ELMO: Emulating Leaks for the ARM Cortex-M0 without Access to a Side Channel Lab

David McCann, Carolyn Whitnall and Elisabeth Oswald

Abstract

Power (as well as EM, cache and timing) leaks are a great cause for concern for developers who have to deal with cryptographic components as part of their overall software implementation, in particular in the context of embedded devices. Whilst there are some tools to detect timing and cache leaks, progress towards pinpointing power and EM leaks has been hampered by the limited information available about the physical components from which such leaks originate. We suggest a novel modelling technique that is capable of producing instruction-level power (and/or EM) models and use it to develop ELMO, the first leakage simulator for the ARM Cortex M0. We show that our methodology is capable of capturing differential data-dependent effects as neighbouring instructions in a sequence vary. We also explore register effects, and verify our models across several measurement boards to comment on board effects and portability. Finally we evaluate ELMO's performance and utility, and show that it can be used to spot even subtle leaks in implementations.

1 Introduction

Inspired by modern approaches to linear regression-based profiling [3, 21] we develop a novel strategy to capture complex leakage from a modern microprocessor with the aim of creating accurate instruction-level leakage models appropriate for use with leakage simulators. As advocated in very early research [6, 18] such simulators could be useful to assess side channel leakage near the start of the (software) design process. More recent works [5, 20] emphasise the same motivation but take a different view of flexibility (the latter contribution works on a high-level code representation in C++, while the former is based on assembly code). Among these existing works only [5] emphasises the importance of, but also the complexities involved in, deriving leakage models when building such a simulator.

A goal related to modelling the instantaneous power consumption is that of modelling the energy consumption. Foundational research here showed clearly that accurate instruction-level (energy) models must take into account effects from previous instructions [19]. One would expect then that this is also relevant when profiling the instantaneous power consumption. In the context of profiling implementations of cryptographic algorithms (e.g. AES) for differential power analysis (DPA) attacks, there are no differential effects due to prior instruction sequences, because a fixed code sequence (e.g. AES) is executed. Hence, any 'prior-sequence effect' is captured automatically in this context.

However, in our context, i.e. the profiling of **general instructions**, no target code sequence is available and consequently **all** combinations of sequences (of some fixed length) need to be considered—a task with an infeasibly high combinatorial complexity given the number of instructions in a modern microprocessor. Further, if indeed the created leakage profiles are to be used as part of a leakage simulator, there is a need to ascertain how **portable** the profiles are. In this specific context, portability refers to minimising measurement setup and board dependent effects, such that the models reflect the microprocessor leakage in its 'purest' form and port well across similar devices (albeit on different boards). Recent contributions [4, 9, 14] emphasise the interest in portability and evaluate practically the challenge of creating portable templates (albeit for the purpose of improving attacks), when estimating the parameters of the leakage distribution.

1.1 Motivation and Approach

Given the popularity of embedded micro-processors in Internet-of-Things (IoT) devices, the problem of early evaluation of the leakage properties of security critical code becomes a pressing concern in this domain. Unlike developers within the traditional smart-card community, IoT developers typically do not enjoy access to or support from either in-house side-channel testing facilities or external evaluation labs (even supposing they are conscious of the threats associated with power and EM attacks). Furthermore, the way in which evaluators and independent testers work is not easily compatible with the rapid pace of current advancement within the IoT community, or with the financial resources available to small start-ups for their various projects. On this basis we argue that there is a need for an easy-to-use simulator, compatible with established software development practice.

IoT developers generally develop in C and/or Assembly, depending on the performance requirements of a given application. A user-friendly tool will be one which 'slots in' accordingly; this can be relatively easily achieved by building models for Assembly instructions. Using a suitable (for the specific processor) instruction-set simulator, it is possible to extract an accurate data-flow model, based on which the instruction-level models can be used to produce power traces. We put this idea into practice for a chosen 'interesting' processor in the IoT context: the ARM Cortex M0. For this processor there are number of development boards available which allowed us to acquire measurements to create the instruction-level profiles. In addition, there are a number of instruction-set simulators; for the purposes of (eventually) releasing a full open-source version of our tool, we selected a popular open-source one of these.

1.2 Our Contributions

This paper sets out our approach for producing an accurate and easy-to-use instruction-level simulator for the ARM Cortex-M0. To this end we aim to understand and usefully model the data-dependent leakage of instructions on an ARM Cortex-M0 processor. We identify 15 key instructions and investigate how the operands to these (two 32-bit words in most cases) influence the form of the power consumption at the relevant points in the trace. Note that outputs from one instruction will form the inputs to another, so that all possible target values in an implementation are covered by such an approach.

Drawing on some *a priori* knowledge about the device and its leakage, our methodology is to fit linear regression models (as introduced to the side channel community by Schindler et al. [16]), via ordinary least squares (OLS) estimation in function of the input bits to a given instruction and the bit-wise transitions between these and the inputs to the previous instruction. In other words, we consider sequences of three instructions, in order to model one (i.e. the middle) instruction. We find this to be adequate in the case of the M0 processor.

Modelling leakage in this way enables us to identify which (groups of) bits or bit-flips have a significant effect on the power consumption and which can be disregarded (we use F-tests to assess joint significance). As well as increasing our understanding of device leakage, it is substantially more comprehensive than relying on Hamming weight or Hamming distance simplifications, as it allows each bit or bit transition to contribute differently. If such differences are truly present then we thereby explain an increased proportion of the exploitable variation, providing models which (for example) better simulate the true power consumption behaviour of a device, and/or attain larger correct-hypothesis scores in standard correlation DPA attacks.

Using the same process we include and test other potentially important factors such as board and register variation to see if they impact differentially on leakage data-dependency and need to be taken into account (we conclude not, in both cases—although with some provisos in the latter). We use clustering to verify our intuition that certain groups of instruction will leak similarly, thereby reducing the complexity of the model building task to the realms of practicality. Finally, show how to utilise these models to detect leaks in cryptographic implementations.

A key feature of our work is that it is data-driven and decisions are made on the basis of rigorous statistical tests. In the process, we discover several leakage features specific to the Cortex-M0,

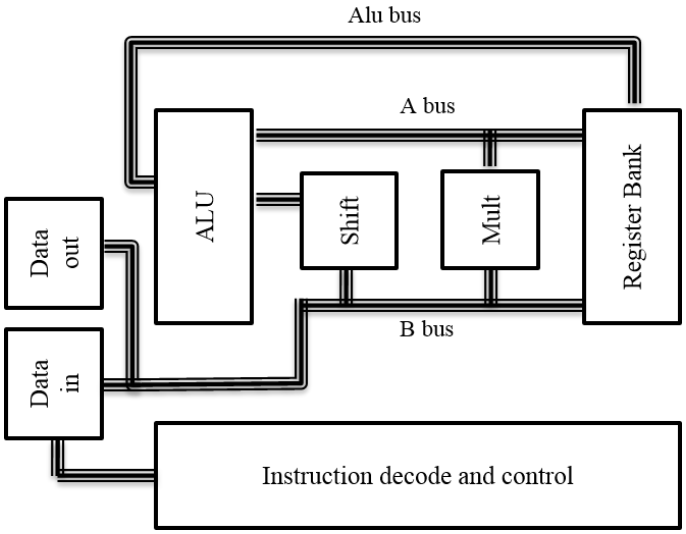

Figure 1: Simplified ARM CPU architecture (redrawn from [7]) for a 3-stage pipeline architecture .

which are of independent interest.

The remainder of the paper proceeds as follows: in Section 2 we describe the device, the measurement setup, and our carefully tailored acquisition procedure; in Section 3 we propose our methodology for leakage characterisation; in Section 4 we explore the data-dependent leakage of each considered instruction taken individually, and the possibility of board and register effects; in Section 5 we show how to model instructions in sequence, allowing for the effects of neighbouring instructions—which entails first reducing the full set of 15 to a demonstrably representative subset of 5 using clustering analysis. In Section 6 we explain how to use the models to simulate power traces, analyse them, and then conclude on leaking instructions. Closing remarks and open questions follow in Section 7.

2 Background and Setup

To keep our paper reasonably self-contained, we begin this section with a brief summary of some device-specific features. Thereafter we describe our measurement setup and acquisition strategy, and outline some basic assumptions that are necessary for our novel modelling strategy.

2.1 Arm Cortex-M0 Processor and Choice of Instructions

The ARM Cortex-M0 processor [1] is the smallest of the ARM Cortex processor family. It is widely advertised as ARM's flagship processor for the Internet of Things. It uses a 32-bit von-Neuman architecture with a three stage pipeline. Whilst its exact CPU architecture is not publicly available, it can be assumed that it resembles the basic architecture of ARM cores, which is explained in detail in [7]. Figure 1 shows a simplified version of the basic architectural components: besides the arithmetic-logic unit (ALU), there exists a hardware multiplier, and a (Barrel) shifter. The register banks feed via two busses into the ALU, one of the busses is also connected to some data in/out registers. There is a third bus that connects the output of the ALU back into the register banks.

