usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

Towards Practical Tools for Side Channel Aware Software Engineering: 'Grey Box' Modelling for Instruction Leakages

David McCann, Elisabeth Oswald, and Carolyn Whitnall, *University of Bristol*

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/mccann

This paper is included in the Proceedings of the
26th USENIX Security Symposium

August 16–18, 2017 • Vancouver, BC, Canada

ISBN 978-1-931971-40-9

Towards Practical Tools for Side Channel Aware Software Engineering: ‘Grey Box’ Modelling for Instruction Leakages

David McCann
University of Bristol

Elisabeth Oswald
University of Bristol

Carolyn Whitnall
University of Bristol

Abstract

Power (along with EM, cache and timing) leaks are of considerable 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 exist some compiler tools to detect timing leaks, similar progress towards pinpointing power and EM leaks has been hampered by limits on the amount of information available about the physical components from which such leaks originate.

We suggest a novel modelling technique capable of producing high-quality instruction-level power (and/or EM) models without requiring a detailed hardware description of a processor nor information about the used process technology (access to both of which is typically restricted). We show that our methodology is effective at 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. We confirm its versatility by demonstrating the basic technique on two processors (the ARM Cortex-M0 and M4), and use the M0 models to develop ELMO, the first leakage simulator for the ARM Cortex M0.

1 Introduction

Early evaluation of the leakage properties of security-critical code is an essential step in the design of secure technology. A developer in possession of a good quality explanatory model for (e.g.) the power consumption or electromagnetic radiation of a particular device can use this to predict the leakage traces arising from a particular code sequence and so identify (and address) possible points of weakness. Whilst the smart-card community is accustomed to support from side-channel testing facilities (either in-house or via external evaluation labs), there is a distinct lack of equivalent dedicated expertise in the fast-growing realm of the Internet-of-Things (IoT). This new market is rife with small start-ups whose limited budgets and rapid pace of advancement are incompatible with the prices and typical workflow of independent evaluators. Thus there arises a pressing need for user-friendly tools which easily integrate with established software development practice—typically in C and/or assembly, depending on the performance requirements of a given application—to assist in making that practice more security-aware.

Whilst there are tools to identify timing leaks such as `ctgrind` [16], there are no such easy tools for detecting power or EM leaks in programs. The reason for this is easily explained: timing information for instructions is readily available, however, accurate models for the instantaneous power consumption or EM emanations are not available.

The challenge of *acquiring* a good quality power model is in **choosing what to include** far more than it is in choosing between particular statistical techniques. The power consumption of a device bears a complex relationship with its various components and processes, necessitating a trade-off between precisely capturing as many details as possible and keeping within reasonable computational and sampling bounds. Transistor (respectively cell) netlists can be used to derive accurate gate-level models [17, Ch. 3], but for the purposes of development-stage side-channel testing in software, we require assembly level models.

Earlier efforts focused on assembly instructions to ensure covering vulnerabilities which might be introduced at compilation time, but (over-) simplified the modelling aspect by relying on Hamming weight and distance assumptions [7, 26]. A more recent, higher-level proposal based on C++ code representation also uses simplified leakage assumptions, although the author does acknowledge the potential for more sophisticated profiling [28]. Among the existing works only Debande et al. [6] emphasise the importance of (and the complexities involved in) deriving realistic leakage models empirically. They fit linear models in function of the state bits and state transitions using the techniques of linear regression.

However, such models are still considerably simplified relative to what is known about the complex factors driving device power consumption. For instance, much earlier efforts to model *total energy consumption* for the purposes of optimising code for constrained devices [27] showed clearly that the power consumed by a particular instruction varies according to the instructions previous in the sequence.

Another important aspect of power model construction, as emphasised by recent contributions in the template building literature [5, 11, 20], is **portability** between different devices of the same design.

1.1 Our Contributions

We present a strategy for building refined assembly code instruction-level power trace simulators and show that it is applicable to two processors relevant to the IoT context: the ARM Cortex M0 and M4. We develop the tool fully for the M0 and verify its utility for side-channel early evaluation.

The first part of our contribution is a side-channel modelling procedure novel for the thoroughness it attains by incorporating established linear regression model-selection techniques. We combine *a priori* knowledge about the M0 and the M4 with power (respectively EM) leakage samples obtained in carefully designed experiments, and use ordinary least squares (OLS) estimation and joint F-tests to decide between candidate explanatory variables in pursuit of models which account well for the exploitable aspects of the side-channel leakage whilst avoiding redundant complexity. The effects that we explore include instruction operands, bit-flips between consecutive operands, data-dependent interactions with previous and subsequent instructions in a sequence, register interactions, and higher-order operand and transition interactions. We verify portability by testing for board effects, which show no evidence of varying differentially with the processed data. We also show (via clustering analysis) that a set of 21 key M0 instructions can be meaningfully reduced to just five similarly-leaking classes, thereby greatly reducing the complexity of the modelling task. As well as enhancing the accuracy and nuance of our predicted traces relative to previous work, our systematic method of selecting and testing potential explanatory variables provides valuable insights into the leakage features of the ARM Cortex devices examined, which are of independent interest.

The second part of our contribution is a procedure to extract the data flows of arbitrary code sequences which can subsequently be mapped to trace predictions via our carefully refined models. We do this for the M0 by adapting an open-source instruction set simulator, chosen to enable us to eventually release a full open-source version of our own tool. We then demonstrate the utility of the simulator for flagging up even unexpected leaks in cryptographic implementations, by performing leakage detection tests against simulated and real measurements associated with an imperfectly-protected code sequence.

The remainder of the paper proceeds as follows: in Section 2 we review the previous work on leakage modelling, and provide a very brief overview of the key features of the ARM Cortex architecture and the Thumb instruction set, alongside some information about our tailored acquisition procedure. In Section 3 we outline our methodology for leakage characterisation and for testing for significant contributory effects. In Section 4 we explore the data-dependent leakage characteristics of each considered instruction taken individually, and empirically confirm the natural clustering of like instructions. In Section 5 we build complex models for the M0, allowing for the effects of neighbouring instructions and higher-order interactions and testing for the possibility of board and register effects. In Section 6 we explain how to use the models to simulate power traces, analyse them, and draw conclusions about leaking instructions. Closing remarks and open questions follow in Section 7.

2 Background

In this section we aim to provide enough context for our paper to be reasonably self-contained for a reader not familiar with the tasks of leakage modelling and model evaluation (Sect. 2.1), the ARM Cortex-M processor family (Sect. 2.2), assembly code instructions (Sect. 2.3) and/or typical side-channel measurement setups (Sect. 2.4).

2.1 Leakage Modelling Techniques

Modelling power consumption always involves a trade-off between precision and economy (with respect to time, memory usage and input data required). The most detailed (‘white box’) efforts take place at the analog or logic level and aim to characterise the power consumed by every component in (part of) a circuit. For the purposes of side-channel analysis, simpler, targeted (‘black box’) models can be estimated from sampled traces for particular intermediate values. Instruction-level models of the type we propose represent a (‘grey box’) middle ground, combining some relatively detailed knowledge of the implementation with empirical analysis of carefully sampled leakage traces. We briefly overview these three research directions below, followed by a summary of some typical approaches to the difficult task of model quality evaluation.

