Towards Micro-Architectural Leakage Simulators: Reverse Engineering Micro-Architectural Leakage Features is Practical

Si Gao[1], Elisabeth Oswald[1], and Dan Page[2]

[1] Digital Age Research Center (D!ARC), University of Klagenfurt, Austria
{si.gao,elisabeth.oswald}@aau.at
[2] Department of Computer Science, University of Bristol, UK
daniel.page@bristol.ac.uk

Abstract. Leakage simulators offer the tantalising promise of easy and quick testing of software with respect to the presence of side channel leakage. The quality of their build in leakage models is therefore crucial, this includes the faithful inclusion of micro-architectural leakage. Micro-architectural leakage is a reality even on low- to mid-range commercial processors, such as the ARM Cortex M series. Dealing with it seems initially infeasible in a "grey box" setting: how should we describe it if micro-architectural elements are not publicly known?
We demonstrate, for the first time, that it is feasible, using a recent leakage modelling technique, to reverse engineer significant elements of the micro-architectural leakage of a commercial processor. Our approach first recovers the micro-architectural leakage of each stage in the pipeline, and the leakage of elements that are known to produce glitches. Using the reverse engineered leakage features we build an enhanced version of the popular leakage simulator ELMO.

1 Introduction

Securing a specific implementation of a cryptographic algorithm on a concrete device is never a trivial task. In recent years, a proposal to help with this challenge has emerged: instead of testing implementations in a costly lab setup, leakage simulators like ELMO [1], MAPS [2], and ROSITA[3] have surfaced, which all claim to capture significant leakage of the respective devices that they apply to. A comprehensive survey of existing simulators was recently published [4]. This survey puts forward a range of challenges that are yet to be solved, among which is the inclusion of more micro-architectural effects (of the resp. processor).

Micro-architectural leakage can render the provable properties of modern masking schemes meaningless in practice. Let us consider for instance an implementation of a masked multiplication using the 2-share masking scheme originally proposed in [5]. We consider its implementation using Thumb Assembly on a microprocessor with the ARM Cortex M3 architecture (see the program code

in this section). The masked multiplication computes the shared out product $c = (c_1, c_2)$ of two shared out numbers a, b (with $a = (a_1, a_2)$ and $b = (b_1, b_2)$) using an independent random number r.

```
1        ISWd2:
2        ...
3        //r1=a(1), r2=a(2), a(1)+a(2)=a
4        //r3=b(1),r4=b(2), r5=r, b(1)+b(2)=b
5        mov  r6, r1  //r6=a(1)
6        ands r6, r3  //r6=a(1)b(1)
7        mov  r7, r4  //r7=b(2)
8        ands r7, r2  //r7=a(2)b(2)
9        ands r1, r4  //r1=a(1)b(2)
10       eors r1, r5  //r1=a(1)b(2)+r
11       ands r3, r2  //r3=a(2)b(1)
12       eors r1, r3  //r1=a(1)b(2)+r+a(2)b(1)
13       eors r6, r1  //c(1)=a(1)b(2)+r+a(2)b(1)+a(1)b(1)
14       mov  r0, r9  //r0=output address
15       eors r7, r5  //c(2)=r+a(2)b(2)
16       ...
```

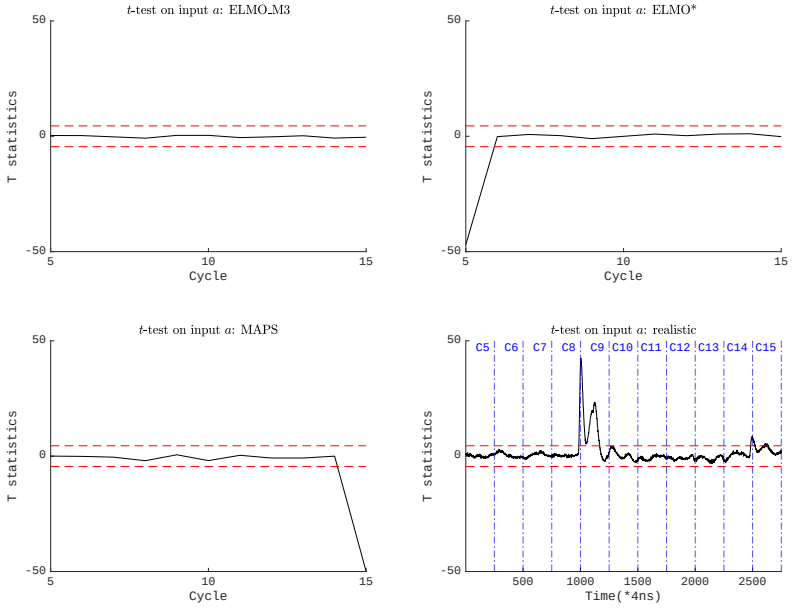


Fig. 1: 1st order TVLA on input a on existing simulators and realistic measurements

The multiplication rule produces the shared out product in a manner that guarantees that no information is revealed about a or b assuming individual intermediate values do not leak jointly. Consequently in practice it typically shows leakage because this assumption is often not justified. Using a leakage simulator, we can pinpoint the instructions that violate the assumption, then try to avoid the leakage via revising the corresponding instructions.

With real measurements, the first order fixed-versus-random t-test on the input a show leakage during the execution of the instructions in line 9 and 15 (the bottom right corner in Figure 1). But when we execute the same piece of code in ELMO (we use the recently released M3 version of ELMO [6]), no leakage can be found (see the upper left graph in Figure 1). The ROSITA tool uses an "upgraded version" of ELMO, which they call ELMO* [3]. Leakage detection reports leakage for the instruction on line 5 (upper right in Figure 1). According to the realistic detection, this is a false-positive leak, while the true leaks in line 9 and 15 are not reported. The white-box M3 simulator MAPS [2] reports leakage for the instruction on line 15, but fails to report leakage for line 9 (bottom left in Figure 1).

All three simulators fail to capture some leak(-s) in this example, and one finds a leak where there is none. Our motivation is thus to develop a technique that leads to more accurate leakage models and ultimately simulators.

1.1 Our contributions

The challenge to include micro-architectural effects is a non-trivial one when working with many interesting cores. This is because many processors of interest feature pipelining and have multiple unknown micro-architectural elements that leak. Consequently we need to reverse engineer their leakage behaviour (note that we do not actually need to reverse engineer the entire core itself).

Side-channel leakage has been used in the past for reverse engineering of both programs and hardware [7], [8], [9], [10]. In these works the authors used standard DPA style attacks (with and without using device leakage models) to confirm hypotheses about the internals of the respective devices/implementations, which were relatively simple. In order to tackle devices that feature pipelining, and/or a more interesting memory subsystem, a better approach is needed. In recent work [11], Gao and Oswald pick up the methodology from [1] and extend it so it can capture considerably more complex leakage models. They also argue that important leakage has been missed in recent attacks [12] and simulators.

We show that their novel modelling technique can not only be used to reason about the quality of leakage models, but that it is actually a tool for reverse engineering the micro-architectural leakage features of devices. We use it to dissect the leakage from a commercial processor based on the ARM Cortex M3 architecture and reveal its micro-architectural leakage characteristics. Doing so is all but straightforward: [11] are clear that the test itself provides "clues" about the internal mechanisms, but one needs to design additional confirmatory experiments to actually verify the micro-architectural meaning of these clues (this ties in with another recent paper [13]).

To put our results into the context of the existing leakage simulation literature, we then compare leakage predictions that are based on the reverse-engineered micro-architectural leakage with the predictions of the most sophisticated simulators ELMO and MAPS[3].

Whilst our methodology currently involves intensive manual effort, we argue such effort is worthwhile, because:

- it enriches our understanding of micro-architecture effects in relevant processor architectures,
- it significantly improves the state-of-the-art leakage modelling of micro-architectural elements,
- it showcases that many existing leakage models and tools miss significant micro-architectural effects.

1.2 Methodology and Paper Organisation

In the following three sections, we discuss step by step how to reverse engineer the micro-architectural leakage elements of a close-sourced commercial processor. In contrast to previous works that captured only simple micro-architectural leakage, and led to the simulators ELMO, ELMO* and MAPS, we aim to comprehensively recover all micro-architectural leakage.

In a grey-box setting, we cannot take advantage of a detailed hardware description, but we can utilise publicly available architectural information to guide our analysis. Therefore, our methodology is based on the following key steps:

1. Build an abstract diagram from the public available information (e.g. architecture reference [14], ISA [15], etc.) and make some safe architectural inferences (Section 2).
2. Recover the relevant micro-architectural details through analysing the side-channel leakage. Specify the data flow for each instruction and construct a micro-architectural leakage model for each pipeline stage (Section 3).
3. Evaluate the overall micro-architectural leakage for the target processor, further adding more subtle micro-architectural leakages (e.g. glitches) or discarding non-significant factors (Section 4).

We then challenge the resulting micro-architectural leakage model of our M3 by a comprehensive comparison in Section 5, and we conclude this paper in Section 6.

