeline. As a second contribution, we analyze the impact of pipeline registers on the leakage of masked ciphers.

2 State of the Art

Over the years, numerous power simulators have been developed; the interested reader can find a survey in [24, Sect. 5.3]. We focus here on recent simulators that perform high-level simulations rather than analog or transistor/gate-level simulations. While low-level simulators are, in general, more accurate, they are relatively slow and rely on VLSI-technology-specific data (e.g. netlists, parasitic components, back-annotated delays), which is usually not publicly available.

Gagnerot introduced in his thesis [10] a power simulator that was developed for leakage assessment of cryptographic implementations. It is able to generate power traces by monitoring all read/write operations on the registers and buses of a complete system (e.g. a smart card). No concrete details of the system are described because the power simulator was developed in collaboration with an industry partner. However, what was stated is that it contained a 16-bit RISC processor, UART interfaces, as well as two coprocessors. The simulator accepts a compiled binary object file as input. Neither the simulator nor its source code are publicly available; hence, it is not known how detailed the modeling of the processor is, e.g. whether it includes the pipeline registers or not.

SILK stands for "Simple Leakage Simulator" and was presented by Veshchikov in 2014 [23]. It is not tied to a specific processor architecture but generates power traces using a high level of abstraction. The power model is very flexible and can be easily adapted to support different leakage scenarios. SILK accepts a C file as input. The source code is publicly available on Github[1].

Also Reparaz described in [19] a simulator capable to generate power traces from a high-level C description of a cryptographic algorithm. The values of the intermediate variables are traced after the implementation has been compiled with a modified version of the LLVM compiler. Thus, the simulator is not tied to a specific architecture. Yet, it is fast and also provides debugging capabilities that help a developer to pinpoint the sources of leakage.

ELMO ("Emulator for Power Leakage for Cortex M0") was introduced in 2016 by McCann et al. [17]. It is dedicated to the Cortex-M0 and M4 families of processors and takes a compiled binary object file as input. ELMO is based on an existing ARM v6-M emulator, which was "back-annotated" with leakage information. This leakage information was extracted using elaborate statistical processing that was applied to power measurements performed on a hardware setup. Therefore, ELMO belongs to the category of profiled simulators. Due to limitations of the underlying emulator, it supports only the Thumb instruction set but not Thumb-2. The reported leakages are potentially very accurate since the hardware measurements include various leakage effects such as glitches and coupling. However, adding a new target to ELMO is very challenging because it requires an elaborate measurement setup and the statistical processing has to be done again since it depends on the characteristics of the target instruction set and micro-architecture, e.g. pipeline depth. ELMO is publicly available[2].

3 Cortex-M3 Architecture-Specific Leakages

3.1 Cortex-M3 Overview

The Cortex-M3 is a 32-bit RISC processor developed by ARM that implements version v7-M [1] of the ARM instruction set. It is one of the most widely-used embedded platforms because it combines an efficient and compact instruction

[1] `https://github.com/nikita-veshchikov/silk`
[2] `https://github.com/bristol-sca/ELMO`

set with a high-quality tool chain. The Cortex-M3 has a Harvard architecture with both 16-bit and 32-bit instructions as well as a 32-bit data path. It does not include a data cache and comes with a pre-fetch buffer instead of a more complex instruction cache. Like other 32-bit ARM processors, the Cortex-M3 is equipped with 16 registers; besides 13 general-purpose registers (`r0-r12`) there is a stack pointer (`r13`), a link register (`r14`), and a program counter (`r15`).

The arithmetic and logical instructions operate solely on registers. A barrel shifter located between the register file and the Arithmetic-Logic Unit (ALU) allows one to combine an ALU operation with a shift or rotation of the second operand. Most ALU instructions execute in one cycle; the only exceptions are `mul` (multiply), `div` (divide), and operations targeting the program counter.

The pipeline is made of three stages. In the first stage, the instruction gets fetched from instruction memory. Thereafter, the instruction is decoded in the second stage, and finally executed in the third stage. Conditional branches are speculated (i.e. one of the alternative instructions is speculatively executed and eventually discarded if it turns out that the speculation was wrong). Store to memory instructions (e.g. `str`) are buffered and executed in one cycle, whereas load from memory instructions (e.g. `ldr`) introduce a wait-state. The typical Clock-Per-Instruction (CPI) figure for embedded software is close to 1.

3.2 Cortex-M3 HDL Analysis

ARM makes the entire HDL source code of the Cortex-M3 processor available to universities via the *DesignStart Pro Academic* program. The source package contains the M3 core, which is described in a set of Verilog files, and a minimal system that connects the core with the memories through AMBA ("Advanced Microcontroller Bus Architecture") buses. The system also includes peripherals like communication and debugging interfaces to enable developers to trace the execution of a program. By default, the Verilog simulation of the system loads and executes a C program cross-compiled for the ARM v7-M architecture.