2.1.1 Model Building Utilising Processor/Implementation Specific Information

Netlists describing all the transistor connections in a circuit, along with their parasitic capacitances, can be used to perform analog simulations of the whole or a part of the circuit. This process involves solving numerous difference equations and is highly resource intensive. A less costly (but also less precise) logic-level alternative uses cell-level netlists, back-annotated with information about signal delays and rise and fall times. These are used to simulate the *transitions* occurring in the circuit, which are subsequently mapped to a power trace according to knowledge of the capacitive loads of the cell outputs. Alternatively, the *number* of transitions occurring can be taken as a simplified approximation of the power consumption, which implicitly amounts to the assumption that all $0 \rightarrow 1$ transitions contribute equally to $1 \rightarrow 0$ transitions (and similarly for $0 \rightarrow 0$ and $1 \rightarrow 1$ transitions). See Chapter 3 of [17] for more details. Note that even these most exhaustive of strategies, which may be collectively classed as 'white box' modelling due to their reliance on comprehensive implementation details, fail to account for influences on the leakage outside the information provided by the netlist (for instance crosstalk) and therefore represent simplifications of varying imperfection.

2.1.2 Model Building for Intermediate Instructions

For the purposes of side-channel analysis and evaluation, it suffices to build models only for power consumption which (potentially) bears a relationship to the processing of security-sensitive data or operations. These strategies bypass the requirement for detailed knowledge of the implementation and may be thought of as 'black box' modelling. A typical approach has been to focus on (searchably small) target intermediate values of interest (for example, the output of an S-box). By measuring large numbers of leakage traces as the output of the target function varies in a known way, it is possible to estimate the parameters of (for example) a multivariate Gaussian distribution associated with each possible value taken by the intermediate. Traces acquired from an equivalent device with an unknown key (and therefore unknown intermediates) can then be compared against these fitted models ('templates') for the purposes of classification [3, 5]. Linear regression techniques can be used to reduce the complexity of the leakage characterisation [23, 29]; the assumption of normality can be avoided, for example by building models using machine learning classification techniques [14].

2.1.3 Model Building for Processor Instructions

In order to simulate leakage of arbitrary code sequences on a given device we opt for ('grey box') instruction-level characterisation. Previous instruction-level (and higher code-level) simulations for the purposes of side-channel analysis have settled for Hamming weight or Hamming distance assumptions [26, 28] or have estimated simple models constrained to be close to such approximations [7, 6]. However, much earlier work by Tiwari et al. [27] explores more complex model configurations for the purposes of simulating and minimising the *total power cost* of software to be run on resource-constrained devices. The authors find that not only do instructions have different costs, but that those costs are influenced by preceding instructions in a circuit. Their models are thus comprised of instruction-specific average base costs additively combined with instruction-pair-specific average circuit state overheads. This methodology is not adequate for our purposes, as it essentially averages over all possible data inputs—precisely the source of variation that most needs to be captured and understood in a side-channel context. Hence, we combine similar instruction and instruction-interaction terms with data-state, -transition and -interaction terms, drawing on modern approaches to linear regression-based profiling [4, 29] to handle the considerable added complexity.

2.1.4 Evaluating Model Quality

To build a model is to attempt to capture the most important features of an underlying reality which is (at least in the cases where such an exercise is useful and interesting) *unknown*. For this reason it is generally not possible to definitively establish the quality of any model (i.e., the extent to which it matches reality). However, there *do* exist methods, depending on the various model-fitting strategies adopted, for indicating whether the output result is suitable for its desired purpose. An approach popular in the side-channel evaluation literature is to estimate the amount of information (in bits) in the true leakage which is successfully captured by an evaluator's model for that leakage with a metric called the *perceived information* (PI) [21, 8]. This retains the usual shortfalling in that the question of how *good* the model is essentially corresponds to the one of how close the PI is to the true (as always, unknown) mutual information. However, in [8] the authors show how to combine cross-correlation and distance sampling to increase confidence (or highlight problems) in models used for evaluation.

Nevertheless, for our purposes, the established tools traditionally associated with linear regression model building are better suited as they allow disentangling the contributions of component parts of the model as well as commenting on overall model quality. The coefficient of determination, or R^2, is a popular goodness-of-fit measure which can be thought of as the proportion of the total variation in the sample which is explained by (i.e. can be predicted by) the model. However, the R^2 is notoriously difficult to interpret as it always increases with the number of explanatory variables, hampering attempts to compare models of different sizes. It can be adjusted by penalising for the number of variables, but it is normally recommended to compute the F-statistic (see Sect. 3.1) to test for the statistically significant improvement of one model over another. The F-test can also be used to test *overall* model significance, which is useful in our case where the exploitable (i.e. data dependent) variation may only represent a small fraction of the total variation in the traces (which includes noise and unrelated processes). That is, a low R^2 need not imply that a model is unfit for purpose, as long as it represents a statistically significant data-dependent component of the leakage; conversely, a high R^2 need not indicate a model as more fit for purpose if the extra variation explained is irrelevant to the data-dependent side-channel leakage or is a result of over-fit.

However, F-tests offer no reassurance that other important contributory factors have not been *omitted* from the model. In the context of modelling for side-channel detection, it is established practice to verify the *adequacy* of the trace simulations by demonstrating that they reliably reveal the same vulnerabilities as real trace measurements. Previously, this has largely been attempted by performing DPA attacks [7, 26]; for the purposes of rigour, we propose to also utilise the leakage detection framework of [10] (see Sect. 6.3).

2.2 ARM Cortex-M Processor Family

The ARM Cortex-M processor family[19] was first introduced by ARM in 2004 to be used specifically within small microcontrollers, unlike the Cortex-A and Cortex-R families which, although introduced at the same time, are aimed at higher-end applications. Within the family there are six variants of processor: the M0, M0+, M1, M3, M4 and M7, where the M0 provides the lowest cost, size and power device, the M7 the highest performing device, and the M0+, M3 and M4 processors sit in-between. The M1 is much the same as the M0 however it has been designed as a “soft core” to run inside a Field Programmable Gate Array (FPGA). The M0 and M3 share the same architecture as the M0+ and M4 respectively, though the M0+ and M4 have additional features on top of the basic processor architecture to provide them with greater performance. The M7 processor is the most recent (2014) and high performing of the Cortex-M family.

Whilst the exact CPU architecture of the Cortex-M devices is not publicly available, it can be assumed to resemble the basic architecture of ARM cores, as detailed in [9]. 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 into the ALU via two buses, one of which 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.

We select the Cortex-M0 and M4 processors (see Tab. 2 in the Appendix for comparison) to evaluate using our clustering profiling methodology and go on to further analyse and produce a leakage emulator for the Cortex-M0. The reason for the selection of these two processors is that they represent either ends of the spectrum for the older, more widely used, of the Cortex-M family, allowing us to demonstrate that our methodology can be applied to a range of processors.

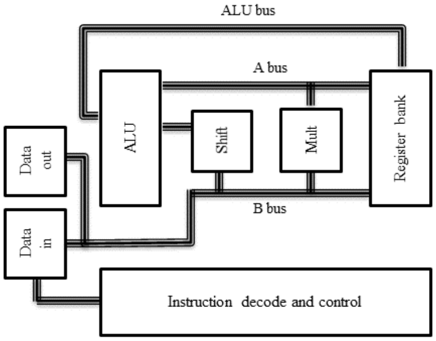

Figure 1: Simplified ARM CPU architecture (redrawn from [9]) for a 3-stage pipeline architecture.

