MIRACLE: MIcRo-ArChitectural Leakage Evaluation

A study of micro-architectural power leakage across many devices, and implications for evaluation of masking schemes and leakage modelling

Ben Marshall, Dan Page and James Webb

Department of Computer Science, University of Bristol,
Merchant Venturers Building, Woodland Road,
Bristol, BS8 1UB, United Kingdom.
{ben.marshall,daniel.page,james.webb}@bristol.ac.uk

Abstract. In this paper, we describe an extensible experimental infrastructure and methodology for evaluating the micro-architectural leakage, based on power consumption, which stems from a physical device. Building on existing literature, we use it to systematically study 14 different devices, which span 4 different instruction set architectures and 4 different vendors. The study allows a characterisation of each device with respect to any leakage effects stemming from sources within the micro-architectural implementation; we use it, for example, to identify and document several novel leakage effects (e.g., due to speculative instruction execution), and scenarios where an assumption about leakage is non-portable between different yet compatible devices.

Ours is the widest study of its kind we are aware of, and highlights a range of challenges with respect to 1) the design, implementation, and evaluation of masking schemes, 2) construction of accurate fine-grained leakage models, and 3) selection of suitable devices for experimental research. For example, in relation to 1), we cast further doubt on whether a given device can or does uphold the assumptions required by a given masking scheme; in relation to 2), we ultimately conclude that real-world leakage models (either statistical or formal) *must* include information about the micro-architecture of the device being modelled; in relation to 3), we claim the near mono-culture of devices that dominates existing literature is insufficient to support general claims regarding security. This is particularly important in the context of the FIPS 140-3 standard for non-invasive side-channel evaluation.

Keywords: side-channel attack, micro-architectural leakage, leakage modelling

1 Introduction

(Micro-)architecture as a concept. In the context of processor design, the term architecture[1] is understood as describing the interface between hardware and software. It defines how hardware and software interact, i.e., what is "visible" to the programmer, and so will typically include a definition of 1) state, 2) instructions that act on said state, and 3) an execution model for said instructions. The term Instruction Set Architecture (ISA) is often used synonymously, with micro-architecture[2] describing an implementation of the

[1] The term architecture seems to stem from the IBM System/360 design, which considered it as capturing "the attributes of a system as seen by the programmer" [ABB64, Page 84]; before this, the more nebulous term organisation was typical.

[2] It seems likely the term micro-architecture stems from use of micro-coded implementations, where it can be read as the architecture controlled by a micro-program (cf. architecture controlled by a program written using the ISA).

associated ISA, i.e., as a specific processor core. As such, the ISA represents a logical abstraction of an underlying, physical micro-architectural implementation.

Beyond pure definitional precision, this abstraction enables an important property that we now, to some extent, expect and even rely on. Principally, it is responsible for allowing behavioural diversity while ensuring functional compatibility, or, put another way, maximising flexibility with respect to implementation while retaining consistency with respect to usage: a by-design disconnection between behavioural and functional semantics of instruction execution means different micro-architectures can realise the same ISA, but, in doing so, employ features that differ in their design and/or implementation. By harnessing this fact, specific[3] micro-architectures can, for example, aggressively optimise instruction execution to suit a given market or use-case, without placing a burden on the programmer. Doing so often acts within a broader strategy to address limitations on scaling (e.g., clock frequency) that stem from Moore's Law.

(Micro-)architecture as an attack vector. Viewed from a different perspective, however, the same property *can* be problematic. For example, consider that development of high-assurance software typically requires both detailed knowledge of, and control over both functional and behavioural semantics of instruction execution. When met, such requirements permit 1) formal reasoning, and guarantees about functional correctness, and 2) management of the associated (implementation) attack surface, e.g., by instrumenting suitable countermeasures. The abstraction of a micro-architecture by an ISA necessarily limits the degree to which this is true, however, so means the requirements are often *not* met (or at least not sufficiently so).

The way in which micro-architectural side-channel attack techniques (see, e.g., [Sze19, Section 4] and [GYCH18, Section 4]) are enabled and/or exacerbated offers an pertinent example of this problem. At a high level, such techniques harness leakage that stems from sources in a particular micro-architecture: one can classify such leakage as either

1. *discrete* (or *digital*), meaning it relates to logical, or functional characteristics, e.g., data-dependent instruction execution latency (i.e., number of cycles) caused by micro-architectural state and execution model, or
2. *analogue*, meaning it relates to physical, or behavioural characteristics, e.g., data-dependant power consumption [KJJ99, MOP07] or EM [GMO01, AARR02] emission caused by the behaviour of CMOS transistors which constitute the micro-architecture.

In a sense, the latter acts as a superset of the former: because analogue leakage can capture fine-grained, potentially sub-cycle features, discrete forms of leakage will typically be captured by it indirectly. Either way, however, micro-architectural abstraction implies 1) the security properties of software are difficult to reason about, and may even differ depending on the micro-architecture it is executed on, and therefore 2) development of robust software-based countermeasures is a significant challenge. Such implications have led to arguments (see, e.g., [GYH18]) for migration of traditionally opaque ISAs toward more (semi-)transparent alternatives in which (selected) micro-architectural features are visible. Likewise, they have motivated hardened micro-architectural designs (see, e.g., [KGBR19, MGH19]) which mitigate the lack of such transparency.

Remit and organisation. The *majority* of micro-architectural side-channel attacks harness some form of discrete leakage. Equally, however, there is an increasing body of work which explores analogue forms of leakage. To a significant degree, such work has been motivated by observations about the security of software-based masked implementations of

[3] For example, modulo trifurcation into mobile, application, and real-time profiles, the same ISA has been harnessed across a wide range of low(er)-end (e.g., ARMv7-M ISA, ARM Cortex-M3 micro-architecture), mid-range (e.g., ARMv7-A ISA, ARM Cortex-A17 micro-architecture), *and* high(er)-end (e.g., ARMv7-A ISA, Qualcomm Krait micro-architecture) micro-architectural implementations.

cryptography: an exemplar is the "gap" between a theoretically, *provably* secure masking scheme proposed by Rivain and Prouff [RP10], versus attacks on a practical implementation thereof by Balasch et al. [BGG+14].

