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 leakages are inferred from an analysis of the 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 for 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].

In light of the real-world threat posed by DPA, 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 countermeasure is masking [8, 11], which can be realized in both hardware and software. Masking aims to conceal every key-dependent variable with a random value, called “mask,” in order to break the link between the intermediate values that are computed on the device and the (unmasked) intermediate values of the algorithm. This principle is related to the idea of secret sharing since every sensitive variable is split 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 must be processed separately during the execution of the algorithm and then re-combined in the end to yield the correct result. The main attraction of masking is that its security can be formally proven in the framework of Isai, Sahai, and Wagner [12]. Despite the strong theoretical security guarantees, it turned out that masking is extremely challenging to implement in practice without introducing any unintended leakage. For example, it was shown in [16] that a masked hardware implementation of a block cipher can be broken by exploiting glitches at the output of logic gates. On the other hand, software implementations of masking can still be vulnerable to DPA attacks due to unintended violations of the Independent Leakage Assumption (ILA), which can result from certain micro-architectural effects or features [19]. Therefore, it is important to check whether a masked implementation of a cipher meets the theoretical security promises also in practice (i.e. does not show any DPA-exploitable leakage), which can be achieved by e.g. performing a leakage detection test [6] or mounting a full DPA attack.

Developing a masked software implementation of a block cipher is a tedious and highly iterative task. 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]. In order to decrease development time, one can use a power simulator, such as ELMO [17], to obtain power consumption traces that can be used for leakage assessment. However, to get realistic power traces, the simulator needs to take into account certain micro-architectural effects, such as inter-instruction dependencies in the power consumption (and, therefore, leakage) of a processor. For example, due to pipelining effects, the power consumption of a given instruction does not only depend on the operands/results and from/to which registers they are read/written, but also on the preceding instructions that are in the pipeline at the same time. ELMO takes such 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 consists of three stages.

While ELMO is a useful tool, it has a few shortcomings. In particular, getting realistic instruction-level power models is a tedious 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 pipeline at the same time (i.e. the number of pipeline stages). This approach achieves promising results, as demonstrated through a number of experiments by the authors of [17], but is only viable for processors with few (e.g. up to three) pipeline stages. However, there exist embedded processors with five, seven or even ten pipeline stages, which makes it extremely costly to develop power models for groups of instructions. Our simulator, *MAPS* (Micro-Architectural Power Simulator), uses a different approach and takes the inter-instruction dependency of the power consumption into account by developing a more refined micro-architectural model of the target processor. In particular, *MAPS* models all pipeline registers and validates these models through simulations with the HDL description of the target micro-architecture. Therefore, *MAPS* has two major advantages over ELMO, namely (i) the power model does not require measurements, especially no measurements of inter-instruction dependencies, and (ii) *MAPS* is suitable for embedded processors with deep(er) pipelines of more than three stages.

Our contributions. Our first contribution is *MAPS* itself. To the best of our knowledge, it is the first open-source power simulator for leakage assessment targeting the Cortex-M3 architecture, a processor widely used in embedded products [18]. In addition to being fast and easy to use, it models the architecture specific leakages based on a structural analysis. As a second contribution, we analyze for the first time in the open literature the impact of the pipeline registers on the leakage of masked software.

2 State of the art

Over the years, many simulators have been developed; the interested reader can find a detailed survey in [24, Section 5.3]. We focus here on the most recent simulators that perform high-level simulation by opposition to analog or HDL simulators. Those low-level simulators, while being generally more accurate because they rely on source files (netlists, parasitic components, back-annotated delays) that are usually not publicly available, are extremely slow.

Gagnerot described in his thesis [10], published in 2013, a power simulator developed for leakage assessment of cryptographic implementations. It generates power traces by tracing all writes to the registers and buses of a complete system. The details of the system are not known since the project was conducted in collaboration with a private company. All we know is that it contained a 16-bit RISC processor, some UART interfaces, a DES and an RSA co-processor. Its input is a compiled binary object. The simulator and its souce code are not publicly available and, therefore, it is not known how detailed the modeling of the architecture is, i.e. whether it includes the pipeline registers or not.

SILK stands for Simple Leakage Simulator and was proposed in 2014 by Veshchikov [23]. It is not tied to a specific architecture and simulates the power traces at a high level of abstraction. However, its power model is very flexible and can be adapted to emulate many situations. It accepts a C source files as input. The source code is publicly available [1].

