ARMISTICE: Microarchitectural Leakage Modeling for Masked Software Formal Verification

Arnaud De Grandmaison, Karine Heydemann, Quentin L. Meunier

ARMISTICE: Micro-Architectural Leakage Modelling for Masked Software Formal Verification

Arnaud de Grandmaison, Karine Heydemann, Quentin L. Meunier

Abstract—Side channel attacks are powerful attacks for retrieving secret data by exploiting physical measurements such as power consumption or electromagnetic emissions. Masking is a popular countermeasure as it can be proven secure against an attacker model. In practice, software masked implementations suffer from a security reduction due to a mismatch between the considered leakage sources in the security proof and the real ones, which depend on the micro-architecture. We propose ARMISTICE, a framework for formally verifying the absence of leakage in first-order masked implementations taking into account modelled micro-architectural sources of leakage. As a proof of concept, we present the modelling of an Arm Cortex-M3 core from its RTL description and leakage test vectors, as well as the modelling of the memory of a STM32F1 board, exclusively using leakage test vectors. We show that, with these models, ARMISTICE pinpoints vulnerable instructions in real world masked implementations and helps the design of masked software implementations which are practically secure.

Index Terms—Side Channel Attacks, Masking, Verification, Micro-architectural Leakage

I. INTRODUCTION

Side Channel Attacks (SCA) exploit physical measurements, like power consumption or electromagnetic (EM) emissions, during the execution of an application to recover secret data. They constitute a powerful class of attacks, allowing to break software and hardware implementations of cryptographic algorithms otherwise proven secure at the algorithmic level.

Introduced in the early 2000s [1], masking countermeasures at order d aim at encoding a secret data into $d+1$ parts called *shares*, such that any combination of less than $d+1$ shares is statistically independent from the secret. This theoretically prevents SCA, as the power consumption and EM emissions are directly linked to the values manipulated by the program. A masking countermeasure can be proven secure against an attacker model, e.g. an attacker able to probe t measurements per execution, which leads to the notion of t-probing security (or probing security at order t) [1], [2]. Proofs are conducted at a given abstraction level, most often at algorithmic or source code level, and are based on a given leakage model. The latter is typically either the value-based leakage model, in which the leakages are the values of intermediate computations; or the transition-based leakage model, in which the leakages are combinations between two consecutive values in some

A. de Grandmaison was with ARM, Paris, France (email: arnaud.degrandmaison@arm.com)

K. Heydemann and Quentin L. Meunier were with Sorbonne University, CNRS, LIP6, FR-75005 France (emails: karine.heydemann@lip6.fr, quentin.meunier@lip6.fr)

elements, such as variables at algorithmic level or registers at assembly level.

Implementing a masking scheme at software level without unmasking secrets and detecting such unmasking by hand are far from trivial tasks. Consequently, some verification techniques and tools have been recently proposed to help designers detect flaws in their implementations [2]–[4]. We argue that verification should be carried out on the compiled code for two main reasons. To start with, compilers perform code transformations, from simplifications of the expressions to the reordering or removal of some instructions, possibly harming or deconstructing the carefully added masking scheme along the way. Then, proofs require the knowledge of the sources of leakage, and are then limited to what is visible at the chosen abstraction level. The lowering of the source code into assembly code introduces data transfers between memory and CPU registers manipulated by instructions, and the leakage relative to successive writes into the same architectural register and consecutive memory accesses are most often not related to the same source-level variable. Some recent work propose some compilation approaches for removing such issues [5], [6]. There also exist some tools for verifying masked assembly code [7], [8]. However, analysis at the ISA level is still not sufficient: masked software implementations suffer from a security reduction due to the mismatch between the considered sources of leakage and the real ones, which depend on the target micro-architecture and may not be visible at the ISA level [9], [10]. For example, some remnant effects may appear, or some internal resources, like pipeline registers or buses, can lead to unmasking.

Yet higher-order masking is supposed to be effective against more powerful attackers, with a major impact on performances and code size. However, using it in an attempt to shield from the architecture implementation details is not a solution [11]. A promising approach is to take into account leakage at processor design time [12], [13], but no such leakage-free architecture is available yet. Accurate modelling of leakage sources for an actionable verification of masked software running on generic off-the-shelf processors is still an open research question.

An actionable verification needs to be conducted with much more details in the processor micro-architecture, in order to accurately detect all possible data combinations in a processor implementation and pinpoint to the precise location and root cause of a leakage. Existing approaches [14]–[16] all have limitations regarding the considered sources of leakage. In this article, we propose the ARMISTICE framework to fill the gap between formal verification approaches and real sources of leakage. From an application running on a processor model,

ARMISTICE extracts all intermediate values computations as symbolic masked expressions and formally verifies the absence of leakage in them. As proof of concept, we build a model of the Cortex-M3, by studying the RTL available through the Arm Academic Access[1] (AAA) program. The *white-box* modelling enabled by analysing the micro-architecture RTL description is experimentally verified with specifically designed leakage test vectors, to both confirm the model correctness and rank the leakage sources according to their impacts. Some areas of the model, related to the memory subsystem, require a *black-box* approach only based on dedicated leakage test vectors. We show that ARMISTICE is able to recover where and what leaks on two different masked AES implementations as well as on 5 other benchmarks. We also show experimentally on two of these masked implementations the accuracy of the found leakages on a real hardware target and how ARMISTICE can help designers remove them.

This paper makes the following contributions:

- ARMISTICE, a framework for formally verifying first-order masked implementations at binary level, and precisely pinpointing where and why secret data leak; this makes ARMISTICE an invaluable and actionable tool for designers to verify masked implementations and fix them if need be.
- A model of the Cortex-M3 with meaningful and additional details to some recent papers (e.g. [17], [18]) that studied the same processor.
- A qualitative and quantitative analysis conducted with dedicated leakage test vectors of the Cortex-M3 leakage sources, including an in-depth study of the memory subsystem.

The paper is organized as follows: Sec. II gives some background and experimentally motivates this paper; the modelling of the Arm Cortex-M3 processor core and memory subsystem is detailed in Sec. III; Sec. IV and Sec. V present respectively the ARMISTICE framework and experimental results before discussing the closest related work in Sec. VI and concluding in Sec. VII.

II. BACKGROUND AND MOTIVATING EXAMPLE

While real attacks can be attempted on a device, this is not so practical when assessing the device's immunity to side channel leakages. Practitioners have thus proposed several leakage assessment methods: some based on formal methods, some based on statistical analysis. Formal verification approaches of either a masked algorithm or a masked software or hardware implementation are all based on a symbolic representation of the variables or input data, which are tagged either secret, public or mask [2], [3], [7], [19]. The verification consists in checking that a set of symbolic expressions, built upon these symbolic variables and representing internal values in the program or circuit, has a distribution which is independent from all the secret variables. Such a check can be performed using either a type inference [7], or an iterative substitution process trying to soundly remove secrets from the expressions [3], [19]. Regarding statistical approaches, Test

Value Leakage Assessment [20] (TVLA) is probably the most popular one. It has two variants: specific and non-specific. For both variants, two sets of traces are compared. In the non-specific case, one set is generated with a fixed (secret) data, and the other with random (secret) data. A t-test analysis on these sets allows to detect any possible leakage without any assumption on the leakage model, i.e. which part of the computation is leaking in which part of the implementation. The non-specific t-test requires using a specific randomly-interleaved procedure in order to avoid false-positive results [21]. In the specific t-test, the traces are split into two sets according to the leakage model (e.g. hamming weight) of a known intermediate value. Other statistical tools are available to practitioners, like the Pearson correlation [22] which is often used in place of the specific t-test.

Let us now consider the proven first-order probing secure masking scheme from Ishai, Sahai and Wagner (ISW) [1] and apply it to a software boolean AND computation. `isw_and`, shown in C language in Lst. 1, takes as inputs two secret values `a` and `b`, each split across two shares `a0` and `a1` (resp. `b0` and `b1`). It produces as a result a secret value `c`, which corresponds to `a & b`, split across 2 shares `c0` and `c1`, without ever exposing any secret value. As pointed out in the introduction, compilation can harm the carefully designed masking scheme. In this case, `GCC` does simplify the expressions, breaking the ISW scheme altogether. Some intermediate computations are thus protected with the `enforce` macro which forces `GCC` to materialize the intermediate value in a register rather than leaving the expression open to optimisations.