[3] ELMO* [3] offers an extension to ELMO that captures some more leakage from the memory subsystem. ELMO offers also such an extension (in the follow-up development), yet both are drawn from experimental guesses. Nevertheless, our focus in this paper still lies in pipelined core, where the entire ELMO family sticks with the original ELMO model [1].

Experimental setup Throughout this paper, we preserve the same experimental setup:

- Target: NXP LPC1313 (ARM Cortex-M3) running at 1 MHz with only Thumb instructions
- Measurement point: voltage at a 100 Ohm shunt resistor at the VCC end
- Pre-processing: on-board 22db amplifier (NXP BGA2801)
- Oscilloscope: Picoscope 5243D running at 250 MSa/s

Unless stated otherwise, each tested code snippet takes 50k traces. Our setup ensures leakage does not last for more than 1 cycle, which helps to identify how leakage changes from cycle to cycle. Thus, most experimental results in the following two sections have been cropped to the exact cycle, which contains 250 sample points.

Statistical reverse engineering methodology As a reverse engineering tool we use the methodology from [11]. Their methodology extends the modelling technique of ELMO. We provide an informative explanation of their technique, which essentially enables to "compare" two leakage models. A leakage model consists of a function and some variables that correspond to actual leakage elements (e.g. architectural registers, buffers, etc.). The models that we compare consist of the same function (which includes possible interactions between variables), but they contain different variables. Specifically we reduce a model by removing a variable (or the interaction between variables) that represents an unknown leakage element. The statistical test from [11] then checks if this additional variable explains statistically significantly more of the the observable leakage. If so, then we conclude that the removed variable represents a significant leakage element in the processor.

To facilitate explaining our research in the following sections, we need to introduce some notation and formalism around the leakage modelling and testing process. We will refer to any model that we test with M and if there are multiple models we distinguish them by their subscript, e.g. we may want to test two models M_0 and M_1. A leakage model consists of a function and some variables. The function defines if and how the included variables interact with each other. The methodology in [11] works with nominal models, thus all coefficients are either 0 or non-0 in the resulting functions. Thus we drop these coefficients for readability, and instead use a set notation to indicate how variables interact with each other: if two variables X, and Y fully interact with each other then we write their respective model as $\{XY\}$; if the two variables only leak independently then we write their respective model as $\{X, Y\}$.

In our work we hope to find if a simpler model $M_1 = \{X, Y\}$ already suffices, or if a fully interactive model $M_0 = \{XY\}$ is necessary. We note at this point that the simpler model M_1 is indeed included in M_0: in fact, M_0 includes all interaction terms and also the individual variables X, Y. A statistical test can be applied to tell whether M_0 is a "better" model than M_1, which implies whether X and Y interact with each other in the measurements. The problem of this

approach is that the variables that we consider are all "large" in the sense that statistically every 32-bit variable (the M3 operates on 32 bit data words) leads to 32 independent statistical variables. Testing multiple large variables then leads to the problem that a test requires a large number of leakage observations to produce statistically significant results. To circumvent this problem we use the trick of "collapsing models" from [11].

2 Step 1: Identifying Safe Architectural Assumptions

Although exploring every concrete detail is not possible in a grey-box scenario, there is always some public information available that can be used to construct an initial, abstract architectural view. For instance, from Figure 2, reproduced from [14, Figure 1.2], we know the Cortex-M3 processors use a 3-stage pipeline [14]: the stages are termed **Fe(tch)**, **De(code)**, and **Ex(ecute)**. More specifically, while executing instruction $i-2$, instruction $i-1$ is being decoded by the instruction decoder, and instruction i is being fetched from the memory to the instruction register. Since there is no dedicate write-back stage, the Arithmetic Logic Unit (ALU) output is written-back to the register file (or memory) immediately after the **Execute** stage.

Although not directly provided in [14], we believe the following details can be safely inferred

- A set of pipeline registers exists between stages, meaning, for example, an instruction register between **Fetch** and **Decode** and pipeline register(s) between **Decode** and **Execute**.
- Figure 2 explicitly claims that "register read" occurs within the **Decode** stage; this implies the pipeline registers between **Decode** and **Execute** stores control signals *and* operands read from the register file.
- Many Thumb instructions [15] use 2 operands, which suggests the register file should have at least 2 read ports; this implies there are (at least) 2 operand pipeline registers between **Decode** and **Execute**.

3 Step 2: Recovering major micro-architectural leakage elements

Previous works such as [13,16] have shown that side-channel leakage can reveal some micro-architectural details. In this spirit, but utilising the F-test methodology for nested models, we set out to recover the major micro-architectural leakage elements of our Cortex-M3 core. We do so by analysing each of three pipeline stages separately.

3.1 Fetch

The **Fetch** stage fetches one or several instructions from the memory to the instruction register (i.e. block **Fe** in Figure 2). Based on the publicly information

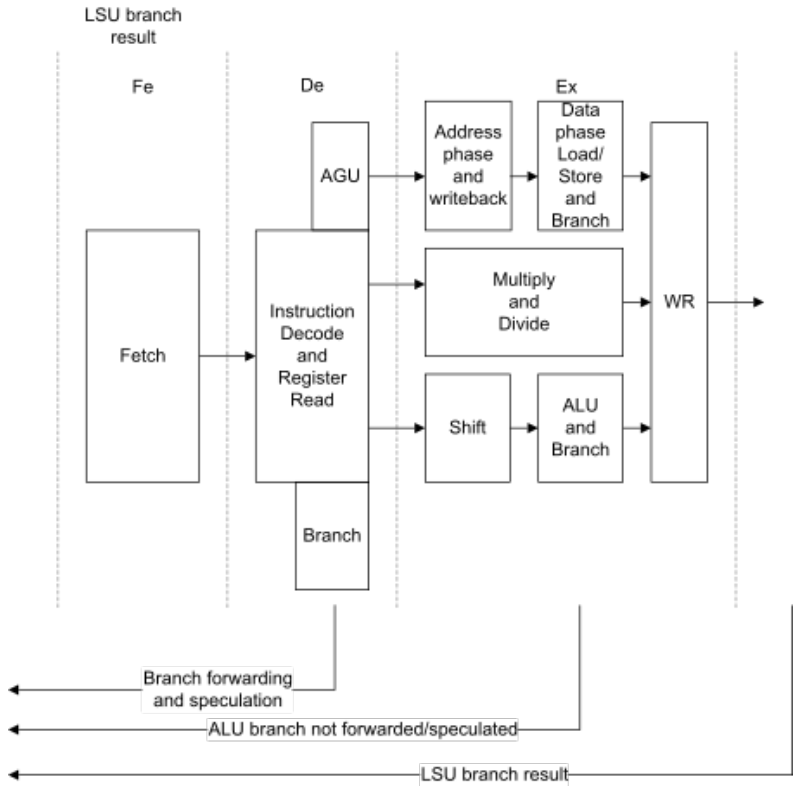
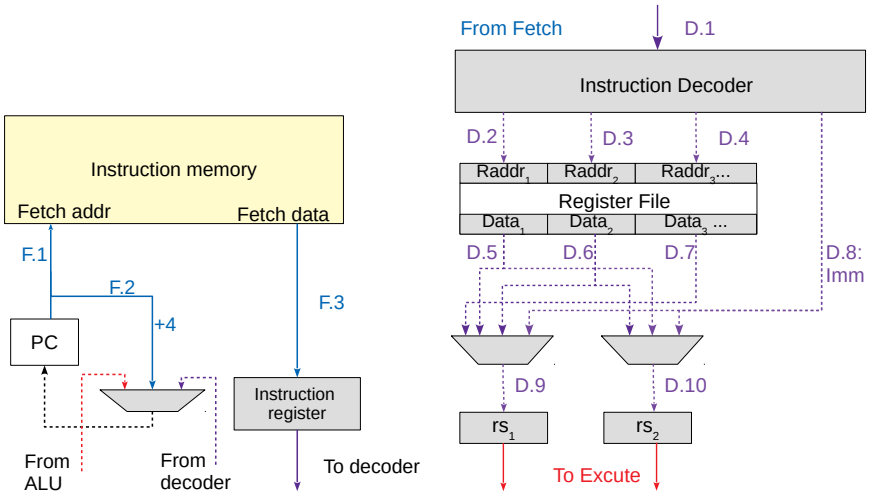


Fig. 2: The Cortex-M3 pipeline [14, Figure 1.2].

provided in the ARM reference manual, we envision the micro-architecture of the **Fetch** stage to look as depicted in Figure 3a.

Functionally, the fetched instruction's address is stored in Program Counter (PC, aka R15 in ARM): therefore we plot F.1 which sends PC value to the instruction memory. PC can be incremented automatically (F.2), or accepts new address for branching (from ALU or decoder). F.3 loads the instruction(-s) to the instruction register, which marks the beginning of **Decode**. We plot all wires in this stage as blue lines in Figure 3a.

In terms of micro-architectural ambiguity, there is none in Figure 3a. In fact, the wires F.1-3 are fully determined by the value of PC. Unless the program performs data dependent branches, all leakage from this stage is constant between executions. We further exclude the leakage from data dependent branches in our analysis: compared with leakage modelling, information flow analysis is a much easier solution for that issue.