2.3 Instructions

In this work we focus on profiling a select number (21) of Thumb instructions that are highly relevant for implementing symmetric cryptography, which run on both the Cortex-M0 and M4 processors: `ldr`, `ldrb`, `ldrh`, `str`, `strb`, `strh`, `lsls`, `lsrs`, `rors`, `muls`, `eors`, `ands`, `adds`, `adds` *#imm*, `subs`, `subs` *#imm*, `orrs`, `cmp`, `cmp` *#imm*, `movs` and `movs` *#imm*. Note that `adds` #imm and `subs` #imm use 3-bit immediate values rather than 8-bit values. All non-memory instructions use the `s` suffix and so update the conditional flags and, in the case of the Cortex-M0, can only use low registers. The implementation of the `muls` instruction takes a single cycle to execute on both processors. We made this selection to include core instructions with particular use within (symmetric key) cryptographic algorithms, which tend to perform operations on the set of unsigned integers. We also focus on the instructions which contain the `s` suffix to comply with restrictions required for many of the Cortex-M0 instructions and, where there is the option to use the non-suffixed instruction with higher registers (as with the `adds`, `subs` and `movs` instructions), we chose the suffixed version to maintain consistency with the other instructions.

Understanding and interpreting the input format of the instructions is necessary in order to correctly model them and the interactions between them. From Fig. 1 we would expect three buses to be used for the ALU instructions, as well as for shift and multiply instructions where the barrel shifter and hardware multiplier are present: the A bus for operand 1, the B bus for operand 2 and the output of the operation on the ALU bus. In our analysis we do not consider the effects of ALU outputs, as we assume the output of an instruction to be used as an input to a following instruction; we focus on the two operands of the operation which we would expect to leak via the A and B buses. We therefore take these to correspond to operands 1 and 2 respectively. For memory instructions we expect the data being loaded or stored to leak on bus B, as well as the data bus. To include this leakage and any interactions it may have with the previous data value that was on this bus, we set the data to be loaded or stored as the value of operand 2 for all memory instructions. How we model these operands based on the register selection of the instructions is described below.

For the majority of non-memory instructions (i.e. those other than `ldr`, `ldrb`, `ldrh`, `str`, `strb`, `strh`), 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.

To simplify 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 have one main operand for memory instructions which is the data in r_t for store instructions and the value in the memory address of r_n ($data[r_n]$) for load instructions which we set to operand 2 in both instances. For store instructions, we set the data in memory which is to be overwritten ($data[r_n]$) and for load instructions the register into which the data is to be loaded (r_t) to be random data which we model as operand 1 in both cases. This is to include any potential leaks that could come from either of these sources, however we would do expect this to include bit interactions with operand 1 of the previous instructions as we do not expect either of these data values to be transmitted on bus A in Fig. 1.

2.4 Measurement Setups

We work with implementations of the two processors by ST Microelectronics on STM Discovery Boards, with the ARM Cortex-M0 being implemented on an STM32F0 (30R8T6) Discovery Board and the ARM Cotex-M4 on the STM32F4 (07VGT6). These boards both feature an ST-Link to flash programs to the processor and provide on-chip debugging capabilities as well as on-board RC oscillator clock signals (8Mhz and 16Mhz for the STM32F0 and STM32F4 respectively). Further details about the devices can be found in datasheets [24] and [25].

In order to get accurate power measurements for the Cortex-M0, we modified the STM32F0 board by extracting the power pins of the processor, and passing the power supply through a 360Ω resistor over which a differential probe was connected. This was to minimise the potential for board and setup effects. We also verified the stability of our power supply. To measure the EM emissions on the Cortex-M4 processor we placed a small EM probe over the output of one of the capacitors leading to one of the power supply pins of the processor.

We used a Lecroy Waverunner 700 Zi scope at a sampling rate of 500 MS/S for both the power and EM analyses. The sampling rate was selected by observing DPA outcomes on the Cortex-M0 across different sampling rates: 500 MS/S was the lowest sampling rate at which the best DPA outcomes were achieved. The clock speed of the Cortex-M0 was set to 8Mhz and the Cortex-M4 set to 16 Mhz. To lower the independent noise, we averaged over five acquisitions per input for the power measurements for the M0 (as this was found to be the lowest number that brought the most signal gain) and 10 for the

EM measurements for the M4 (to further reduce the additional noise associated with this method of taking traces). No filtering or further signal processing took place for the Cortex-M0 power measurements, however a 48Mhz low-pass filter was used before amplifying the EM signal for the Cortex-M4.

We note that our measurement of EM uses only one probe over one of multiple power inputs to the processor (for the M0 we reduced the number of power inputs to a single one over which to measure) and that, whilst we have applied *some* pre-processing to the (noisier) EM measurements, we could have attempted more thorough techniques to enhance the signal. We view, therefore, our two measurement setups for the different boards to represent different ends of the spectrum in terms of the time and effort invested to get improved measurements. In this way we aim to gain an understanding of how our profiling methodology adapts to different setup scenarios as well as for different processors.

3 A Novel Methodology to Characterise a Modern Microprocessor

In principle all components (i.e. on the lowest level of gates and interconnects) contribute to the side channel leakage in the form of power or EM and so could be modelled as predictor variables. The skill and challenge in model building is then to select and test (and possibly discard) potential predictors in a systematic manner, manoeuvring the trade-off between infeasible complexity and oversimplification. We opt for a 'grey box' approach which does not require detailed hardware descriptions but *does* assume access to assembly code in order to construct models at the instruction level. We concentrate on predictor variables that can be derived from assembly sequences (i.e. input data, register locations), but we also want to potentially account for board-specific effects.

Linear regression model-fitting techniques have been used by the research community for some years already to profile side-channel leakage [23]. We refine the adopted procedures according to well-established statistical hypothesis testing strategies, in order to better understand the true functional form of the leakage and to make informed judgements about candidate explanatory variables. Specifically, we perform F-tests for the joint significance of groups of related variables, and include or exclude them accordingly, thus producing meaningful explanatory models which are not unnecessarily complex.

3.1 Model Building

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 [12]):

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

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 b^{th}-bit, $\mathbf{x_i}$ denotes the i^{th} operand to a given instruction, $\mathbf{z_i}$ denotes the i^{th} 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.[1] 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.

3.2 Selecting Explanatory Variables

The innovations we propose over previous uses of linear regression for modelling side-channel leakage are with respect to informed model selection. The task of selecting a meaningful subset from a large number of candidate explanatory variables is well-recognised as non-trivial. Techniques such as stepwise regression [15] fully automate the procedure by iteratively adding and removing individual terms according to their contribution to the current configuration of the model. This approach is sensitive to the order in which terms are introduced and prone to over-fitting, and has attracted criticism for greatly understating the uncertainty of the finalised models as typically reported. Stepwise regression has been used to achieve so-called 'generic-emulating' DPA [30]; it is effective in this context because attack success does not derive from the actual construction of the produced models but requires only that the proportion of variance accounted for is greater under the correct key hypothesis than under the alternatives. However, we require our models to be *meaningful*, not just (artificially) close-fitting. Thus we adopt a more conservative and traditional approach towards model building by which informed intuition about likely (jointly) contributing factors precedes formal statistical testing for inclusion or exclusion.