Listing 1. ISW AND
```
1  // Inputs: - Secrets a = a0 ^ a1, b = b0 ^ b1
2  //         - Mask m
3  // Output: - Secret c = c0 ^ c1
4
5  #define enforce(r) __asm__ __volatile__ \
6          ("@enforce\n\t" : "+r" (r) ::)
7
8  aux0 = m ^ enforce(a0 & b1);
9  aux1 = aux0 ^ enforce(a1 & b0);
10 c0 = (a0 & b0) ^ m;
11 c1 = (a1 & b1) ^ aux1;
```

Listing 2. ISW AND (GCC assembly output)
```
1  ; r0:a0, r1:b0, r2:a1, r3:b1, r6:c[] r7:m
2  and.w   r4, r0, r3   ; a0 & b1
3  eors    r4, r7       ; aux0 = (a0 & b1) ^ m
4  and.w   r5, r2, r1   ; a1 & b0
5  ands    r0, r1       ; a0 & b0
6  ands    r3, r2       ; b1 & a1
7  eors    r4, r5       ; aux1 = aux0 ^ (a1 & b0)
8  eors    r0, r7       ; c0 = (a0 & b0) ^ m
9  eors    r4, r3       ; c1 = aux1 ^(a1 & b1)
10 str     r0, [r6, #0]
11 str     r4, [r6, #4]
```

Lst. 1 is compiled, with size optimisations for example, for the Arm Cortex-M3 processor, a common processor in embedded devices. As our implementation is masked using the ISW scheme, and protected from compiler optimisations, one can check for himself that the secret values `a`, `b` and `c` do not appear in Lst. 2. Moreover, formal verification with our own ARMISTICE framework confirms the absence of leakage at the ISA level, in the value- and transition-based models.

With power traces captured on a STM32F1 target board

---

[1]The Arm Academic Access (AAA) is a free-of-charge program that provides access to an expanding range of Arm IP for academic purposes only: https://www.arm.com/academicaccess
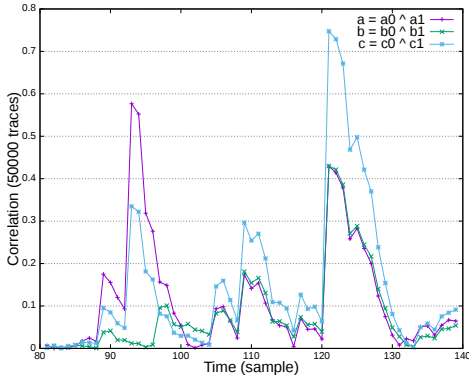
Fig. 1. Pearson correlation results for Lst. 1.

using the experimental setup from Sec. III-B1, a Pearson correlation [22] using a (a.k.a a0^a1), b (a.k.a b0^b1) and c (a.k.a a&b) as intermediate values reveals, as shown in Fig. 1, that all secret values are leaked during the execution of the `isw_and` function. Some of the peaks can be explained, e.g. the two consecutive `str` instructions (lines 10-11) are likely to leak the transition of the data written to memory [17], [23], explaining partly the peak on the blue curve at samples 120 - 128 in Fig. 1. One can note that leakages are increasing again after sample 132, although the ISW AND sequence is over: this illustrates the fact that despite being no longer explicitly used by the source code, the secrets a, b and c continue to leak due to *some* remnant effect. The Pearson correlation metric is used in this example because it makes the leakage on c more visible than with a t-test, which is affected by the low probability of having more than 16 bits set in c due to the underlying boolean AND operation.

Despite the apparent simplicity of Lst. 1, most other peaks do not have a clear explanation from the literature, motivating the work described in this article: understanding the Cortex-M3 micro-architecture in enough details so that it can be modelled and used with a formal analysis tool to provide clear explanations for these leakages.

III. CORTEX-M3 MODELLING

In this section, we present the modelling of the Arm Cortex-M3 data path. We also discuss the leakage profile from off-core components, i.e. memory related components.

A. Abstract Model

As we have access to the Cortex-M3 RTL description through AAA, we go for a white-box approach to model the processor micro-architecture, which involves Verilog source files eye-ball analysis and Verilog simulations.

1) Under the Processor's Hood: The Cortex-M3 implements the ARMv7-M ISA composed of 32-bit and 16-bit instructions. It is a 3-stage pipeline processor comprising a Fetch, a Decode and one or two Execute stages depending on the instruction. As the Fetch stage does not manipulate data, we do not consider it further in the context of this paper.

The main components found in the Cortex-M3 core, depicted in Fig. 2, are:
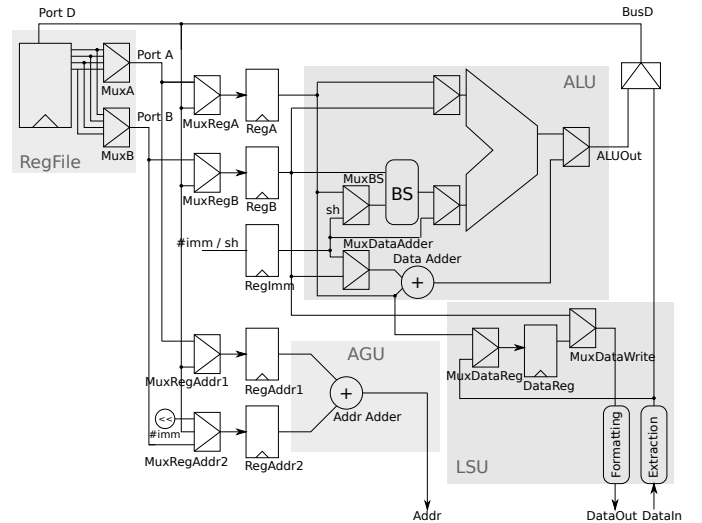

Fig. 2. Data path of the Cortex-M3 processor core

- A register file (RegFile), containing the general purpose registers (GPR), that has two read ports, named `port A` and `port B`, and one write port, named `port D`.
- An arithmetic logic unit (ALU), to perform actual computations on data stored in micro-architectural registers `RegA`, `RegB` and `RegImm`. As most ALU instructions allow shifting the second operand, a barrel shifter (BS) can optionally be used on `RegB`'s output to the ALU.
- An address generation unit (AGU), with a dedicated adder (named `Addr Adder` in Fig. 2), to compute the address used for accessing memory, from two micro-architectural registers `RegAddr1` and `RegAddr2`, containing the base address and the offset respectively.
- A load / store unit (LSU), to deal with data sent to or received from memory, like extracting the relevant part of the received data. The memory is accessed over the AHB-Lite bus, which is pipelined with an address-phase followed by a data-phase one cycle later.

2) How Instructions Use the Micro-architecture: We only consider in this description simple instructions typically used in cryptographic implementations: simple ALU instructions and memory accesses. We also assume that there is a single execution path regardless of the secret input values. This requirement is usually met to avoid timing side channel attacks and is recommended by all secure coding rules. As a consequence, no jump nor branch can depend on secret data.

After being fetched, an instruction goes through the decode stage, which prepares the data subsequently processed by the execute stage. It retrieves operands in the register file or the instruction itself and updates `RegA`, `RegB`, `RegImm`, `RegAddr1` and `RegAddr2`. Depending on RTL-implementation choices, read ports and internal registers that are not used by the current instruction can either keep their previous values, be reset to some default value, or get a value related to a bit field of the instruction which is not semantically relevant. For example, the decoding of the instruction `mov.w Rd,#imm`, which has only an immediate as source operand, selects PC on port A and $R_{imm[3:0]}$ on port B. `RegA` is written, while `RegB` is not. The choice of updating or not an unused register or read port,

and in which way, takes into account 1) the requirement of maintaining an understandable code by the RTL developers and 2) some performance, cost or area trade-offs. In the next section, we illustrate the consequences of these choices regarding potential leakage.

Simple ALU instructions (e.g. add, eor, mov, sign extension or bit selection) are executed in one cycle during the execute stage. For most ALU operations, port A and RegA receive the first register operand, and port B and RegB the second one. This is not the case for pure shift operations for which port B/RegB receives the operand to shift (first operand) while port A/RegA receives the shift amount (second operand) — a necessary twist because the BS is located on RegB's output to support the optional shift of the second operand on ALU instructions. At the end of the execution cycle, the destination register is written into the RegFile through port D.

A single load or store instruction requires two execution cycles: during the first cycle (EXE1) the address is computed, while in the second cycle (EXE2), the data to write (resp. read) is sent to (resp. received from) memory. Two addressing modes are available for memory accesses: an immediate-offset addressing mode and a register-based one. In both cases, the offset is added to a base register (a GPR). The base register is read through port A and written both in RegA and RegAddr1. In case of a register offset, it is read through port B and written both in RegB and RegAddr2. Depending on the addressing mode for write memory accesses, the data sent to memory either goes to RegB and DataReg (immediate offset) or only goes to RegA (register offset). Moreover, a store with a register offset prevents any following instruction from passing the decode stage.

In case of data dependencies, forwarding mechanisms exist in order to avoid pipeline stalls. The ALU result or the data read from memory can be forwarded to RegA, RegB, RegAddr1 and RegAddr2 registers. In case of a data dependency between a load and a pipelined store with immediate offset (for instance ldr Rx, [Ry, Rz], str Rx, [Ry', #imm]), the read value is forwarded to DataReg.

B. Sources of Leakage

In this section, we present the leakage test vectors, or test vectors for short, designed in order to 1) confirm or disprove with experimental measurements our findings from the RTL analysis regarding the data path components involved in the execution of an instruction 2) quantitatively evaluate the leakage due to the different components and 3) analyse and model the memory subsystem using a black-box approach.