(a) **Fetch**. (b) **Decode**.

Fig. 3: Hypothetical micro-architecture: **Fetch** and **Decode**.

3.2 Decode

The **Decode** stage starts from translating the fetched instruction into the control logic, and ends with sending the pre-loaded operand(-s) to the pipeline register(-s)(i.e. block **De** in Figure 2).

Figure 3b plots our view of the micro-architecture for the **Decode** stage. The decoder translates the instruction (D.1) into control signals, including the register indices for the pre-loaded operands (D.2-D.4) and potential immediate numbers (D.8). The corresponding operands are loaded from the register file (D.5-D.7), then sent to the pipeline registers (D.9-D.10). The pipeline registers rs_1 and rs_2 mark the beginning of **Execute**. All the wires in this stage are plotted as purple lines: if the signal is directly read from a register, we use solid line; otherwise, we use dash line to represent the fact that this signal might be affected by glitches (analysed in Section 4.2). Note that there should also be a few pipeline registers storing the control signals and the immediate number: as they are not data-dependent, we simply omit those in Figure 3b.

Unlike **Fetch**, there are a few ambiguities in the **Decode** stage: first, it is unclear how many read ports/operands should exist in Figure 3b. Considering most Thumb instructions take at most 2 operands, previous tools often assume the register file has 2 read ports [1,2] (i.e. connected to D.5 and D.6). We also started with a similar architecture, but some instructions (e.g. *adds Rd, Rn, Rm*) produced leakage that access more than 2 registers. From side-channel leakage alone, we cannot conclude whether there is another read port (i.e. D.7), or such leakage is from a multiplexing route of the existing ports or even an unexpected access from glitches. Either way, we proceed our analysis assuming there are 3 read ports (which is leakage equivalent to the other options).

With multiple read ports existing in the micro-architecture, the next question to ask is *which operand is loaded from which port*. Thus, we design the following experiments and try to find the answer from analysing the realistic measurements.

Testing the read ports. We denote \$a as the the low register r_a where a randomised operand A is stored in. The 3 reading ports in Figure 3b (marked as $Data_{1-3}$, connected to D.5-7), we denote them as $port_1$, $port_2$ and $port_3$ respectively. As we can see from the following code snippet, when executing the first *eors*, the second instruction enters the **Decode** stage. According to Figure 3b, operands C and D should occupy two read ports on the register files (therefore also two connected buses D.5 and D.6), while the previous values on these ports should be A and B. Thus, within the cycle that is decoding the second instruction, as long as we observe a leakage that corresponds to the interaction of A and C, it is expected that A and C should share the same reading port/operand bus.

```
1 Testing_port:
2 ...
3 eors $a,$b
4 INSTR $c, $d
5 nop
6 eors $0,$0 //$0=register that stores 0
7 ...
```

Specifically, let us assume in *eors \$a,\$b*, A takes port 1 (i.e. D.5)[4]. From here, whenever an interaction is detected between A and C, we set C to port 1. Otherwise, if an interaction is detected between A and D, we set D to port 1.

This leads to testing the following two models using real device data:

- $M_0 = \{AC\}$, $AC = \{x | x = a||c, a \in A, c \in C\}$
- $M_1 = \{A, C\}$ (similarly BD, AD, BC)

If the test concludes that there is no enough evidence that M_1 is significantly worse than M_0, we conclude that there is no strong evidence of A and C interact with each other, therefore it is less likely A and C shares the same reading port/operand bus in the micro-architecture. Otherwise, A is clearly interacting with C: if the interaction is indeed coming from the micro-architecture[5], it is likely A and C share the same reading port/operand bus.

Altogether we tested 55 Thumb instructions, which covers almost the entire instruction set (versus 23 cryptography-relevant instructions in ELMO [1]).

[4] If it is the other way around, what we learned is a "mirrored specification", which will be remedied by a *mirrored* leakage model later.

[5] In theory, it is also possible that the interaction is caused by glitches, or physical defaults such as coupling [17]. In our experiments, we find the magnitude of wire transition leakage is usually larger than the other options, which makes it possible to make a distinction.

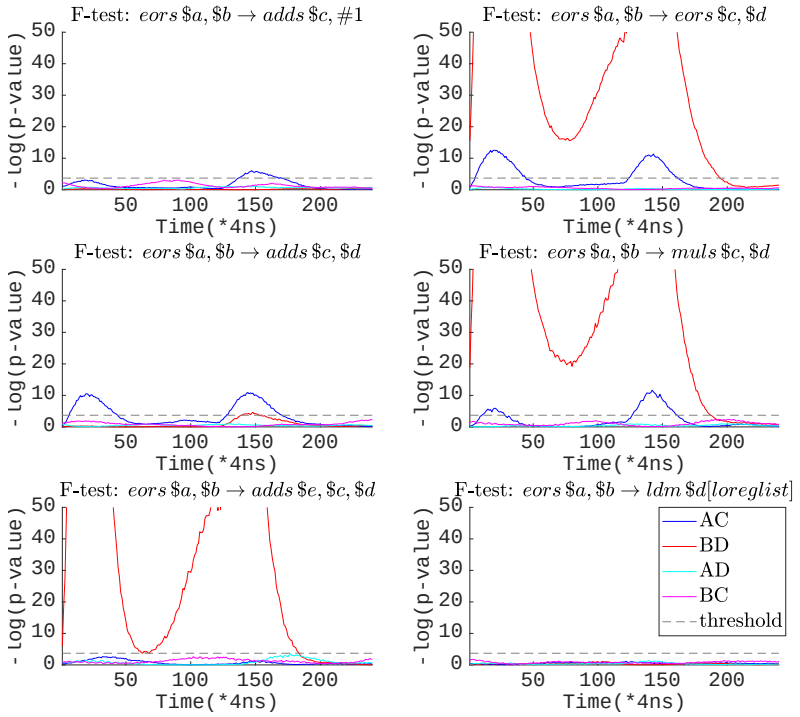


Fig. 4: Leakage analysis on register access in the decoding stage.

Table 1 gives a concise summary of the instructions and our findings through leakage analysis, which we explain subsequently.

Results. Our analysis shows that the decoding leakage (i.e. which operand is loaded through which port) strongly depends on the instruction encoding. More specifically, the column *Encoding* in Table 1 demonstrates the encoding bit-field of each instruction: ARM often uses R_d to represent the destination register and R_m/R_n represent the source registers. The assembler instruction uses those explicitly, yet did not explicitly explain the distinction (especially for R_m and R_n), or whether it links to any micro-architecture element. From our following analysis, it seems there is at least some connection.

Let us first look at some concrete F-test results as given in Figure 4. In this figure, the black dashed line gives the F-test threshold, and any of the coloured lines that exceed the threshold indicates that the corresponding term cannot be dropped (or in other words, it needs to be included as a micro-architectural leakage element).

There are six sub-figures, which correspond to different cases:

– *adds* $c, #1 (Type I): only interaction AC appears, which suggests C is loaded to port 1.

Table 1: Summary of tested Thumb-16 instructions.

Group		Assembler	Encoding				Decoding			Executing	
	Operand		Type	Rd	Rm	Rn	Port 1	Port 2	Port 3	RS_1	RS_2
ALU	0	MOVS Rd, #<imm8>	I	10-8	-	-	Rd	-	-	-	-
	1	INSTR_a Rd, Rm(, #<imm3/5>)	II	2-0	5-3	-	Rd	Rm	-	Rm	-
		INSTR_b Rd, Rm(, #<imm3/5>)	II	2-0	5-3	-	Rd	Rm	-	-	Rm
		INSTR Rd, Rd, #<imm8>	I	10-8	-	-	Rd	-	-	Rd	-
		INSTR Rd, Rm	III	7,2-0	6-3	-	Rd	Rm	-	-	Rm
	2	INSTR Rd, Rm	II	2-0	5-3	-	Rd	Rm	-	Rd	Rm
		INSTR Rd, Rn, Rm	IV	2-0	8-6	5-3	Rd	Rm	Rn	Rn	Rm
		ADD Rdn, Rm	III	7,2-0	6-3	-	Rdn	Rm	-	Rdn	Rm
		MUL Rdm, Rn	IV	2-0	-	5-3	Rdm	Rn	-	Rdm	Rn
LOAD	Imm	LDR(H/B) Rd, [Rn, #<imm>]	IV	2-0	-	5-3	Rd	Rn	-	Rn	-
	Reg	LDR(H/B) Rd, [Rn, Rm]	IV	2-0	8-6	5-3	Rd	Rn	Rm	Rn	Rm
	Multiple	LDM Rn!, <loreglist>	V	-	-	10-8	-	-	Rn	Rn	-
	Pop	POP <loreglist>		-	-	-	-	-	-	C	-
STORE	Imm	STR(H/B) Rd, [Rn, #<imm>]	IV	2-0	-	5-3	Rd	Rn	-	Rn	Rd
	Reg	STR Rd, [Rn, Rm]	IV	2-0	8-6	5-3	Rd	Rn	Rm	Rn->Rd	Rm
	Multiple	STM Rn!, <loreglist>	V	-	-	10-8	-	-	Rn	Rn	-
	Push	PUSH <loreglist>		-	-	-	-	-	-	C	-