The criterion for model inclusion is based on 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

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

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:

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

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)[2], F is larger than the 'critical value' of the $F_{p_B - p_A, n - p_B}$ distribution[3] 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 (in Sect. 5) for register, board and adjacent instruction effects on the operand and bit-flip contributions by computing F-statistics for the associated sets of interaction terms.

4 Identifying Basic Leakage Characteristics

We first investigate the instruction-dependent form of the leakage in a simple setting, where differential effects from other factors do not yet play a role. For this purpose, we perform the same fixed sequence `mov`-`instr`-`mov` 5,000 times for each selected instruction, as the two 32-bit operands vary. We measure the power consumption (or EM, in the case of the M4) associated with each sequence, and identify as a suitable point the maximum peak[4] in the clock cycle during which the instruction leaks. We fit the model (1) to the (drift-adjusted) vector of measurements at this point.

Table 3 in the Appendix confirms the overall significance of the model for each M0 instruction. This supports our point selection and the intuition that the leakage depends in part on the data being operated on. However, some differences can be observed in the contributing factors:

- The load instructions depend only on the bits of the operands (operand 2 or both for `ldrh`) and not the bit flips.
- The store instructions depend only on the bits and the bit transitions of the second operands.
- The operations on immediate values essentially have *no* second operand on which to depend.
- For all the other instructions all tested sets of explanatory variables are judged significant at the 5% level, which the exception of the second operand bit transitions for the `mul`.[5]

4.1 Further observations and indicators for model quality

Although we caution in the background section that over-interpreting the 'raw' value of resulting R-squareds is not advisable, their relative values can provide some evidence about the relative quality between (same-type) models obtained via e.g. different setups and devices.

Hence we now discuss same-type models for the M4, which we obtained using traces from a deliberately weaker measurement setup. Table 4, also in the Appendix, shows the model results for the M4. The model for the `mov` instruction is not found to be significant, implying that there is insufficient evidence to conclude that the EM radiation of `mov` depends on its data operands and bus transitions. The models for other instructions are overall significant, but fewer of the data-dependent terms are identified as contributing.

- Both operands to the ALU instructions contribute, except in the case of those involving immediate values (which, again, essentially have no second operand).
- Only the second operand to the load and store instructions contributes significantly.
- Bus transitions contribute to the instructions on immediate values, and also to `cmp`.

Whilst we again advise against over-interpreting the R-squareds (see Section 2.1), a comparison between the first rows of 3 and 4 indicates that, in the case of the ALU and shift instructions, model (1) accounts for substantially less of the variation in the M4 EM traces than it does of the variation in the M0 power traces. Although this could be taken as evidence that the instructions in question leak more in the case of the M0 and less in

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

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

[4] This choice is specific to our measurements and is by no means the only option.

[5] 'Significant at the 5% level' is a shorthand way of saying that the null hypothesis of 'no effect' is *rejected* by the F-test when the probability of a false rejection (Type I error) is fixed at 5%.

the case of the M4, it is more likely that the M4 model is weaker because of the weaker setup as discussed in Sect. 2.4). We take this as further evidence that statistical measures that we suggest as part of our methodology are suitable to judge model quality.

4.2 Clustering Analysis to Identify Like Instructions

We eventually want to allow for possible differences in the leakage behaviours of instructions depending on adjacent activity in sequences of code (as per [27]). This will be much easier to achieve if we can reduce the number of distinct instructions requiring consideration. For instance, we might expect instructions invoking the same processor components (as visualised in Fig. 1) to leak similarly: ALU instructions as one group (i.e. `adds`, `adds` *#imm*, `ands`, `eors`, `movs`, `movs` *#imm*, `orrs`, `subs`, `subs` *#imm*, `cmp`, `cmp` *#imm*), shifts as another, albeit closely-related group (`lsls`, `lsrs`, `rors`), loads (`ldr`, `ldrb`, `ldrh`) and stores (`str`, `strb`, `strh`) that interact with the data in/out registers as two more groups, and the multiply instruction (`muls`) as a group on its own with a distinct profile due to its single cycle implementation.

We compare this intuitive grouping with that which is empirically suggested by the data by performing clustering analysis (see, e.g., Chapter 14 of [12]) on the per-instruction data term coefficients $\boldsymbol{\beta}$ obtained by fitting model (1) for both the M0 and the M4. We use the average Euclidean distance between instruction models to form a hierarchy of clusters (represented by the dendrograms in Fig. 6). Adjusting the inconsistency threshold[6] between 0.7 and 1.2 produces the groupings reported in Tables 5 and 6. In the case of the M0, these align nicely with our intuitive grouping: at threshold 0.9 the match is exact; at a threshold of 1.1 the shifts join the ALU instructions; at a threshold of 1.2 the instructions form a single cluster. In the case of the M4, the intuition is confirmed to a degree: at threshold 0.8, the ALU instructions are spread out over four groups, and the store operations over two; but the shift operations cluster together, as do the loads, and the `mul` is again identified as distinct. There is no overlap between the nine groups until they form a single cluster at threshold 1.0.

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 i^{th} object as $S_i = \frac{b_i - a_i}{\max(a_i, b_i)}$, where a_i is the average distance from the i^{th} object to the other objects in the same cluster, and b_i is the minimum (over all clusters) average distance from the i^{th} object to the objects in a different cluster [22]. Fig. 2 plots the M0 cluster silhouettes for a selection of the arrangements in Tab. 5. The consistency threshold of 0.9 is associated with the highest median silhouette value (0.56), supporting our *a priori* intuition.

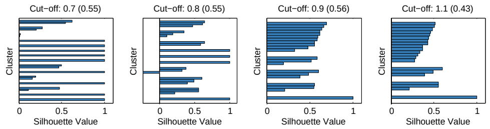

Figure 2: Silhouette plots for each M0 cluster arrangement (numbers in parentheses report median silhouette indices).

4.3 Functional Form of the Leakage

We next look more closely at the form of the estimated leakage models. Fig. 3 plots the mean data-dependent coefficients associated with the different terms in the model equations, for each of the five M0 groups suggested by the clustering analysis with a threshold of 0.9.

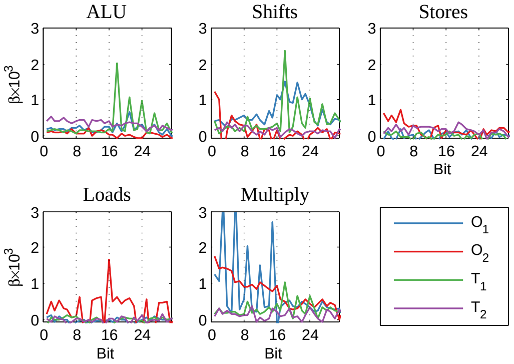

Figure 3: Average estimated coefficients on the model terms for each 'found' M0 instruction cluster

The differences between the groups are immediately clear. We make the following observations for the M0:

- ALU instructions (`adds`, `ands`, `cmps`, `eors`, `movs`, `orrs` and `subs`, and their immediate value equivalents where relevant) 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.
- Shifts (`lsls`, `lsrs`, `rors`) appear to leak in the first operand (which contains the data being shifted)

[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`).