Set within the associated literature, we position this paper as contributing in the following ways. First, versus the Rosita framework of Shelton et al. [SSB+20], for example, we deliberately attempt to broaden the scope by 1) focusing on multiple, physical (vs. single, emulated) processor cores, and 2) increasing the diversity and complexity of micro-architectural features considered. Second, we place explicit value on increasing the extent to which "folk-law" observations are explainable. To a greater degree than previously, doing so provides a formal basis for the topic. Third, we place explicit value on the reusability of associated artefacts. We posit, for example, that the infrastructure and data sets stemming from our work can support forms of leakage-aware verification, such as that of Barthe et al. [BGG+20], which demand accurate, fine-grained leakage models. More specifically then, the paper is organised as follows:

- Section 2 surveys existing literature related to micro-architectural leakage. Based on this, Section 3 then attempts to define a precise, unified terminology. This allows development of a structured classification for leakage sources and effects, and so clearer discussion and evaluation of associated work.
- Section 4 describes an extensible experimental infrastructure and methodology for evaluating the micro-architectural leakage, based on power consumption, which stems from a processor core: we dub this MIRACLE, which is a backronym for MIcRo-ArChitectural Leakage Evaluation.
- Section 5 analyses specific data sets produced by the infrastructure, in order to document several novel, low-level leakage effects, and to explore some overarching high-level hypotheses. These include 1) to what extent common assumptions, such as Only Computation Leaks (OCL) [MR04] and Independent Leakage Assumption (ILA) [RSVC+11, Section 2.2], hold in practice, and 2) whether and how identical instruction sequences leak on different but compatible devices, and therefore how "portable" some forms of countermeasure are.
- Section 6 then, finally, attempts to summarise the implications for situations when consideration of micro-architectural leakage is important; we pitch this as a limited set of design or implementation guidelines for digital design and cryptographic engineers.

We carefully limit the remit of constituent work in the following ways. First, we focus exclusively on analogue micro-architectural leakage related to power consumption. We use the term micro-architectural leakage synonymously from here on, but stress that most of our results are more general, e.g., apply to instances such as EM emission. Second, we focus exclusively on identifying and documenting micro-architectural leakage; this means we deem exploitation of, and countermeasures based on leakage effects out of scope.

2 Background

In this section, we survey existing literature related to micro-architectural leakage: in however limited a sense, the goal is to 1) organise and summarise relevant work, which spans different research fields and focuses, and 2) clarify the relationship between this paper, and relevant work which we either build on and/or produce implications for.

(Micro-)architectural leakage effects. There have been many efforts to study particular micro-architectural sources of power leakage, and how they can undermine masking based countermeasures. In [BGG+14], the authors discuss how physical effects in a hardware device ("glitches and transition-based leakages") can halve the security order of a masked implementation. This was followed by [PV17], which details several *specific*

micro-architectural effects which can be repeatedly demonstrated as undermining a masked software implementation. This includes the "overwrite-effect" where registers are overwritten with sensitive values, the "memory remnant" effect where values written or read from memory are buffered unexpectedly and the "neighbour-leakage" effect, where explicit accesses to one register can cause implicit accesses to an adjacent register. Subsequent works such as [CGD18] and [SSB+20] replicate some or all of these effects in different devices, and introduce either new variants, or more specific cases of known variants. In [MMT20], the authors survey four devices and explicitly build on the work of [PV17]. We consider [MMT20] the work most similar to this one, in that it also surveys a wider range of effects and devices, confirming (as discussed later) that leakage effects vary widely between devices.

Other works have examined particular processor cores in great detail from a micro-architectural perspective [CGMA+15, BP18, DAK19], but have not looked more widely at families of cores, such as all ARM Cortex-M devices.

(Micro-)architectural leakage modelling and tooling. In [MWO16], the authors build a statistical leakage model of ARM Cortex-M0 and Cortex-M4 cores by examining tuples of adjacent instructions, and using statistical regression to model the resulting information leakage. This successfully captures even very esoteric kinds of leakage in instructions which are executed very closely together. However, as explored in [SSB+20], it can fail to capture cases where instructions which are executed far apart in time can induce leakage in one another. The best example being load and store instructions, which is discussed in [SSB+20, Section IV(C)], and examined in detail in Section 5.1 of this work. Another limitation of statistical modelling techniques is ensuring that a representative sample of instruction executions and sequences are used to build the model. The authors of [MWO16] found it was possible to group instructions into similarly leaking groups to minimise the problem space, but the problem of generating large amounts of initial stimulus remains.

Complementing statistical approaches to leakage modelling are formal modelling approaches. These have mostly been applied to masked *hardware* implementations, as with the `maskVerif` [BBC+19] and REBECCA [BGI+18] tools, or more recently, SILVER [KSM20].

Such techniques have also been applied to software masking. In [BGG+20], the authors use a Domain Specific Language (DSL) to describe the individual instructions of a processor core, and the state elements they update. This model of the micro-architecture is supplied by the user, and may need to be reverse engineered from an existing device. Assuming the accuracy of the micro-architecture model, this is a compelling approach. The TORNADO tool [BDM+20] is another similar approach, which, combines the Usuba bitslicing compiler [MDLG18] and the tightProve [BGR18] tools to generate C programs with formal side-channel security assurance in the register probing model.