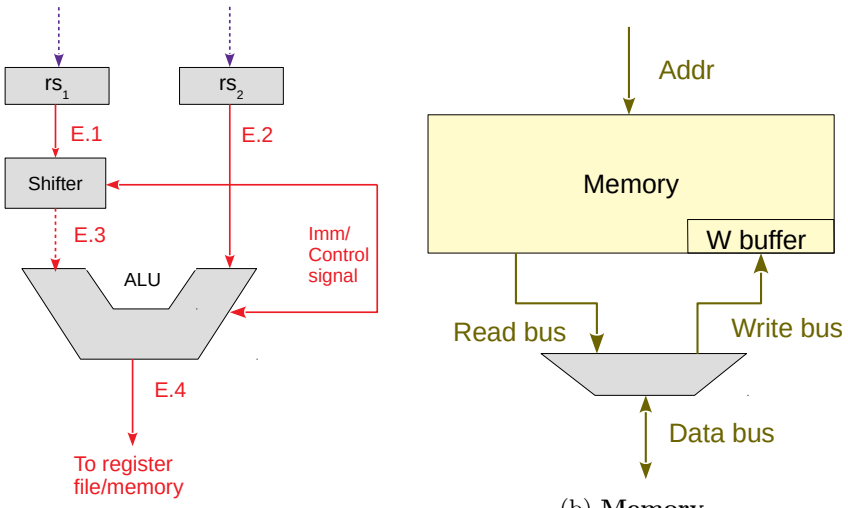
- *eors* \$c, \$d (Type II): as expected, this group shows interaction AC and BD, even if R_d is not required by the functionality (e.g. $rsbs\, R_d, R_m$). Required or not, C/D is loaded to port 1/2 respectively.
- *adds* \$c, \$d (Type III): as showed in Table 1, the only difference here is both C and D can come from a high register (R_{8-13}). Although the interaction is significantly weaker, we saw the same interaction as Type II in Figure 4 (i.e. $A \rightarrow C$ and $B \rightarrow D$).
- *muls* \$c, \$d (Type IV): unlike the previous cases, Type IV explicitly uses another register R_n (see Table 1). For *mul* and *ldr*, the leakage form is consistent: R_d (C) is connected to port 1 and R_n (D) is connected to port 2, therefore all transitions of AC and BD remain the same. We assume R_m (if used) is loaded from the extra port 3.
- *adds* \$e, \$c, \$d (Type IV, exceptional): 3-register instructions (i.e. *adds*,*subs*) are exceptional: they connect R_m instead of R_n to port 2. R_d is still loaded, yet not interacting with operand A or B. Although no concrete evidence, we set R_d to port 1 and leave R_n to port 3.
- *eors* \$a, \$b $\rightarrow$ *ldm* \$d, [*loreglist*] (Type V): this group shows no interaction; we assume R_n connects to port 3.

push and *pop* do not load any operand (other than the non-data-dependent stack register SP) in the decoding stage, therefore have been excluded from the decoding part of Table 1. Corresponding to the 3 purple dash lines (D.5-D.7) in Figure 3b, Table 1 documents the operand on each port for each instruction. Note that in a grey-box scenario, Table 1 represents the "reasonable conjectures" from leakage analysis: without reviewing the source code, this is the best possible guess we can come up with. D.9 and D.10 connect to the pipeline registers rs_1 and rs_2, which will be inspected in the **Execute** stage.

3.3 Execute

On the contrary, the **Execute** stage is relatively simple: preloaded operands start from the pipeline register (E.1 and E.2 in Figure 5a), then go through the computation logic within the ALU. The ALU's output (E.4) is then sent back to the register file or memory, depending on the specific instruction. There might be some more complicated computation logic (e.g. the multiplier in Figure 2), but from a leakage point of view, since they all connect to the pipeline registers, we simply combine everything into the equivalent ALU. Most previous tools assume there are two pipeline registers that store the operands: in our analysis, we found that 2 registers could already explain our observed leakage, therefore we stick with 2 registers in Figure 5a.

In previous tools, **Execute** is often regarded as the critical part: for instance, ELMO [1] captures the leakage/transition leakage from the 2 operands on the data buses E.1 and E.2 in Figure 5a. MAPS [2] on the other hand, captures the transition leakage on the pipeline registers rs_1 and rs_2, as well as the destination register transition in the register file (the assignment for rs_1 and rs_2 may or may not be identical to NXP's implementation). Both tools ignore the **Fetch**



(a) **Execute**. (b) **Memory**.

Fig. 5: Hypothetical micro-architecture: **Execute** and **Memory**.

and **Decode** stage and focus on part of the **Execute** stage's leakage. Recall that our analysis in the previous section did not reveal D.9 or D.10. Even if we knew what appears on D.9 and D.10, the pipeline registers rs_1 and rs_2 could still preserve their own values (driven by their control signal). Thus, the fundamental question to answer in this stage, is *which value enters* rs_1/rs_2*?*

We can perform a similar analysis as for the **Decode** stage. Specifically, let us consider the same code snippet, but targeting at the latter *eors*.

```
1 Testing_rs1rs2:
2 ...
3 eors $a,$b
4 INSTR $c,$d
5 nop
6 eors $0,$0 //$0=register that stores 0
7 ...
```

Assuming *eors* sets rs_1 to A and rs_2 to B, as the latter *eors* should have the same micro-architectural effect as the previous one, thus it would set both rs_1 and rs_2 to 0. We have tested beforehand that *nop* does not touch the pipeline registers in our target core, which is also confirmed in [13]. The purpose of having this *nop* is separating the pipelined leakage: in a 3-stage processor, when executing the latter *eors*, it is expected that the target instruction *INSTR* has already committed its result, therefore does not further affect the leakage. Thus, we can test if the operands A or B still affects the leakage for the latter *eors*: if so, the pipeline register transits as

$$rs_1 : A \rightarrow A \rightarrow 0, HD = A$$

otherwise, we further test whether C or D affects the leakage. If C is presented in the leakage, it suggests:

$$rs_1 : A \rightarrow C \rightarrow 0, HD = C$$

Considering that the observed leakage for executing the latter *eors* is not affected by the decoding stage of $INSTR$, we can have a higher confidence that C enters rs_1.

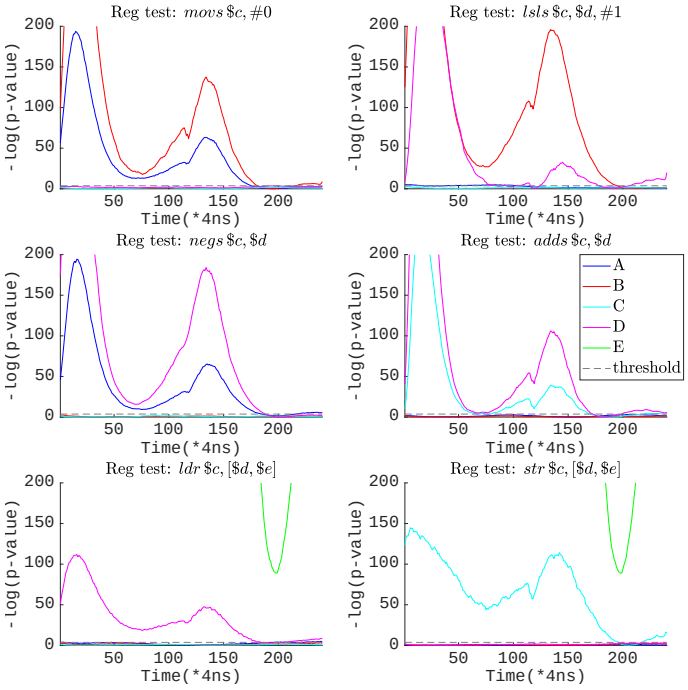


Fig. 6: Pipeline register analysis in the executing stage.

Following this approach, we have tested all instructions in Table 1. A few representative results are presented in Figure 6, namely:

- *movs $c,#imm* does not store the immediate in pipeline registers, therefore both rs_1 and rs_2 keep their previous values (i.e. A and B).
- There are two types of 1-operand ALU instructions (Table 1): *mov*, *shift*-s, and *add/sub* use only rs_1, while *neg/mvn*, *reverse*, and *extend* instructions utilise only rs_2.
- 2-operand ALU instructions always use both rs_1 and rs_2. Further analysing the transition shows that the left operand always goes to rs_1: that is to say, R_d goes to rs_1 if R_d contains a necessary operand; otherwise R_n enters rs_1 (i.e. *INSTR Rd, Rn, Rm*).

- For *ldr*-s, the base address (R_n) enters rs_1, while the offset (if not constant), goes to rs_2. If the offset is constant, rs_2 preserves its previous value.
- For *str*-s, the first cycle works the same way as *ldr*, while the second cycle sends R_d to rs_1.
- *pop* and *push* clear rs_1 with the address in SP, which according to our assumption, should be a constant.