The M0 implements the ARMv6-M Thumb instruction set, which includes the majority of the standard 16-bit Thumb instruction set as well as a number of 32-bit Thumb2 instructions. We work with an implementation of this processor by ST Microelectronics, which is situated on an STM32F0 (30R8T6) Discovery Board [17]. This board features an ST-Link to flash programs

to the processor and provide on-chip debugging capabilities. It also has an on-board 8MHz RC oscillator clock signal.

In this work we focus only on a select number (15) of 16-bit Thumb instructions. The register configuration of these Thumb instructions is the same as that of the general ARM assembly language with certain additional constraints, such as the majority of instructions being able only to operate on registers 0-7. The 15 Thumb instructions that were selected for profiling, as being commonly used in implementations of (symmetric) cryptographic primitives, are: `ldr`, `ldrb`, `str`, `strb`, `lsl`, `lsr`, `ror`, `mul`, `eor`, `and`, `add`, `sub`, `orr`, `cmp` and `mov`. Many of the other instructions available in the Thumb instruction set tend to fall into similar categories (such as loading or storing a half word). Memory addresses of branching instructions were not considered.

Understanding and interpreting the input format of the instructions is necessary in order to correctly model them. For the majority of non-memory instructions (i.e. those other than `ldr, ldrb, str, strb`), three different registers may be selected for use in the format "`inst` r_d, r_n, $r_m/\#imm$", where r_d is the destination register for the output, r_n the register holding the first operand and r_m the register containing the second operand. However, `mov` and `cmp` instructions each have only two registers: r_d, r_n and r_n, r_m respectively. It is possible for a number of these instructions (`add`, `sub` and `cmp`) to use an immediate ($\#imm$) value instead of r_m, but we consider modelling for these to be outside the scope of this current study. To simply our configuration for modelling instructions, and to ensure enough registers for the analysis of three instructions (where each register must be fixed beforehand), r_d was the same as r_n for all of these, limiting the number of registers required for each instruction to 2. This method also allowed us to more easily assess switching effects in the destination register. We therefore took operand 1 to be r_d/r_n and operand 2 to be r_m.

Memory instructions have a slightly different configuration as the second operand needs to be a valid memory address. They typically have the form "`inst` $r_t, [r_n, r_m/\#imm]$" where r_t is the register to which the data is to be stored or from which it is to be loaded (according to the functionality of the instruction), r_n is the memory address and $r_m/\#imm$ is the offset to this memory address which can either be in a register r_m or input as an immediate value ($\#imm$). The `ldr` instruction analysed was of this form rather than the alternative form which loads the memory address of a label. For our analysis we did not consider the leakage of memory addresses and so the value of the offset was simply set to 0 for all memory instructions with the memory address of r_n fixed beforehand. We therefore only really have one operand for memory instructions which is the data in r_t for store instructions and the value in the memory address of r_n ($r_n[data]$) for load instructions. These values are fixed before a sequence of instructions is executed (see the following section for further details) and the value of r_t was considered operand 1 and $r_n[data]$ operand 2.

2.2 Non-Profiled Instructions

Rather than profile *all* available instructions, we focus (at this stage) on those which are commonly used in cryptographic implementations. When analysing arbitrary pieces of code, it is important that the simulator deals appropriately with any non-profiled instructions it encounters. One option would be to assign them a leakage value according to the 'closest' profiled instruction. However, we opt for a more conservative approach, with an emphasis on transparency: a set pattern is defined for all of them, and their indices are logged so as to inform a developer of their location in the code. (The set pattern is that of an ALU instruction with zero operands).

2.3 Measurement Setup

We modified the STM32F0 boards by extracting the power pins of the processor, and passing the power supply through a 360Ω resistor. This was to minimise the potential for board and setup effects. We also verified the stability of our power supply. Eight boards were modified in this way to allow for experiments testing for board effects.

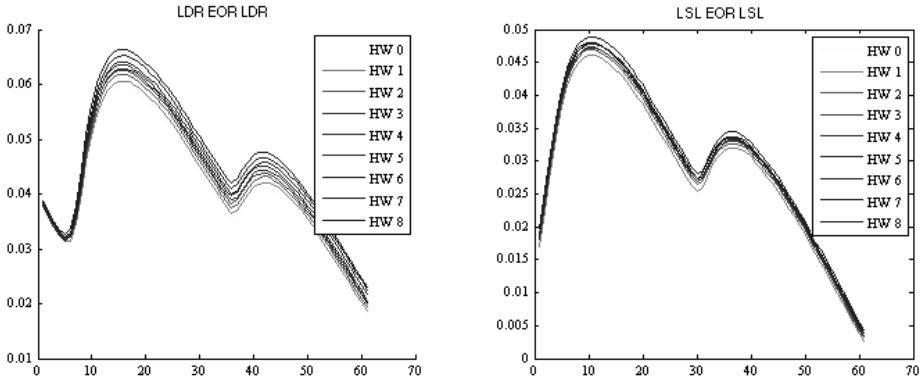

Figure 2: Leakage traces (labelled by their Hamming Weights) for an `eor` instruction neighboured by an `ldr` (left) and an `lsl` (right).

We used a Lecroy Waverunner 700 Zi scope at a sampling rate of 500 MS/S with a differential probe. The sampling rate was selected by observing DPA outcomes across different sampling rates: 500 MS/S was the lowest sampling rate at which the best DPA outcomes were achieved (with the board oscillator set to 8MHz). To lower the independent noise, we averaged over five acquisitions per input (this was the lowest number that brought the most signal gain). No filtering or further signal processing took place.

We do not seek to claim this as the best setup; rather, simply to show that we chose a sensible configuration.

2.4 Generating Sequences

It has been recognised before that the previous state of the microprocessor impacts on the instantaneous power profile of any target instruction [19]. Fig. 2 demonstrates that this is not just a theoretical consideration. It shows the visible difference in the power leakage traces for an `eor` instruction when neighboured by two different instructions (`ldr` on the left panel and `lsl` on the right).

To incorporate the effects of neighbouring instructions into our models, we need to sample traces across the range of possible triplets[1], as data inputs vary, keeping all other factors fixed as far as possible to avoid conflating instruction with other effects. In particular, the different sequences must execute as part of the *same acquisition*, to avoid acquisition or setup effects, and the measurements associated with each possible triplet must be *evenly spread* across the duration of the acquisition, to ensure that any drift or environmental fluctuation is independent of the varying sequence. Thus, we interleave the traces associated with the different sequences—as per the advice of [8] in the context of avoiding systemic bias in leakage detection.

This precludes the simple sampling solution of flashing a given sequence to the processor, gaining a set of traces and then doing the same for a different sequence at a later time (which would also be impractical and time consuming). One option we explored was to store multiple sequences on the board and to execute them in a specific order. Here, however, the number of sequences that can be implemented at one time is constrained by the memory size on the board (through experimentation, to around 350 sequences).

We eventually arrived at *dynamic compilation* as a means of configuring and running a given sequence on-the-fly. The sequence of instructions is stored in binary and then, using user-input for the different instructions required in the triplet, the correct sequence, along with the accompanying harness code (which loads the correct data into the correct registers, starts and ends the trigger

[1] Arguably, for more complex processors one could expect that more than just the preceeding and following instruction strongly impact on the instruction to be modelled.

and generates the sequence of `nop` instructions in which the triplet is embedded[2]) is generated and stored in memory at run-time. Once the required binary data has been generated and stored in this way, the program branches to the appropriate location in memory and executes the code. This is repeated as many times and for as many sequences as the user wishes.

We use this strategy to take our traces for analysis, as follows. The program first reads in three bytes of data via serial, each representing the desired instruction in the order in which they are to occur. Six 32-bit words of random data for each of the operands of the three instructions are then read. Registers `r1` and `r7` are reserved for operand 1 and operand 2 (respectively) of the first instruction, `r3` and `r4` for the second instruction and `r5` and `r6` for the third instruction. These correspond respectively to r_d/r_n and r_n/r_m for non-memory instructions and r_t and $r_n[data]$ for memory instructions in the notation of the previous section. Further analysis of the effects of different register combinations on leakage and a justification for keeping set registers for the different instructions is given in Section 4.

If the instruction selected was a memory instruction, the second operand is set to an address in memory which contains the data of operand 2. This ensures that the `ldr` and `ldrb` instructions load the expected data from a viable memory address and the `str` and `strb` instructions store data to a viable memory address.