In [GHP+20], the authors describe techniques for co-design and formal verification of hardware and side-channel resistant software using the REBECCA tool [BGI+18]. They give a worked example of their tool using the Ibex RISC-V core, and the parts of the hardware which gave rise to various micro-architectural sources of leakage. Their work is an excellent guide to explaining why certain leakages might be visible in the black-box processor cores analysed in this work.

3 Terminology

Per Section 2, study of micro-architectural leakage exists at the intersection of several different research fields, and so, in part as a result of this, associated terminology has evolved which is imprecise or inconsistent. This can, for example, make it difficult to clearly

discuss and evaluate associated work. Before any technical contributions we attempt to address this issue, doing so in two steps, i.e., first addressing architectural then micro-architectural concepts. In each such step, we use a model for instruction execution to classify associated leakage sources. Doing so allows one to reason about how and why leakage occurs, and thus how it may be avoided (or not).

It is important to note that *none* of this will be deemed innovative to digital design engineer. As well as supporting the remainder of this paper, however, we posit that shared terminology and understanding is a necessary starting point to enable 1) digital design engineers (cf. [BMT16]) to reason about the impact of leakage from their (hardware) designs and implementations, and 2) cryptographic engineers to develop useful leakage models, and thus leakage-free (software) designs and implementations.

3.1 Notation

Let $\mathsf{MEM}[i]$ denote the i-th element of memory: $\mathsf{MEM}[i]^j$ is used to specify an access granularity of j bytes where appropriate, with $j = 1$ (implying memory is byte addressed) assumed if/when omitted. Let $\mathsf{GPR}[i]$ denote the i-th General Purpose Register (GPR); we refer to a given Special Purpose Register (SPR) by name, including an optional field where appropriate. Within the specific context of ARMv7-M, for example, $\mathsf{PC} \equiv \mathsf{GPR}[15]$ might denote the Program Counter (PC), whereas $\mathsf{CPSR}[C] \equiv \mathsf{CPSR}_{29}$ might be used to denote the carry flag within the Current Program Status Register (CPSR).

Let $\mathbb{E}_i$ and $\mathbb{C}_i$ denote some i-th execution and clock cycle respectively. We describe $\mathbb{E}_i$ and $\mathbb{E}_j$ (resp. $\mathbb{C}_i$ and $\mathbb{C}_j$) as being separated by a distance of n if $|i - j| = n$, noting that the specific case where $n = 1$ implies they are consecutive.

3.2 Architectural leakage

3.2.1 Model

We assume an ISA will include a definition of (at least)

- a set of architectural state, namely storage elements such as GPRs, SPRs, and memory,
- a set of architectural instruction semantics: any given instruction may read values from state elements, performs computation, and write values to state elements, and
- an instruction execution model.

An ISA will usually adopt a in-order execution model, wherein each execution cycle will atomically and independently fetch, decode, then execute a single instruction; only architectural state is guaranteed to be preserved between execution cycles.

3.2.2 Leakage

Definition 1. Architectural leakage can be inferred from the architectural definition of instruction execution, meaning it stems *purely* from architecturally visible detail (and is thus micro-architecture agnostic).

Definition 2. Intra-instruction leakage can be reasoned about with within the context of 1 execution cycle (i.e., execution of 1 instruction). This contrasts with **inter-instruction leakage**, which cannot: it occurs, and thus must be reasoned about over $n > 1$ execution cycles (i.e., execution of n instructions).

Note that inter-instruction leakage is not limited to adjacent instructions, (i.e., those executed in $\mathbb{E}_i$ and $\mathbb{E}_{i\pm1}$). As we explore later, interaction between and thus leakage from *non*-adjacent instructions (i.e., those executed in $\mathbb{E}_i$ and $\mathbb{E}_{i\pm j}$ for some $j > 1$) is also plausible.

Given an instruction sequence, the definitions above imply that associated architectural leakage can be inferred abstractly, i.e., *without* using a concrete implementation of the ISA: viable approaches include the use of an instruction set simulator, or even static analysis. For example, consider the following 3-instruction sequence:

```
1 add a0, a1, a2 // HW(a1) + HW(a2) + HW(a1 + a2) + HD(a0, a1 + a2)
2 add t0, t0, t2 // HW(t0) + HW(t2) + HW(t0 + t2) + HD(t0, t0 + t2)
3 lsl t0, t0, #4 // HW(t0) + HW(t0 << 4) + HD(t0, t0 << 4)
```

Based on an assumed leakage model for register access and bus activity, the annotation captures inferred architectural leakage, i.e., 1) Hamming weight leakage for each operand related to each read from the GPRs, 2) Hamming weight leakage related to each result computed, and 3) Hamming distance leakage related to each write to the GPRs. It is trivial to identify instances of inter- and intra-instruction leakage: the former occurs when an instruction reads operands from the GPRs or performs computation, whereas the latter occurs when an instruction writes results to the GPRs and thereby *over*writes a value already there (as written by a previous instruction, and therefore implying the required interaction).

As an aside, certain sources of architectural leakage are closely related to features in the ISA design. A given compressed instruction format, such as the RISC-V standard C extension [RV:19, Section 16] or ARMv7-M Thumb [ARM18, Chapter A5], will often employ destructive, e.g.,

$$\texttt{add r1, r2} \quad \mapsto \quad \mathsf{GPR}[1] \leftarrow \mathsf{GPR}[1] + \mathsf{GPR}[2],$$

versus non-destructive, e.g.,

$$\texttt{add r0, r1, r2} \quad \mapsto \quad \mathsf{GPR}[0] \leftarrow \mathsf{GPR}[1] + \mathsf{GPR}[2],$$

semantics with respect to the destination register: use of the former forces Hamming distance leakage between $\mathsf{GPR}[1]$ and $\mathsf{GPR}[1] + \mathsf{GPR}[2]$ which can be avoided in (careful) use of the latter. There is an increasingly accepted argument (see, e.g., [RKL+04, RRKH04]) that security should be considered as a first-class metric at design-time, and, as such, one *could* argue this and similar examples should be considered within the ISA design process.