Reparaz [20] presented a simulator in 2016. The input is a C high-level description. The values of the intermediate variables are traced after the implementation has been compiled with a modified version of a LLVM compiler. Therefore it is not tied to a specific architecture. Yet, it is fast and provide debugging capabilities allowing to pinpoint easily the source of the leakages.

ELMO (Emulator for Power Leakage for Cortex M0) has been introduced by McCann et al. [17] in 2016. It is dedicated to the Cortex-M0 processor and takes a compiled binary object as input. It is based on an existing ARM v6-M emulator that has been "back-annotated" with leakage information. The leakage information has been extracted using elaborated statistical processing that was applied to extensive measurements performed on a hardware setup. Hence, ELMO belongs to the category of profiled simulators. Because of the limitations of the underlying simulator, it does not support the Thumb-2 instruction set. The leakages reported by ELMO can be very accurate since the hardware measurements include leakage effects such as glitches or coupling. However, adding a new target to ELMO is very challenging as it requires a fully debugged hardware setup and the statistical processing itself depends on the characteristics of the target architecture and instruction set such as the 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 designed by ARM that implements the version v7-M [1] of the ARM instruction set. It is a very popular and successful platform for embedded products because of its efficient and compact instruction set.

[1] `https://github.com/nikita-veshchikov/silk`

[2] `https://github.com/bristol-sca/ELMO`

The Cortex-M3 has a Harvard architecture with 16-bit and 32-bit instructions and a 32-bit data path. It does not include a data cache and a prefetch buffer replaces a more complex instruction cache. Like other 32-bit ARM processors, the Cortex-M3 contains 16 registers, split in 13 general-purpose registers (`r0-r12`), a stack pointer (`r13`), a link register (`r14`), and a program counter (`r15`).

Arithmetic and logic instructions operate only on registers. A barrel shifter located between the register file and the Arithmetic and Logic Unit (ALU) allows to combine a shift or a rotation of the second operand with an ALU instruction in the same cycle. All ALU operations execute in one clock cycle except `mul` (multiply), `div` (divide) and operations targeting the program counter.

The pipeline is made of three stages. In the first stage, the instruction is fetched from the instruction memory. Then, the instruction is decoded in the second stage. Finally, the instruction is executed in the third stage. Conditional branches are speculated (i.e. one of the alternative instruction is speculatively executed and canceled if the other alternative was actually chosen).

Store to memory instructions like `str` are buffered and thus executed in one cycle, while load from memory instructions `ldr` introduce one wait-state. The typical clock-per-instruction (CPI) figure for embedded software is close to 1.

3.2 Cortex-M3 HDL analysis

The Cortex-M3 processor is described in a set of source files written in Verilog. Those source files are available to academia through the *DesignStart Pro Academic* program. The package contains the HDL description of the processor and a minimal system. The minimal system connects the core to the memories with AMBA (Advanced Microcontroller Bus Architecture) buses. It also adds a set of peripherals like communication and debugging interfaces that allow to extensively trace what happens during the execution of a program. By default, the Verilog simulation of the minimal system loads and executes a C program cross-compiled for ARM v7-M architecture.

Since we have access to the HDL 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 also be captured since those registers also define the pipeline stages.

All registers in the core can be found by looking for signals defined with the Verilog keyword *reg* and assigned in a Verilog *always @(posedge <clock>)* block. Out of those registers, only the ones related to a manipulation of the data are interesting from a leakage-detection point of view. We can further discriminate by selecting the registers with a width of 32 bits. Moreover, the ALU exclusively operates on register operands, so only the 32-bit registers linked with the two output ports of the register file have to be analyzed.

With those criteria, the 16 registers `r0-r15` of the register file, two registers `ra` and `rb` located between the register file and the ALU, and three registers inside the ALU are retained. Nonetheless, the three registers inside the ALU are only used during multi-cycle ALU instructions such as `mula` or `div`. Since those