1) Experimental Setup: All measurements are performed on NewAE's ChipWhisperer Pro (CW1200) with a STM32F1 target board [24] embedding a Cortex-M3 configured to run at 7.37MHz. The CW1200 acquires four samples per CPU cycle. Depending on the test vector, we acquire from 10,000 up to 500,000 power consumption traces, each with different random inputs generated with a Mersenne Twister RNG.

2) Anatomy of a Test Vector: A test vector has several random inputs, named Opi and starts (resp. ends) with a preamble (resp. postamble). The actual payload of a test
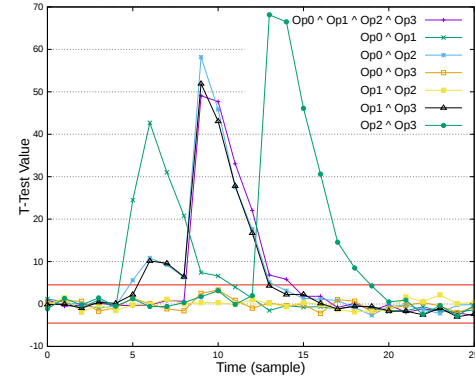

Fig. 3. Specific t-test results using the test vector of Lst. 3 using 20,000 traces.

vector consists in a small assembly instruction sequence that manipulates some random input values. The preamble achieves three goals: the loading of random input values into GPRs, their optional preparation (e.g. extracting a specific byte or masking some parts), and the reset to a specific state, zero or random, of all elements along a data path. The postamble's role is to ensure some form of quietness for a few cycles, in order to ease the experimental measurements and their subsequent analysis.

For each test vector, several specific t-tests are performed using the Hamming Weight of expressions composed of logical or arithmetic operations on the test vector's random input values (32-bit or less). A concluding specific t-test (i.e. with a t-value higher than 4.5) on the exclusive-or between expressions (e.g., input values) shows that there is a transition leakage induced by a micro-architectural component between these expressions.

An example is given in Lst. 3: it is composed of two exclusive-OR instructions (eor), each with different source GPR operands holding different random input values. The destination GPR operands are also distinct.

Listing 3. Test vector example
```
1        @ preamble
2        eor rDst1, rOp0, rOp1
3        eor rDst2, rOp2, rOp3
4        @ postamble
```

We perform a specific t-test on the traces resulting from the execution of the code given in Lst. 3 to measure the interaction between the consecutive values of the first (resp. second) operand of both instructions using the expression HW(Op0^Op2) (resp. HW(Op1^Op3)), where HW denotes the Hamming Weight of an expression. The value of this expression is directly linked to the power consumption of the port A and RegA (resp. port B and RegB). Thus, a high t-test value for this specific expression indicates a leakage in at least one of these two components. Similarly, we also measured the interaction between two consecutive values at the ALU output (or BusD) using the expression HW((Op0^Op1)^(Op2^Op3)).

The result of the specific t-test for the previous example is plotted in Fig. 3. In this figure, we clearly see the interaction between operands on the paths from the RegFile to RegA/RegB (curves named Op0^Op2 and Op1^Op3, samples 5-9) and once
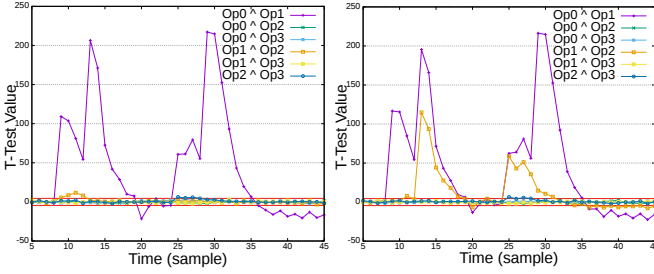
Fig. 4. Specific t-test results when placing a `ldr ROp2, [Raddr, #imm]` between two `eor Rdst, ROp0, ROp1`. On the left, `ldr` is 16-bit encoded; on the right it is 32-bit encoded. 200,000 traces were used.

latched in `RegA`/`RegB` (curves named `Op0^Op2` and `Op1^Op3`, samples 10-13). We can also see the interaction between the results of both instructions (curve named `Op0^Op1^Op2^Op3`, samples 10-13). As the preceding and following instructions manipulate constant values, the results of both instruction leak (curve `Op0^Op1`, samples 5-9 and curve named `Op2^Op3` at samples 14-18).

3) Invisible Source of Leakage at ISA Level: One major outcome of the RTL analysis is the discovery of some potential sources of leakage without any explicit link with data manipulated by instructions. In this section, we give some examples of such invisible leakage at ISA level.

We first illustrate the consequence of the choices of the decoding stage regarding read ports. As previously mentioned, the decode stage of a `mov.w Rd,#imm` instruction selects $R_{imm[3:0]}$ on `port B` without writing it in `RegB`. Using a dedicated test vector and a specific t-test with 50,000 traces, an interaction between the data previously read on port B and the content of the register $R_{imm[3:0]}$ is visible (max t-value of 9.1). This behaviour can reveal secret data, e.g., when the register $R_{imm[3:0]}$ contains one share of a first-order masked data and the second share has been read on `port B` in the previous cycle.

The encoding used for an instruction is also of critical importance, as the `RegA` and `RegB` write enable signals may depend on the instruction encoding. For example, the decoding of the instruction `ldr ROp2, [Raddr, #imm]` writes `ROp2` into `RegB` if and only if the instruction uses a 32-bit encoding. Fig. 4 shows specific t-test results when running a 16-bit or a 32-bit encoded load instruction. We can see that a 32-bit encoding leads to a write of the destination register content into `RegB` as there is an interaction between `Op1` and `Op2` visible at samples 13-18 and 25-31. Considering that `RegB` is always written with the destination register of a load instruction with immediate offset may lead to miss potential interactions between the values written into `RegB` before and after the load if the load is actually 16-bit encoded; alternatively, considering it is never written may lead to miss potential interactions between `Op2` and the values written into `RegB` before and after the `ldr` when it is 32-bit encoded.

Another important knowledge gained from the RTL is related to the forwarding mechanisms and internal registers. As an example, `DataReg` contains the value to be sent to the memory in case of a store with an immediate offset. If there is a dependence on this value with the preceding load instruction, the loaded value is forwarded to the `DataReg`. The

`RegB` register is however still written with the content of the register supposed to contain the value to be sent, but which is in this case the old value held in this register. This may be a source of leakage which can only be determined with the knowledge of both the issue time of instructions and the implemented forwarding mechanisms in the processor.

4) Leakage Test Vector Suite: We have carefully designed 77 test vectors: 31 devoted to the analysis of the data path components involved in each class of instructions, 5 devoted to forwarding mechanisms, 7 devoted to writing back into the register file and 34 devoted to the analysis of the LSU and memory. Test vectors enable us to assess and confirm the potential leakage on the different buses and internal registers as well as leakage due to memory transfers in the Cortex-M3 core of our STM32F1 target. Contrary to the ARM processor core whose RTL must not be modified (apart from selecting some configurable options), the memory subsystem implementation varies between targets with an identical core [17]: the (off-core) memory subsystem on our STM32F1 target is thus a black box. Test vectors dedicated to the memory and LSU enable us to model the memory subsystem.

Unsurprisingly, a small subset of our test vectors are similar to those from MIRACLE [17]. The RTL availability allows us to have an exact knowledge of the data manipulated by instructions. As a consequence, we design more specific test vectors in order to profile the leakage induced by the micro-architectural components and to explain where in-core leakage stems from. We cannot describe all the test vectors here but they are available online with their results [25]. We summarize the main results in the next section.

5) Leaking Components: Our analysis reveals that the RTL version of the Cortex-M3 we have access to is more recent than the one used for the implementation of the Cortex-M3 core in the STM32F1. For example, shifted operands are sent to `RegA` and not `RegB`. Each time a 32-bit instruction does not have a source register operand, `R0` is read in the register file on the corresponding read port (note that it is the default behaviour for the 16-bit instructions decoding). Moreover, some crossed leakages between `RegA` and `RegB` appear in case of consecutive ALU instructions with different operations. The abstract processor model used for the ARMISTICE tool, presented in the next section, reflects our findings.