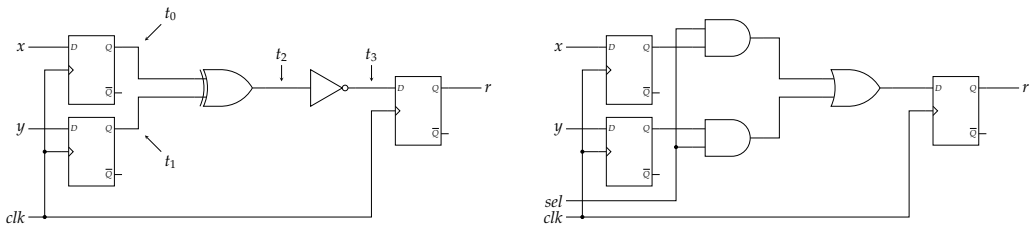
3.3 Micro-architectural leakage

3.3.1 Model

Existing literature (see, e.g., [PV17, CGD18, SSB+20, MMT20]) has now clearly demonstrated that architectural leakage cannot accurately capture every pertinent leakage source or, therefore, effect. To do so, one *must* consider leakage which stems from the implementation of an ISA, i.e., the micro-architecture. We address this fact by extending our original model in Section 3.2: specifically, we

- extend the architectural state with a set of micro-architectural state plus a and a mapping function between architectural and micro-architectural state elements, and
- extend the architectural instruction semantics with a set of micro-architectural instruction semantics.

One could define micro-architectural state elements as being those implicitly supporting execution of instructions (by maintaining any associated values while execution occurs); this contrasts with architectural state elements, which are those explicitly used by instructions. Likewise, micro-architectural semantics describe execution of an instruction in terms of micro-architectural state elements; this contrasts with architectural semantics, which do so in terms of architectural state elements.



(a) Non-glitching combinatorial leakage.

(b) Glitching combinatorial leakage.

Figure 1: Two example designs, illustrating the difference between non-glitching combinatorial leakage and glitching combinatorial leakage. In the top design, only combinatorial leakage is evident, based on how the signals t_0 and t_1 change on each clock cycle; subsequent signals, e.g., t_2 and t_3, change only once per clock cycle. Note this ignores the effect of wire delay on t_0 and t_1. In the bottom design (a multiplexer), if the *sel*, x, and y signals all change, then the delayed *sel* signal will cause Hamming distance leakage on r: this causes it to glitch, i.e., change multiple times per clock cycle.

In contrast with architectural state, micro-architectural state may or may not be preserved across either execution or clock cycles. As such, the micro-architectural state mapping function and the instruction semantics must be considered, to some extent, with respect to an execution context which includes 1) the current micro-architectural state, 2) the "current" instruction (i.e., that executed in $\mathbb{E}_i$), *and* 3) the "surrounding" instructions (i.e., those executed in $\mathbb{E}_{i\pm j}$ for $1 \leq j < m$ given some m). For example: where pipelining is employed, instructions following a branch may or may not be (completely) executed depending whether the branch is taken or not; where pipelining is employed, the (concrete) state element corresponding to an (abstract, architectural) register may be part of the forwarding logic versus the register file; where register renaming is employed, the (concrete) state element corresponding to an (abstract, architectural) register may differ for each write to that register.

3.3.2 Leakage

Definition 3. Micro-architectural leakage cannot be inferred from the architectural definition of instruction execution, meaning it stems *purely* from architecturally invisible detail (and is thus micro-architecture specific).

Definition 4. Intra-cycle leakage can be reasoned about with within the context of 1 clock cycle. This contrasts with **inter-cycle leakage**, which cannot: it occurs, and thus must be reasoned about over $n > 1$ clock cycles.

Definition 5. Sequential micro-architectural leakage occurs when the value of a state element changes.

We specifically use the term sequential leakage, because it stems from the sequential logic elements, flip-flops and registers constructed from them, used within a design. A specific instance of sequential micro-architectural leakage is necessarily classified as inter-cycle, because it occurs at the boundaries of clock cycles. However, it may be classified as *either* intra-instruction, because it can stem from isolated execution of 1 instruction (e.g., during iterative computation of a multiplication), *or* inter-instruction, because it can stem from interaction between execution of $n > 1$ instructions (e.g., a specific state element is updated by different instructions respectively executed in $\mathbb{E}_i$ and $\mathbb{E}_{i\pm j}$ for $j \neq 0$).

Definition 6. Non-glitching combinatorial micro-architectural leakage occurs when the output of a combinatorial logic element toggles; **glitching combinatorial**

micro-architectural leakage occurs when the output of a combinatorial logic element toggles more than once per clock cycle.

In existing literature, it is common for the catch-all term "glitching" to be used for either case. We carefully distinguish between the cases, observing that all leakage caused by glitching is combinatorial leakage but not all combinatorial leakage is caused by glitching. The designs in Figure 1 represent (simple) instances of the non-glitching and glitching cases respectively. Various other instances are arguably more subtle, in the sense they relate more to the physical properties of an implementation than logical properties of a design. For example, it is common to observe combinatorial leakage stemming from an inverter chain: these structures are used, for example, to mitigate issues due to 1) the standard cell library (e.g., to allow a high degree of fan-out), and/or 2) place-and-route (e.g., to drive signals over long wires).

Beyond some niche exceptions (e.g., multi-cycle paths) we deem out of scope, a specific instance of combinatorial micro-architectural leakage is necessarily classified as intra-cycle and therefore intra-instruction. This is because, within the context of a sequential logic design, the value produced by a combinatorial logic element must settle before the next clock cycle (otherwise a violation of the critical path occurs).