Since we have access to the HDL source code, all registers related with the data path can be isolated and then traced. At the logic level, any information leakage could be related to the values held by the registers. The dependencies between the succeeding instructions and the sensitive data will be captured too since these registers also define the pipeline stages. All registers in the core can be easily found by looking for signals defined with the Verilog keyword *reg* and assigned in an "*always @(posedge <clock>)*" block. Of these registers, only the ones involved in the manipulation of data are relevant from a leakage-detection point of view. We can further discriminate by selecting the registers that have a width of 32 bits. In addition, since the ALU operates exclusively on operands read from registers, only the 32-bit registers connected to the two output ports of the register file have to be analyzed. Based on these criteria, the 16 registers of the register file (i.e. `r0` to `r15`), along with two registers `ra` and `rb` located between the register file and the ALU and three registers inside the ALU, are retained. However, the latter three registers are solely used during multi-cycle ALU instructions like `umull`, `umlal`, or `udiv`. Since such instructions are quite

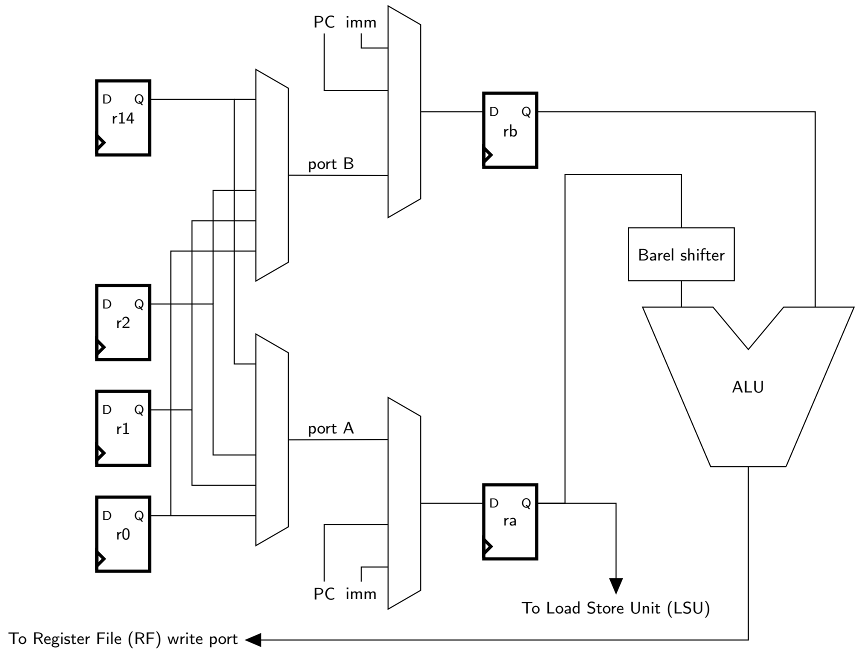

Fig. 1. Simplified structure of the Cortex-M3 pipeline

uncommon in the context of symmetric cryptosystems, we decided to not trace these three registers. The program counter `r15` is also not traced by default in order to limit the length of the power traces. A major requirement for secure cryptographic software is that the control flow is independent of any sensitive data; if this is the case then the program counter can not leak anyway.

The registers `ra` and `rb` are pipeline registers isolating the decode from the execute stage. Their existence and location could have also been inferred from the fact that an ALU instruction can be executed while the succeeding instruction can access the registers. However, our analysis of the HDL code confirmed their exact location and allowed us to find out what values are written to them in each instruction. A simplified version of the pipeline is shown in Fig. 1.

3.3 Cortex-M3 Pipeline Leakages

The registers `ra` and `rb` are specific to the pipeline architecture of the M3 processor. They are a possible source of leakage since they combine operand values of consecutive instructions. Indeed, the power consumption associated with the writing to these registers is directly related to the Hamming distance between the current operand value and the previous one. Both the first and the second operand of ALU instructions can be affected. Since the register `ra` connects the the register file with the barrel shifter, even an ALU instruction with a shifted or rotated second operand may leak through this register.