Due to the CMOS technology, any component (bus, register, ...) is a source of transition leakage and can lead to the reveal of secret data. However the number of traces needed to see a transition leakage depends on the components. The more combinatorial logic the component is connected to, the less the number of traces. The table below gives the number of traces (starting with 10,000 traces) required to see, using a specific t-test, a transition leakage into a specific component. These values correspond to experiments with 32-bit manipulated data. More traces are needed when data have a smaller bit-width: e.g. 500,000 traces are needed to see a transition leakage due to consecutive writes of 8-bit values in a given GPR.
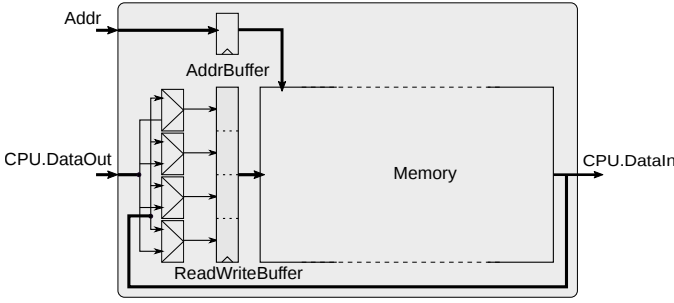
Fig. 5. Model of the memory subsystem of our STM32F1 target

#Traces	Leaking Components
10,000+	`RegA`, `RegB`, `RegAddr1`, `RegAddr2`, `AluOut`, `Addr Bus`, `Data Out`, `Data In`
50,000+	`Port A`, `Port B`, `DataReg`
200,000+	`Bus D`, `GPR`

From the RTL of the Cortex-M3 core, we know that memory is byte-addressed and the width of the data bus is 32 bits. Also, an aligned word is always loaded from the memory: when reading a byte (resp. half-word), the containing 32-bit word is sent to the CPU and then the LSU extracts the requested byte (resp. half-word). Regarding data writes into memory, a data whose width is less than a word is duplicated to form a 32-bit value to be sent to the memory. Our test vectors devoted to the analysis of the LSU and the black-box memory subsystem reveal several interactions from which we build a model of the memory subsystem, as depicted in Fig. 5:

- Read Side: There is a transition leakage between bytes at the same index in words loaded consecutively, be they separated or not by other non-memory instructions. Moreover, when bytes or half-words are requested, some leakage is visible between them with enough traces, even when they are not at the same word index. The data extraction in the LSU induces intra-word leakage when two load instructions are pipelined in the LSU: the data extraction is performed as soon as the extraction pattern is known, but the data arrival is at the end of the cycle. As a consequence, the extraction is performed during a small amount of time on the data read in the previous cycle. This provokes some intra-word leakage in the first loaded data.

- Write Side: Due to the duplication of data in case of byte (resp. half-word) write in the memory, a stored byte can interact with all other bytes (resp. aligned half-word) of the data sent to the memory in the previous cycle. When two store instructions are not issued in consecutive cycles, the stored bytes only interact with the ones at the same index in the previously read or written data. On our STM32F1 target, the `DataOut` bus seems to be reset between writes and there is, seemingly in the off-core memory, a one-word buffer which contains the last piece of data read or written, as explained next.

- Read Write Interaction: Consecutive memory instructions (load-store, store-load), even separated by non-memory instructions, do interact. As there are two data buses (`DataIn` and `DataOut`), the interaction takes place in the memory subsystem. Our test vectors enable us to confirm that there is a one-word buffer containing either the last read word

or the result of the last written word. When storing a byte or half word, the targeted aligned word is retrieved in the memory and the word resulting from the write (i.e. with one or two changed bytes) is written in this buffer. This behaviour is consistent with all the interactions we described before between two consecutive load or store instructions separated by non-memory instructions. Fig. 5 illustrates the inferred memory model and the one-word buffer.

This section has shown that each micro-architectural element can leak, some more obviously than others, i.e. requiring more or less traces. This is by no means specific to the Cortex-M3 though, and all processors are plagued with similar leakage sources. Despite the Cortex-M3 being a simple processor, manually checking all possibilities for securing a piece of code against side channels quickly becomes a daunting task for a developer, thus motivating the need for tooling to automate the task and manage the complexity.

IV. FORMAL VERIFICATION WITH A MICRO-ARCHITECTURAL MODEL

This section introduces ARMISTICE, a framework designed for formally proving the absence of leakage in a masked program when executing it on a specific processor core.

Overview: ARMISTICE implements different symbolic ARMv7-M instruction set simulators, denoted *processor models* in the remainder, targeting different abstraction levels such as ISA level or micro-architectural level. Such a symbolic simulator builds symbolic expressions representing the values passing through the hardware components of the processor when executing a masked program. The symbolic variables in expressions are typed as masks, secrets or public variables (e.g. plaintext). Other variables, whose values do not depend on the inputs (such as loop counters), have concrete values. These expressions provide an accurate model of the leakage due to the manipulated data, and proving that these expressions, or some combinations of expressions, are statistically independent from secret values gives a valuable guarantee regarding secret independence on a real execution. ARMISTICE leverages an external formal analysis tool to prove the absence of leakage in a given leakage model, and allows to avoid a statistical analysis or interpretation as would happen with a leakage trace simulator.

The verification process is depicted in Fig. 6. First, from a binary code, an entry point of the program to analyse and information on symbolic variables with their type, an execution trace is generated using an external symbolic execution engine. This step enables one to retrieve 1) the sequence of instructions executed by the program (we assume a single execution path i.e. no jump depends on a symbolic input) and 2) the symbolic content of relevant memory locations for the program just before its beginning. Then, the execution trace is simulated using the user-specified processor model which computes the symbolic expressions in hardware elements at each execution cycle and performs all formal checks according to the user-specified leakage model, either value-based or transition-based. In the value-based model, the verified expression is the new content of each modelled hardware component. In the
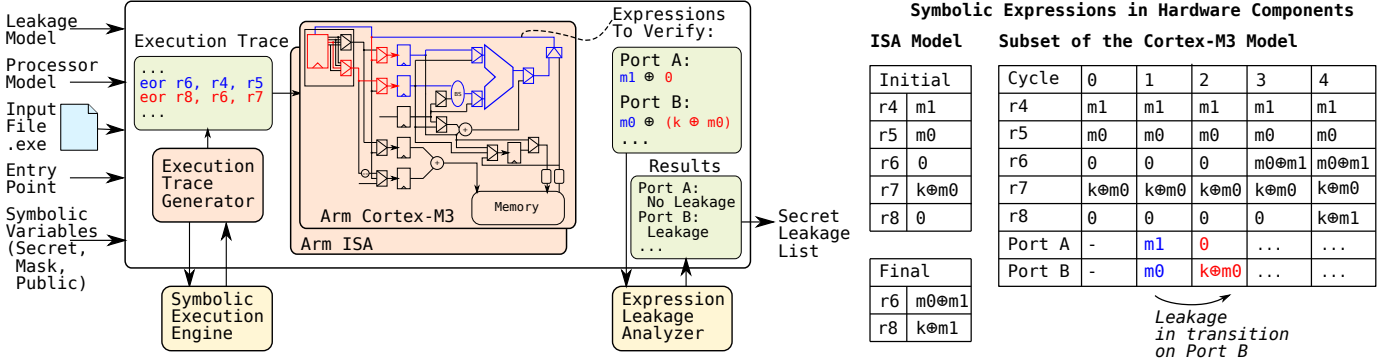
Fig. 6. ARMISTICE framework and example of symbolic expressions construction

transition-based model, it is the exclusive-or between the old and new expressions of each modelled hardware component. ARMISTICE retrieves the verification results from the external verification tool and eventually outputs a synthesis of all the results. It can also give, for each detected leakage, the involved assembly instruction(s), the leaking expression as well as the leaking hardware element. The user can leverage these precise sources of leakage to remove them. For example, he can then reorder instructions or insert specific instructions, and check that the new code is leakage-free.

Implementation: The different parts of ARMISTICE are implemented in Python. Currently, there are two processor models for the Armv7-m ISA: one at ISA level and one at micro-architecture level, denoted `Arm ISA` and `Arm Cortex-M3` respectively. However, ARMISTICE offers an easy way to add new models by implementing a specified interface, comprising a few functions called by a common simulation engine core. In the `Arm ISA` model, each instruction executes in one cycle and only modifies the content of the destination registers or the memory according to its semantics. The `Arm Cortex-M3` model implements a cycle-accurate description of the abstract model presented in Sec. III with a 3-stage pipeline (DEC, EXE1, EXE2). Modelled hardware components comprise GPR, internal CPU and memory registers, data memory as well as multiplexers and buses. At each simulation cycle, each modelled hardware component is updated according to the instruction processed in the corresponding pipeline stage. Note that the `Arm ISA` is a subset of the `Arm Cortex-M3` one. Currently, the execution trace is generated using the symbolic engine angr [26]. For the formal verification of symbolic expressions, we selected LeakageVerif which has been shown to be the most efficient, accurate and appropriate for verifying first-order probing security of expressions representing intermediate computations of masked software [19].