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 three bits of the second operand.

- Stores (`str`, `strb`, `strh`) leak primarily in the first byte of the second operand.
- Loads (`ldr`, `ldrb`, `ldrh`) leak across most bits of the second operand. This shape is closest to the typically-made Hamming weight assumption.
- Multiply (`muls`) leaks mostly 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.

In summary, our exploratory analysis of the data-dependent form of the instruction leakages confirms many of our *a priori* intuitions about the architecture and supports our model building approach as sensible and meaningful. It also indicates that we can lessen the burden of the task by reducing the number of distinct instructions to be modelled to a meaningfully representative subset of the initial 21. Reducing *unnecessary* complexity in the instruction set increases the scope for adding meaningfully explanatory complexity to the models themselves, which we proceed to do in the next section for the power consumption of the M0.

5 Building Complex Models for the M0

From this point forward we concentrate on the M0 and seek to build more complex, sequence-dependent models for five instructions chosen to represent the groups identified by the clustering analysis of Sect. 4.2: `eors`, `lsls`, `str`, `ldr` and `muls`. The model coefficients for each of these are shown in Fig. 7 (see Appendix). As we would hope, they can be observed (by comparing with Fig. 3) 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 are confident that these five are adequate for understanding the leakage behaviour of all 21. Restricting the analysis in this way enables exhaustive exploration of the effects of preceding and subsequent operations when instructions are performed in sequence.[7]

5.1 Exploring Board Effects

To understand if we need to account for variation between boards we replicate the M0 acquisition described at the start of Section 4 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 instructions This also implies that our setup has minimised (or even removed) any measurable impact on the processor's power consumption.

5.2 Exploring 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 `movs` 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, making 625 per pair) as `movs` 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. 7 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 consid-

[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.

erable 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. 7 (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—a legitimate criteria for assessing practical significance according to [2], although we have not carried out the formal visual inspection there proposed. 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 Sect. 5.3.

5.3 Allowing For Sequence Dependency

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.

To achieve this we acquired 1,000 traces for each of the possible 125 combinations of three out of the five instructions, with random data inputs. 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 `muls`, `lsls` and `eors`; 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 `eors` 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:

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

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. Table 8, in the Appendix, 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.

5.4 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 [13, 29] have allowed for these and for other higher-order interactions, increasing the possibility of accounting for even more exploitable variation in the leakage. However, they have failed to investigate if such effects are in fact present.

We therefore test for the inclusion of adjacent and non-adjacent bit interactions in model (3). Table 9, in the Appendix, shows that we find significant effects precisely (and only) where we would expect to: in the leakages of `lsls` and `muls`, instructions which explicitly involve the joint manipulation of bits within the operands. We also test for adjacent bit flip interactions, which are not found to contribute significantly towards any of the instruction leakages. For the purposes of simulation, we therefore elect to use model (3) in the case of `eors`, `str` and `ldr`, and model (3) with the addition of input bit interactions in the case of `lsls` and `muls`.

6 Using and Evaluating our Grey Box Models in a Practical Context

Up until know we have considered short instruction sequences. We have shown that our novel approach produces models which, when evaluated in the context of an instruction triplet, include statistically relevant and architecturally justified terms. Furthermore, our methodology clearly indicates model quality: the models derived from a dedicated setup for monitoring the power consumption showed much better statistics then the models derived from the much less sophisticated EM setup.

However, to make the final argument that our approach results in models that are useful in the context of arbitrary instruction sequences, we need to consider code that is longer and more varied then the triplets that we used for model building. We also need to define a measure that allows us to judge how good the 'match' between model-simulated and real power traces is. We could consider randomly generating arbitrary code sequences (of some predefined length), and defining some distance measure. However, because we have a very clear application for these models in mind, we opt for a more decisive and targeted evaluation strategy. The ultimate test, arguably, is to utilise our models for the M0 to evaluate the security different implementation of a cryptographic algorithm (e.g. AES). To conduct such a test, we build an **E**mulator for power **L**eakages for the **M0** (short: ELMO, elaborated on in the next section). In this context we expect that leakage simulations based our newly constructed models enable to detect leaks that relate to the modelled instructions, but also (maybe more simply) that our models correlate well to measured traces.

6.1 ELMO

As follows from Sect. 3, our instruction-level models work with code that has been compiled down to assembly level, easily obtained via the ARM toolchain. Computing model predictions requires knowledge of the inputs to instructions, which entails emulating a given piece of code in order to extract the data flow. There are a number of instruction-level emulators available for the ARM, Thumb and Thumb2 instruction sets due to the popularity of these processors.

We choose an open source (programmed in C) emulator called Thumbulator[8]. We choose this over more well-known emulators[9] for its simplicity and ease of adaptivity for our purposes. 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 profile any of these. Of course, any of the other emulators could be equally incorporated within our methodology.

The Thumbulator takes as input a binary program in 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 Sect. 5; see Eqn. (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 `lsls` and `muls`). 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.

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

[9] E.g. QEMU `http://wiki.qemu.org/`, Armulator `https://sourceforge.net/projects/armulator/`

6.2 Evaluating Model Correlation to Real Leakages

A simple way to check how well a model corresponds to real leakage behaviour is to compute the (Pearson) correlation between the model predictions for a particular instruction (operating on a set of known inputs) and the measured traces corresponding to a code sequence containing that same instruction (operating on the same inputs). This procedure can be used to demonstrate the improvement of our derived models over weaker, assumed models, such as the Hamming weight.

Figure 4 juxtaposes the correlation traces produced by the Hamming weight prediction of the leakage associated with the first round output as the M0 performs AES (top), and by the ELMO prediction corresponding to the same intermediate being loaded into the register (middle). It can be clearly seen that the ELMO model generates larger peaks, and more of them. The bottom of Figure 4 shows, for comparison, the peaks which are exhibited when the model predictions are correlated with an equivalent set of ELMO-emulated traces. These indicate the same leakage points as displayed in the measured traces, with the advantage of enhanced definition thanks to the lack of (data-independent) noise in the simulations. It thus emerges that Hamming weight-based simulations do not give a full picture of the true leakage of an implementation on an M0, and should not be relied upon for pre-empting data sensitivities. The same picture emerges for the other instructions but we do not include an exhaustive analysis for the sake of brevity. In conclusion, our models represent a marked improvement over simply using the Hamming weight.

6.3 Evaluating Models via Leakage Detection

Further to the capability of our models to improve correlation analysis, we now show that they can also be applied to the task of (automated) leakage detection on assembly implementations. They can thereby be used to spot 'subtle' leaks – that is, leaks that would be difficult for non-specialist software engineers to understand and pinpoint.

To aid readability we briefly overview the leakage detection procedures proposed by Goodwill et al. [10]. 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 against traces associated with random inputs. In all cases the Welch's two-sample t-test for equality of means is then performed; results that are larger than a defined threshold, which we indicate via a dotted line in our figures, are taken as evidence for a leak.