Listing 1. Code fragment with second-operand leakage

```
; r2 and r3 contain the two shares
; r4 and r5 contain random and unrelated values
; r6 and r7 are initialized to 0
and r6, r4, r2, lsl 4
orr r7, r5, r3, ror 5
```

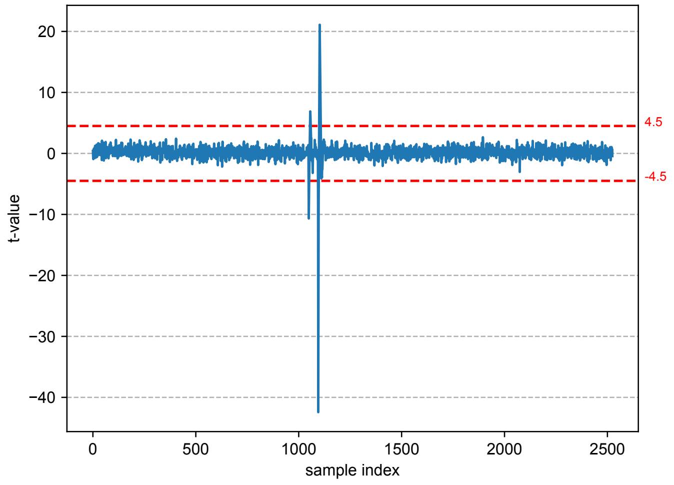

Fig. 2. T-test confirming second-operand leakage (hardware measurements)

Register Transfer Notation 1. Equivalent to Listing 1

```
1: $\mathtt{rb} \leftarrow \mathtt{r4}$
2: $\mathtt{ra} \leftarrow \mathtt{r2}$
3: $\mathtt{r6} \leftarrow \mathtt{rb} \wedge (\mathtt{ra} \ll 4)$
4: $\mathtt{rb} \leftarrow \mathtt{r5}$
5: $\mathtt{ra} \leftarrow \mathtt{r3}$                              $\triangleright\ \mathit{Power}(\mathtt{ra}) = \mathit{HW}(\mathtt{r2} \oplus \mathtt{r3})$
6: $\mathtt{r7} \leftarrow \mathtt{rb} \vee (\mathtt{ra} \ggg 5)$
```

Listing 1 illustrates such a leakage. In this code fragment, the two registers `r2` and `r3` hold the shares of a secret value, which is ($\mathtt{r2} \oplus \mathtt{r3}$). Register `r4` and `r5` contain arbitrary values unrelated with the content of other registers. From an architectural view, there should be no leakage. However, our measurements on an actual Cortex-M3 processor show that there is a leakage, as illustrated in Fig. 2. The measurements were taken on an Atmel Cortex-M3 SAM3X8E chip using a Langer EM probe connected to a LeCroy WR 8254M oscilloscope sampling at 500 MSamples/s. This leakage is not difficult to explain when we take all register transfers involving `ra` and `rb` into account. Listing 1 is equivalent to the Register Transfer Notation 1. As expected, ($\mathtt{r2} \oplus \mathtt{r3}$) leaks through `ra`.

Listing 2. Code fragment with `str` instruction leakage

```
; r2 and r3 contain the two shares
str r2, [r0, 0]
str r3, [r0, 4]
```

Register Transfer Notation 2. Equivalent to Listing 2

```
1: rb ← r0
2: ra ← r2
3: rb ← r0
4: ra ← r3                                  ▷ $Power(\mathtt{ra}) = HW(\mathtt{r2} \oplus \mathtt{r3})$
```

In general, every instruction using a value read from a general-purpose register is affected, not only the ALU instructions. For example, all memory store instructions will leak when executed one right after another, as in Listing 2 and its equivalent Register Transfer Notation 2. The leakage of the `str` instructions extends further to the `push` and the store-multiple (`stm`) instructions since the latter are actually a shorthand for a sequence of `str` instructions.

3.4 Guidelines to Reduce Cortex-M3 Pipeline Leakage

The Cortex-M3 pipeline leakages can be reduced or even entirely circumvented in a few different ways, listed below in ascending order of their implementation cost in terms of execution time and code size.

1. Simply swap the operands of commutative instructions.
2. Schedule instructions so that the two shares are not processed by successive instructions. This may be difficult to achieve because of the relatively small number of general-purpose registers.
3. Overwrite the pipeline registers with unrelated values, which can sometimes be done by just using more complex instructions for certain operations. To give a concrete example, the statement "`mov r0, 0`" to clear register `r0` can be replaced by "`eor r0, rx, rx`" where `rx` is an arbitrary register. In the former version using `mov`, the registers `ra` and `rb` are not written since the immediate value `0` gets directly transferred from the instruction decoder to the register `r0`. In the second version, `ra` and `rb` are written with the value of `rx` before `r0` is cleared. This version can, depending on which register is actually used as `rx`, increase code size by two bytes at most.
4. Explicitly set the registers `ra` and `rb` to a value unrelated to any sensitive data. This can be done by a statement of the form "`orr r0, r0, r0`" where `r0` contains some random data; for example, `r0` could be the address of an input buffer. The cost is a clock cycle and two or four bytes of code size.

Note that inserting a `nop` instruction will not prevent the leakage since the `nop` instruction does not pass the instruction decoder and, consequently, it can not modify the two pipeline registers `ra` and `rb`.