Example: The example in Fig. 6 illustrates how symbolic expressions are constructed in both models, focusing on `Port A` and `Port B` for the `Arm Cortex-M3` model. We consider a sequence of two instructions, namely `eor r6, r4, r5` and `eor r8, r6, r7`, performing a remasking of a secret `k` from the mask `m0` to the mask `m1`. The 'initial' column in the `ISA model` ('cycle 0' column in the `Arm Cortex-M3` model) gives the content of the relevant GPR just before the sequence: e.g. `r4` contains the symbolic variable `m1`, `r6` contains the concrete value 0. In the `Arm ISA` model, the registers are updated after the execution of each instruction: `r6` contains the symbolic values `m0 ⊕ m1` then `r8` contains `k ⊕ m1`. In this model, ARMISTICE guarantees the absence of leakage both in value and transition. In the `Arm Cortex-M3` model, in cycle 1, the instruction `eor r6, r4, r5` is in the decode stage, and so `Port A` and `Port B` contain respectively the expressions `m1` and `m0`. In cycle 2, this instruction advances in the execute stage and the instruction `eor r8, r6, r7` goes into the decode stage, what updates `Port A` with value 0 (as `r6` has not been updated yet) and `Port B` with value `k ⊕ m0`. Focusing on the components `Port A` and `Port B`, this creates the following two transitions: `m1 → 0` and `m0 → k ⊕ m0`. The resulting expressions, respectively `m1` and `k`, show that there is no secret leakage on `Port A`, but that a secret leakage in transition is present on `Port B`. The subsequent writes in `r6` and `r8` do not result in a secret leakage, as for the `Arm ISA` model.

Back to the motivating example: Let us now consider again our software implementation of the ISW AND algorithm, with a C code implementation shown in Lst. 1, and its compiler-emitted assembly code from Lst. 2. The results of the ARMISTICE analysis on Cortex-M3 in the transition leakage model are reported in the tables from Fig. 7. The upper table displays, for each instruction, which micro-architectural elements are leaking, together with a link to the expression and the secret values (`a`, `b`, `c`) leaked by that expression in the lower table. The correlation plot is the one from Fig. 1, with annotations to explain the leakage origins using the expressions from the tables. It is worth reminding here that due to the nature of the operation performed by the ISW AND, a bitwise and, a leakage on a or b implies a leakage on c.

As made obvious in Fig. 7, no processor cycle is leakage-free and all secrets are leaking in the transition leakage model most of the time. The leakages found by ARMISTICE match perfectly those found experimentally on the STM32F1. Next section will dive deeper in the details of the ARMISTICE results with real benchmarks.

V. EXPERIMENTAL EVALUATION

We show that our framework ARMISTICE modelling the Cortex-M3 allows us to find theoretical secret leakages in the micro-architecture, even when the masking scheme has been shown to be secure in their considered leakage model. Finally, we show on examples that the secret leakages found are experimentally visible.

	Instructions	Leaks: *expr. name*
I1	`and.w r5, r2, r1`	MuxRegA, RegA: $e0$ RegB: $e1$
I2	`ands r0, r1`	PortA, RegA: $e2$ AluOut: $e3$
I3	`ands r3, r2`	AluOut: $e4$
I4	`eors r4, r5`	RegB: $e5$
I5	`eors r0, r7`	AluOut: $e6$
I6	`eors r4, r3`	AluOut: $e7$
I7	`str r0, [r6, #0]`	-
I8	`str r4, [r6, #4]`	PortB, RegB, DataReg, DataOut, BufferMem: $e7$

Name	Expression	Leaks
$e0$	$a0 \cdot b1 \oplus a1$	a, c
$e1$	$a0 \cdot b1 \oplus b0$	b, c
$e2$	$a0 \oplus a1$	a, c
$e3$	$a0 \cdot b0 \oplus a1 \cdot b0$	a, c
$e4$	$a0 \cdot b0 \oplus a1 \cdot b1$	a, b, c
$e5$	$a1 \cdot b0 \oplus b1$	b, c
$e6$	$a0 \cdot b0 \oplus a0 \cdot b1 \oplus a1 \cdot b0$	a, b, c
$e7$	$a0 \cdot b0 \oplus a0 \cdot b1 \oplus a1 \cdot b0 \oplus a1 \cdot b1$	a, b, c

Pipeline stages:

Pipeline stages			
DEC	I1	I2	I3	I4	I5	I6	I7	I8			
EXE1		I1	I2	I3	I4	I5	I6	I7	I8		
EXE2									I7	I8	
MEM										I7	I8
Expressions	$e0$	$e0, e1$ $e2$	$e2, e3$	$e4$	$e5$	$e6$	$e7$	$e7$	$e7$	$e7$	$e7$

Plot: Correlation (50000 traces) vs Time (sample), with curves:
- $a = a0 \wedge a1$
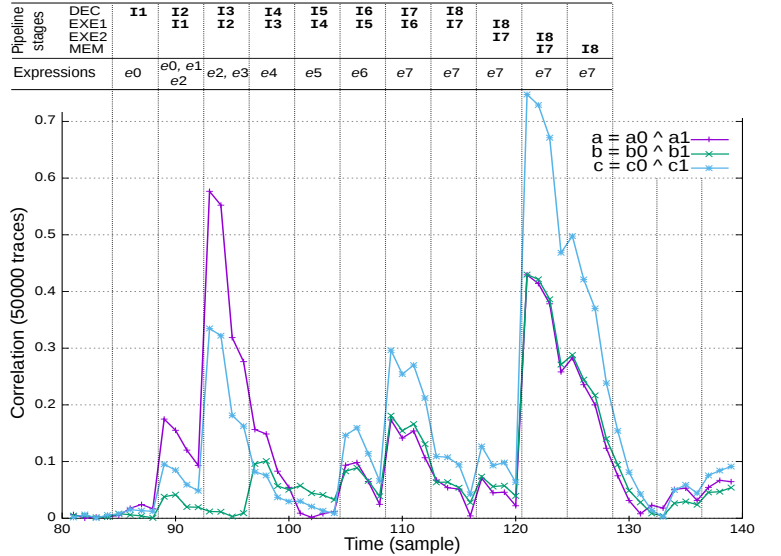- $b = b0 \wedge b1$
- $c = c0 \wedge c1$

Fig. 7. Detailed analysis of ISW AND in the transition leakage model. The tables display the ARMISTICE results. The Pearson correlation plot from Fig. 1 is annotated with vertical bars to show the processor cycles, the leaking expressions and the instructions that are in the different pipeline stages at each cycle.

A. Benchmarks

In order to illustrate the relevance of ARMISTICE, we consider 4 C-source implementations masked in the first-order value-based probing security model: `Secmult`, the popular secure Galois field multiplication [27]; `AES-Herbst` and `AES-KS` a masked version of AES and its key schedule, following the masking scheme proposed by Herbst *et al.* [28]; `AES-Yao` another masked version of AES [29]. The main difference in `AES-Yao`, compared to `AES-Herbst`, is the multiplication by constants 2 and 3 in the MixColumns step that is performed via table lookups. As the key schedule in `AES-Yao` is not masked, it is not considered in the evaluation. C-source benchmarks were compiled using `GCC` 10.2, at optimisation level `O2`. These benchmarks are first-order value-based masked, hence they are not secure in the transition-based leakage model. ARMISTICE is likely to detect leakages at ISA and micro-architectural level.

We also select 4 Arm assembly-level masked implementations. `Arm-Add` and `Arm-Add-opt` are two masked implementations of an addition of two secrets split in two shares [30]. They are provided as Arm assembly code and are, to the best of our understanding, designed to be masked in the first-order transition-based probing security considering GPR. ARMISTICE should then not detect leakages due to writes in the RegFile. `Dil-And` is a 32-bit masked logical AND and `Dil-A2B` is a 32-bit conversion from arithmetic to boolean masking. These two functions are provided as part of a masking scheme of the Dilithium algorithm for the Cortex-M3 [31]. Both versions have been implemented using an enhanced version of the MAPS simulator. As they are supposed to take into account the Cortex-M3 micro-architecture, it is interesting to know if ARMISTICE can find remaining secret leakages or not at micro-architectural level.

B. Leakage Analysis

For each benchmark, a verification is performed on both the `Arm ISA` and `Arm Cortex-M3` processor models.

Tab. 8 presents a synthesis of the analysis results for the `Cortex-M3` processor model per benchmark. Namely, it gives for each modelled component of the Cortex-M3 the number of secret data leakages in the value-based leakage model (`#VL` column) and in the transition-based one (`#TL` column). As expected, the results for the GPR (line "R0 to R14") are the same, and so stand for both this model and the `Arm ISA` processor model, in which only the GPR are considered as sources of leakage. This also comprises values read from and written to memory as they necessarily pass through a GPR.

At ISA level, for `Secmult`, `AES-KS`, `AES-Herbst` and `AES-Yao`, the analysis in the value-based leakage model confirms there is no leakage in the GPR, hence nor in memory, as claimed by their authors.

For the `Arm Cortex-M3` processor model, for the same benchmarks, we can first notice that as expected there is no leakage in the value-based leakage model, except for the memory related components (`DataOut`, `DataIn` and `ReadWriteBuffer`). This is because when a masked byte of the AES state is read, the whole memory word is actually read in memory and transferred, containing four state bytes. As there is a single mask byte for masking each of these four bytes, this constitutes a leakage. We can also notice that there is an enormous amount of leakages in transition on micro-architectural components for these benchmarks. In fact, all of the components except `MuxRegAddr2`, `RegAddr2`, `MuxBS`, `MuxDataAdder` and 5 GPR have at least a transition revealing secret information. While a verification in the value-based leakage model at the ISA level is a required first step towards leakage-free implementations, it leaves many secret leakages on the table.