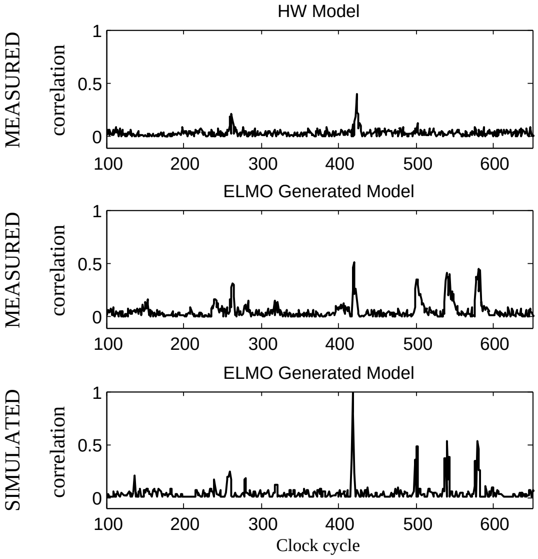

Figure 4: Correlation traces for ELMO-predicted intermediate values (top) and Hamming weight model predictions (middle) in 500 real M0 traces; correlation trace for ELMO-predicted intermediate values in the equivalent set of ELMO-emulated traces (bottom).

6.3.1 Detecting 'Subtle' Leaks

We now choose a code sequence relating to a supposedly protected AES operation. The code sequence implements a standard countermeasure called masking [1]. Masking essentially distributes all intermediate variables into shares which are *statistically independent*, but whose composition (typically by way of exclusive-or) results in the (unmasked) variables. Consequently, standard DPA attacks [18] no longer succeed. The ease of implementation in software and ability to provide some sort of proof of leakage resilience has made masking a popular side channel attack countermeasure, on the receiving end of considerable attention from academia and industry alike. However, it is also well-known that implementations of masking schemes can produce subtle unanticipated leakages [17].

We faithfully implemented a masking scheme for AES (as described in [17]) in Thumb assembly to avoid the potential introduction of masking flaws by the compiler (from C to assembly). The code sequence, which we will analyse and discuss, relates to an operation called

ShiftRows which takes place as part of the AES 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 1 shows the assembly code for ShiftRows. An experienced and side-channel aware implementer who has detailed leakage information about the M0 would now be able to spot a problem with this code: because the `ror` instruction also leaks a function of the Hamming distance to its predecessor, there could be problem if the prior instruction is protected by the same mask. Clearly an inexperienced implementer, or somebody who does not have the necessary profiling information, would not be able to make this inference.

We now show that ELMO traces (for this same code sequence) can be used for the purposes of (pre-emptive) leakage detection. Since we do not expect any specific, simple leaks to be detectable under masking, we configured a 'fixed-versus-random' test to check instead for *arbitrary leaks*. Figure 5 shows that the analysis of our model-simulated traces indicates the presence of leaks in several instructions (see also Tab. 1 where they are colour-coded in red). These leaks are precisely due to the `ror` leakage properties that we discussed in the previous paragraph. The figure shows that all real-measurement leaks can be identified from the simulations, with the exception of some lingering leakage in the cycles after the final `ldr`. We believe this results from the fact that our models are constructed at instruction level rather than clock-cycle level—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 simulations, it has the big advantage that we can easily track back to the 'offending' instruction.

In short, our grey box approach to modelling side-channel leakage proves highly successful at capturing and replicating potentially vulnerable data-dependency in arbitrary sequences of assembly code.

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

Table 1: Thumb assembly implementation of ShiftRows showing (colour-coded in red) leaky instructions as indicated by the model-simulated power consumption.

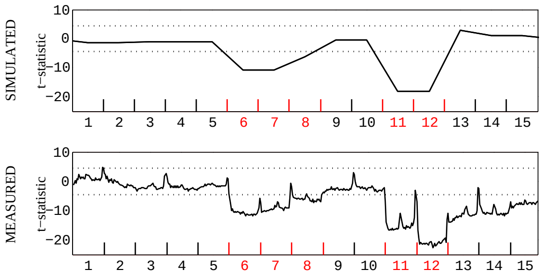

Figure 5: Fixed vs random t-tests against the (simulated and real) power consumption of masked ShiftRows. (Dotted lines indicate the ±4.5 threshold for t-test significance).

7 Conclusion

We have shown how to combine a 'grey box' view of a cryptographic device with well-understood statistical techniques for model construction and evaluation in order to profile and simulate instruction-level side-channel leakage traces. Our methodology enables informed and statistically-testable decisions between candidate predictor variables, as well as empirically-verified clustering of like instructions. In this way, *redundant* complexity can be removed to increase the scope for additional *explanatory* complexity in our models. The procedure is appropriate for use with different devices and side-channels, and is self-equipped with the capability to identify scenarios where the measurements in question contain little of interest (i.e. minimal data-dependency). In addition to the valuable insights this methodology provides into leakage behaviours, which are of immediate interest to the side channel experts, it has considerable practical application via the integration of our models into a side-channel simulator (ELMO). We are thereby able to produce leakage traces for arbitrary sequences of code which demonstrably exhibit the same vulnerabilities as the same code sequences running on a real device. This capability suggests a variety of highly beneficial possible uses, such as the automated detection of leakages in the software development stage and the automated insertion (and testing) of countermeasures, as well as hugely promising prospects for optimisation with respect to protection level and energy efficiency.

8 Acknowledgments

This work has been supported in part by EPSRC via grant EP/N011635/1 and by the European Union's H2020 Programme under grant agreement number 731591, as well as by a studentship from GCHQ.

9 Availability

Our trace emulator tool (ELMO) can be downloaded from GitHub: `https://github.com/bristol-sca/ELMO`.

References

[1] J. Blömer, J. Guajardo, and V. Krummel. Provably secure masking of AES. In H. Handschuh and M. A. Hasan, editors, *Selected Areas in Cryptography – SAC '04*, volume 3357 of *LNCS*, pages 69–83. Springer, 2004.

[2] A. Buja, D. Cook, H. Hofmann, M. Lawrence, E.-K. Lee, D. F. Swayne, and H. Wickham. Statistical inference for exploratory data analysis and model diagnostics. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 367(1906):4361–4383, 2009.

[3] S. Chari, J. Rao, and P. Rohatgi. Template Attacks. In B. Kaliski, Ç. Koç, and C. Paar, editors, *CHES 2002*, volume 2523 of *LNCS*, pages 51–62. Springer Berlin / Heidelberg, 2003.

[4] O. Choudary and M. Kuhn. Efficient Stochastic Methods: Profiled Attacks Beyond 8 Bits. In *CARDIS 2014*, volume 8968 of *Lecture Notes in Computer Science*, pages 85–103. Springer, 2014.

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

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

[7] J. den Hartog, J. Verschuren, E. P. de Vink, J. de Vos, and W. Wiersma. PINPAS: A tool for power analysis of smartcards. In *International Conference on Information Security (SEC2003)*, volume 250 of *IFIP Conference Proceedings*, pages 453–457. Kluwer, 2003.

[8] F. Durvaux, F.-X. Standaert, and N. Veyrat-Charvillon. How to Certify the Leakage of a Chip? In P. Q. Nguyen and E. Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, pages 459–476, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

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

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

[11] 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*, pages 226–231. IEEE, 2014.