3 A Methodology to Characterise a Modern Microprocessor

Linear regression has been used by the research community for some years already to profile side-channel leakage [16]. We introduce the use of rigorous hypothesis testing to build, refine and understand meaningful explanatory models, making informed trades between model simplicity and exhaustiveness. Groups of related variables are jointly tested for their relevant contribution to the leakage, and included or excluded accordingly.

We fit models of the following form (written in matrix notation) to the measured leakage of different instructions via OLS estimation (see, e.g., Chapter 3 of [10]):

$$\mathbf{y} = \delta + [\,\mathbf{O_1} \,|\, \mathbf{O_2} \,|\, \mathbf{T_1} \,|\, \mathbf{T_2}\,]\, \boldsymbol{\beta} + \boldsymbol{\varepsilon} \tag{1}$$

where $\mathbf{O_i} = [\,\mathbf{x_i}[0] \,|\, \mathbf{x_i}[1] \,|\, \ldots \,|\, \mathbf{x_i}[31]\,]$ is the matrix of operand bits across bus $i = 1, 2$, $\mathbf{T_i} = [\,\mathbf{x_i}[0] \oplus \mathbf{z_i}[0] \,|\, \ldots \,|\, \mathbf{x_i}[31] \oplus \mathbf{z_i}[31]\,]$ is the matrix of bit transitions across bus $i = 1, 2$ (i.e., $[b]$ denotes the bth-bit, $\mathbf{x_i}$ denotes the ith operand to a given instruction, $\mathbf{z_i}$ denotes the ith operand to the previous instruction, and '|' denotes matrix concatenation). The scalar intercept δ and the vector of coefficients $\boldsymbol{\beta}$ are the model parameters to be estimated, and $\boldsymbol{\varepsilon}$ is the vector of error terms (noise), assumed for inference to have constant, uncorrelated variance across all observations.[3] If the noise can additionally be assumed to be normally distributed then the validity of the hypothesis tests holds without need of recourse to asymptotic properties of the test statistics.

The joint statistical significance of the operands and/or bit-flips in the overall leakage can be tested using the F-test. Consider two models, A and B, such that A is 'nested' within B—that is, it has $p_A < p_B$ parameters associated with a subset of model B's fitted terms (e.g. $\mathbf{y} = \delta' + [\,\mathbf{O_2} \,|\, \mathbf{T_1} \,|\, \mathbf{T_2}\,]\, \boldsymbol{\beta'} + \boldsymbol{\varepsilon'}$ versus (1) above). We are interested in the joint significance of the terms omitted from A (in our example case, the bits of the first operand). The test statistic is computed via the residual sums of squares (RSS) of each model, along with their respective numbers of parameters p_A, p_B and the sample size n as follows:

$$F = \frac{\left(\frac{\mathrm{RSS}_A - \mathrm{RSS}_B}{p_B - p_A}\right)}{\left(\frac{\mathrm{RSS}_B}{n - p_B}\right)} \tag{2}$$

[2]The surrounding `nop`s (equivalent to `mov r8, r8` in our architecture) are designed to minimise noise from other activity before and after the triplet of interest

[3]By mean-centering each trace prior to analysis we remove drift, which could otherwise introduce auto-correlation.

Under the *null hypothesis* that the terms have no effect, F has an F-distribution with $(p_B - p_A, n - p_B)$ degrees of freedom. If then, for a given significance level (usually $\alpha = 5\%$, as we opt for throughout)[4], F is larger than the 'critical value' of the $F_{p_B - p_A, n - p_B}$ distribution[5] we reject the null hypothesis and conclude that the tested terms *do* have an effect. If F is smaller than the critical value, we say that there is no evidence to reject the null hypothesis.

In the same way, we can add other terms to model (1) and test appropriate subsets in order to rigorously explore which factors influence the form of the leakage and should therefore be taken into account in the final model. We are especially concerned with sources of variation that have a *differential* impact on the *data-dependent* contributions, as these will determine how well we are able to proportionally approximate the exploitable part of the leakage (whereas 'level' (average) effects will simply shift the model by an additive constant). In particular, we test for register and board effects on the operand and bit-flip contributions by computing F-statistics for the associated sets of interaction terms.

Having thus established the basic leakage characteristics of each instruction considered separately (see Section 4), we then turn our attention towards instructions in sequence. We build models allowing the data-dependent leakage to vary depending on the previous and subsequent instructions in series of three. However, the number of possible triplets of fifteen instructions is too many to explore exhaustively: with the aim of reducing the pool to a manageable number of representative instructions, which together cover the range of leakage behaviours within the full set, we perform clustering analysis (see, e.g., Chapter 14 of [10]) on the data term coefficients ($\boldsymbol{\beta}$) from model (1) for each instruction, and show that they naturally group into five similarly-leaking classes (see Section 5.1). We are then able to sample exhaustively over all 125 possible combinations of our 5 chosen group-typifying instructions, and build the sequence-varying models accordingly (see Section 5.2).

4 Establishing Basic Leakage Characteristics

To investigate the instruction-dependent form of the leakage in the simplest setting we measure 5,000 power consumption traces for each of the 15 instructions performed in the same fixed sequence `mov`-instr-`mov` as the two 32-bit operands vary. We identify the interesting point by inspection as the peak in the clock cycle during which the instruction is performed, and fit model (1) to the (drift-adjusted) leakage at this point.

Tab. 1 summarises the model fit. The first row presents the R-squared, which may be thought of as the proportion of the variation 'explained' by the fitted terms. In most cases, the input bits and transitions account for between 0.1 and 0.2 of the total. It falls below this range for `ldrb` and `str`.

The remaining rows present F-tests for pertinent subsets of the explanatory variables. We have shaded in grey all those which are not found to be significant. The `ldr`/`ldrb` instructions depend only upon the bits of the 2nd operand; `str` and `strb` depend only on the bits of the operands (both in the former case, just the first in the latter) and not the bit flips. For all the other instructions the null hypotheses of 'no effect' are rejected at the 5% level for all tested sets of explanatory variables.

Having confirmed (as anticipated) that the leakage (usually) significantly depends on the data operands, and the bit-flips between these and the previous values on the same bus, we now investigate whether or not the form of this dependency is influenced by board or register variation.

[4]The significance level should be understood as the probability of rejecting the null hypothesis when it is in fact true.

[5]The number large enough to imply inconsistency with the distributional assumption fixing the probability of error at α.

		add	and	cmp	eor	ldr	ldrb	lsl	lsr	mov
	R-squared	0.127	0.126	0.120	0.110	0.125	0.050	0.198	0.175	0.116
F-statistic	Operand 1	7.49	2.67	7.81	1.83	1.05	1.02	21.24	18.25	1.82
	Operand 2	3.18	2.61	2.29	2.02	18.49	5.46	4.13	2.49	2.39
	Transition 1	7.34	11.93	8.64	10.38	0.83	0.68	9.00	9.66	8.92
	Transition 2	3.61	4.61	1.69	4.18	1.23	0.92	2.99	1.38	6.51
	Combined	5.56	5.46	5.17	4.73	5.45	2.01	9.41	8.05	4.98

		mul	orr	ror	str	strb	sub	df1	df2	Critical value
	R-squared	0.373	0.104	0.186	0.070	0.139	0.135			
F-statistic	Operand 1	52.90	2.08	12.67	7.61	21.61	6.58	32	4871	1.45
	Operand 2	29.92	1.83	10.35	1.60	1.07	3.53	32	4871	1.45
	Transition 1	4.64	7.42	9.36	0.72	0.96	10.27	32	4871	1.45
	Transition 2	1.56	5.94	1.76	0.97	0.66	2.64	32	4871	1.45
	Combined	22.67	4.44	8.68	2.85	6.13	5.96	128	4871	1.22

Table 1: F-tests for significant joint data effects; tests which fail to reject at the 5% level are shaded grey.

4.1 Testing for Board Effects

To test for variation between boards we replicate the acquisition described above for a further 7 boards. We find the leaking point for each acquisition and pool the data. We then fit model (1) with the addition of a dummy for (level) board effects and we compare this against a model with the further addition of board/data interaction dummies, in order to test the joint significance of the latter.

We find a remarkable degree of consistency in the data-dependent leakage of the different boards. F-tests for the joint interaction between board and data effects do *not* reject the null hypothesis of 'no effect' for *any* of the 15 instructions (see Tab. 7 in Appendix A).