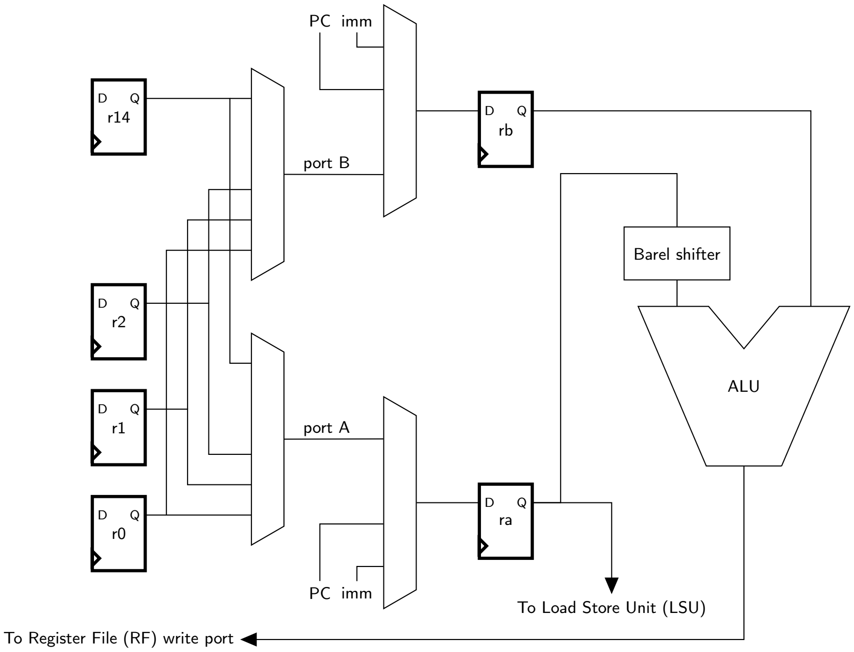

Fig. 1. Simplified structure of the Cortex-M3 pipeline

instructions are rarely found in the coding of symmetric primitives, we decided to not trace those registers. The program counter `r15` is also not traced by default in order to limit the length of the power traces. The first requirement of a secure implementation is that it must be constant flow anyway.

The registers `ra` and `rb` are the pipeline registers isolating the decoding stage from the execution stage. Their existence and location could have been inferred from the fact that an ALU instruction may be executed while the next instruction may access the registers. However, the analysis of the HDL code confirms their exact location and also specifies what value are assigned to them for each instruction. A simplified version of the Cortex-M3 pipeline is depicted in Fig. 1.

3.3 Cortex-M3 pipeline leakages

The registers `ra` and `rb` are specific to the Cortex-M3 pipeline architecture. They are a possible source of leakage since they combine the operand values of consecutive instructions. Indeed, the power consumption associated with writing those registers is related to the Hamming distance between the current operand value and the previous value.

Both the first operand and the second operand of the ALU instructions may be affected. Since `ra` is connected between the register file and the barrel shifter, even an instruction with a shifted or rotated second operand will be affected.

Listing 1. Code fragment with 2^{nd} operand leakage

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

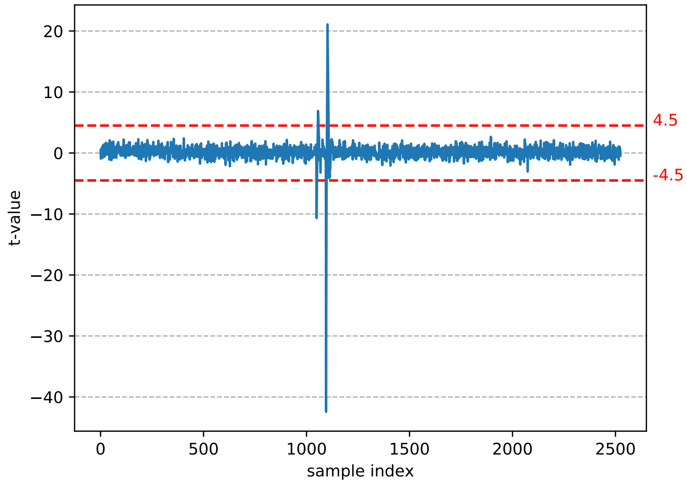

Fig. 2. 2^{nd} operand leakage, hardware measurements

Register Transfer Notation 1. Equivalent to Listing 1

```
$\mathtt{rb} \leftarrow \mathtt{r4}$
$\mathtt{ra} \leftarrow \mathtt{r2}$
$\mathtt{r6} \leftarrow \mathtt{rb} \wedge (\mathtt{ra} \ll 4)$
$\mathtt{rb} \leftarrow \mathtt{r5}$
$\mathtt{ra} \leftarrow \mathtt{r3}$    $\triangleright$ $Power(\mathtt{ra}) = HW(\mathtt{r2} \oplus \mathtt{r3})$
$\mathtt{r6} \leftarrow \mathtt{rb} \vee (\mathtt{ra} \ggg 5)$
```