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

[13] A. Heuser, W. Schindler, and M. Stöttinger. Revealing side-channel issues of complex circuits by enhanced leakage models. In *Design, Automation and Test in Europe (DATE 2012)*, pages 1179–1184, 2012.

[14] A. Heuser and M. Zohner. Intelligent Machine Homicide. In W. Schindler and S. Huss, editors, *COSADE 2012*, volume 7275 of *LNCS*, pages 249–264. Springer Berlin Heidelberg, 2012.

[15] R. R. Hocking. The Analysis and Selection of Variables in Linear Regression. *Biometrics*, 32(1):1–49, 1976.

[16] A. Langley. ctgrind: Checking that functions are constant time with Valgrind. `https://github.com/agl/ctgrind`, 2010.

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

[18] 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.

[19] T. Martin. *The Designer's Guide to the Cortex-M Processor Family: A Tutorial Approach*. Newnes, 2013.

[20] 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.

[21] M. Renauld, F.-X. Standaert, N. Veyrat-Charvillon, D. Kamel, and D. Flandre. A Formal Study of Power Variability Issues and Side-Channel Attacks

for Nanoscale Devices. In K. G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 109–128. Springer, 2011.

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

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

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

[25] ST Microelectronics. *STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced ARM-based 32-bit MCUs*, 9 2016. Rev 13.

[26] C. Thuillet, P. Andouard, and O. Ly. A smart card power analysis simulator. In *Proceedings of the 12th IEEE International Conference on Computational Science and Engineering, CSE 2009*, pages 847–852. IEEE Computer Society, 2009.

[27] 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.

[28] N. Veshchikov. SILK: high level of abstraction leakage simulator for side channel analysis. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop, PPREW@ACSAC 2014*, pages 3:1–3:11. ACM, 2014.

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

[30] C. Whitnall, E. Oswald, and F.-X. Standaert. The Myth of Generic DPA...and the Magic of Learning. In J. Benaloh, editor, *CT-RSA*, volume 8366 of *LNCS*, pages 183–205. Springer, 2014.

A Supplementary Tables and Figures

Feature	Cortex M0	Cortex M4
Architecture	Von-Newman	Harvard
Word size	32 bit	32 bit
Multiplier	Single cycle	Single cycle
Instruction set	Thumb (complete) Thumb-2 (some)	Thumb (complete) Thumb-2 (complete) Additional DSP and FPU
Barrel shift instructions	No	Yes
Total instructions	56	137; Optional 32 for FPU

Table 2: Comparison between Cortex-M0 and Cortex-M4 microprocessors. Information taken from [24] [25] [19].

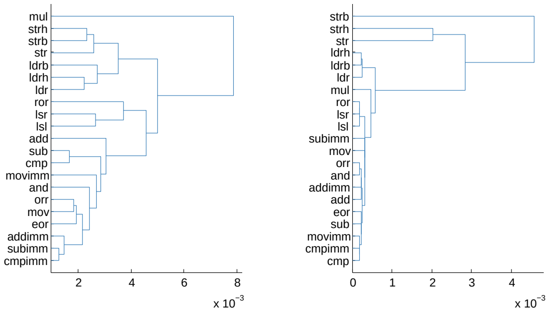

Figure 6: Dendrograms representing the hierarchical clustering of the M0 (left) and M4 (right instructions according to the fitted leakage models.

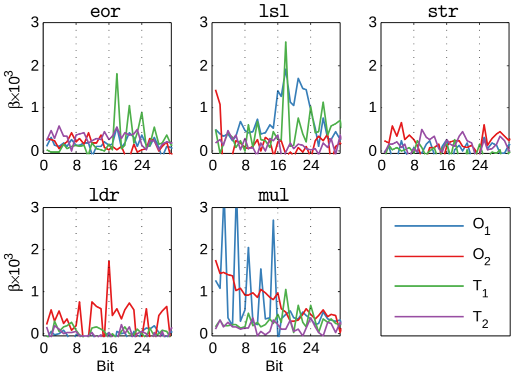

Figure 7: Estimated coefficients on the model terms for each chosen representative M0 instruction.

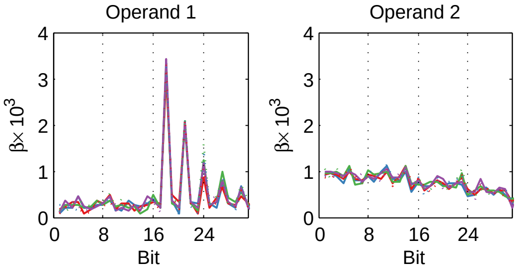

Figure 8: Estimated coefficients on the data terms as source register varies.

		adds	adds #*imm*	ands	cmp	cmp #*imm*	eors	ldr	ldrb	ldrh	lsls	lsrs
	R-squared	0.276	0.289	0.253	0.227	0.260	0.202	0.147	0.107	0.187	0.296	0.292
F-statistic	Operand 1	19.36	6.09	5.20	15.60	7.32	4.29	0.93	0.82	1.53	32.88	32.70
	Operand 2	10.23	-0.00	7.25	4.70	0.00	3.92	22.55	14.59	30.78	7.22	5.18
	Transition 1	23.35	29.52	30.40	20.25	24.12	19.35	0.93	1.18	1.40	20.18	18.88
	Transition 2	5.20	24.11	9.11	3.76	20.95	10.66	1.22	1.43	0.67	1.92	2.96
	Combined	14.51	15.44	12.89	11.19	13.40	9.63	6.55	4.54	8.78	15.98	15.69

		movs	movs #*imm*	muls	orrs	rors	str	strb	strh	subs	subs #*imm*
	R-squared	0.255	0.455	0.278	0.214	0.315	0.061	0.075	0.067	0.237	0.271
F-statistic	Operand 1	3.18	8.80	32.68	3.17	24.02	1.24	0.72	1.12	13.78	5.38
	Operand 2	3.93	-0.00	20.93	3.52	20.28	4.99	7.36	5.18	3.45	0.00
	Transition 1	22.83	53.03	2.25	15.44	23.60	1.06	1.29	1.21	24.07	27.96
	Transition 2	20.71	63.83	1.05	17.25	1.90	2.46	2.66	3.55	4.68	23.53
	Combined	13.04	31.71	14.68	10.34	17.50	2.46	3.10	2.73	11.82	14.16

Table 3: F-tests for significant joint data effects in the power consumption of the M0; tests which fail to reject at the 5% level are shaded grey. Critical values shown in brackets in the row headings. Degrees of freedom for the F-tests are (128,4871) for the combined test, (32,4871) for the rest.