3.4 Summary

Definition 7. A given **leakage source** can be classified as being either

1)	**architectural**	≡	**A-class**
2)	**sequential micro-architectural**	≡	**SMA-class**
3)	**non-glitching combinatorial micro-architectural**	≡	**NCMA-class**
4)	**glitching combinatorial micro-architectural**	≡	**GCMA-class**

and further qualified as manifesting on an a) **intra-instruction**, b) **inter-instruction**, c) **intra-cycle**, or d) **inter-cycle**, basis. A leakage source is therefore defined as the design or implementation feature that enables leakage to occur.

We stress that, due to our focus on the architectural and micro-architectural levels within what is a larger stack of abstractions, this classification should be viewed as necessary but not sufficient: it cannot and thus does not capture *every* leakage source. A pertinent example is that of capacitive coupling between wires (see, e.g., [CBG+17, CEM18, LBS19]), which, although important to model, we regard as a separate problem. Our justification is that there is no architecture or micro-architecture at that level of abstraction, only standard cells are wires: capacitive coupling leakage can therefore be regarded as at best indirectly related to any particular instruction sequence and execution of it, versus architectural or micro-architectural leakage where the same relationship is more direct.

Definition 8. A **leakage effect** is the form of information observable whenever a leakage source causes leakage due to execution of an instruction sequence we term a **leakage trigger**.

The term micro-benchmark[4] refers to a well established concept: it is a short, self-contained instruction sequence specifically designed to analyse (e.g., evaluate) some feature in the processor core that executes it. For example, the performance counter based nanoBench framework of Abel and Reineke [AR19] was used to determine (or reverse engineer) the latency, throughput, and port usage of x86 instructions, and cache architecture of x86 processor cores. We adapt the term as follows:

[4]Different, context-specific terms are sometimes used for what is essentially the same concept. For example, the term litmus test is common within the context of concurrent hardware or software.

Definition 9. A **leakage micro-benchmark** is a specific instruction sequence constructed to prove or disprove a hypothesis about, e.g., the existence of a leakage source (and hence associated leakage effect).

4 Infrastructure

Fundamentally, we are interested in evaluating the (micro-)architectural leakage stemming from a given processor core. In this section we describe the components of MIRACLE, our experimental infrastructure and methodology for doing so. We start by introducing high-level terminology related to the components and processes involved, then, in subsequent subsections, offer more, lower-level detail.

We refer to the physical integrated circuit containing some System on Chip (SoC) as a **device**; such a device will contain one or more **cores**[5]. Most devices cannot be used in a stand-alone manner, because, for example, they require surrounding infrastructure for power delivery. We refer to this infrastructure as the host **platform**, noting that both general-purpose (i.e., support multiple devices), and special-purpose (i.e., be specific to, and even integrated with a device) instances are possible. As a result, the **target** of evaluation is defined by a 3-tuple of core, device, and platform. An evaluation, which we refer to as an **experiment**, is coordinated by a **controller** (i.e., a workstation) in two steps: it 1) acquires a set of power consumption **traces** during execution of some **micro-benchmark** by the target, then 2) subjects the traces to some form(s) of analysis, attempting to prove or disprove an associated **hypothesis**. We refer to the power consumption traces as the **trace data set** and results of analysis as the **analysis data set**.

Note that we avoid direct comparison of data sets stemming from different targets and therefore devices. For example, a conclusion such as "*leakage is stronger in target X than target Y*" is not possible. However, we do compare targets with respect to the associated hypothesis with a particular micro-benchmark. For example, a conclusion such as "*target X and Y both exhibit leakage effect Z*" is possible.

4.1 Devices

Let $\mathsf{f}_i^{\mathsf{d}}$ denote a device where f is the family identifier (e.g., ARM), d is the core identifier (i.e., the specific processor core), and i is the instance number (where more than one exits). Where necessary, we permit further annotation st.

$$
\begin{array}{ll}
\bar{\mathsf{f}}_i^{\mathsf{d}} & \implies \text{an ASIC-based (or "hard") device} \\
\widetilde{\mathsf{f}}_i^{\mathsf{d}} & \implies \text{an FPGA-based (or "soft") device} \\
\mathsf{f}_i^{\mathsf{d}}\,[x\text{MHz}] & \implies \text{a device operating at a specific clock frequency (of } x\text{MHz)}
\end{array}
$$

We use this notation in Table 1, which describes the entire set of 14 different devices (currently) supported by MIRACLE. Note that each FPGA-based device is synthesised using Xilinx Vivado 2019.1; default synthesis settings are used, with no effort invested in synthesis or post-implementation optimisation.

4.2 Platforms

Each device is supported by and so situated within a specific platform. MIRACLE (currently) supports 3 different platforms, described in detail by the following:

[5] In our case these are processor cores, but note that the terminology could be extended to accommodate, e.g., hardware accelerators.