4.2 Testing for Register Effects

The ARM Cortex-M0 architecture distinguishes between low (r0–r7) and high (r8–r15) registers. The latter, which can only be accessed by the `mov` instruction, are used for fast temporary storage. These were observed by inspection to have different leakage characteristics to the low registers. However, due to their singular usage we consider them outside of the scope of this particular analysis and focus only on the low registers. For the purposes of future extensions to our methodology, we propose modelling high register `mov`s as an additional distinct instruction.

We test for variation between the eight low registers by collecting 5,000 traces for each source register (r_n) and destination register (r_d) (evenly distributed over the possible source/destination pairs) as `mov`s are performed on random inputs. We then fit model (1) with the addition of dummy variables for source register and for destination register, and compare this against a model with the further addition of register/data interaction dummies, in order to test the significance of the latter.

We find that the registers *do* have a jointly significant effect on the leakage data-dependency (see LHS of Tab. 8 in Appendix A). Considered separately, only the source register effect remains significant; at the 5% level we do not reject the null hypothesis that the destination register has no effect. Moreover, the effect can be isolated (by testing one 'source register interaction' at a time relative to the model with no source register interactions) to just half the source registers (`r0`, `r1`, `r4` and `r7`).

This analysis suggests that the inclusion of (some) source register effects would increase the ability of the model to accurately approximate the data-dependent leakage. However, such an extension would add considerable complexity; it is important to examine the *practical* significance of the effects as well as the *statistical* significance which, in large sample sizes (such as we deal

with here), will eventually be detected even for very small differences. The figure on the right of Tab. 8 (Appendix A) shows the estimated coefficients on the data terms as the source register varies. The 'significant' effect is at least small enough that it cannot be easily visualised; we judge it acceptable, for now, to exclude it from the model in order to incorporate more important factors such as the effect of previous and subsequent instructions, which we consider in Section 5.

5 Profiling Instructions in Sequence

In this section we work towards extending our instruction level models to control (and test) for the possible effects of the previous and subsequent instructions in a given sequence. The first step is to explore the scope for simplifying the problem by reducing the number of distinct instructions requiring consideration.

5.1 Clustering Instructions Into Similar Classes

Our *a priori* intuition about the grouping, based on knowledge of the device architecture (as explained in Fig. 1), is as follows. We expect ALU instructions to leak as a group (i.e. `add`, `and`, `eor`, `mov`, `orr`, `sub`,
tt cmp), load and store instructions that interact with the data in/out registers as two groups (`str`, `strb`) and (`ldr`, `ldrb`), the shift instructions that are done on the Barrel shifter as a group (`lsl`, `lsr`, `ror`), and finally we expect the multiply instruction to show a distinct profile (`mul`).

We compare this intuitive grouping with that which is empirically suggested by the data by performing clustering analysis on the per-instruction coefficients (i.e. the $\boldsymbol{\beta}$ vectors) obtained in Section 4. We use the average Euclidean distance between instruction models to form a hierarchy of clusters (represented by the dendrogram in Appendix A, RHS of Fig. 7). Adjusting the inconsistency threshold[6] between 0.7 and 1.2 produces the groupings reported in Tab. 2. These align nicely with our intuitive grouping: at threshold 0.9 the match is near-exact, with the exception that `cmp` is not singled out but falls into the large group of arithmetic instructions. At a threshold of 1.2 the instructions form a single group.

A 'good' cluster arrangement will achieve high similarity within groups and high dissimilarity between groups. The *silhouette value* is a useful measure to gauge this, defined for the ith object as $S_i = \frac{b_i - a_i}{\max(a_i, b_i)}$, where a_i is the average distance from the ith object to the other objects in the same cluster, and b_i is the minimum (over all clusters) average distance from the ith object to the objects in a different cluster [15]. Fig. 3 plots the silhouettes of each of the arrangements in Tab. 2. The consistency threshold of 0.9 is associated with the highest median silhouette value (0.48). One of the instructions in cluster 2 (`ror`; see ordering in Tab. 2) is not particularly well-matched, however it is not *wrongly* matched (as would be indicated by a negative silhouette value).

We next look at the form of the estimated leakage models. Fig. 4 plots the mean data-dependent coefficients associated with the different terms in the model equations, for each of the groups suggested by the clustering analysis with a threshold of 0.9.

The differences between the groups are immediately clear. We make the following observations:

- Group 1 (`add`, `and`, `cmp`, `eor`, `mov`, `orr` and `sub`) leak primarily in the transition between the first operands given to the current and previous instruction. However, not all the bits of this transition contribute; most of the explained leakage is in three bits of the third operand byte and one in the fourth.
- Group 2 (`lsl`, `lsr`, `ror`) appears to leak in the first operand and the transition between that and the first operand for the preceding instruction. The coefficients are largest for the third and (to a lesser extent) the fourth bytes. The transition leakage applies only to a few bits, while the operand leakage is more spread out between the bits. There is some evidence of leakage from the first two bits of the second operand.

[6]The *inconsistency coefficient* is defined as the height of the individual link minus the mean height of all links at the same hierarchical level, all divided by the standard deviation of all the heights on that level (see Matlab's `cluster` command: `http://uk.mathworks.com/help/stats/cluster.html`.

CT	Intuitive group 1	2	3	4	5	6	Instructions (in descending order of SI)
0.7	2	0	0	0	0	0	`mov eor`
	1	0	0	0	0	0	`and`
	1	0	0	0	0	0	`orr`
	1	0	0	0	0	0	`add`
	1	0	0	0	1	0	`cmp sub`
	0	2	0	0	0	0	`lsl lsr`
	0	1	0	0	0	0	`ror`
	0	0	2	0	0	0	`strb str`
	0	0	0	2	0	0	`ldr ldrb`
	0	0	0	0	0	1	`mul`
0.8	5	0	0	0	0	0	`and eor mov add orr`
	1	0	0	0	1	0	`sub cmp`
	0	3	0	0	0	0	`lsl lsr ror`
	0	0	2	0	0	0	`strb str`
	0	0	0	2	0	0	`ldr ldrb`
	0	0	0	0	0	1	`mul`
0.9	6	0	0	0	1	0	`orr mov eor and cmp add sub`
	0	3	0	0	0	0	`lsl lsr ror`
	0	0	2	0	0	0	`strb str`
	0	0	0	2	0	0	`ldr ldrb`
	0	0	0	0	0	1	`mul`
1.0	6	3	0	0	1	0	`lsr cmp orr sub lsl ror mov eor add and`
	0	0	2	0	0	0	`strb str`
	0	0	0	2	0	0	`ldr ldrb`
	0	0	0	0	0	1	`mul`
1.1	6	3	0	0	1	0	`lsr cmp sub orr lsl mov eor ror and add`
	0	0	2	2	0	0	`ldrb ldr strb str`
	0	0	0	0	0	1	`mul`
1.2	6	3	2	2	1	1	(all; SI undefined)

Table 2: Found clusters compared with intuitive grouping as the consistency threshold (CT) increases.

- Group 3 (`str`, `strb`) leaks primarily in the first byte of the first operand.
- Group 4 (`ldr`, `ldrb`) leaks across all bits of the second operand. This shape is closest to the typically-made Hamming weight assumption.
- Group 5 (`mul`) leaks in the first two bytes of the first and second operand. The coefficients on the first operand are large for just six of the bits while the second operand coefficients are medium-sized across all bits of the first two bytes.

The purpose of clustering the instructions is to allow us to restrict further analysis to a reduced set of representative instructions, one from each group. We choose `eor`, `lsl`, `str`, `ldr` and `mul`. The model coefficients for each of these are shown in the first panels of Fig. 8 (see Appendix A). As we would hope, they can be seen (by comparing with Fig. 4) to match well the mean coefficients for the groups that they represent, with the possible exception of `str`, which has smaller coefficients on the first byte than the average within its group.

We take a separate look at the leakage model for `ror`, as its silhouette index identifies it as a weak match within its group. Compare the bottom right panel of Fig. 8 (Appendix A) with the top middle panels in Figs. 4 and 8: the leakage on the third byte of the first operand appears reduced, while there is slightly more leakage in the first byte of the second operand. Overall, though, the location of the leakage seems reasonably similar; it certainly bears more in common

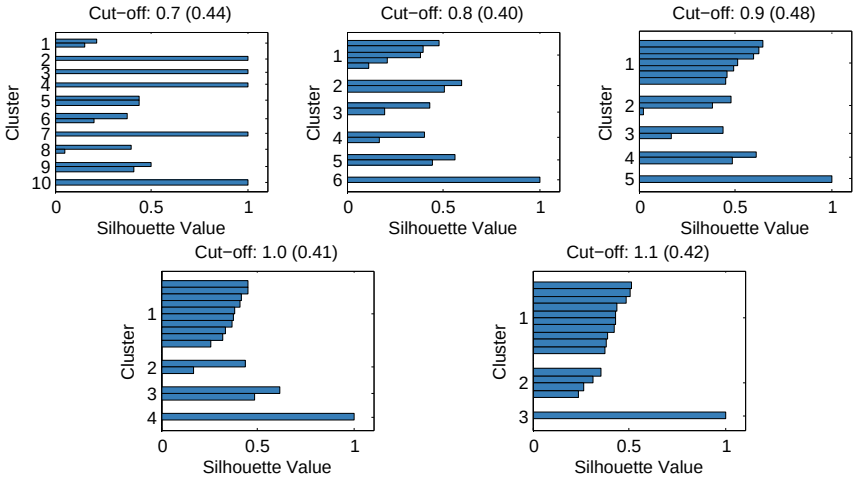

Figure 3: Silhouette plots for each cluster arrangement (numbers in parentheses report median silhouette indices).