It is worthwhile to mention that most our results in Table 1 regarding the pipeline registers are consistent with [2][6]. The fact that their conclusion is drawn from analysing the source code of Cortex-M3 from ARM is reassuring: our technique did successfully recover the underlying micro-architecture elements. The only exception we found is shift-s: in MAPS [2], the target operand is always set to rs_2; while our test suggests the operand goes to rs_1. Either this difference is because NXP indeed changed the design, or ARM has multiple versions of Cortex-M3 design.

3.4 Register write-back

Technically speaking, a 3-stage pipeline often does not have a dedicate register write-back stage. However, as we can see in Figure 2, the ALU output has to be written back at the end of the **Execute** cycle, which leads to a transition leakage that affects the next cycle. As both the ALU output and the value destination register are explicitly defined by the Instruction Set Architecture (ISA) and the executing code, there is no need for further investigation for such leakage.

3.5 Memory sub-system

It is well known that the memory subsystem produces various "unexpected" issues [16,3,13,18,19]. The main challenge is that while the ISA specifies *what should happen in the processor*, it certainly does not specify the detailed design of any asynchronous component (i.e. the memory). More specifically, in our case, ARM specifies the memory interface through the AMBA APB Protocol [20]: the protocol defines how the processor should communicate with the peripheral, performing read/write operations. However, the peripheral is asynchronous (aka self-timed) to the processor, therefore the response time as well as internal interactions are completely up to the peripheral. Take *ldr/str* instructions for instance, although it is often assumed they take 2 cycles, in practice, the situation is much more complicated. The peripheral can prolong the transfer by adding wait states [20], or for certain instructions, the ALU can proceed without the peripheral finishing its task.

As a consequence, without a timing-accurate memory simulator, the chance of constructing a timing-accurate leakage model for the memory sub-system seems gloomy. In Figure 5b, we construct a hypothetical view that captures various known issues (e.g. from [13,3]). Specifically, we assume our memory system works as follows:

[6] Available in their code repository, not in the paper.

- Each load/store produces leakage on the entire word (32-bit), even if the target is only one byte (see Section 5.2 [13]).
- The memory system has only one (shared) address bus (specified by [20]).
- The memory buses preserve their values until the next access (recommended in [20]).
- Read and write share the same data bus (consistent with Section 5.1 [13]).
- There is also a dedicate bus/buffer that holds the write value (our own experiments).

Although Figure 5b does not specify the timing behaviour, fortunately, the accurate timing is not required for many application scenarios (e.g. leakage simulators [1,2,3]/verifiers [21]): for developers, it is essential to learn *why such leakage appears*, but less crucial *when*.

Take two adjacent store instructions for instance, as long as we know there exists a transition leakage on the write bus, we do not necessarily care about *whether this leakage appears at clock cycle* x *or* $x + y$. Whilst a more detailed investigation on timing characteristics might be possible, they becomes increasingly unrewarding in a grey-box setting.

4 Step 3: Refining the micro-architectural leakage model

In the previous section, we reverse engineered where operands are stored in the micro-architecture, and we developed a first understanding of the interactions between operands across the three pipeline stages. Now we set out to refine this understanding and characterise the interactions.

4.1 Considering components with stable signals

Fetch. Stable signals are from micro-architectural components that do not have glitches. Because we assume our target program does not contain any "data-dependent branches", we do not need to consider elements from this stage.

Decode. Because we do not consider data dependent instructions, we can also exclude all purple wires before the register file (D.1-4, D.8) as they do not produce data-dependent leakage (i.e. remain the same between each execution). After accessing the register file, each purple wire must be considered, as it carries an operand that varies from trace to trace.

Based on the information in Table 1, we can build a simplified micro-architectural leakage model that only contains the "stable" signals in the circuit for D.5-D.7 (aka read ports 1-3). The outputs of two operand MUX-s are trickier: when rs_1 is updated, D.9 carries the updated value. However, when rs_1 preserves its previous value (e.g. *rsbs Rd, Rm*), we cannot determine the value on D.9 easily. Considering the same leakage could come from various equivalent micro-architectures, we consider them separately in Section 4.2.

Thus, the assumed micro-architectural leakage for the decoding stage is:

$$L_d = \{port_1 \otimes port'_1, port_2 \otimes port'_2, port_3 \otimes port'_3\}$$

where $port'_1$ represents the value on port 1 from the previous instruction decoding. If both values on port 1 are not constant,

$$port_1 \otimes port'_1 = \{(x||y)|x \in port_1, y \in port'_1\}$$

Otherwise, if one of the values is a constant, this term can be simplified to only $port_1$ or $port'_1$. This leakage is a super set of both the standard HW and HD model, covering not only the leakage of the values but also any transition occurring on the wire.

Using again the *collapsed F-test* [11], we can interrogate if this model explains all observable leakage (in the decoding stage). Figure 7 plots the evaluation for the same instructions in Figure 4: for all but one instruction L_d is correct. Only for the 3-operand *adds*, the test result suggests L_d cannot explain all the observed leakage within this decoding stage (which will be further studied in Section 4.2).

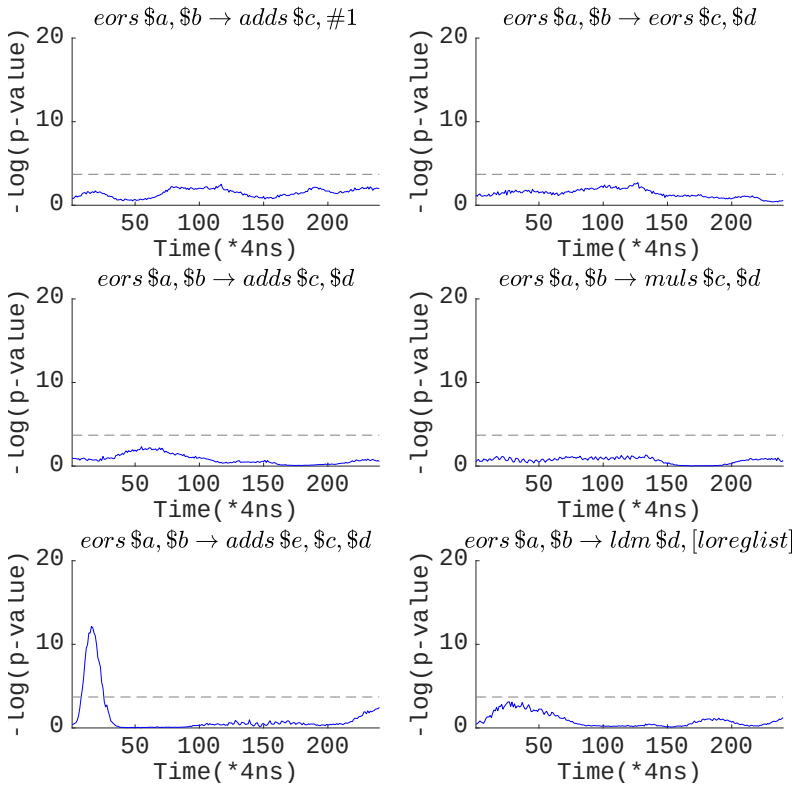


Fig. 7: Model completeness analysis in the decoding stage.

Execute. Similarly, for the **Execute** stage, we ignore the immediate number and the control signals, and focus on the wires E.1, E.2 and E.4. Obviously, the entire leakage of this stage depends on the two operands in rs_1 and rs_2.

Unlike the **Decode** stage, these two operands deliberately interact with each other in the ALU. Thus, it is expected that there is some cross-operand leakage. Considering that the ALU is a relatively complicated piece of combinational logic where multiple computations run in parallel (i.e. not "gated" [18,19]), finding the exact form of L_E presents its own challenge. Therefore, we leave L_E conservatively as:

$$L_E = \{rs_1 \otimes rs_2 \otimes rs_1' \otimes rs_2'\}$$

Clearly L_E includes all possible glitchy states on the red wires in Figure 5a. The data-dependent bits in the Current Program Status Register (CPSR) rely on the ALU's output, and are therefore also covered by L_E. For conciseness we refer to [11] where one of the examples analyses just this situation; besides, Figure 9 also evaluates the entire leakage model, including the execute leakage here.

Register write-back. Although a write-back stage does not exist in this 3-stage pipeline, updating the destination register still happens after the **Execute** stage: thus, we need a separate micro-architectural leakage element L_{WB} to capture such leakage. Denote the ALU output from the last cycle as Res and the previous value of the destination register as R_d. The register write-back leakage element L_{WB} can be written as:

$$L_{WB} = \{Res \otimes R_d\}$$

Note that Res is defined by the ISA and R_d is architecturally visible, therefore does not take any further investigation.

Memory. Following Section 3.5,

we denote Bus as the shared bus and Bus_w as the dedicate write bus, where $Addr$ represents the address bus. The micro-architectural leakage of the memory subsystem is :

$$L_M = \{Bus \otimes Bus', Bus_w \otimes Bus_w', Addr \otimes Addr'\}$$

Although some of above leakage only appears for memory access instructions, considering the APB protocol explicitly recommends to keep the remaining values on the bus [20], we always keep L_M as part of the leakage model, even if the instruction does not access memory.