Listing 1 illustrates such a leakage. In this fragment, `r2` and `r3` are the two shares; the secret is $(\mathtt{r2} \oplus \mathtt{r3})$. Registers `r4` and `r5` contain random values unrelated with any other register values. From an algorithmic view, there should be no leakage. However, the measurements on an actual chip show that there is a leakage, as observed on Fig. 2. The measurements were performed on an Atmel Cortex-M3 SAM3X8E chip with a Langer EM probe connected to a Lecroy WR8254M oscilloscope sampling at 500 MSamples/s.

The leakage can be explained by explicitly writing the registers transfers involving `ra` and `rb`. Listing 1 is equivalent to the Register Transfer Notation 1. As expected, $(\mathtt{r2} \oplus \mathtt{r3})$ is leaking.

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

```
rb ← r0
ra ← r2
rb ← r0
ra ← r3                                      ▷ $Power(\texttt{ra}) = HW(\texttt{r2} \oplus \texttt{r3})$
```

Basically, every instruction using a value read from a register is affected, not only the ALU instructions. For example, all memory store instructions will leak if scheduled one right after another, as in the Listing 2 and its equivalent Register Transfer Notation 2.

The leakage of the `str` instructions extends to the `push` instructions and to the store-multiple `stm` instructions since they are actually a shorthand for a sequence of `str` instructions.

3.4 Coding for Cortex-M3 pipeline leakage

The Cortex-M3 pipeline leakages can be circumvented in a few different ways, listed below in ascending order of their implementation cost:

1. simply swap the operands of commutative instructions.
2. schedule instructions so that the two shares are not processed by succeeding instructions. This might prove difficult because of the limited number of registers.
3. use more complicated versions of some instructions, so that the pipeline registers are written with unrelated values. For example "`mov r0`, 0" may be replaced with "`eor r0`, `rx`, `rx`" where `rx` may be any register. In the short version, the registers `ra` and `rb` are not written since the immediate value 0 is directly transferred from the instruction decoder to the register `r0`. In the long version, `ra` and `rb` are written with the value of `rx` before `r0` is cleared. The cost is two bytes of program memory at maximum, depending on which register`rx` is used.
4. in any case, the registers `ra` and `rb` may be set to a value unrelated to the data by the instruction "`orr r0`, `r0`, `r0`" if `r0` is unrelated to the sensitive data. For example, `r0` may be the address of an input buffer. The cost is one clock cycle and two or four bytes of program memory.

Note that inserting a `nop` instruction will not solve the leakage because the `nop` instruction does not propagate past the instruction decoder and hence does not modify the registers `ra` and `rb`.

4 Our simulator: *MAPS*

In this section we give 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 masked implementations of lightweight cryptographic primitives for the Internet of Things (IoT). The design goals are explained below.

Easy to use. Implementation and testing of a masked algorithm requires a good understanding of cryptographic engineering. Moreover, this highly iterative process requires a lot of time and scrutiny. Our simulator is easy to use for implementers and provides a convenient way for automated leakage assessment of cryptographic implementations. Hence, it considerably improves the whole development and testing process.

Only one set of source files. In the absence of a leakage simulator for Cortex-M3, one would resort to imulated leakages, which are typically generated using a modified or totally different implementation of the assessed algorithm. Having two different sources files, one for the simulation and one for hardware measurements, may lead to errors due to inconsistencies and adaptations required by either the simulator or the hardware. Therefore *MAPS* supports C and assembly implementations.

Fast simulation-debug cycle. Implementing a secure masked version of a cryptographic primitive is not an easy task as one has to work in assembly to have full control over the instructions that will be executed. The allocation of the registers and the selection of the operands may require several tries. Large simulation times do not allow the designer to try several "what-if" scenarios. Ideally, the complete cycle "write-compile-test" should take less than a few minutes.

Easy debugging. In this context, debug has two meaning. The first one relates to the debugging of the functionality at the first stage of the implementation process. Our simulator can interact with GDB through a GDB server. The second usage refers to identifying which instructions cause an information leakage. *MAPS* generates an index file linking the program counter and the power trace sample index which allows fast identification of the instruction that leaks.