4 Our Simulator: *MAPS*

In this section, we provide an overview of the main properties (i.e. features and limitations) of *MAPS* and briefly describe its operation.

4.1 Features

MAPS has been created to aid and simplify the development of masked implementations of cryptographic primitives. Its main features are as follows.

Easy to Use. The implementation and testing of a masked primitive requires advanced skills in cryptographic engineering. In addition, it is a highly iterative task that takes a lot of time, effort, and scrutiny. Our simulator is easy to use (even for non-experts) and provides a convenient way to do automated leakage assessment of cryptographic implementations. In this way, *MAPS* simplifies the whole development and testing process.

Advanced Debug Support. In this paper, the word debug has actually two meanings; the first relates to the debugging of the functionality of the primitive to achieve (algorithmic) correctness. Our simulator is able to interact with the GNU debugger GDB through a GDB server. The other meaning refers to identifying which instructions cause information leakage. *MAPS* generates an index file linking the program counter and the power trace sample index, which allows for easy identification of the instruction that leaks.

Fast(er) Development Cycles. Securely-masked versions of a cryptographic algorithm are typically implemented in Assembly language to have full control over the instructions that will be executed. The allocation of registers and the selection of operands may require several tries, but long simulation times make it costly to try various “what-if” scenarios. *MAPS* is very fast so that an entire “compile-simulate-test” cycle can be completed in just a few minutes.

Only One Set of Source Files. In absence of a leakage simulator for Cortex-M3, implementers commonly resort to emulated leakages, which, as mentioned in Sect. 2, are typically generated using a high-level C implementation instead of the Assembler implementation that will actually be deployed. Having to deal with two separate code bases can easily lead to mistakes due to inconsistencies and may require adaptations of one or both source codes. Using *MAPS* avoids such problems since the leakage assessment can be carried out with exactly the same implementation that will finally end up on the target device.

Target-Specific Leakages. Our simulator reports algorithmic leakages and as many as possible target-specific leakages. The power waveforms are computed from the trace of all registers related to the data being processed, including the pipeline registers.

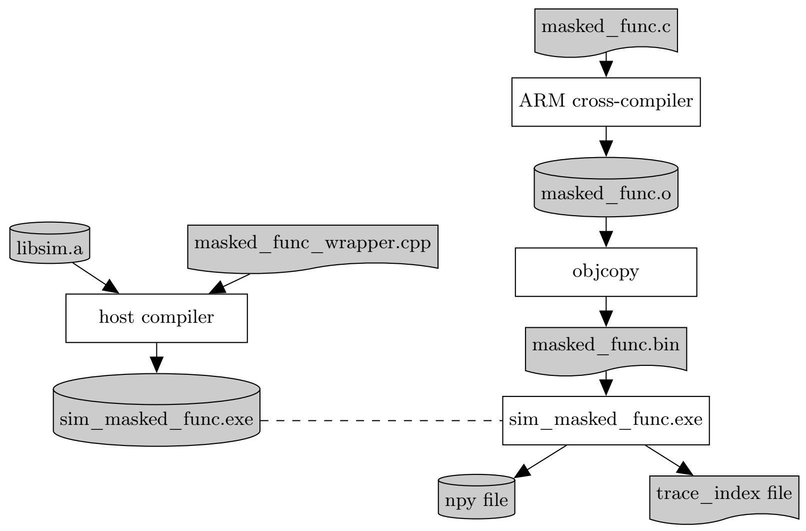

Fig. 3. *MAPS* flow

Open-Source. *MAPS* is open-source software[3] and may be used and modified without restrictions. In addition, anyone can contribute to the further development of *MAPS* by adding not-yet-supported instructions or new features.

4.2 Simulation Flow

A high-level view of the operation of *MAPS* is depicted in Fig. 3. At first, one has to produce a simulator executable, which is labeled *sim_ masked_ func.exe* in Fig. 3. The executable is tasked with loading and simulating the function to be analyzed. It "glues" together the Cortex-M3 simulation engine, the interface functions, and the test functions, all written in C++ 11.

The Cortex-M3 simulation engine is a C++ object with the usual methods such as `load()`, `step()`, `run()`, and so on. It is also responsible for tracing the register writes: each time a register is written, the Hamming distance between the previous register value and the new value is added as a new sample to the power trace. A power trace is a `std::vector` that can be manipulated after the end of the simulation. The Cortex-M3 simulation engine as well as some useful functions, such as a default `main()` function handling common command-line options, are grouped together in a library named *libsim.a*.

The file *masked_func_ wrapper.cpp* contains both the test functions and the interface functions. The latter functions wrap the call to the simulator engine so that the function to be analyzed regarding leakage appears like a host-domain function. It "abstracts" the process of passing parameters from the host to the

[3] The full source code of *MAPS* is available on Github under the GNU General Public License version 3 (GPLv3): `https://github.com/cryptolu/maps`.

simulated function. All parameters are simply copied into the simulated target memory as required by the ARM Application Binary Interface (ABI) [2].