Identifier	Instances	Platform	Vendor	Device	Package	Core	Micro-architecture	ISA	Flash	SRAM	References
$\overline{\text{ARM}}^{\text{N0}}$	1	SCALE	NXP	LPC812M101JDH16	TSSOP-16	ARM Cortex-M0+	32-bit 2-stage pipeline 1-cycle multiplier	ARM6-M	16 kB	4 kB	[ARMb, NXPa]
$\overline{\text{ARM}}^{\text{N1}}$	1	SCALE	NXP	LPC1114FN28/102	DIP-28	ARM Cortex-M0	32-bit 3-stage pipeline	ARM6-M	32 kB	4 kB	[ARMc, NXPb]
$\overline{\text{ARM}}^{\text{N2}}$	1	SCALE	NXP	LPC1313FBD48/151	LQFP-48	ARM Cortex-M3	32-bit 3-stage pipeline 1-cycle multiplier	ARMv7-M	32 kB	8 kB	[ARMd, NXPc]
$\overline{\text{ARM}}^{\text{N3}}$	3	CW308	NXP	LPC1115FBD48/303	LQFP-48	ARM Cortex-M0	32-bit 3-stage pipeline	ARM6-M	64 kB	8 kB	[ARMc, NXPb]
$\overline{\text{ARM}}^{\text{S0}}$	1	CW308	STM	STM32F071RBT6	TQFP-64	ARM Cortex-M0	32-bit 3-stage pipeline	ARM6-M	128 kB	16 kB	[ARMc, Mea, New]
$\overline{\text{ARM}}^{\text{S1}}$	1	CW308	STM	STM32F100RBT6B	TQFP-64	ARM Cortex-M3	32-bit 3-stage pipeline 1-cycle multiplier	ARMv7-M	128 kB	8 kB	[ARMd, Meb, New]
$\overline{\text{ARM}}^{\text{S2}}$	1	CW308	STM	STM32F215RET6	TQFP-64	ARM Cortex-M3	32-bit 3-stage pipeline 1-cycle multiplier	ARMv7-M	512 kB	128 kB	[ARMd, Mec, New]
$\overline{\text{ARM}}^{\text{S3}}$	1	CW308	STM	STM32F303RCT7	TQFP-64	ARM Cortex-M4	32-bit 3-stage pipeline 1-cycle multiplier	ARMv7-M	256 kB	40 kB	[ARMe, Med, New]
$\overline{\text{ARM}}^{\text{S4}}$	1	CW308	STM	STM32F405RGT6	TQFP-64	ARM Cortex-M4	32-bit 3-stage pipeline 1-cycle multiplier	ARMv7E-M	1 MB	192 kB	[ARMe, Mee, New]
$\overline{\text{ARM}}^{\text{S5}}$	3	CW308	STM	STM32F051C8T6	LQFP-48	ARM Cortex-M0	32-bit 3-stage pipeline	ARM6-M	64 kB	8 kB	[ARMc, Mef]
$\widetilde{\text{MB}}^{\text{X0}}$	1	SASEBO-GIII	XLNX			MicroBlaze v10.0	32-bit 3-stage pipeline	MicroBlaze			[Xila]
$\widetilde{\text{MB}}^{\text{X1}}$	1	SASEBO-GIII	XLNX			MicroBlaze v10.0	32-bit 5-stage pipeline	MicroBlaze			[Xila]
$\widetilde{\text{MB}}^{\text{X2}}$	1	SASEBO-GIII	XLNX			MicroBlaze v10.0	32-bit 8-stage pipeline	MicroBlaze			[Xila]
$\widetilde{\text{RV}}^{\text{PRV}}$	1	SASEBO-GIII				PicoRV32	32-bit multi-cycle	RV32IMC			[Wol]

Table 1: Pertinent technical detail for each device (currently) supported by MIRACLE. Note that we use a short-hand for vendors: NXP denotes NXP, STM denotes STMicroelectronics, INTC denotes Intel, XLNX denotes Xilinx.

```
extern volatile void payload( word_t* inputs );

void driver( ctx_t* ctx ) {
  word_t* inputs = ctx->receive_inputs();
  ctx->device_set_trigger();
  payload( inputs );
  ctx->device_clear_trigger();
}
```

(a) The goal-agnostic, high-level driver (implemented in C).

```
.global payload

payload: push { r4, r5, r6, r7, lr } // Preserve callee save GPRs
         <clear callee save registers>
         <load inputs>
kernel:  <execute kernel>
         <clear used registers>
         pop  { r4, r5, r6, r7, pc } // Restore  callee save GPRs
```

(b) The goal-specific, low-level payload (implemented in assembly language).

Figure 2: The 2-part structure of each micro-benchmark.

- SCALE describes a platform based on the SCALE[6] host board; it supports a range of interchangeable target boards, and thus devices. The trace acquisition pipeline includes an on-board NXP BGA2801 amplifier (with 22 dB gain), and an on-board 2.6 MHz low-pass filter.
- CW308 describes a platform based on the ChipWhisperer CW308 (or UFO)[7] host board; it supports a range of interchangeable target boards, and thus devices. The trace acquisition pipeline includes an off-board Agilent 8447D amplifier (with 25 dB gain), and an off-board MiniCircuits SLP-30+ 32 MHz low-pass filter.
- SASEBO-GIII describes a platform based on the SASEBO-GIII [HKSS12] side-channel analysis platform; it houses two FPGAs, a Xilinx Kintex-7 (model `xc7k160tfbg676`) target FPGA, and a Xilinx Spartan-6 (model `xc6slx45`) support FPGA, and thus can be reconfigured to support a range of devices. The trace acquisition pipeline includes an off-board MiniCircuits BLK+89 D/C blocker, an off-board Agilent 8447D amplifier (with 25 dB gain), and an off-board MiniCircuits SLP-30+ 32 MHz low-pass filter.

The trace acquisition process is coordinated by a controller, which is connected to and so communicates with both the platform and a PicoScope 5000 series oscilloscope which terminates the trace acquisition pipeline. Although a platform- and/or device-specific approach to configuration and programming is required (e.g., via a Xilinx Platform Cable USB II for the SASEBO-GIII platform, or a Segger J-Link and OpenOCD for ARM-based devices), communication between the controller and platform is supported in a more uniform manner: each platform uses an FTDI-based FT232 USB-to-UART bridge then exposed to the device, which is on-board for the SCALE platform and off-board for the CW308 and SASEBO-GIII platforms.

4.3 Micro-benchmarks

Notation. MIRACLE (currently) supports 23 different micro-benchmarks, each of which we refer to using MAJOR/MINOR as a short-hand: MAJOR is the major micro-benchmark identifier (or class), and MINOR is the minor micro-benchmark identifier (i.e., instance within said class). Where clear from the context, we use the minor micro-benchmark identifier alone.