Target-specific leakages. Our simulator reports the 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.

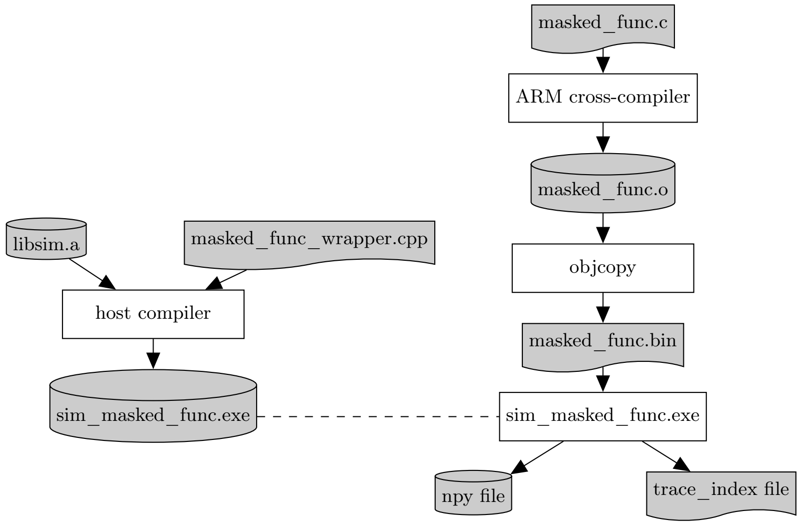

Fig. 3. *MAPS* flow

Open-source. *MAPS* is open-source software[3]. It may be used and modified without restrictions. Moreover, anyone can contribute to the further development of *MAPS* by adding support for other instructions or new features.

4.2 Simulation flow

A high-level view of the operation of *MAPS* is described in Fig. 3. First, a simulator executable, labeled *sim_ masked_func.exe* in Fig. 3, has to be produced. The executable is tasked with loading and simulating the function to be tested. 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 value and the new value is stored as a new sample in the power trace. The 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 the common command-line options are grouped into a library *libsim.a*.

The file *masked_func_ wrapper.cpp* contains the test functions and the interface functions. The interface functions wrap the call to the simulator engine so that the function to be tested appears like a host-domain function. It abstracts the process of passing parameters from the host to the simulated function. All

[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`.

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

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

The function to be tested is written in C in the file *masked_func.c*. It may use inline assembly and macros. It is cross-compiled for the ARM v7-M and converted into a 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. The Welch t-test is computed over the collected power traces and stored in a Numpy (.npy) file that can be easily visualized using Python. A trace index file is also generated. This file maps the t-test sample index to the simulated program counter so that the address of an instruction causing a leakage can be easily reported.

4.3 Validation

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

4.4 Limitations

In this subsection, we summarize briefly the limitations of our tool:

- only the Cortex-M3 target is supported.
- not all assembly instructions of the Cortex-M3 described in [1] are supported. The instructions that are not supported are: conditional instructions, table branch instructions, saturation instructions, multiply instructions, packing instructions, hint instructions. The unsupported instructions are unlikely to be found in an implementation of a lightweight cryptographic 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"` instruction leaks ($\mathtt{r2} - \mathtt{r3}$) on the actual hardware but does not leak on the simulator.
- only the processor is simulated. No peripheral or interface is modeled. Data can only be transferred between the host and the targets using the ABI and the target memory.
- the simulator traces only the registers of the Cortex-M3 *core*. Other registers that may be located outside of the core, such as in a memory interface, are not taken into account.
- the simulator is not cycle-accurate.

4.5 Performance

The speed performance of *MAPS* is summarized in Table 1. All test cases correspond to a fixed-vs-random Welch t-test as in [6] for one million measurements (i.e. two million executions of the simulated function). All tests are performed on a Intel i7-6700 processor running at 3.4 GHz. For comparison, we recall that the acquisition speed of the setup used for the DPA contest V4 implementing AES is approximately 0.9 traces/s [24].

Table 1. *MAPS* performance for three masked lightweight block ciphers (generation of one million traces)

Algorithm	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 considered for our evaluation first-order protected implementations of three lightweight block ciphers (i.e. Simon, Speck, and Rectangle), which are briefly described next.

Simon-64/128 [5] is a lightweight block cipher with an And-Rotation-Xor structure. The tested implementation is a 2-share masked implementation protected with the Trichina AND-gate [22].