The `Arm-Add` and `Arm-Add-Opt` benchmarks both contain secret leakages in GPR in the value-based and transition-based leakage models. This suggests that there may be a flaw in the design of these programs contrary to the claim of the authors. Unsurprisingly, there are many other secret leakages in other hardware elements of the core, as those were not taken into

	BENCHMARK	Secmult		AES-KS		AES-Herbst		AES-Yao		Arm-Add		Arm-Add-Opt		Dil-And		Dil-A2B	
	NB OF ANALYSED INSTR.	365		3001		5409		4527		122		106		79		443	
	COMPONENT	#VL	#TL	#VL	#TL	#VL	#TL	#VL	#TL	#VL	#TL	#VL	#TL	#VL	#TL	#VL	#TL
GPR	R0 to R14	0	0	0	99	0	190	0	155	71	83	66	72	0	0	0	0
DECODE STAGE	Port A	0	1	0	144	0	10	0	46	44	68	50	58	0	0	0	0
	Mux Reg A	0	1	0	144	0	19	0	36	44	68	51	60	0	0	0	0
	Reg A	0	0	0	144	0	0	0	36	43	67	51	60	0	0	0	0
	Port B	0	16	0	541	0	226	0	244	61	78	35	44	0	0	0	0
	Mux Reg B	0	16	0	553	0	226	0	243	64	78	45	44	0	0	0	0
	Reg B	0	23	0	283	0	137	0	145	59	72	41	50	0	0	0	0
	Mux Reg Addr 1	0	0	0	18	0	0	0	0	1	2	0	0	0	0	0	0
	Reg Addr 1	-	-	0	18	0	0	0	0	0	0	0	0	0	0	0	0
	Mux Reg Addr 2 / Reg Addr2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ALU DATA PATH	Mux BS / Mux Data Adder	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	Barrel Shifter	0	0	0	246	0	70	0	92	55	81	61	69	0	0	0	0
	Data Adder	0	0	0	18	0	0	0	0	0	0	0	0	0	0	0	0
	ALU / Mux ALU Out	0	0	0	27	0	0	0	0	71	87	66	75	0	0	0	14
AGU & LSU DATA PATH	Addr Adder	0	0	0	18	0	0	0	0	0	0	0	0	0	0	0	0
	Mux Data Reg	0	0	0	18	0	70	0	47	2	2	0	0	0	0	0	0
	Data Reg	0	2	0	32	0	378	0	438	2	2	0	1	0	1	0	1
	Mux Data Write / Data Out	0	0	0	18	0	70	0	47	2	2	0	0	0	0	0	0
	Data In	7	7	122	577	1444	1444	1136	1199	2	4	0	0	0	0	0	0
	Data Extract.	0	0	0	99	0	70	0	191	0	0	0	0	0	0	0	0
MEMORY	Addr Buffer	0	0	0	18	0	0	0	0	0	0	0	0	0	0	0	0
	Read Write Buffer	7	10	122	864	1444	1838	1136	1259	2	4	0	0	0	1	0	1
WB TO RF	Mux Bus D	0	0	0	153	0	70	0	191	71	87	66	75	0	0	0	14

Fig. 8. ARMISTICE analysis results for the 8 benchmarks. Columns `#VL` (resp. `#TL`) gives the number of value (resp. transition) leakages found.

consideration in the masking scheme design.

Finally, for the `Dil-And` and `Dil-A2B` benchmarks, almost all the secret leakages in value and transition have been eliminated. Yet, a few secret leakages in transition remain: they occur on the path from the ALU to the `RegFile`, in the `DataReg`, and in the `ReadWriteBuffer`. We first looked at leakages in the path from the ALU to the RegFile in `Dil-A2B`, as it should have been taken into consideration in the design of these programs. Unfortunately, the leaking expressions are too big for a manual analysis (more than 800 KB per leaking expression). Therefore, we rather performed a manual analysis of the leaking expressions reported for the `Dil-And` program. Both leakages on `DataReg` and `ReadWriteBuffer` happen when writing back in memory the two shares of the result. If the two stores have been separated by a logical instruction to clean the data path, this instruction does neither erase the content of the `Data Reg`, nor the one of the `ReadWriteBuffer`. Consequently, the two stores of the two shares reveal the secret results in these two hardware elements. We show in the next section that this secret leakage can be experimentally observed at the ARMISTICE's reported time in the power trace.

These results show that the ISA level or the value-based leakage model are not good abstractions of the real power consumption. Instead, there is a need to take into account a more detailed description of the device executing the program. The RTL level combined with the transition-based leakage model is a better abstraction for leakage analysis, as together they are much closer to the hardware transitions, responsible for most of the power consumption. Besides, the `Dil-And` and `Dil-A2B` results show that it is difficult to take into account micro-architectural details without a formal approach, as manual approaches will fail to take into account every aspect, even with detailed knowledge. These results thus show the relevance of the proposed approach, by being able to pinpoint the instructions, the moments and hardware elements in the core responsible for secret leakages, even for codes designed to be secure for the Cortex-M3 core.

C. Accuracy and Exploitability

In order to illustrate the accuracy of the approach, we run three experiments. First, in order to show that the leakages found on the `Dil-And` benchmark are real, we capture 500,000 traces with the setup described in Sec. III-B1. We run a specific t-test on the leaking expression and look at the t-test value for the samples corresponding to the cycles at which ARMISTICE detected the leakages: the results, shown in Fig. 10, demonstrate a perfect match.

In a second experiment, we repeat the same process for all of the simple secret leaking expressions in the first round of the Key Schedule in the transition leakage model. By simple, we mean expressions with at most one binary operator. There are height such expressions. In Fig. 9, we can see that all the secret leakages experimentally observed are detected by ARMISTICE at the expected location in the trace. However, four secret leakages detected by ARMISTICE do not translate into concluding t-tests, making them *seemingly* false positives. However, after further investigations:

- Two of these leakages result from writing a byte in R0 (ex6 and ex7). As shown in section III-B5, transitions in R0 on a single byte can hardly be observed with 500,000 traces. Here, in both cases, as there is still a remnant effect from a big previous leakage, these leakages on R0 cannot be distinguished on the trace.
- One of these leakages results from the transition between two register values put on the Bus B, but not written into Reg B (ex5). In fact, the two corresponding instructions do not read the registers, but the latter are selected because of the immediate values contained in the instructions. We designed a specific test vector for this case which confirmed that no leakage can be observed in this case, at least with our experimental setup.
- The last leakage corresponds to an already observed leakage situation, since it occurs on `ALU out` (ex2). After investigation, it appears that in some cases, the processor stalls for one cycle, which is not taken into account in our model. While we have not yet found out the reason why
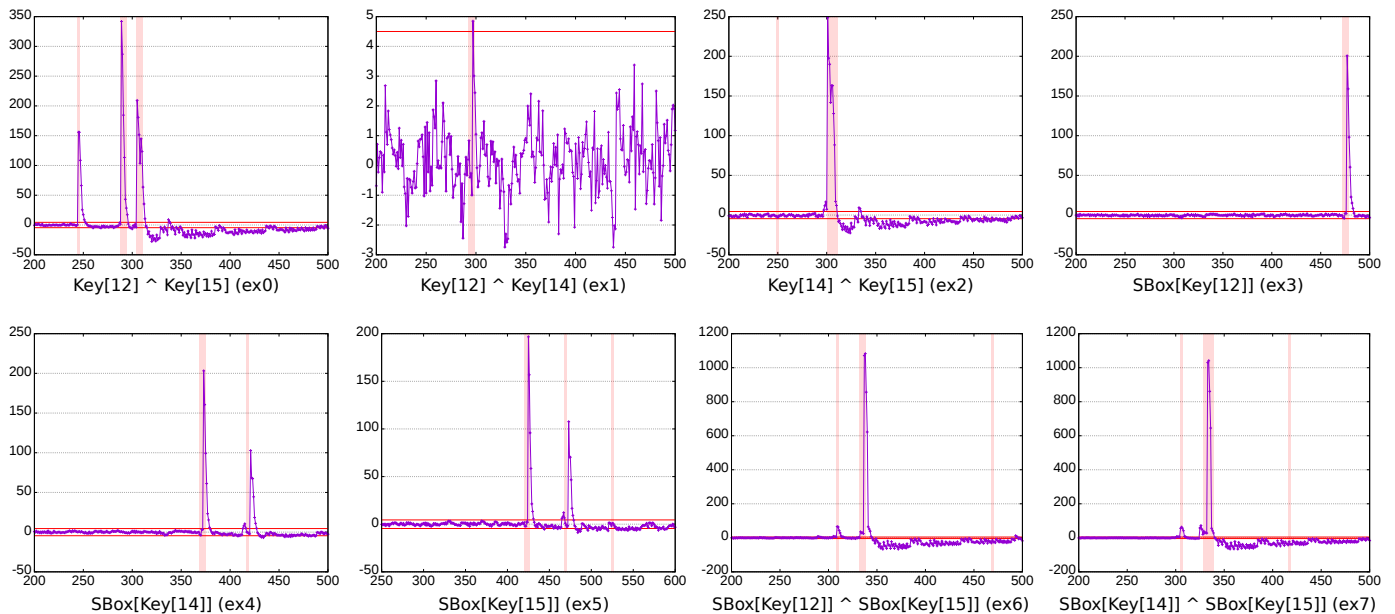
Fig. 9. Comparison between the secret leakages found by ARMISTICE in the first round of `AES-KS`, and the corresponding specific t-tests. The pink rectangles indicate the cycles at which a leakage was found in ARMISTICE for the corresponding expression.

such stall cycles occur, we suspect they come from the fetch of some instructions in the Flash memory. Adding an extra instruction to shift the two non leaking consecutive instructions makes the stall cycle disappear and the leakage found by ARMISTICE visible.