4.2 Glitch & Multiplexer

Glitchy register access. The lower left figure in Figure 7 suggests that considering only the stable signals is not always enough. In a more realistic scenario, the situation can be even worse: in order to achieve a concrete understanding of *which operand is read from which port*, we deliberately designed our setup (see

Section 3.2) to avoid various "known issues". For instance, it is reported back in 2017 by Papagiannopoulos and Veshchikov that some processors might implicitly access an adjacent register while accessing a target register [16]. It was latter explained in [18] such leakage is likely to be caused by the address decoding in the register file. When setting up our experiments, we deliberately use only the odd registers (i.e. r_1, r_3, r_5, r_7): although there is no guarantee that such LSB-neighbouring effect is the only type of neighbouring effect in our target processor, within 50k traces, we did not find this effect in our analysis.

Nonetheless, the so-called "neighbouring effect" [16] can be extended to more general glitchy accesses within the register file: in a 3-stage core, considering the decoding and operand pre-loading are happening in the same cycle, it is expected that the signal glitch starts even earlier, say from the decoded register addresses (i.e. D.2-D.4 in Figure 3b). Back to our exceptional *adds*: as one can see in Table 1, the previous *eors* loaded R_m from bit 5-3 of the instruction, while the current *adds* requires R_m from bit 8-6 instead. Considering this change of field needs to be initiated by the decoder, we can expect that for a short time after the clock edge, the decoder still outputs R_m as the bit 5-3 of the new *adds* instruction (i.e. $R_n = C$), and then switches back to bit 8-6, which gives $R_m = D$. In other words, although the stable signal on $port_2$ changes as:

$$B \rightarrow D$$

the glitchy signal switches through:

$$B \rightarrow C \rightarrow D$$

which might give the transition of $B \otimes C$ and $C \otimes D$.

As we can see in Figure 8, the interaction $C \otimes D$ is clearly visible in the upper-left. Without including this micro-architectural leakage our constructed leakage model does not fully explain the observed leakage.

The lower half of Figure 8 demonstrates another case of this effect. Specifically, if we try the following code:

```
1 glitchy_reg:
2 ...
3 eors r5,r7    //r5=C, r7=D
4 adds r3,#1    //r1=A, r3=B
5 ...
```

Following our discussion above, when decoding *adds*, there might be a short time period when the decoder still decodes in the style of *eors*. According to Table 1, this means the immediate number 1 will be taken as register r_1 (bit 2-0 from the instruction *eors*). In the lower left of Figure 8, clearly value A is loaded in this cycle. In fact, as the signal transition goes $C \rightarrow A \rightarrow B$, there could be interaction of $C \otimes A$ and $A \otimes B$, which is exactly what we see in Figure 8. Our completeness test confirms that we can capture all of this micro-architectural leakage as long as these terms are added in.

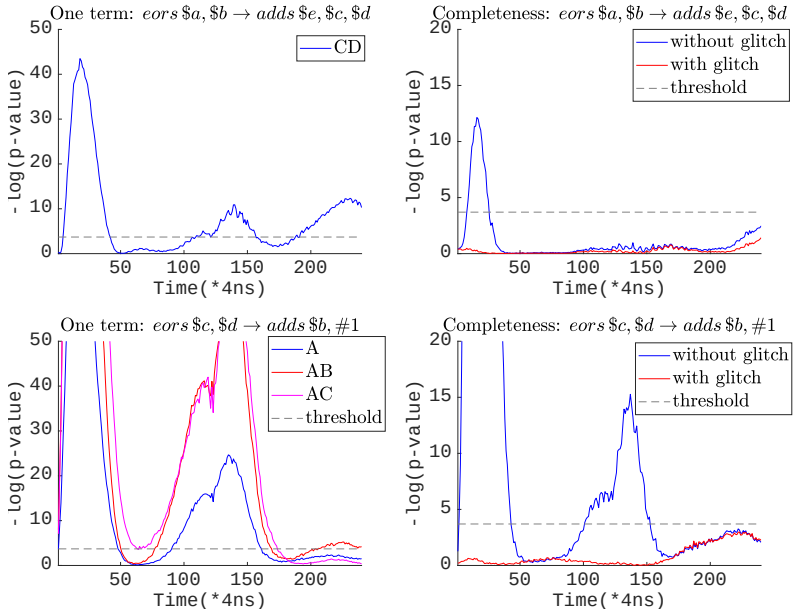


Fig. 8: Glitchy register access in the decoding stage.

Taking glitches into consideration, we add one glitchy term for each port: for port 1, denoted as $port_{1g}$, representing the glitchy accessed value on port 1. The glitchy decoding stage leakage can be regarded as

$$L_{Dg} = L_D + \{port_{1g} \otimes port_1, port_{1g} \otimes port_1'\}$$

where $port_{1g}$ could be:

- Implicitly access caused by decoding: decoding the current instruction in the previous style
- Implicitly access caused by register address: the neighbouring effect, needs to be tested on the specific device

$port_{2g}$ can be added following the similar rules. Considering such an effect has a relatively small magnitude and enormous test space (i.e. the entire decoding space must be considered), we did not further identify which factor must be added and which can perhaps be ignored. A conservative micro-architectural leakage model will include everything, if more implementation details were available then certain elements could be excluded (with the resulting model checked via the F-test).

Multiplexer. Grey-box simulators usually discard them, because their contribution to the overall leakage is relatively limited. We follow this approach in our work here[7].

4.3 Putting it all together

We constructed a micro-architectural leakage model for each of the three pipeline stages, and the memory subsystem. The overall device leakage is then the sum of the micro-architectural leaks: $L = L_D + L_E + L_M + L_{WB}$, and we can, using the F-test methodology, enquire if it is possible to drop or simplify some terms. Because we know that the micro-architectural leakage from the memory subsystem is always significant, there is no point in trying to simplify or drop this. However, we can check if the decode and execute leakage is significant enough (when considering it as all the pipeline stages are active). With the same code in Section 3.2 ("Testing_port"), we first test if removing L_D or L_E can provide a valid model: as we can see in the left half of Figure 9, both fail our test easily, which suggests both stages' leakage must be kept.

Thus, we further test in the right half of Figure 9 if using a linear model (i.e. a weighted HW/HD model) is good enough. The upper right figure suggests if the executed instruction is *eors*, having a linear L_D or a linear L_E passes F-test, although L_D and L_E cannot be linear at the same time. This is in fact consistent with the observations in [11]: if the instruction is relatively simple, using a linear model of the ALU inputs/output can be a valid option. The lower right of Figure 9 shows that for an *adds* instruction, the execute stage must utilise a non-linear model. But, the decoding leakage L_D can always be set to linear in our experiments: considering the decoding stage only contains buses that load values and flip from one value to another, this is quite natural. Hence we always restrict the decoding leakage L_D to be a linear micro-architectural leakage element, denoted as L_{Dl}. Similarly, as the write-back logic is relatively simple, we also simplify L_{WB} to be a linear micro-architectural leakage element (L_{WBl}). Because of the known byte-wise interactions on the memory bus [13], L_M is left without any restriction.

$$L = L_{Dl} + L_E + L_{WBl} + L_M$$

5 Putting our Micro-Architectural Model to the Test

In this section, we further show how existing simulators fail to find leakage, but our new micro-architectural model reveals it, and helps to develop concrete attacks to exploit it. Then we report on our integration of the new model in the existing emulator ELMO.

[7] One recent white-box tool, Coco [18]), takes a conservative approach: if we have MUX(s,a,b) (where s is the selecting signal), they simply allow any possible leakage by considering $a \otimes b \otimes s$.

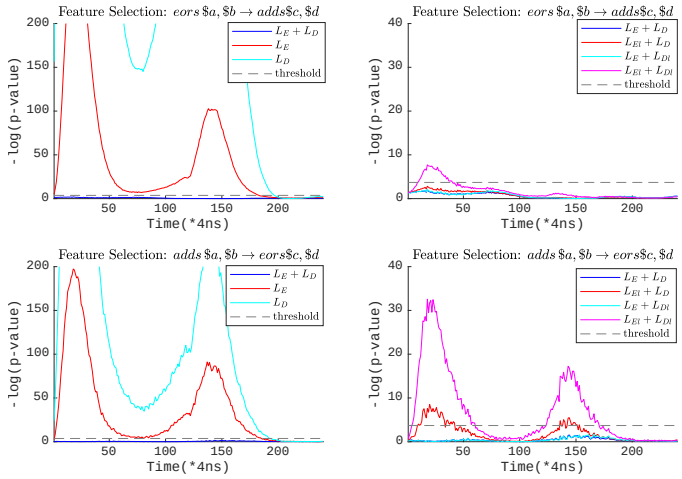


Fig. 9: Feature selection for our leakage model.

5.1 Exploiting decoding port leakage

Existing simulators typically do not define any explicit decoding leakage. In particular, ELMO (and ELMO*) always set $rs_1 = R_d$ and $rs_2 = R_m$, even if R_d is never used in certain instructions (e.g. *movs* R_d, R_m in Table 1). This is not correct (at least on the core that we utilised), but if we consider the "operand buses" in ELMO to be the decoding read ports, then decoding leakage is captured by ELMO (and ELMO*), albeit in a different clock cycle.