The test functions implement a basic fixed-vs-random Welch's t-test leakage assessment as described in detail in [6]. However, the assessment method is independent of the simulation engine and can therefore be easily replaced. The test functions and interface functions are not part of the library *libsim.a* since different functions to be analyzed will have different interfaces.

The function to be analyzed regarding leakage needs to be written in C and can contain inlined Assembly code as well as macros. It must be stored in the file *masked_func.c*, which is cross-compiled for ARM v7-M and converted into binary format. When the simulator executable is run, it loads the result of the cross-compilation and applies the fixed and random inputs as instructed by the test functions. Welch's t-test is computed over the collected power traces and stored in a Numpy (*.npy*) file that can be conveniently visualized using Python scripts. A so-called trace index file is also generated, mapping the t-test sample index to the simulated program counter. Thanks to this file, the address of an instruction causing leakage can be quickly spotted.

4.3 Validation

In order to ensure that the Cortex-M3 processor is correctly modeled, both its functionality and leakage generation features were carefully tested in a specific test environment. All supported instructions were collected in a C file that was cross-compiled for the Cortex-M3. Then, they were simulated using *MAPS* as well as ARM's Verilog-based minimal system testbench. For each simulation, a trace of the registers was created and the two traces were compared. The trace generated by our simulator exactly matched the one produced by the system testbench, which guarantees that *MAPS* behaves like the actual processor.

4.4 Limitations

The current version of *MAPS* has the following limitations:

- Only the Cortex-M3 target is supported.
- Not all instructions of the Cortex-M3 described in [1] are supported. The currently-not supported instructions include conditional instructions, table branch instructions, saturation instructions, multiply instructions, packing instructions, as well as hint instructions. However, all these instructions are unlikely to be found in an implementation of a symmetric primitive.
- The simulator traces only the registers. Glitches or the power consumption of the ALU are not taken into account. For example, a "`cmp r2, r3`" leaks ($\texttt{r2} - \texttt{r3}$) on the actual hardware, but does not leak on the simulator.
- No peripheral components or interfaces are modeled, and data can only be transferred between host and targets using the ABI and target memory.
- The simulator traces only registers of the Cortex-M3 core. Other registers located outside of the core, e.g. in a memory interface, are not considered.
- The simulator is not cycle-accurate.

4.5 Performance

The simulation speed of *MAPS* is summarized in Table 1. All test cases correspond to a fixed-vs-random Welch's t-test as in [6] for a million measurements (i.e. two million executions of the simulated function). The tests were executed on an Intel i7-6700 processor running at 3.4 GHz. For comparison, it should be noted that the acquisition speed of the setup employed by the organizers of the DPA contest v4 to measure AES power traces was about 0.9 traces/s [24].

Table 1. *MAPS* simulation performance for three first-order masked block ciphers (generation of one million traces)

Algorithm	No. of instructions	Simulation time [s]	Traces/s
Simon-64/128	1194	113	17700
Rectangle-64/128	2279	220	9091
Speck-64/128	6055	488	4098

We used for our performance evaluation first-order masked implementations of three well-known lightweight block ciphers, namely Simon and Speck [5], as well as Rectangle [25]. Simon-64/128 is a hardware-oriented cipher with an And-Rotation-Xor structure. The version we tested is a 2-share masked implementation protected with the Trichina AND gate [22]. Speck-64/128 is a more software-optimized cipher based on an Addition-Rotation-Xor structure. The tested implementation is protected by a 2-share Boolean masking, whereby the modular addition is performed directly on the Boolean shares according to the Kogge-Stone Adder (KSA) technique introduced in [9]. Rectangle-64/128 is a bit-sliced lightweight cipher designed on basis of a substitution-permutation network. The tested implementation is protected by a 2-share Boolean masking using the Trichina AND gate [22] and the OR gate from Baek et al. [3] with an additional random variable to mirror the AND gate.

5 Case Study

In this section, we show how *MAPS* can be used to implement a secure version of Simon-64/128 on a Cortex-M3 processor. All results we will present in the following are based on a leakage assessment using Welch's t-test on power traces generated by *MAPS* in a fixed-vs-random setting. For each experiment, 10,000 traces with fixed inputs and 10,000 traces with random inputs were collected.

First, Fig. 4(a) shows the result of a naive implementation of Simon-64/128 protected using Trichina AND gates. This naive implementation minimizes the number of execution cycles and places intermediate results of the computation in the next free register. Any Hamming distance effect due to the reuse of some registers was not taken into account and *MAPS* was configured to not trace the pipeline registers `ra` and `rb`. Unsurprisingly, this implementation leaks.