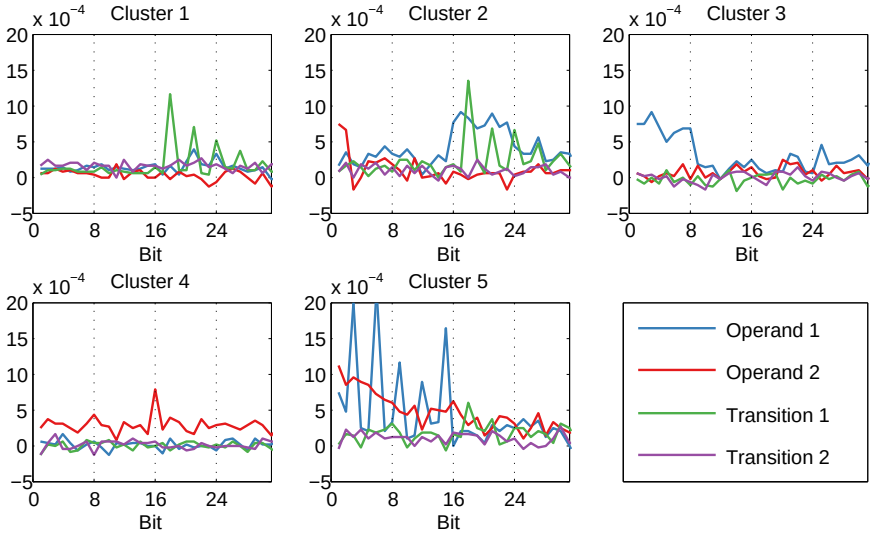

Figure 4: Average estimated coefficients on the model terms for each 'found' cluster.

with the group to which it has been assigned than it does with any of the other groups.

We are confident that our reduced set of five representative instructions is adequate for understanding the leakage behaviour of all 15. Restricting further analysis to these five enables exhaustive exploration of the effects of preceding and subsequent operations when instructions are performed in sequence.[7]

5.2 Building Sequence-Dependent Models

To explore the effect of previous and subsequent instructions we acquired 1,000 traces for each of the possible 125 combinations of three out of the five instructions, with random data inputs.

[7]Such an approach implicitly makes the further assumption that instructions within each identified cluster are affected similarly by the sequence of which they are a part.

We alternated the sequences within a single acquisition to minimise the possibility of conflating instruction sequence effects with drift or acquisition effects, and mean-centered them to adjust for any overall drift. We compressed the traces to a single point (the maximum power peak) in each clock cycle, and selected the clock cycle most strongly associated with the data inputs to the target (middle) instruction. For the `ldr` instruction (which is two cycles long) the relevant point was one cycle ahead of that of the `mul`, `lsl` and `eor`; for `str`, the relevant point was three clock cycles ahead, implying that the data leaked during the subsequent instruction.

Using these relevant points, we then built models for each target instruction in function of its operands, as in model (1), with the addition of dummy variables for previous and subsequent instructions. We further allow for the data-dependent component to vary via four sets of interaction terms: the product of the instruction dummies with the Hamming weights of each operand and also with the corresponding Hamming distances (the sum of bit-flips). This enables a degree of flexibility in estimating the form of the data dependency whilst avoiding the introduction of an infeasible number of instruction/data bit interaction terms into the model equation.

For ease of presentation consider the following groups of variables which together comprise the full set of explanatory variables:

- $\mathbf{I_p}$: The previous instruction in the sequence, fitted as a dummy variable (with `eor` as baseline to preserve linear independence).
- $\mathbf{I_s}$: The subsequent instruction in the sequence, fitted similarly to I_p.
- $\mathbf{D} = [\,\mathbf{O_1} \,|\, \mathbf{O_2} \,|\, \mathbf{T_1} \,|\, \mathbf{T_2}\,]$: All 128 operand bit and transition dummies.
- $\mathbf{DxI_p} = [\,\mathbf{O_1xI_p} \,|\, \mathbf{O_2xI_p} \,|\, \mathbf{T_1xI_p} \,|\, \mathbf{T_2xI_p}\,]$: The Hamming weights of the two 32-bit operands and their Hamming distances from the previous two inputs, interacted with the 'previous instruction' dummies (i.e. the products of the four summarised data terms with each of the four instruction dummies).
- $\mathbf{DxI_s} = [\,\mathbf{O_1xI_s} \,|\, \mathbf{O_2xI_s} \,|\, \mathbf{T_1xI_s} \,|\, \mathbf{T_2xI_s}\,]$: The Hamming weights of the two 32-bit operands and their Hamming distances from the previous two inputs, interacted with the 'subsequent instruction' dummies, as above.

The extended model, in our matrix notation, is therefore:

$$\mathbf{y} = \delta + [\,\mathbf{I_p} \,|\, \mathbf{I_s} \,|\, \mathbf{D} \,|\, \mathbf{DxI_p} \,|\, \mathbf{DxI_s}\,]\, \boldsymbol{\beta} + \boldsymbol{\varepsilon} \tag{3}$$

The first row of Tab. 3 reports the R-squared values of model (3) fitted for each instruction. These are very high because of the large amount of variation introduced into the data by the varying sequences, and subsequently explained once we control for this. From rows 2 and 3 we gather that the previous instruction dummies account for about twice as much of the variation as the subsequent instruction dummies. The proportion explained by the data alone (see row 4) is reduced to around a tenth of what it was in the constant sequence acquisition (compare Tab. 1).

For the purposes of building comprehensive instruction-level models we are especially interested in confirming (or otherwise) the presence of sequence-varying data-dependency, which we again achieve by performing F-tests for the contribution of the interaction terms. The second part of Tab. 3 shows that the full set of interaction terms are jointly significant (at the 5% level) in all cases, as are the previous and subsequent instruction interactions considered separately. We also divide the interaction terms into four groups according to the operand or transition with which they are each associated, in order to test whether the varying data-dependency arises from all or just a subset (in which case we could reduce the complexity of the model). Only for the `str` model do we fail to find evidence of significant effects for all four, suggesting (in that case) the possibility of removing operand 1 and transition 2 terms without cost to the model.

We thus conclude that the form of the data-dependent leakage depends significantly on the previous and subsequent instructions within a sequence, and recommend that they be taken into account (as we have done here) when seeking to build comprehensive instruction level models.

		eor	lsl	str	ldr	mul
R^2	Extended model	0.935	0.902	0.779	0.953	0.874
	$\mathbf{I_p}$ only model	0.550	0.579	0.572	0.629	0.524
	$\mathbf{I_s}$ only model	0.372	0.294	0.194	0.316	0.292
	$\mathbf{D}$ only model	0.013	0.031	0.015	0.012	0.057
F	$\mathbf{DxI_p}$, $\mathbf{DxI_s}$ (32)	17.7	21.0	5.6	4.5	24.1
	$\mathbf{DxI_p}$ (16)	21.1	26.8	4.2	3.7	31.7
	$\mathbf{DxI_s}$ (16)	14.1	15.0	7.1	5.3	16.7
	$\mathbf{O_1xI_p}$, $\mathbf{O_1xI_s}$ (8)	9.1	5.5	0.7	2.5	4.1
	$\mathbf{O_2xI_p}$, $\mathbf{O_2xI_s}$ (8)	33.8	26.4	4.7	5.0	12.0
	$\mathbf{T_1xI_p}$, $\mathbf{T_1xI_s}$ (8)	43.6	25.2	11.3	2.6	15.6
	$\mathbf{T_2xI_p}$, $\mathbf{T_2xI_s}$ (8)	8.8	5.3	0.4	2.0	23.3

Table 3: R-squareds for subsets of the model, and F-statistics for the marginal contributions of the interaction terms. df1 is shown in parenthesis; df2 is 24,831 in all cases. Tests which fail to reject at the 5% level are shaded grey.