The following code snippet is from another 2-share bitwise ISW multiplication [5], where $a_1(b_1)$ and $a_2(b_2)$ represent the two input shares of $a(b)$. Leakage reports from ELMO [8] and MAPS are:

- ELMO. Line 7 is leaking from the first operand's bit-flip ($r_2 = a_2 * b_1 \rightarrow r_6 = a_1 * b_1$).
- MAPS. No leakage.

```
1 ISWd2:
2 ...          // r1=a1*b2+r, r2=a2, r3=b1,
3 ...          // r6=a1*b1, r9=output address
4 ands r2, r3 // r2=a2*b1
5 eors r1, r2 // r1=a1*b2+r+a2*b1
6 mov  r2, r9 // Get back output address
7 eors r6, r1 // r6=a1*b1+a1*b2+r+a2*b1
8 ...
```

[8] The ELMO* [3] extension does not find any additional leakage.

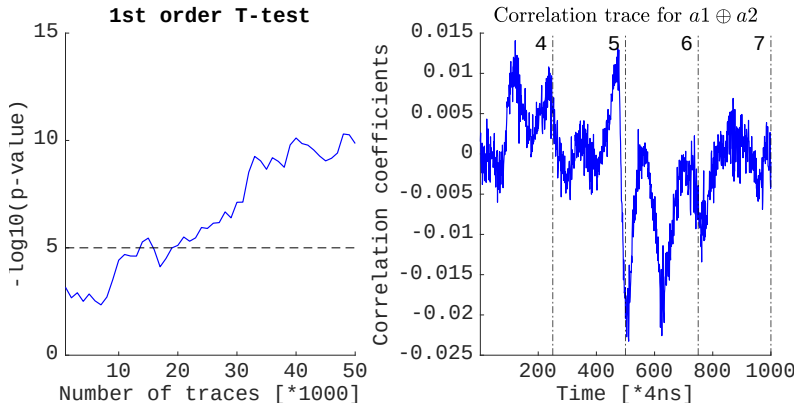


Fig. 10: Experiments with 50k traces on 2-share bitwise ISW multiplication.

According to Table 1, line 7 should not show any leakage from rs_1, since line 6 never loaded r_2 to rs_1. However, the decoding stages of line 6 and 7 load these operands into $port_1$, which suggests a leakage can be found in the decoding cycle of line 7. The right half of Figure 10 illustrates the correlation trace using $HW(a_1 \oplus a_2)$: the correlation peaks appear in the execute cycle of line 6 (i.e. the decoding cycle of line 7). Besides, the TVLA trend in the left half of Figure 10 shows such leakage is relatively weak: it takes more than 20k traces before the leakage can be stably detected.

To show that missing out on the explicit inclusion of decoding leakage matters, we further investigate a 3-share bitwise ISW multiplication (where no first order leakage can be found in this implementation).

```
1 ISWd3:
2 ...
3 mov  r7, r9        //r7=a3
4 mov  r5, r11       //r5=b3
5 ands r5, r7        //r5=a3*b3
6 ands r4, r6        //r6=a2*b2
7 ands r3, r1        //r3=a1*b1
8 ldr  r7, [r0, #0] //r7=r12
9 ...
```

In theory, there should not be any attack that combines less than three intermediates (leakage points), which increases the required number of traces. We show that a third order attack indeed does not succeed with a too limited number of traces in the upper-left of Figure 11)[9].

[9] Our implementation only uses LSB to compute the bit-sliced S-box; therefore the measured trace has been averaged 50 times before analysis, in order the increase the SNR.

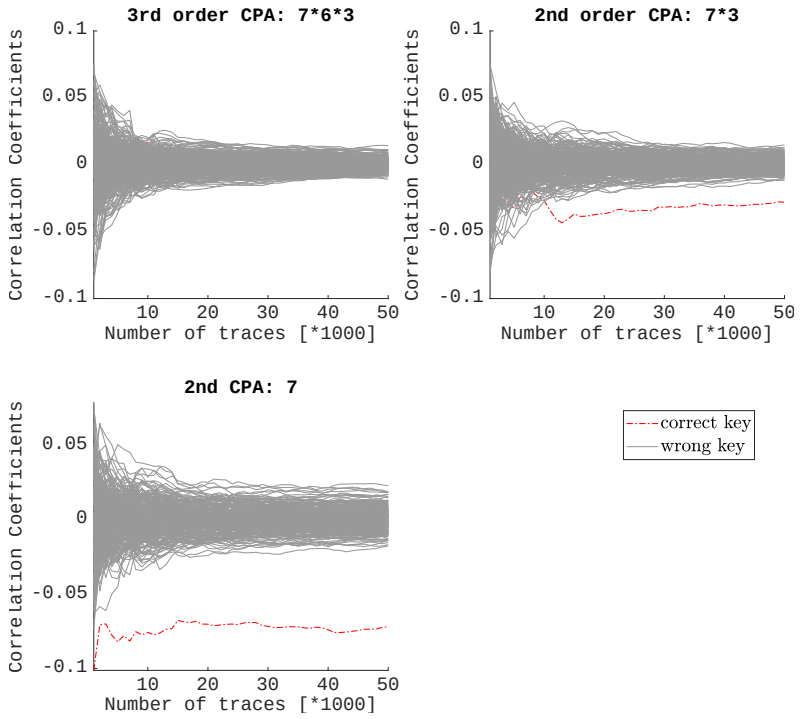


Fig. 11: Experiments with 50k traces on 3-share bitwise ISW multiplication.

However, given our micro-architectural leakage model we may safely assume that the processor will inadvertently combine masking shares for us. Indeed, even the simpler ELMO model can predict such leakage: specifically, the first operand bit-flip from line 6 and 7 gives the leakage of $a_1 \oplus a_2$, which should reveal the secret a if combined with the leakage of line 3 (i.e. a_3). We have also confirmed this leakage in Figure 11: the upper-right figure shows using this combination, the correct key can be found within 10k traces.

With the decoding leakage added in, our micro-architectural leakage model extensively expands the region of potential leakage: considering the execute cycle of line 7, we know from Table 1 that *ldr* does load r_7 in the decoding cycle, which provides the leakage of a_3. As a consequence, we can use the second order moment of the measurements from line 7 alone, which avoids the combination of noise from different time samples. In our experiments, this is indeed the best option: the correct key guess can be found with only 1k traces. In other words, the supposedly provably secure scheme can be attacked in a univariate manner, but none of the existing simulators would reveal this weakness.

5.2 Consequences of incorrectly assigning pipeline registers

When it comes to the pipeline registers, many instructions do not follow the default $rs_1 = R_d$ and $rs_2 = R_m$ setup as Table 1 shows. Considering the pipeline registers may even be preserved through a few instructions, this is clearly an issue if some previous operand is believed to be cleared out of the context while in reality it is sitting somewhere within the processor.

```
1 Scverif_Ref:
2 ...                  // r1=output address
3 ...                  // r5=a2+r,r3=a1+r
4 str  r5, [r1, #4]    //
5 pop  {r4-r5}         // Reload r4 and r5
6 eors r3, r3          // Clear r3
7 ...
```

The above code is from a 2-share refreshing gadget verified by scVerif [21]. Clearly, any transition between r_3 and r_5 leaks the secret a. Let us focus on line 5: scVerif treats *pop* as several *load*-s, where each *load* clears both rs_1 and rs_2 (see [21, Alg.3]). According to Table 1, *pop* on our target device only clears rs_1, but not rs_2. Thus, when executing line 6, rs_2 remains the previous value set by line 4. According to Table 1, there is a transition between $a_2 + r$ and $a_1 + r$ on rs_2. Leakage reports from ELMO and MAPS are:

- ELMO/ELMO*. No leakage.
- MAPS. Leakage from line 6, pipeline registers[10].

Here we begin to witness the benefit of having the accurate pipeline register assignment: MAPS clearly points out this leakage, while neither ELMO nor ELMO* finds any leakage. This is because both ELMO and ELMO* stick with ELMO's leakage model, which also believes *pop* clears both rs_1 and rs_2. At least on our M3 core, this is not the case: as we can see in the right half of Figure 12, a clear correlation peak in line 6 suggests rs_2 still keeps the value from line 4. Since this transition is from rs_2 (versus the port transition in Figure 10), the left half of Figure 12 shows its leakage is much easier to detect compared with Figure 10.

5.3 Towards a micro-architectural simulator: μElmo

Our goal is to extract micro-architectural leakage to improve simulation tools: thus we included the reverse engineered micro-architectural leakage elements into the instruction set simulator that underpins ELMO and created an upgraded version of ELMO, denoted as μELMO. The original ELMO already emulate rs_1 and rs_2 (in two variables $op1$ and $op2$), therefore the required revisions are a) updating rs_1 and rs_2 according to Table 1 and b) adding the decoding ports/memory buses as new variables in μELMO.

[10] MAPS needs the command line argument "-p" to calculate the pipeline registers' leakage.