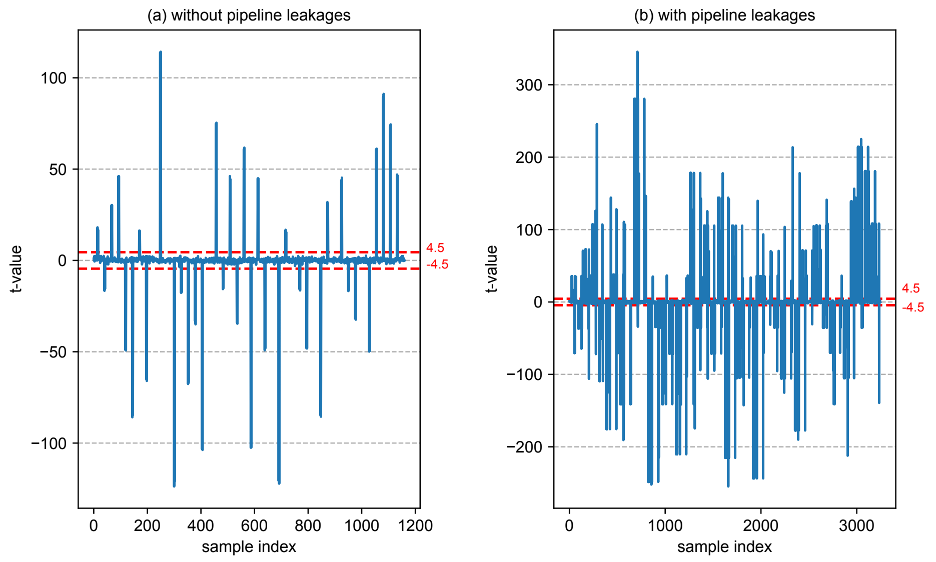

Fig. 4. Leakage assessment of the naive implementation of Simon-64/128, simulated (a) without and (b) with pipeline leakages

Figure 4(b) visualizes the result of the leakage assessment test for the same naive implementation, but this time the tracing of the pipeline registers `ra` and `rb` is enabled in the simulator. Many more leakage points can be observed.

Next, the naive implementation was improved by fixing the leakages due to the reuse of registers. The obtained result of the leakage assessment is depicted in Fig. 5(a), whereby the simulator was configured to not trace register `ra` and `rb`. Now the leakages seem to be fixed. However, Fig. 5(b) illustrates that this improved version still leaks through the pipeline registers when their tracing is enabled. In fact, most of the leakage comes from the two pipeline registers.

Table 2. Comparison of three masked implementations of Simon-64/128

Version	No. of instructions	Penalty factor
(1) naive	1106	1.00
(2) fixed register-reuse leakages	1194	1.08
(3) fixed pipeline leakages	1285	1.16

Table 2 lists the number of instructions executed by the three Simon implementations. Version (1) is the naive implementation and version (2) the naive implementation with fixed register-reuse leakage effects. Version (3) represents an improvement of version (2) to fix all pipeline leakages using the techniques given in Subsect. 3.4. It should be noted that the number of instructions differs

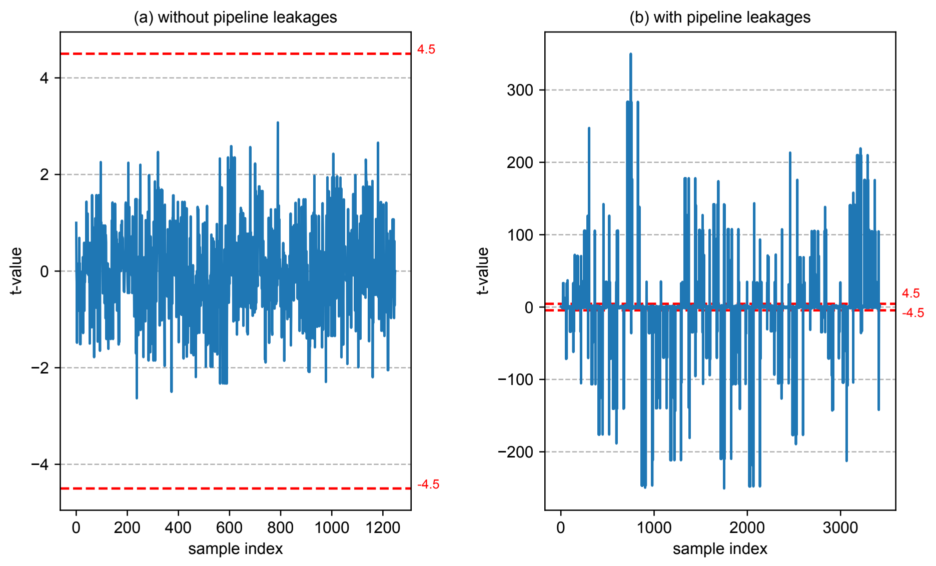

Fig. 5. Leakage assessment of the improved implementation of Simon-64/128 (register-reuse leakages corrected), simulated (a) without and (b) with pipeline leakages

from the number of clock cycles. For example, replacing one `stm` instruction by several `str` instructions does not increase the number of cycles.