5.3 Exploring Higher-Order Effects

An obvious limitation of model (3) is that it restricts the relationship between the bits/transitions and the leakage to be linear. In practice, it is reasonable to suppose (for example) that bits carried on adjacent wires may produce some sort of interaction. Previous analyses fitting linear regression models to target values [11, 21] have allowed for these and for other higher-order interactions, increasing the possibility of accounting for even more exploitable variation in the leakage (if present). We test for the inclusion of adjacent bit interactions in model (3), and find (see row one of Table 4) that they are significant for instructions `lsl` and `mul` only. Adjacent bit flip interactions do not contribute significantly in addition to the adjacent bit interaction effect for any of the instructions. However, non-adjacent pairs *do* contribute relative to the model with adjacent bit interactions in both cases where the latter are significant. We therefore elect to simulate traces based on model 3 in the case of `eor`, `str` and `ldr`, and based on model 3 with the addition of input bit interactions in the case of `lsl` and `mul`.

Tested interactions	eor	lsl	str	ldr	mul
Adjacent bits	1.007	3.890	1.083	0.908	13.330
Adjacent bit flips	1.121	0.832	0.950	0.981	1.308
Non-adjacent bits	1.077	1.286	0.936	0.969	1.384

Table 4: F-tests for significant pairwise bit interaction effects (adjacent and non-adjacent); tests which fail to reject at the 5% level are shaded grey. Degrees of freedom are (62,24769), (62,24707) and (930,23839) respectively.

6 ELMO: An Accurate Power Simulator for the M0

Using the model coefficients obtained via our profiling strategy, we can now implement our method for a practical simulation of the power consumption of an ARM Cortex-M0 running arbitrary assembly code. These power traces are then suitable for conducting attacks, and crucially, they are suitable to detect vulnerabilities, which can be related back to specific assembly instructions. We now explain how our novel tool ELMO achieves this.

6.1 Generating ELMO traces

As it follows on from Section 3, we require code that has been compiled down to assembly level for our instruction-level models. This can be easily achieved using the ARM toolchain. The instruction-level models further require knowledge of the inputs to instructions, i.e. we need to extract the data flow model for a given piece of code, which can be achieved via emulation of that code. There are a number of instruction-level emulators available for the ARM thumb and thumb2 instruction set due to the popularity of these processors.

Well-known examples include QEMU [8] and the ARMulator [9]. We opt for the slightly less widely-used Thumbulator: written in C [10], we choose it for its simplicity and ease of adaptivity. One disadvantage of this choice is that it is inevitably less well-tested than its more popular rivals; it also omits the handful of Thumb-2 instructions which are available in the ARMv6-M instruction set, although we did not attempt to profile any of these and their use would be unlikely in cryptographic operations. Of course, any of the other emulators could be equally incorporated within our methodology.

The Thumbulator takes as input a binary program of Thumb assembly, and decodes and executes each instruction sequentially, using a number of inbuilt functions to handle loads and stores to memory and reads and writes to registers. It provides the capability to trace the instruction and memory flow of a program for the purpose of debugging. Our data flow adaptation is built around a linked list data structure: in addition to the instruction type, the values of the two operands and the associated bit-flips from the preceding operands are stored in 32-element binary arrays.

The operand values, and associated bit-flips from the preceding operations, are then used as input to the model equations (as derived in Section 5.2, see Equation (1)), one for each profiled instruction group. Summarising, simulating the power consumption requires deriving from the data flow information the variables corresponding to the terms in the equations: the previous and subsequent instructions, the bits and the bit-flips of each operand, the Hamming weight and Hamming distances, and the adjacent bit interactions where relevant (i.e. for `lsl` and `mul`). The variables are then weighted by the appropriate coefficient vector and summed to give a leakage value, which is written to a trace file and saved.

We enhance the functionality of ELMO via the addition of a virtual UART port, which enables data to be read into and written from the program, and a trigger signal to be started and stopped. A developer can thus model the power profile of a selected portion of a program only. Data is printed from the emulator by storing `str` to address 0xE0000000. The trigger is started or stopped by toggling address 0xE0000004 from 0 to 1 or 1 to 0 respectively. Data to be read must be stored in hex format delimited by new lines; the file path is set in ELMO and the `ldr` instruction used with address 0xE1000000 to load up to 32 bits. If random numbers are required for testing purposes, these can be cheaply obtained using the `ldr` instruction with address 0xE1000008. This loads one byte of random data, which is currently generated using the random number generator in C seeded with the time, into the register stated by the instruction. The random data is printed to a file in case the developer needs this information later. Of course this is not intended to replace any sound implementation of a cryptographic random number generator, but only aids simple testing of side channel countermeasures.

To facilitate interaction with the emulator we developed a number of assembly functions to carry out these tasks: readdata(unsigned chr *address), which writes a data to address; randdata(unsigned chr *address), which writes a random byte of data to address; printhex(unsigned chr *address), which prints the byte contained in address address to a file in hex format; starttrigger() and endtrigger(), which start and end the trigger. Other UART ports and functions can easily be added if further functionality is required by the user.

To assist with diagnostics, ELMO outputs a full list of the instructions (along with their memory location and machine code representation) in the order in which they are executed. Any

[8] Source code at: `http://wiki.qemu.org/`

[9] Source code at: `https://sourceforge.net/projects/armulator/`

[10] Source code at: `https://github.com/dwelch67/thumbulator.git/`

activity identified as vulnerable (e.g. by leakage detection tests) can thus be easily tracked back to the original assembly source code. ELMO also outputs the index of any non-profiled instructions encountered in the code.

Finally, we tested the correctness of ELMO's implementation of the instruction-level models by comparison to the Model equations being evaluated in Matlab against a set of test vectors.

6.2 Testing Performance and Utility of ELMO

We provided sound statistical evidence about the quality of our modelling approach before. However, since the primary motivation for our instruction-level power consumption simulation is the early identification of potential side-channel vulnerabilities, we further test the utility of ELMO by running leakage detection tests as proposed by Goodwill et al. [8]. These are based on classical statistical hypothesis tests, and can be categorised as *specific* or *non-specific*. Specific tests divide the traces into two subsets based on some known intermediate value such as an output bit of an S-box or the equality (or otherwise) of a round output byte to a particular value. The non-specific 'fixed-versus-random' test acquires traces associated with a particular fixed data input and compares them with traces associated with random inputs. In all cases the Welch's two-sample t-test for equality of means is then performed: for each point in the trace, the t-statistic

$$T = \frac{\bar{X}_A - \bar{X}_B}{\sqrt{\frac{s_A^2}{n_A} + \frac{s_B^2}{n_B}}} \tag{4}$$

is computed (where A and B are the two groups induced by the partition rule, $\bar{X}$ is the mean and $s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X})^2$ the unbiased estimate of the variance). Values of T exceeding a certain threshold, typically recommended to be 4.5[11], are flagged as evidence of exploitable leakage.

Using leakage detection tests (instead of running concrete attacks) is preferable for two reasons. First, in their generic form they require no understanding of the current state-of-the art of side channel attacks on specific algorithms. This makes them attractive to developers as they cannot afford to plough through the large volume of published attacks. Secondly, once a generic test spots a potential leak, it can be already traced back to a specific instruction sequence, which can be analysed further if necessary, for instance with a specific test. A cautionary note though is that the data acquisition for the fixed-versus-random tests can be challenging in practice, because the acquisition of fixed and random input traces has to be strictly interleaved to avoid any systematic bias in the data collection.

6.2.1 Analysis of an AES implementation

As a first example we utilise an AES implementation of mbed TLS [12]. This is an unprotected implementation of AES in C, and we expect it to leak everywhere. This was confirmed when we ran a generic detection test. To demonstrate the utility of ELMO we want to show here that the results of specific leakage detection tests on ELMO traces mirror the results that we achieved on real measured traces.

Figure 5 compares the results of the specific test for S-box leakage in round 4 of an mbed implementation of AES running on ELMO and on an ARM Cortex-M0. The simulated and measured traces are partitioned on each of the 128 bits of the SubBytes output in turn. The detected leakages correspond well overall (note that the *direction* of the peaks is irrelevant), although in some places the leakage from the measured traces can be seen to linger over more clock cycles. This is a result of the fact that ELMO uses instruction-level models rather than clock-cycle level models—so the leakage arising from a particular instruction is tied to the cycle in which it is performed. Whilst this degrades the *visual* similarity of our leakage traces, it has the big advantage that we can easily track back to the 'offending' instruction.