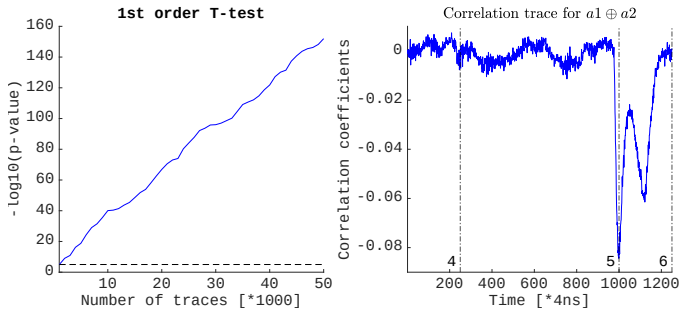


Fig. 12: Experiments with 50k traces on 2-share mask refresh from scVerif.

With μELMO, we now revisit the 2-share ISW multiplication in Section 1. We study the simple case where only a single bit is actually encoded in each processor word (recent work has shown that any more clever form of bit or share-sclicing would be insecure [12]). Thus except for a single bit (representing a share) the other bits are always constant 0. This implies that any term $rs_1 \otimes rs_2$ can easily be simplified as $\{rs_1, rs_2, rs_1 \oplus rs_2\}$: in the 1-bit version, the later can express any joint leakage from $rs_1 \otimes rs_2$. Using similar converting rules for the entire L, we can easily derive a model that only contains xor-sum terms, not any joint term. We then perform 1st order t-test on each individual term within this cycle separately (e.g. rs_1, rs_2 and $rs_1 \oplus rs_2$) and summarise the leakage of the entire cycle from multiple t-statistics.

Of course, this masking implementation is inefficient: instead one would attempt to simultaneously compute other 1-bit multiplications. If it was possible to ensure their mutual independence we can still simplify the multi-bit $\{rs_1 \otimes rs_2\}$ as $\{HW(rs_1), HW(rs_2), HW(rs_1 \oplus rs_2)\}$ as before (if not then their interaction terms would need to be considered as well).

We started this paper by showing how this ISW example leaks in practice but all simulators fail to correctly identify the leaks in Section 1. Using μElmo traces leads to the detection result in the left of Figure 13. The detection correctly idenfies the two leaks.

6 Conclusion

We utilised a recent statistical tool for statistical model building to reverse engineer the micro-architectural leakage of a mid-range commodity processor (the NXP LPC1313). This reverse engineering effort enables us to build more accurate leakage models, which are essential for accurate leakage simulators. As a side effect our model provides, to some extent, an in-depth picture of how the ARM Cortex M3 architecture is implemented in the LPC1313.

Our research was motivated by the observation that the most recent leakage simulators are inaccurate and consequently traces produced by them will not

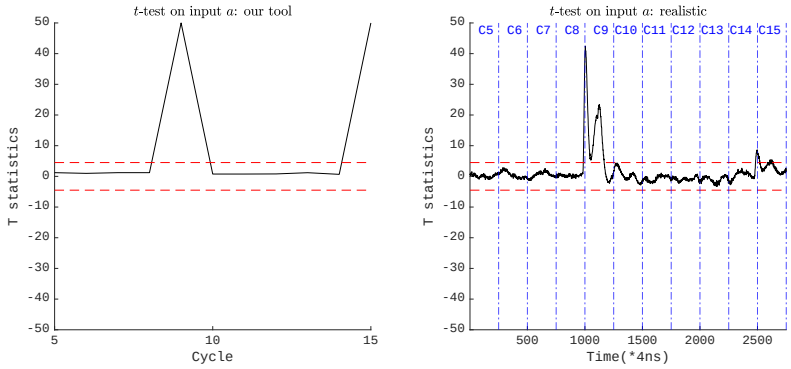


Fig. 13: Comparison of our tool and realistic measurements.

show all leaks that can be found in real devices (or they show leaks where in actual fact are none). We integrate our reverse engineered micro-architectural elements into the simulator that underpins ELMO and then demonstrate that the resulting simulator produces leakage traces that are more faithful to real device traces in the context of enabling the detection of leaks and when they occur.

Our methodology is generic in the sense that it relies on a statistical test that can deal with large numbers of variables (with limited data). For now we need to manually instrument the statistical tests to recover micro-architectural leakage information. However, merging our results with the recently introduced idea of software kernels in [13] could enable automation of our method in the future. This would be a significant step towards being able to produce highly accurate leakage simulators for a range of off-the-shelf processors.

Acknowledgments

We would like to thank Ben Marshall for his invaluable insights, which guided us through various mazes in our leakage modelling efforts. Si Gao and Elisabeth Oswald were funded in part by the ERC via the grant SEAL (Project Reference 725042). This work has been supported in part by EPSRC via grant EP/R012288/1, under the RISE (`http://www.ukrise.org`) programme.

References

1. McCann, D., Oswald, E., Whitnall, C.: Towards Practical Tools for Side Channel Aware Software Engineering: 'Grey Box' Modelling for Instruction Leakages. In: 26th USENIX Security Symposium (USENIX Security 17), Vancouver, BC, USENIX Association (2017) 199–216

2. Corre, Y.L., Großschädl, J., Dinu, D.: Micro-architectural Power Simulator for Leakage Assessment of Cryptographic Software on ARM Cortex-M3 Processors. In Fan, J., Gierlichs, B., eds.: Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings. Volume 10815 of Lecture Notes in Computer Science., Springer (2018) 82–98
3. Shelton, M.A., Samwel, N., Batina, L., Regazzoni, F., Wagner, M., Yarom, Y.: Rosita: Towards Automatic Elimination of Power-Analysis Leakage in Ciphers. CoRR **abs/1912.05183** (2019)
4. Buhan, I., Batina, L., Yarom, Y., Schaumont, P.: SoK: Design Tools for Side-Channel-Aware Implementions. `https://arxiv.org/abs/2104.08593` (2021)
5. Ishai, Y., Sahai, A., Wagner, D.A.: Private Circuits: Securing Hardware against Probing Attacks. In Boneh, D., ed.: 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., Springer (2003) 463–481
6. McCann, D.: ELMO. `https://github.com/bristol-sca/ELMO` (2017)
7. Clavier, C.: Side Channel Analysis for Reverse Engineering (SCARE) - An Improved Attack Against a Secret A3/A8 GSM Algorithm. IACR Cryptol. ePrint Arch. **2004** (2004) 49
8. Goldack, M.: Side-Channel based Reverse Engineering for Microcontrollers (2008)
9. Wang, X., Narasimhan, S., Krishna, A., Bhunia, S.: SCARE: Side-Channel Analysis Based Reverse Engineering for Post-Silicon Validation. In: 2012 25th International Conference on VLSI Design. (2012) 304–309
10. Oswald, D., Strobel, D., Schellenberg, F., Kasper, T., Paar, C.: When Reverse-Engineering Meets Side-Channel Analysis – Digital Lockpicking in Practice. In: Selected Areas in Cryptography – SAC 2013, Springer Berlin Heidelberg (2014)
11. Gao, S., Oswald, E.: A Novel Completeness Test and its Application to Side Channel Attacks and Simulators. IACR Cryptol. ePrint Arch. (2021) `https://eprint.iacr.org/2021/756`.
12. Gao, S., Marshall, B., Page, D., Oswald, E.: Share-slicing: Friend or Foe? IACR Transactions on Cryptographic Hardware and Embedded Systems **2020**(1) (Nov. 2019) 152–174
13. Marshall, B., Page, D., Webb, J.: MIRACLE: MIcRo-ArChitectural Leakage Evaluation. IACR Cryptol. ePrint Arch. (2021) `https://eprint.iacr.org/2021/261`.
14. ARM Limited: ARM®v7-M Architecture Reference Manual. `https://developer.arm.com/documentation/ddi0337/e` (2005)
15. ARM Limited: Thumb® 16-bit Instruction Set Quick Reference Card. `https://developer.arm.com/documentation/qrc0006/e` (2008)
16. Papagiannopoulos, K., Veshchikov, N.: Mind the Gap: Towards Secure 1st-Order Masking in Software. In Guilley, S., ed.: 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., Springer (2017) 282–297
17. De Cnudde, T., Ender, M., Moradi, A.: Hardware Masking, Revisited. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**(2) (2018) 123–148
18. Gigerl, B., Hadzic, V., Primas, R., Mangard, S., Bloem, R.: Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs. In Bailey, M., Greenstadt, R., eds.: 30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021, USENIX Association (2021) 1469–1468

19. De Meyer, L., De Mulder, E., Tunstall, M.: On the Effect of the (Micro)Architecture on the Development of Side-Channel Resistant Software. IACR Cryptol. ePrint Arch. **2020** (2020) 1297
20. ARM Limited: AMBA® APB Protocol. `https://developer.arm.com/documentation/ihi0024/c/` (2010)
21. Barthe, G., Gourjon, M., Grégoire, B., Orlt, M., Paglialonga, C., Porth, L.: Masking in Fine-Grained Leakage Models: Construction, Implementation and Verification. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(2) (2021) 189–228