Each micro-benchmark must be implemented for each device, or, rather, for each unique ISA. The 7 different cores (currently) supported by MIRACLE span 4 different ISAs, however, which presents a challenge with respect to how to describe them. We use one of two approaches, depending on who or what the user of such a description is. First, our human-readable "on paper" description uses the combination of 1) a written explanation of the underlying goal, plus 2) an illustrative, pseudo-code example implementation modelled on the use of ARMv7-M; in some instances, we employ stylistic alterations[8] to improve their readability. On one hand, we use ARMv7-M because we expect this to be the most familiar of those ISAs support (and so add most value with respect to illustrating the underlying goal). On the other hand, although concrete ARMv7-M assembly language syntax and instruction mnemonics are used, for example, we opt for abstract, symbolic notation for values, register identifiers, etc. More specifically, we use the following notation:

- `A` through `G` represent variables; note these are not hexadecimal literals, which would be prefixed by `#`. Unless otherwise noted or self-evident, any two variables, say `A` and `B`, are always allocated different architectural registers.
- `rA` denotes a register which contains some variable `A` which is relevant to the associated experiment.
- `rX` and `rY` denote registers which contain an address or variable which is irrelevant to the associated experiment.
- `rZ` denotes a register which contains zero.

Second, our machine-readable "as source code" description is then of course a functionally equivalent but ISA-specific realisation of the pseudo-code; in doing so there is need to cope with differences between ISAs (e.g., the availability of a specific instruction type, or addressing mode), but, in all cases, we carefully avoid impact on the underlying hypothesis. Beyond formulation of the micro-benchmark goal, we found this to be the most fragile and therefore challenging aspect of the development process.

Structure. Each micro-benchmark is implemented using the 2-part structure described in Figure 2: the goal-agnostic, high-level micro-benchmark **driver** (Figure 2a) is implemented using C, whereas the goal-specific, low-level micro-benchmark **payload** (Figure 2b) is implemented using assembly language for the appropriate ISA. The driver part is identical for all devices, but specific to a given micro-benchmark: it is responsible, e.g., for 1) receiving a set of inputs, which are generated uniformly at random then communicated by the controller, 2) managing aspects of trace acquisition (e.g., the trigger signal), and 3) invoking the associated payload. Note that `word_t` is `typedef`'ed to reflect the word size, e.g., to `uint32_t` or `uint64_t`. The payload part is responsible for realising the micro-benchmark itself; the micro-benchmark **kernel** is surrounded by additional instructions, which, for example, 1) clear callee-save registers before execution to ensure a fixed architectural state and prevent interaction with the input, and 2) clear all registers used by the kernel after one execution to prevent interaction with subsequent executions.

[6] https://github.com/danpage/scale-hw
[7] https://wiki.newae.com/CW308_UFO_Target
[8] Examples include additional space or indentation, or labels which are either shorter or more meaningful given the associated description.

Analysis. For a given device, we execute each each micro-benchmark n times and so acquire n associated traces of power consumption; these, along with the input variables that allows us to completely all intermediate values, constitute the trace data set for a given experiment. Unless otherwise stated, and without loss of generality, we fix $n = 20,000$ for ASIC-based devices and $n = 30,000$ for FPGA-based devices; the traces are padded or truncated appropriately so each one contains m samples.

MIRACLE (currently) supports 14 devices and 23 benchmarks, meaning a total of 322 such trace data sets. We subject each one to Hamming weight and Hamming distance based correlation analysis, opting for this approach over techniques such as Test Vector Leakage Assessment (TVLA) [GJJR11] or Welch's t-test [Wel47] because it allows more confidence in a qualitative assessment of 1) whether or not there was *an* interaction between variables, and 2) exactly *which* interactions take place if so. Finally, the analysis data set that results is manually used to reason about a conclusion to the hypothesis associated with the experiment.

4.4 Artifacts

Among the goals of MIRACLE, we view accessibility, reproducibility, and extensibility as important[9] and so in need of explicit consideration: other researchers should, as far as possible, be able to access all the associated artefacts (i.e., source code and data sets), reproduce (and/or contest!) our conclusions, and develop and use, for example, additional micro-benchmarks for their own platforms and/or devices.

Similar goals are now best practice for research in general, but, within the context of MIRACLE specifically, we address them in two ways. First, all source code for MIRACLE is available online at

`https://github.com/scarv/miracle`

Second, we developed a web-based interface at

`https://miracle.scarv.org`

which offers a straightforward way to 1) inspect the binary or disassembled form of each micro-benchmark, and 2) view and compare the associated analysis data sets. Note, however, that we do *not* (currently) retain any trace data sets. These (currently) represent more than 1 TB, which means the monetary cost and logistics of long-term storage and access are (currently) problematic. We hope to find a way to resolve this in the future.

5 Case studies

Note that *all* devices (currently) supported by MIRACLE, per Table 1, have a 32-bit data-path and could be described as micro-controller class. Although MIRACLE is general-purpose, this (initial) selection was intended to facilitate an exploration of (micro-)architectural leakage stemming from ostensibly similar (i.e., from the same class, and so designed to satisfy similar use-cases) devices commonly used in existing literature.

Using the MIRACLE infrastructure outlined in Section 4, this section presents said exploration via a set of case studies: each has the same structure, in the sense it 1) describes the set of micro-benchmarks used, 2) summarises the resulting analysis data sets, then 3) discusses those results, e.g., attempting to explain their occurrence, relevance, and/or implication. Some case studies replicate and generalise results in existing literature, while others, to the best of our knowledge, are novel (either in terms of the leakage effect and/or associated source).