[11] A critical value of 4.5 corresponds to a $\approx 0.001\%$ probability of a false positive, i.e. concluding that there is a leak where no leak exists

[12] Source code at: `https://tls.mbed.org/`

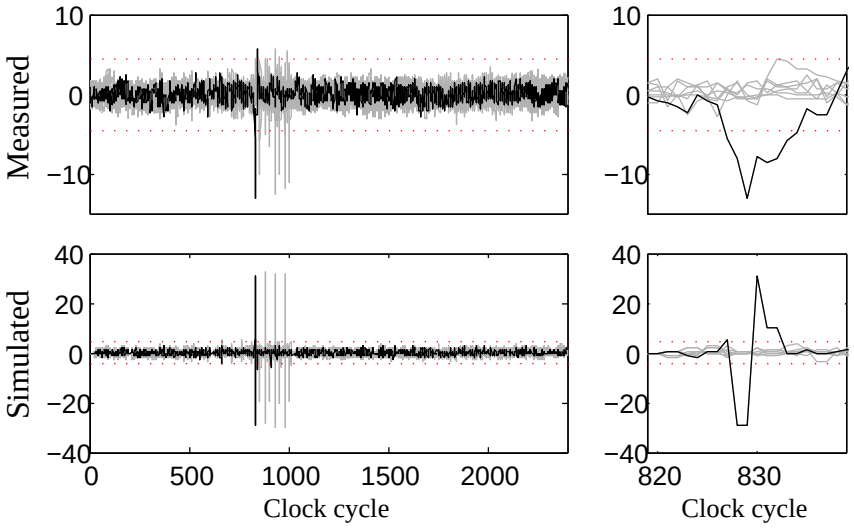

Figure 5: Round 4 S-box t-tests against the (simulated and real) power consumption of the mbed implementation of AES.

Table 5 shows the corresponding source code sequence to Fig. 5. Each instruction is associated with one (or two) clock cycles. Lines printed in blue were found to leak, but one of the instructions in the associated triplet was not profiled. Lines printed in red were found to leak and resulted from a fully profiled instruction sequence.

Cycle No.	**Address**	**Machine Code**	**Assembly Code**
825	0x08000A86	0x4BBA	ldr r3,[PC+#0x2E8]
826	0x08000A86	0x4BBA	ldr r3,[PC+#0x2E8]
827	0x08000A88	0x0089	lsl r1,r1,#0x2
828	0x08000A8A	0x58CB	ldr r3,[r1,r3]
829	0x08000A8A	0x58CB	ldr r3,[r1,r3]
830	0x08000A8C	0x405A	eor r2,r3
831	0x08000A8E	0x69BB	ldr r3,[r7,#0x18]
832	0x08000A8E	0x69BB	ldr r3,[r7,#0x18]
833	0x08000A90	0x0A1B	lsr r3,r3,#0x8
834	0x08000A92	0x21FF	mov r1,#0xFF
835	0x08000A94	0x4019	and r1,r3

Table 5: Section of Thumb assembly code for mbed TLS implementation of AES showing leakage of first bit of the SBox output as indicated by ELMO.

6.2.2 Finding subtle masking flaws

Our first example showed that ELMO produces traces for arbitrary code sequences that are adequate to test for side channel leaks, in the sense that detected leaks in ELMO traces compare well with leaks in real traces for the same source code. We now want to go one step further and analyse how well ELMO traces capture features which are meant to be 'hidden' by implementations of standard countermeasures, such as masking [2]. Masking essentially shares out all intermediate variables into shares which are statistically independent, but whose composition (typically this is an exclusive-or) results in the (unmasked) variables. Consequently, standard DPA attacks [13] cannot succeed. The ease of implementation in software and ability to provide some sort of proof

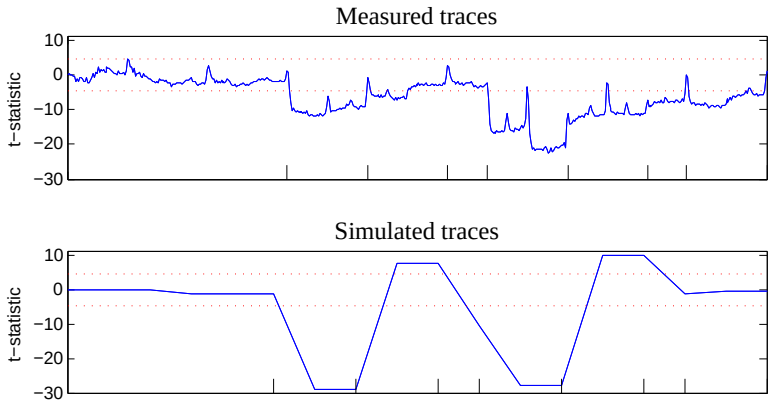

Figure 6: Fixed vs random t-tests against the (simulated and real) power consumption of masked ShiftRows.

of leakage resilience has made masking a popular side channel attack countermeasure that has received a lot of attention from academia and industry alike. It is also well known that implementations of masking can lead to subtle flaws [12].

We implemented a simple masking scheme (as described in [12]) in Thumb assembly to avoid the compiler (from C to Assembly) introducing potential masking flaws. In AES, an operation called ShiftRows takes place as part of the round function. In a masked implementation, this results in a masked row (i.e. which would typically be stored within a register) being rotated and then written back into memory. Table 6 shows the Assembly code for ShiftRows (the colour coding in the table reflects if or not an instruction is found to be leaking in the same way as in the previous example).

We used ELMO to create power traces for this implementation, and we ran a leakage detection test on them. Because we would expect that in a masked implementation there would be no specific, simple leaks to detect, we configured a 'fixed-versus-random' test to check for arbitrary leaks. The results of this test are given in Fig. 6. Clearly, the leakage detections on both simulated and real traces show that leaks are present in several instructions. On first glance this might be surprising: after all, there is no apparent flaw in the masking scheme or its implementation. However, the *combinations* of instructions (i.e. their sequence) lead to leakages, most likely because masked data items are moved consecutively across the same data bus. The leakage detection results on ELMO's traces identify nearly all leaks, the only difference is in the last instruction, which leaks on the measured traces but not on ELMO traces. This is because ELMO simulates the last instruction in a triplet in which one instruction has not been characterised (it relates to the trigger signal that can be set). This is a shortcoming that we would like to address in future work on ELMO.

7 Conclusion

We combine the techniques of linear regression-based leakage profiling with an established emulator for the Thumb instruction set to build a leakage simulator for the ARM Cortex-M0. Our models, which make use of statistical hypothesis testing to incorporate meaningful information whilst avoiding over-complexity, allow for interactions between neighbouring instructions in a sequence and between adjacent bits in operands. Our resulting simulator, ELMO, is able to produce leakage traces for arbitrary pieces of code of high quality. We demonstrate ELMO's utility by performing leakage detection tests for AES implementations on ELMO and real traces.

Cycle No.	Address	Machine Code	Assembly Code
1	0x08000206	0x684C	ldr r4,[r1,#0x4]
2	0x08000206	0x684C	ldr r4,[r1,#0x4]
3	0x08000208	0x41EC	ror r4,r5
4	0x0800020A	0x604C	str r4,[r1,#0x4]
5	0x0800020A	0x604C	str r4,[r1,#0x4]
6	0x0800020C	0x688C	ldr r4,[r1,#0x8]
7	0x0800020C	0x688C	ldr r4,[r1,#0x8]
8	0x0800020E	0x41F4	ror r4,r6
9	0x08000210	0x608C	str r4,[r1,#0x8]
10	0x08000210	0x608C	str r4,[r1,#0x8]
11	0x08000212	0x68CC	ldr r4,[r1,#0xC
12	0x08000212	0x68CC	ldr r4,[r1,#0xC]
13	0x08000214	0x41FC	ror r4,r7
14	0x08000216	0x60CC	str r4,[r1,#0xC]
15	0x08000216	0x60CC	str r4,[r1,#0xC]

Table 6: Thumb assembly implementation of ShiftRows showing leaky instructions as indicated by ELMO.

References

[1] ARM. *Cortex-M0 Devices: Generic User Guide*, 10 2009. Issue A.

[2] J. Blömer, J. Guajardo, and V. Krummel. Provably secure masking of AES. *SAC '04*, vol. 3357 of *LNCS*, pp. 69–83. Springer, 2004.

[3] O. Choudary and M. Kuhn. Efficient Stochastic Methods: Profiled Attacks Beyond 8 Bits. In *CARDIS 2014*, vol. 8968 of *LNCS*, pp. 85–103. Springer, 2014.

[4] O. Choudary and M. Kuhn. Template Attacks on Different Devices. In *COSADE 2014*, vol. 8622 of *LNCS*, pp. 179–198. Springer Berlin Heidelberg, 2014.