In order to deal with the *non-visible* leakages, one can decide to deactivate leakage analysis in some hardware elements for a given data size if the leakages in these elements are experimentally too small to be observable. Regarding the unexplained stall cycles, not modelling them only results in false positives as the stall resets some signal values. Yet, future work includes understanding why such stall cycles occur.

For the third experiment, we modify the assembly code of the previous experiment to remove all the possible secret leakages detected by ARMISTICE. Some of them, occurring in the memory, were not taken into consideration as they require modifying the memory data layout of the program. For each detected secret leakage, we add one or several instructions in order to clean the parts of the data path involved in the leaking transition. The added instructions can have three different kinds: for transitions happening in the arithmetic and logic data path, we use a `orr lr,lr,lr` instruction, not a `nop` neither a `mov` instruction, in order to be sure to clean `regA` and `regB`; for transitions happening in the AGU, LSU or memory, we use either a load or a store by making sure that the address is secret independent and that the value read or written is a constant; finally, for transitions happening in a GPR, we use a `orr rX,lr,lr` to erase the content of the register with a constant value before the new one. Once all instructions are added, the new code is run again on ARMISTICE to ensure that no more leakage is found.

Fig. 11 shows the t-test values obtained for all the expressions after patching the code: all leakages have disappeared. This second experiment shows how ARMISTICE can help designers remove all the potential secret leakages in a code,
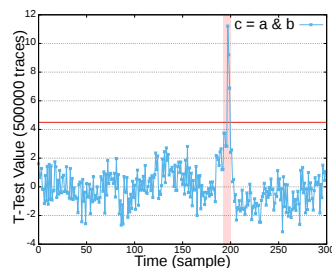

Fig. 10. Comparison between the secret leakages found by ARMISTICE in the `Dil-And` benchmark, and the corresponding specific t-test. The pink rectangle indicates the cycles at which a leakage was found in ARMISTICE.
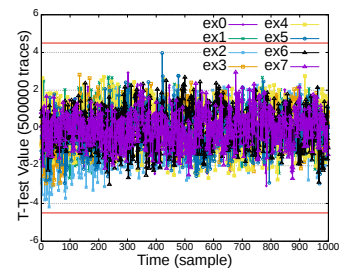

Fig. 11. t-test values of the eight originally leaking expressions (ex0 to ex7) in `AES-KS`, after patching the assembly code. No more expression is leaking.

by exhibiting the specific leakages, along with where and when they happen. This information allows ones to write the minimal code patch to remove such leaking transitions. The patched code can eventually be analysed by ARMISTICE in order to verify the absence of leakage.

To conclude, ARMISTICE is a valuable tool that detects secret leakages happening during the execution of a masked software on a processor. It can also help patch a code to suppress leakages, and verify a patched code. These results show that a RTL description is a relevant abstraction level to perform a formal leakage analysis. Although some modelled transitions may not translate into visible leakages, removing all RTL transitions leaking a secret provides a valuable security guarantee.

Currently, glitches are not taken into consideration in ARMISTICE. While nothing theoretically prevents considering them, we believe that considering them "as such" will result in a lot of detected leakages that are unlikely to be observed on a real target device. A more detailed analysis of the circuit

is required in order to select existing glitches, which is left for future work.

VI. Related work

Power or electromagnetic trace simulators using a model inferred from experimental measurements intrinsically take into account micro-architectural leakage [32], [33]. ELMO [32] pioneered such trace simulators with a power model of the Arm Cortex-M0 for a subset of the ISA composed of the most frequent instructions in cryptographic implementations. The accuracy of such simulators depends on the set of experiments used to learn the model. Recently, ELMO has been extended into ELMO* in order to better model register or memory reuse as well as interaction between non consecutive instructions [23]. As for MAPS [14], it has been built using the RTL description of the Arm Cortex-M3. It focuses on the potential leakage due to internal registers at the entrance of the ALU: our experiments show that ARMISTICE outperforms MAPS regarding the modelling of the processor core in addition to help remove the found leakage by pinpointing the leaking instructions and the leaking hardware elements.

MIRACLE [17] is an infrastructure proposing several test vectors for highlighting micro-architectural leakages due to interactions between operands of instructions or memory accesses, or due to speculative execution. While this work shows that the leakage varies between architectures and even implementations of the same processor core, it does not explain how to use test vectors to understand or describe the micro-architecture to model the leakage sources. Gao *et al.* [18] go one step further by reverse-engineering some micro-architectural leakage features of an Arm Cortex-M3 processor running 16-bit instructions only. Using test vectors and their recently so-called collapsed models [34], they deduce a model of the different components of the Arm Cortex-M3. Barenghi *et al.* explore how to infer the structure of different Arm Cortex-A and Cortex-M pipelines, using a framework of microbenchmarks and the CPI metric (Clock cycles Per Instruction) [35], [36]. They highlight the potential side channel leakages resulting from the pipeline structures and some architectural choices. As we have access to the RTL of the Cortex-M3 processor, we can use a white-box approach for the processor core; we go deeper in the modelling of the leakage sources, like considering both the 16-bit and 32-bit variants of the instruction set. We give clear explanations of the leakages found on the Cortex-M3 and reported by those related work. However, as previous work, we resort to a black-box approach based on leakage test vectors for modelling the memory subsystem but we go one step further as this model is used by ARMISTICE for detecting leakages due to interaction in the memory subsystem.

In order to remove the micro-architectural leakage, the ROSITA tool [23] automates leaking pattern replacement detected by using ELMO* and non specific t-tests. The iterative process is able to remove most leakages from three implementations but fails for one implementation. The modelling may be incomplete, or, as ELMO* cannot help pinpointing the source of leakage, the replacement patterns may not be

fully secure. In addition, these patterns may not be as small as they could be with a better leakage source understanding. Also, when the patching fails, developers are left clueless with the leakage, advocating for a solution able to explain it. On the hardware side, the usage of dedicated logic or design tools can help reduce the sources of potential leakage [12], [13]. Such design-time solutions do not help for existing processors. FENL [37] extends the ISA with additional instructions for cleaning off in-core resources that may be the source of leakage. This is an appealing solution as it offers developers means to avoid in-core leakage at a lower cost than using tediously designed cleaning gadgets, but it still does not help for existing processors. Moreover, automated solutions such as ARMISTICE are required to help determine how and where to use such dedicated instructions.

Finally, two recent works target the verification of masked software on a low-level processor description.

Barthe *et al.* propose a Domain Specific Language for modelling assembly implementations and for specifying fine-grained leakage models [16]. The semantics of assembly instructions can then be enriched with the leakage effects of the instruction. In particular, different leakage effects due to some internal registers or some buffers in memory can be made explicit. They propose the scVerif tool as a front end of MaskVerif [2], [3] for formally verifying masked assembly gadgets and implementations. scVerif allows one to build micro-architectural leakage aware gadgets, be they masking gadgets or cleaning gadgets. Such gadgets can then be used in an automatic composition in order to reduce the time required to implement secure solutions [38]. The proposed approach enables one to take into account the interactions inside the pipeline or due to memory access during the formal verification. However, it relies on the knowledge of the micro-architectural source of leakage and requires the explicit modelling of leaking components (e.g. internal registers or memory buffers). These papers do not explain how the modelling of the Arm Cortex-M0+ targeted in the experiments was carried out. As shown in our paper, the decoding stage of the Arm Cortex-M3 processor induces different leakage effects depending on the binary encoding of the instruction, thus requiring as many instruction specifications as the number of encodings per instruction. Moreover, the user would have to explicit the instruction encoding in his implementation expressed in the scVerif's input language. A verification using the binary code as we propose seems more appropriate for avoiding putting the burden on the user. Finally, as scVerif executes sequentially the instructions, it cannot model the effects of the forwarding mechanism.