Figure 6 shows the result of the t-test for the further-improved implementation of Simon-64/128 where we tried to fix all pipeline leakages. The t-test was performed using measured traces (acquired with the hardware setup that was also used for the t-test shown in Fig. 2) in a fixed-vs-random setting. As can be seen in Fig. 6, this implementation is still not entirely leakage-free, but the t value exceeds the threshold of 4.5 only insignificantly compared to the naive implementation in Fig. 4. Performing the t-test with this implementation on simulated traces did not show any leakage anymore, i.e. the t value was always well below the threshold of 4.5. Therefore, an implementer can conveniently use *MAPS* in the early stages of the leakage elimination process until the t-test on simulated traces is free of leakage. The final step is then the “fine-tuning” of the implementation until the t-test on measured traces does not show any leakage anymore. However, thanks to *MAPS*, an implementer needs to measure traces only at the very end of the implementation process, but not in the early stages of the implementation, which significantly reduces the development time.

With our setup, the measurement of power traces took roughly eight hours for 8,000 encryptions with a fixed input and 8,000 encryptions with a random input. Each encryption was repeated eight times and then averaged to reduce the noise. On the other hand, obtaining simulated traces with *MAPS* for 8,000 fixed-input/random-input encryptions took just 1.2 seconds altogether, which is 24,000 times faster than the eight hours we needed to measure the traces.

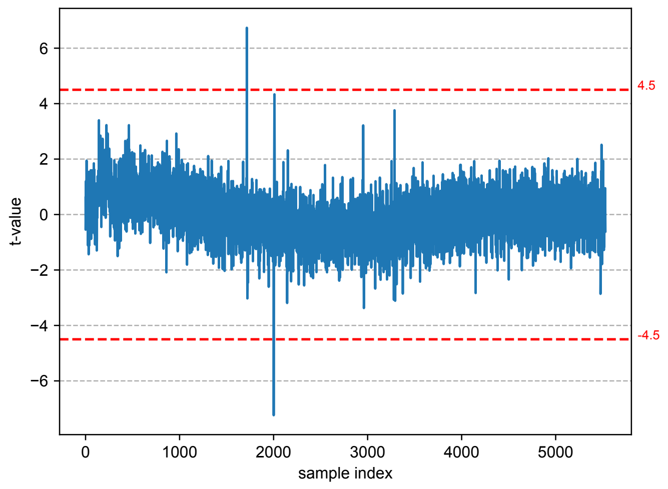

Fig. 6. Leakage assessment of the further-improved implementation of Simon-64/128 (all pipeline leakages corrected) based on measured power traces

6 Conclusions and Future Work

In this paper, we presented the design of *MAPS*, a simulator for fast leakage assessment of cryptographic software on ARM Cortex-M3 processors, which are widely used in the embedded domain. We demonstrated that our simulator can greatly speed up the implementation of masked block ciphers by identifying the architecture-specific leakages early in the development phase. Furthermore, we analyzed Cortex-M3-specific leakages introduced by the pipeline registers and showed that they are significant. In this way, we contribute to a better understanding of which micro-architectural properties and features of a Cortex-M3 processor actually cause the leakage that can be exploited in a DPA attack. We also provided a number of guidelines on how to take the pipeline leakages into consideration when developing a masked implementation of a cipher.

Our approach to analyze architecture-specific leakages can be easily applied to other targets without the need of complex profiling procedures, provided the HDL code of the processor is available. A possible candidate is Cortex-M0 since it is also part of the *DesignStart Pro Academic* program. The simulation speed may be further improved by optimizing the t-test implementation following the recent proposal of Reparaz et al. [20].

References

1. ARM Limited. ARM v7-M Architecture Reference Manual. Available for download at `http://static.docs.arm.com/ddi0403/eb/DDI0403E_B_armv7m_arm.pdf`, 2010.