Speck-64/128 [5] is also a lightweight block cipher but with an Addition-Rotation-Xor structure. The tested implementation is protected by a 2-share boolean masking. The modular addition is protected by a KSA scheme [9].

Rectangle-64/128 [25] is a bit-slice lightweight block cipher based on 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 showcase how our simulator can be used to code a secure version of Simon-64/128 on a Cortex-M3 processor. In the following, all figures are the result of a leakage assessment using a Welch t-test on the power waveforms generated by our simulator in a fixed-vs-random setting. For each experiment, 10,000 traces with fixed inputs and 10,000 traces with random inputs are collected.

First, Fig. 4 (a) shows a naive coding of Simon-64/128 masked using Trichina AND-gates [22]. A naive implementation minimizes the number of execution cycles and maps the intermediate steps of the computations to the next free register. Any Hamming distance effect due to the reuse of some registers is not taken into account and the simulator is not configured to trace the pipeline registers `ra` and `rb`. Unsurprisingly, the naive implementation leaks.

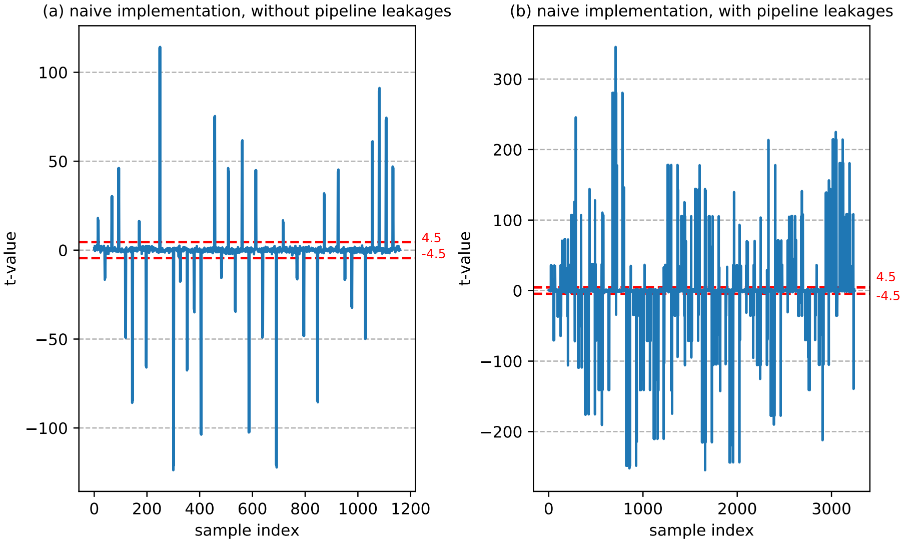

Fig. 4. Simon-64/128, naive coding, simulated (a) without and (b) with pipeline leakages

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

Next, the naive implementation is corrected to take into account the reuse of the registers. The corresponding leakages are depicted in Fig. 5 (a). The simulator is configured to not trace the registers `ra` and `rb`. As expected, the leakages seem to be fixed.

However, as demonstrated on Fig. 5 (b), the corrected implementation do leak through the pipeline registers when the tracing of `ra` and `rb` is enabled. Actually most of the leakage comes from the pipeline registers.

Table 2. Number of instructions for each masked implementations of Simon-64/128

Version	Number of instructions	Increase factor
(1) naive	1106	1.00
(2) corrected for register reuse	1194	1.08
(3) corrected for pipeline registers	1285	1.16

Table 2 lists the number of instructions executed by the three implementations. The implementation (1) is the naive implementation and implementation (2) is the naive implementation corrected for the register reuse leakage effects. The implementation (3) corrects the implementation (2) for pipeline leakage using the methods given in Section 3.4. Please note that the number of instructions