Coco [15] is an approach for formally verifying if a masked software would securely be executed on a CPU. Leakages are searched for at gate-level, comprising glitches. Authors show several weaknesses in the RISC-V IBEX core related to the register file, ALU and LSU units that can be removed with small changes. Coco requires the analysed masked implementation to satisfy two constraints: 1) shares related to a same secret must not be processed by consecutive instructions 2) registers and memory locations which contain a share must not be overwritten with the second share. If these constraints may

be sufficient for the IBEX core simple micro-architecture, they are not sufficient for the Cortex-M3 in which non consecutive instructions may write into a same micro-architectural register. The first constraint is not necessary either as consecutive instructions do not lead to a secret leakage when the explicitly or implicitly manipulated shares are not combined inside the processor. Moreover, meeting such constraints requires dedicated tools, such as formal verification tools at ISA level. ARMISTICE has no constraint on the analysed masked implementation, it is able to find leakage due to any modelled hardware element: GPR, internal register, bus, memory buffer or memory location. In addition, it helps designers to remove such interactions thanks to its outputs.

This comparison shows that ARMISTICE outperforms the existing approaches as it automates formal verification taking into account all the micro-architectural details of a processor, and makes a link between the secret leakages found, the hardware, and the executed software.

VII. CONCLUSION

We presented ARMISTICE, a framework for formally verifying the absence of leakage considering first-order value-based or transition-based probing security in masked software implementations. Using an accurate model of a target core and memory subsystem, ARMISTICE allows to take into account micro-architectural leakages in the verification process. We presented the modelling of an Arm Cortex-M3 based STM32F1 board based both on a white-box analysis of the core and on a black-box analysis of the memory subsystem *via* leakage test vectors. We experimentally showed that our modelling is relevant for detecting micro-architectural leakages as well as for manually removing them thanks to the ARMISTICE's output, comprising leaking instructions and hardware components where leakages stem from. Future work will consider higher order leakage verification, the automation of model extraction from a RTL design and automatic code patching from an analysis result.

REFERENCES

[1] Y. Ishai, A. Sahai, and D. Wagner, "Private circuits: Securing hardware against probing attacks," in *CRYPTO*, 2003.
[2] G. Barthe, S. Belaïd, G. Cassiers, P.-A. Fouque, B. Grégoire, and F.-X. Standaert, "maskverif: Automated verification of higher-order masking in presence of physical defaults," in *ESORICS*, 2019.
[3] G. Barthe, S. Belaïd, F. Dupressoir, P.-A. Fouque, B. Grégoire, and P.-Y. Strub, "Verified proofs of higher-order masking," in *CHES*, 2015.
[4] G. Pengfei, X. Hongyi, P. Sun, J. Zhang, F. Song, and T. Chen, "Formal verification of masking countermeasures for arithmetic programs," *IEEE Trans. on Soft. Eng.*, 2020.
[5] J. Wang, C. Sung, and C. Wang, "Mitigating power side channels during compilation," in *ACM ESEC/FSE*, 2019.
[6] H. Seuschek, F. De Santis, and O. M. Guillen, "Side-channel leakage aware instruction scheduling," in *CS2*, 2017.
[7] I. B. El Ouahma, Q. L. Meunier, K. Heydemann, and E. Encrenaz, "Side-channel robustness analysis of masked assembly codes using a symbolic approach," *J. Cryptographic Engineering*, vol. 9, no. 3, 2019.
[8] Q. L. Meunier, I. B. El Ouahma, and K. Heydemann, "SELA: a Symbolic Expression Leakage Analyzer," in *PROOFS*, 2020.
[9] J. Balasch, B. Gierlichs, V. Grosso, O. Reparaz, and F.-X. Standaert, "On the cost of lazy engineering for masked software implementations," in *CARDIS*, 2015.
[10] K. Papagiannopoulos and N. Veshchikov, "Mind the gap: Towards secure 1st-order masking in software," in *COSADE*, 2017.
[11] T. Moos and A. Moradi, "On the easiness of turning higher-order leakages into first-order," in *COSADE*, 2017.
[12] P. Slpsk, P. K. Vairam, C. Rebeiro, and V. Kamakoti, "Karna: A gate-sizing based security aware eda flow for improved power side-channel attack protection," in *IEEE/ACM ICCAD*, 2019.
[13] D. ijai, J. Balasch, B. Yang, S. Ghosh, and I. Verbauwhede, "Towards efficient and automated side-channel evaluations at design time," *J. of Cryptographic Engineering*, vol. 10, 11 2020.
[14] Y. Le Corre, J. Großschädl, and D. Dinu, "Micro-architectural power simulator for leakage assessment of cryptographic software on arm cortex-m3 processors," in *COSADE*, 2018.
[15] B. Gigerl, V. Hadzic, R. Primas, S. Mangard, and R. Bloem, "Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs," in *USENIX Security*, Aug. 2021.
[16] G. Barthe, M. Gourjon, B. Grégoire, M. Orlt, C. Paglialonga, and L. Porth, "Masking in fine-grained leakage models: Construction, implementation and verification," *IACR Trans. on Crypt. Hard. and Emb. Syst.*, 2021.
[17] B. Marshall, D. Page, and J. Webb, "Miracle: Micro-architectural leakage evaluation," 2021, cryptology ePrint Archive, Report 2021/261.
[18] S. Gao, E. Oswald, and D. Page, "Reverse engineering the micro-architectural leakage features of a commercial processor," 2021, cryptology ePrint Archive, Report 2021/794.
[19] Q. L. Meunier, E. Pons, and K. Heydemann, "LeakageVerif: Scalable and Efficient Leakage Verification in Symbolic Expressions," cryptology ePrint Archive, report 2021/1468.
[20] G. Goodwill, B. Jun, J. Jaffe, and P. Rohatgi, "A Testing Methodology for Side-Channel Resistance Validation," in *NIST NIAT*, 2011.
[21] T. Schneider and A. Moradi, "Leakage assessment methodology - a clear roadmap for side-channel evaluations," 2015, cryptology ePrint Archive, Report 2015/207.
[22] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*, ser. Springer Series in Statistics, 2001.
[23] M. A. Shelton, N. Samwel, L. Batina, F. Regazzoni, M. Wagner, and Y. Yarom, "Rosita: Towards automatic elimination of power-analysis leakage in ciphers," in *NDSS*, 2021.
[24] C. OFlynn and Z. D. Chen, "Chipwhisperer: An open-source platform for hardware embedded security research," in *COSADE*, 2014.
[25] "ARMISTICE: Micro-Architectural Leakage Modelling for Masked Software Formal Verification." [Online]. Available: https://www-soc.lip6.fr/armistice
[26] "angr: a platform-agnostic binary analysis framework." [Online]. Available: https://github.com/angr/angr
[27] M. Rivain and E. Prouff, "Provably secure higher-order masking of AES," in *CHES*, 2010.
[28] C. Herbst, E. Oswald, and S. Mangard, "An AES smart card implementation resistant to power analysis attacks," in *ACNS*, vol. 3989, 2006.
[29] Y. Yao, M. Yang, C. Patrick, B. Yuce, and P. Schaumont, "Fault-assisted side-channel analysis of masked implementations," in *IEEE HOST*, 2018.
[30] B. Jungk, R. Petri, and M. Stöttinger, "Efficient side-channel protections of ARX ciphers," *IACR Trans. on Crypt. Hard. and Emb. Syst.*, 2018.
[31] V. Migliore, B. Gérard, M. Tibouchi, and P.-A. Fouque, "Masking dilithium," in *ACNS*, 2019, pp. 344–362.
[32] D. McCann, E. Oswald, and C. Whitnall, "Towards practical tools for side channel aware software engineering: 'grey box' modelling for instruction leakages," in *USENIX Security*, 2017.
[33] N. Sehatbakhsh, B. B. Yilmaz, A. Zajic, and M. Prvulovic, "EMSim: A Microarchitecture-Level Simulation Tool for Modeling Electromagnetic Side-Channel Signals," in *IEEE/ACM HPCA*, 2020.
[34] S. Gao and E. Oswald, "A novel completeness test and its application to side channel attacks and simulators," 2021, cryptology ePrint Archive, Report 2021/756.
[35] A. Barenghi and G. Pelosi, "Side-channel security of superscalar CPUs: Evaluating the Impact of Micro-Architectural Features," in *DAC*, 2018.
[36] A. Barenghi, L. Breveglieri, N. Izzo, and G. Pelosi, "Exploring cortex-m microarchitectural side channel information leakage," *IEEE Access*, 2021.
[37] S. Gao, B. Marshall, D. Page, and T. Pham, "FENL: an ISE to mitigate analogue micro-architectural leakage," *IACR Trans. on Crypt. Hard. and Emb. Syst.*, Mar. 2020.
[38] A. Abromeit, F. Bach, L. A. Becker, M. Gourjon, T. Güneysu, S. Jorn, A. Moradi, M. Orlt, and F. Schellenberg, "Automated masking of software implementations on industrial microcontrollers," in *DATE*, 2021.