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

[6] J. den Hartog, J. Verschuren, E. P. de Vink, J. de Vos, and W. Wiersma. PINPAS: A tool for power analysis of smartcards. In *SEC2003*, vol. 250 of *IFIP Conference Proceedings*, pp. 453–457. Kluwer, 2003.

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

[8] G. Goodwill, J. J. B. Jun, and P. Rohatgi. A testing methodology for side channel resistance validation. NIST non-invasive attack testing workshop, 2008.

[9] N. Hanley, M. O'Neill, M. Tunstall, and W. P. Marnane. Empirical evaluation of multi-device profiling side-channel attacks. In *Workshop on Signal Processing Systems (SiPS) 2014*, pp. 226–231. IEEE, 2014.

[10] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction (Second Edition)*. Springer-Verlag, New York, 2009.

[11] A. Heuser, W. Schindler, and M. Stöttinger. Revealing side-channel issues of complex circuits by enhanced leakage models. In *DATE 2012*, pp. 1179–1184, 2012.

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

[13] S. Mangard, E. Oswald, and F.-X. Standaert. One for All – All for One: Unifying Standard DPA Attacks. *IET Information Security*, 5(2):100–110, 2011.

[14] D. P. Montminy, R. O. Baldwin, M. A. Temple, and E. D. Laspe. Improving cross-device attacks using zero-mean unit-variance normalization. *J. Cryptographic Engineering*, 3(2):99–110, 2013.

[15] P. J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *J. Computational and Applied Mathematics*, 20:53–65, 1987.

[16] W. Schindler, K. Lemke, and C. Paar. A Stochastic Model for Differential Side Channel Cryptanalysis. In J. Rao and B. Sunar, editors, *CHES 2005*, vol. 3659 of *LNCS*, pp. 30–46. Springer Berlin / Heidelberg, 2005.

[17] ST Microelectronics. *Reference manual: STM32F0x1/STM32F0x2/STM32F0x8 advanced ARM-based 32-bit MCUs*, 7 2015. Rev 8.

[18] C. Thuillet, P. Andouard, and O. Ly. A smart card power analysis simulator. In *Proceedings of CSE 2009*, pp. 847–852. IEEE Computer Society, 2009.

[19] V. Tiwari, S. Malik, A. Wolfe, and M. T.-C. Lee. Instruction level power analysis and optimization of software. *VLSI Signal Processing*, 13(2-3):223–238, 1996.

[20] N. Veshchikov. SILK: High level of abstraction leakage simulator for side channel analysis. In *Proceedings PPREW@ACSAC 2014*, pp. 3:1–3:11. ACM, 2014.

[21] C. Whitnall and E. Oswald. Profiling DPA: Efficacy and Efficiency Trade-Offs. In *CHES 2013*, vol. 8086, pp. 37–54. Springer, 2013.

A Supplementary Tables and Figures

		adds	**ands**	**cmp**	**eors**	**ldr**	**ldrb**
	R-squared	0.501	0.168	0.157	0.161	0.275	0.578
F-stat	Interactions	0.76	0.77	0.70	0.74	0.75	0.75
	Data effects	34.48	32.38	31.02	30.72	52.98	11.79
	Board effects	4972.62	455.46	398.84	423.78	1076.19	7483.81

		lsls	**lsrs**	**mov**	**muls**	**orrs**	**rors**
	R-squared	0.415	0.383	0.612	0.444	0.363	0.276
F-stat	Interactions	0.83	0.73	0.70	0.76	0.67	0.80
	Data effects	60.50	56.62	29.91	163.08	27.73	56.25
	Board effects	2821.12	2399.77	8345.27	1410.81	2645.13	1032.41

		str	**strb**	**subs**	**df1**	**df2**	**Critical value**
	R-squared	0.678	0.557	0.190			
F-stat	Interactions	0.85	0.72	0.81	28	39960	1.08
	Data effects	30.78	64.04	36.05	4	39988	1.21
	Board effects	11300.45	5897.54	567.15	7	39988	2.01

Table 7: F-statistics for board and data effects as board varies (tests which fail to reject at the 5% level are shaded grey). Interactions are tested with respect to the 'full' model ('Data + Board + Data×Board'); data and board level effects are tested with respect to the reduced model 'Data + Board'.

Interaction effect	F-stat	Degrees of freedom	Crit. value
All registers	1.207	(896, 39025)	1.080
Source registers	1.357	(448, 39025)	1.113
Destination registers	1.034	(448, 39025)	1.113
Source register = 0	1.398	(64, 39409)	1.308
Source register = 1	1.689	(64, 39409)	1.308
Source register = 2	1.300	(64, 39409)	1.308
Source register = 3	1.151	(64, 39409)	1.308
Source register = 4	1.496	(64, 39409)	1.308
Source register = 5	1.025	(64, 39409)	1.308
Source register = 6	1.838	(64, 39409)	1.308
Source register = 7	1.098	(64, 39409)	1.308

Table 8: F-statistics for register interaction effects (tests which fail to reject at the 5% level are shaded grey).

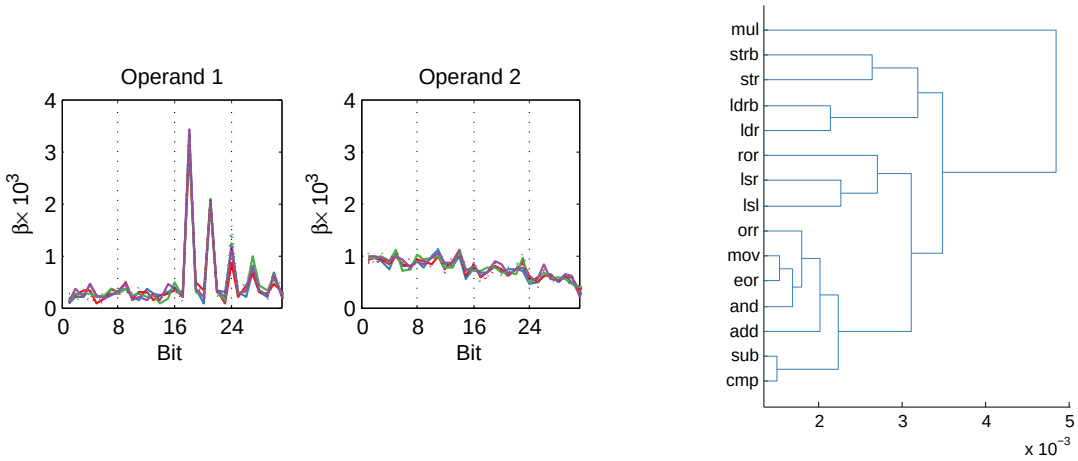

Figure 7: Left: Estimated coefficients on the data terms as source register varies. Right: Dendrogram representing the hierarchical clustering of the instructions according to their fitted leakage models.

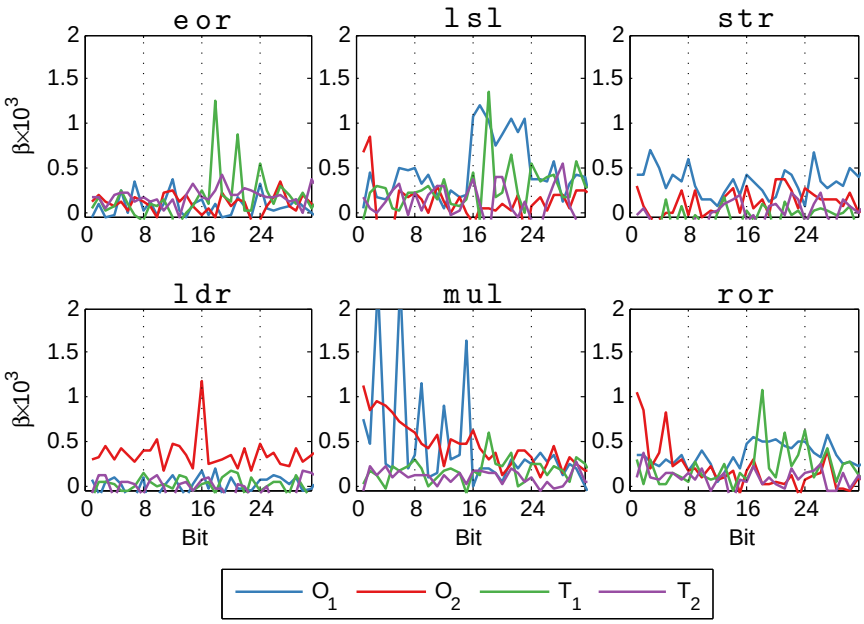

Figure 8: Estimated coefficients on the model terms for each chosen representative instruction, as well as `ror`.