2. ARM Limited. Procedure Call Standard for the ARM Architecture. Available for download at `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042f/IHI0042F_aapcs.pdf`, 2015.
3. Y.-J. Baek and M.-J. Noh. Differential Power Attack and Masking Method. *Trends in Mathematics*, 8(1):53–67, June 2005.
4. J. Balasch, B. Gierlichs, V. Grosso, O. Reparaz, and F. Standaert. On the cost of lazy engineering for masked software implementations. In M. Joye and A. Moradi, editors, *Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers*, volume 8968 of *Lecture Notes in Computer Science*, pages 64–81. Springer, 2015.
5. R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers. The SIMON and SPECK Lightweight Block Ciphers. In *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, pages 175:1–175:6. ACM, 2015.
6. G. Becker, J. Cooper, E. DeMulder, G. Goodwill, J. Jaffe, G. Kenworthy, T. Kouzminov, A. Leiserson, M. Marson, P. Rohatgi, and S. Saab. Test Vector Leakage Assessment (TVLA) Methodology in Practice. In *International Cryptographic Module Conference*, 2013.
7. A. Biryukov, D. Dinu, and J. Großschädl. Correlation Power Analysis of Lightweight Block Ciphers: From Theory to Practice. In M. Manulis, A. Sadeghi, and S. Schneider, editors, *Applied Cryptography and Network Security - 14th International Conference, ACNS 2016, Guildford, UK, June 19-22, 2016. Proceedings*, volume 9696 of *Lecture Notes in Computer Science*, pages 537–557. Springer, 2016.
8. S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In M. J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.
9. J.-S. Coron, J. Großschädl, M. Tibouchi, and P. K. Vadnala. Conversion from Arithmetic to Boolean Masking with Logarithmic Complexity. In G. Leander, editor, *Fast Software Encryption - 22nd International Workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, Revised Selected Papers*, volume 9054 of *Lecture Notes in Computer Science*, pages 130–149. Springer, 2015.
10. G. Gagnerot. *Étude des attaques et des contre-mesures assoccées sur composants embarqués*. PhD thesis, Université de Limoges, 2013.
11. L. Goubin and J. Patarin. DES and Differential Power Analysis (The "Duplication" Method). In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 1999.
12. Y. Ishai, A. Sahai, and D. A. Wagner. Private Circuits: Securing Hardware against Probing Attacks. In D. Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, 2003.
13. P. C. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In M. J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

14. T. Le, C. Canovas, and J. Clédière. An Overview of Side Channel Analysis Attacks. In M. Abe and V. D. Gligor, editors, *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2008, Tokyo, Japan, March 18-20, 2008*, pages 33–43. ACM, 2008.
15. S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks - Revealing the Secrets of Smart Cards*. Springer, 2007.
16. S. Mangard, N. Pramstaller, and E. Oswald. Successfully Attacking Masked AES Hardware Implementations. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2005.
17. 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.
18. K. Papagiannopoulos and N. Veshchikov. Mind the Gap: Towards Secure 1st-Order Masking in Software. In S. Guilley, editor, *Constructive Side-Channel Analysis and Secure Design - 8th International Workshop, COSADE 2017, Paris, France, April 13-14, 2017, Revised Selected Papers*, volume 10348 of *Lecture Notes in Computer Science*, pages 282–297. Springer, 2017.
19. O. Reparaz. Detecting Flawed Masking Schemes with Leakage Detection Tests. In T. Peyrin, editor, *Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers*, volume 9783 of *Lecture Notes in Computer Science*, pages 204–222. Springer, 2016.
20. O. Reparaz, B. Gierlichs, and I. Verbauwhede. Fast Leakage Assessment. In W. Fischer and N. Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 387–399. Springer, 2017.
21. H. Seuschek, F. De Santis, and O. M. Guillen. Side-channel leakage aware instruction scheduling. In M. Brorsson, Z. Lu, G. Agosta, A. Barenghi, and G. Pelosi, editors, *Proceedings of the 4th Workshop on Cryptography and Security in Computing Systems (CS2@HiPEAC 2017)*, pages 7–12. ACM Press, 2017.
22. E. Trichina, T. Korkishko, and K.-H. Lee. Small Size, Low Power, Side Channel-Immune AES Coprocessor: Design and Synthesis Results. In H. Dobbertin, V. Rijmen, and A. Sowa, editors, *Advanced Encryption Standard - AES, 4th International Conference, AES 2004, Bonn, Germany, May 10-12, 2004, Revised Selected and Invited Papers*, volume 3373 of *Lecture Notes in Computer Science*, pages 113–127. Springer, 2005.
23. N. Veshchikov. SILK: High Level of Abstraction Leakage Simulator for Side Channel Analysis. In M. D. Preda and J. T. McDonald, editors, *Proceedings of the 4th Program Protection and Reverse Engineering Workshop, PPREW@ACSAC 2014, New Orleans, LA, USA, December 9, 2014*, pages 3:1–3:11. ACM, 2014.
24. 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.
25. W. Zhang, Z. Bao, D. Lin, V. Rijmen, B. Yang, and I. Verbauwhede. RECTANGLE: a Bit-slice Lightweight Block Cipher Suitable for Multiple Platforms. *SCIENCE CHINA Information Sciences*, 58(12):1–15, 2015.