		adds	adds #*imm*	ands	cmp	cmp #*imm*	eors	ldr	ldrb	ldrh	lsls	lsrs
	R-squared	0.048	0.052	0.049	0.050	0.086	0.051	0.148	0.135	0.124	0.047	0.055
F-statistic	Operand 1	1.58	3.82	1.72	2.16	10.96	3.80	1.13	1.04	1.11	0.76	1.09
	Operand 2	3.98	-0.00	3.92	3.49	0.00	2.63	22.97	20.18	18.00	4.59	6.09
	Transition 1	1.33	0.63	1.25	0.73	0.81	1.02	0.92	0.90	0.97	0.70	0.51
	Transition 2	0.67	4.07	0.73	1.47	2.25	0.87	0.91	0.67	0.87	1.11	1.07
	Combined	1.94	2.11	1.97	1.98	3.60	2.06	6.60	5.92	5.41	1.88	2.20

		movs	movs #*imm*	muls	orrs	rors	str	strb	strh	subs	subs #*imm*
	R-squared	0.029	0.063	0.038	0.038	0.117	0.546	0.814	0.691	0.046	0.068
F-statistic	Operand 1	1.09	0.76	1.30	2.59	15.85	1.13	1.09	1.05	2.62	7.06
	Operand 2	1.01	0.00	1.87	1.88	2.00	177.01	649.37	329.94	2.47	0.00
	Transition 1	1.13	0.88	1.58	0.77	1.13	0.75	1.01	0.98	0.68	1.57
	Transition 2	1.35	8.29	1.16	0.68	1.06	0.87	1.14	0.72	1.32	2.52
	Combined	1.15	2.54	1.48	1.51	5.05	45.73	166.32	85.10	1.84	2.80

Table 4: F-tests for significant joint data effects in the EM radiation of the M4; tests which fail to reject at the 5% level are shaded grey. Critical values shown in brackets in the row headings. Degrees of freedom for the F-tests are (128,4871) for the combined test, (32,4871) for the rest.

CT	**Intuitive group**					**Instructions (in**
	1	**2**	**3**	**4**	**5**	**descending order of SI)**
0.7	2	0	0	0	0	cmp subs
	2	0	0	0	0	cmp$_{imm}$ subs$_{imm}$
	2	0	0	0	0	orrs movs
	1	0	0	0	0	adds$_{imm}$
	1	0	0	0	0	eors
	1	0	0	0	0	ands
	1	0	0	0	0	movs$_{imm}$
	1	0	0	0	0	adds
	0	2	0	0	0	lsls lsrs
	0	1	0	0	0	rors
	0	0	2	0	0	strh strb
	0	0	1	0	0	str
	0	0	0	2	0	ldrh ldr
	0	0	0	1	0	ldrb
	0	0	0	0	1	muls
0.8	3	0	0	0	0	subimm cmp$_{imm}$ adds$_{imm}$
	3	0	0	0	0	movs eors orrs
	2	0	0	0	0	cmp subs
	1	0	0	0	0	ands
	1	0	0	0	0	movs$_{imm}$
	1	0	0	0	0	adds
	0	3	0	0	0	lsls lsrs rors
	0	0	3	0	0	strb strh str
	0	0	0	3	0	ldr ldrh ldrb
	0	0	0	0	1	muls
0.9	11	0	0	0	0	adds$_{imm}$ movs$_{imm}$ subs$_{imm}$ movs cmp$_{imm}$ ands orrs eors subs cmp adds
	0	3	0	0	0	lsls lsrs rors
	0	0	3	0	0	strb strh str
	0	0	0	3	0	ldr ldrh ldrb
	0	0	0	0	1	muls
1.0	11	0	0	0	0	adds adds$_{imm}$ ands cmp cmp$_{imm}$ eors movs movs$_{imm}$ orrs subs subs$_{imm}$
	0	3	0	0	0	lsls lsrs rors
	0	0	3	0	0	str strb strh
	0	0	0	3	0	ldr ldrb ldrh
	0	0	0	0	1	muls
1.1	11	3	0	0	0	adds adds$_{imm}$ ands cmp cmp$_{imm}$ eors lsls lsrs movs movs$_{imm}$ orrs rors subs subs$_{imm}$
	0	0	3	0	0	str strb strh
	0	0	0	3	0	ldr ldrb ldrh
	0	0	0	0	1	muls
1.2	11	3	3	3	1	(all; SI undefined)

Table 5: M0: Found clusters compared with intuitive grouping (1 = ALU, 2 = shifts, 3 = stores, 4 = loads, 5 = multiply) as the consistency threshold (CT) increases.

CT	**Intuitive group**					**Instructions (in**
	1	**2**	**3**	**4**	**5**	**descending order of SI)**
0.7	2	0	0	0	0	cmp cmp$_{imm}$
	2	0	0	0	0	ands orrs
	1	0	0	0	0	movs$_{imm}$
	1	0	0	0	0	subs
	1	0	0	0	0	eors
	1	0	0	0	0	adds$_{imm}$
	1	0	0	0	0	adds
	1	0	0	0	0	movs
	1	0	0	0	0	subs$_{imm}$
	0	2	0	0	0	rors lsrs
	0	1	0	0	0	lsls
	0	0	2	0	0	strh str
	0	0	1	0	0	strb
	0	0	0	2	0	ldrb ldrh
	0	0	0	1	0	ldr
	0	0	0	0	1	muls
0.8	5	0	0	0	0	cmp cmp$_{imm}$ subs movs$_{imm}$ eors
	4	0	0	0	0	orrs ands adds$_{imm}$ adds
	1	0	0	0	0	movs
	1	0	0	0	0	subs$_{imm}$
	0	3	0	0	0	lsrs rors lsls
0.9	0	0	2	0	0	strh str
	0	0	1	0	0	strb
	0	0	0	3	0	ldr ldrh ldrb
	0	0	0	0	1	muls
1.0	11	3	3	3	1	(all; SI undefined)

Table 6: M4: Found clusters compared with intuitive grouping (1 = ALU, 2 = shifts, 3 = stores, 4 = loads, 5 = multiply) as the consistency threshold (CT) increases.

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

		eors	lsls	str	ldr	muls
R^2	Full model	0.936	0.902	0.780	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
	D only model	0.014	0.031	0.016	0.012	0.057
F	$\mathbf{DxI_p}$, $\mathbf{DxI_s}$ (32)	18.5	20.8	5.5	6.2	23.7
	$\mathbf{DxI_p}$ (16)	23.4	26.7	3.6	5.9	30.5
	$\mathbf{DxI_s}$ (16)	13.6	14.8	7.4	6.6	16.9
	$\mathbf{O_1xI_p}$, $\mathbf{O_1xI_s}$ (8)	8.9	5.3	0.4	2.6	4.4
	$\mathbf{O_2xI_p}$, $\mathbf{O_2xI_s}$ (8)	33.0	25.4	3.3	8.6	11.6
	$\mathbf{T_1xI_p}$, $\mathbf{T_1xI_s}$ (8)	43.5	25.4	11.9	3.2	15.9
	$\mathbf{T_2xI_p}$, $\mathbf{T_2xI_s}$ (8)	8.9	4.8	0.5	2.1	23.5

Table 8: R-squareds for subsets of the M0 instruction models, 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.

Tested interactions	eors	lsls	str	ldr	muls
Adjacent bits	1.026	3.877	1.075	0.885	13.390
Adjacent bit flips	0.977	0.603	1.089	1.019	1.047
Non-adjacent bits	1.068	1.295	0.930	0.969	1.372

Table 9: F-tests for significant pairwise bit interaction effects (adjacent and non-adjacent) in the power consumption of the M0; tests which fail to reject at the 5% level are shaded grey. Degrees of freedom are (62,24769), (62,24707) and (930,23839) respectively.