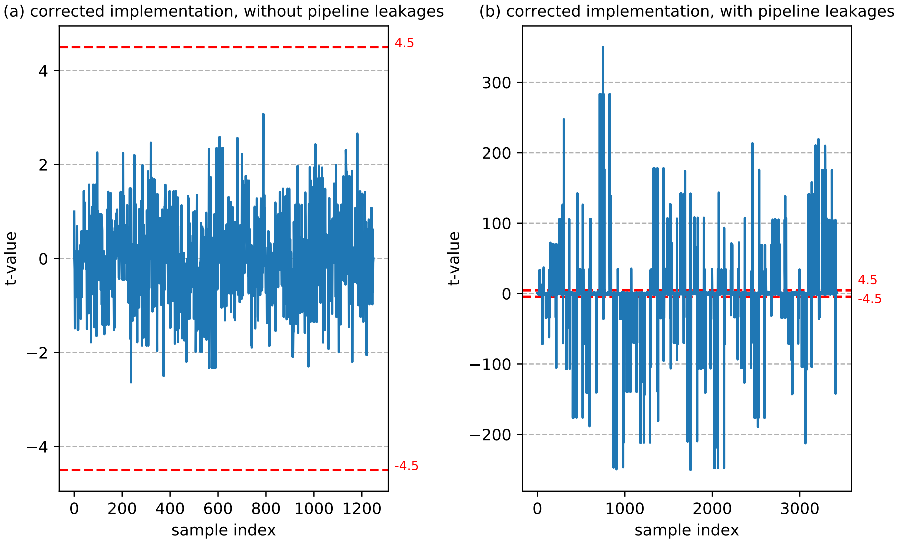

Fig. 5. Simon-64/128, corrected coding, simulated (a) without and (b) with pipeline leakages

differs from the number of cycles. For example, replacing one `stm` instruction by several `str` instructions does not add any cycle.

Finally, Fig. 6 shows the result of the t-test for a further improved implementation of Simon-64/128 where we tried to fix all pipeline leakages. The t-test was performed with measured traces (acquired with the same hardware setup as Fig. 2) in a fixed-vs-random setting. As can be seen in Fig. 6, the implementation is still not completely free of leakage, but the t value exceeds the threshold of 4.5 only slightly compared to the naive implementation in Fig. 2. 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. Consequently, an implementer can 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 also 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 phase, but not in the early stages, which significantly reduces the development time. With our set-up, the measurement of traces took 8 hours for 8,000 encryptions with a fixed input and 8,000 encryptions with random inputs. Each encryption was repeated 8 times and then averaged to reduce the noise. On the other hand, obtaining simulated power traces with *MAPS* for 8,000 encryptions with a fixed input and 8,000 encryptions with random inputs took only 1.2 seconds, which is more than 24,000 times faster than the 8 hours we needed to obtain the measured power traces.

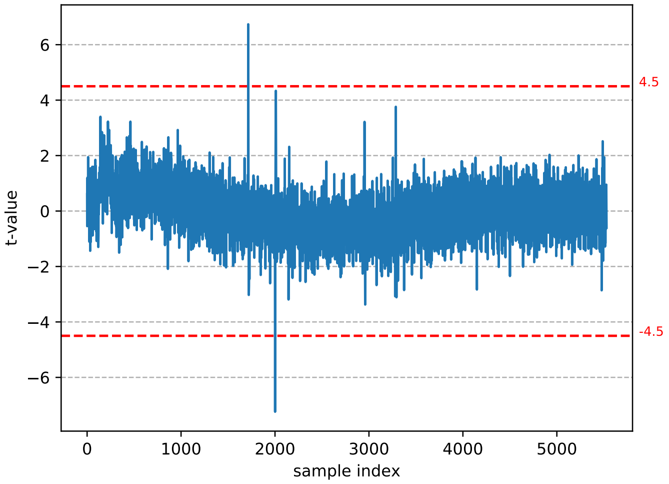

Fig. 6. Simon-64/128, pipeline leakages corrected, measurements on hardware setup

6 Conclusion 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 for IoT applications. We demonstrated that our simulator can significantly speed up the implementation of masked primitives by identifying the architecture-specific leakages very early in the development cycle. We analyzed the Cortex-M3 specific leakages caused 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 processor actually introduce the leakage an attacker can exploit in a DPA. We also provided guidelines on how to take the pipeline leakages into consideration when developing a masked implementation of a cipher.

Our method to analyze the architecture specific leakages can be easily applied to other targets without requiring 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 also improved by optimizing the t-test implementation following the recent proposal of Reparaz et al. [21].

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):1–15, 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. National Institute of Standards and Technology (NIST). Public Comments Received on "Profiles for the Lightweight Cryptography Standardization Process". Available for download at `http://www.nist.gov/sites/default/files/documents/2017/06/20/public-comments-profiles-i-ii-june2017.pdf`, June 2017.
19. 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.
20. 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.
21. 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.
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.