[9] We point, for example, to the TCHES artifact evaluation process at `https://ches.iacr.org/2021/artifacts.php` as evidence for this claim.

```
1           .text
2  kernel:  ldr rA, [rC, #0]
3           eor rE, rE, rE
4           ldr rB, [rD, #0]
```

(a) MEMORY-BUS/LD-LD: load-after-load.

```
1           .text
2  kernel:  ldr rA, [rC, #0]
3           eor rE, rE, rE
4           str rB, [rD, #0]
```

(b) MEMORY-BUS/LD-ST: store-after-load.

```
1           .text
2  kernel:  str rA, [rC, #0]
3           eor rE, rE, rE
4           ldr rB, [rD, #0]
```

(c) MEMORY-BUS/ST-LD: load-after-store.

```
1           .text
2  kernel:  str rZ, [rA, #0]
3           eor rE, rE ,rE
4           str rZ, [rB, #0]
```

(d) MEMORY-BUS/ST-ST-1: store-after-store, overwrite with zero value.

```
1           .text
2  kernel:  str rA, [rC, #0]
3           eor rE, rE ,rE
4           str rB, [rD, #0]
```

(e) MEMORY-BUS/ST-ST-2: store-after-store, overwrite with security-critical value.

```
1           .text
2  kernel:  str rA, [rD, #0]
3           str rB, [rE, #0]
4           str rC, [rD, #0]
```

(f) MEMORY-BUS/ST-ST-3: store-after-store, with intermediate flush.

Figure 3: Pseudo-code for micro-benchmarks described in Section 5.1.1, i.e., those related to the case study on hidden state in the memory access path.

5.1 Memory: hidden state

This case study focuses on the so-called *memory remnant effect*, as observed, for example, by Papagiannopoulos and Veshchikov [PV17, Section 3.2] who describe it as relating to "leakage originating from consecutive SRAM accesses". In short, it captures the fact consecutive memory accesses may interact even if those accesses involve different architectural state This can be a challenging effect to identify and resolve, because 1) intermediate (e.g., ALU) instructions may not prevent leakage, and thus 2) leakage may occur due to instructions which occur far apart.

5.1.1 Micro-benchmarks

Figure 3 includes pseudo-code for the micro-benchmarks used, which can be described as follows:

1. MEMORY-BUS/LD-LD (Figure 3a): an `ldr` instruction, followed by an intermediate `eor` (i.e., ALU) instruction, followed by an `ldr` instruction, none of which access the same architectural state (i.e., general-purpose registers, nor addresses in memory). The aim is to answer the question *is there Hamming distance leakage between the values loaded*, i.e., *is there hidden state in the memory access path for* `ldr` *instructions (implying a possibility they interact)*?
2. MEMORY-BUS/LD-ST (Figure 3b): as MEMORY-BUS/LD-LD, except with the second `ldr` instruction replaced by a `str` instruction. The aim is to answer the question *is there Hamming distance leakage between the values loaded and stored*, i.e., *is there hidden state in the memory access path for* `ldr` *and* `str` *instructions (implying a possibility they interact)*?
3. MEMORY-BUS/ST-LD (Figure 3c): as MEMORY-BUS/LD-ST, but with the order of `ldr` and `str` instructions swapped.

Table 2: A summary of results stemming from the micro-benchmarks in Figure 3, i.e., cases which explore Hamming distance leakage from combinations of `ldr` and `str` instructions. Note that `AC`, for example, indicates that the Hamming distance between `A` and `C` was leaked.

Device	LD-LD	LD-ST	ST-LD	ST-ST-1	ST-ST-2	ST-ST-3
$\mathsf{ARM}^{\mathsf{N0}}$	AB				AB	
$\mathsf{ARM}^{\mathsf{N1}}$	AB				AB	
$\mathsf{ARM}^{\mathsf{N2}}$	AB	AB	AB		AB	
$\mathsf{ARM}^{\mathsf{N3}}$	AB	AB			AB	
$\mathsf{ARM}^{\mathsf{S0}}$	AB	AB	AB		AB	
$\mathsf{ARM}^{\mathsf{S1}}$	AB	AB	AB		AB	
$\mathsf{ARM}^{\mathsf{S2}}$	AB					
$\mathsf{ARM}^{\mathsf{S3}}$	AB	AB	AB		AB	
$\mathsf{ARM}^{\mathsf{S4}}$	AB				AB	
$\mathsf{ARM}^{\mathsf{S5}}$	AB	AB	AB		AB	
$\mathsf{MB}^{\mathsf{X0}}$	AB	AB	AB		AB	
$\mathsf{MB}^{\mathsf{X1}}$						
$\mathsf{MB}^{\mathsf{X2}}$					AB	
$\mathsf{RV}^{\mathsf{PRV}}$						

4. MEMORY-BUS/ST-ST-1 (Figure 3d): an `str` instruction, followed by an intermediate `eor` (i.e., ALU) instruction, followed by an `str` instruction, none of which access the same architectural state (i.e., general-purpose registers, nor addresses in memory). The aim is to answer the question *if a security-critical value is stored in memory, is the Hamming weight leaked by overwriting it with zero*?
5. MEMORY-BUS/ST-ST-2 (Figure 3e): as MEMORY-BUS/ST-ST-1, but with the `str` instructions storing a non-zero value. The aim is to answer the question *is there Hamming distance leakage between the values stored*, i.e., *is there hidden state in the memory access path for `str` instructions (implying a possibility they interact)*?
6. MEMORY-BUS/ST-ST-3 (Figure 3f): as MEMORY-BUS/ST-ST-2, but with 1) the `str` instructions accessing the same address, and 2) the intermediate instruction replaced with another `str`, which stores a zero value. The aim is to answer the question *does the intermediate `str` instruction flush hidden state in the memory access path*, i.e., *is any Hamming weight leakage due to hidden state, or to the memory access itself.*

5.1.2 Results

Each micro-benchmark in this case study is functionally equivalent across the set of devices used, and, where permitted by the ISA, *identical*. Despite this fact, we observe markedly different leakage behaviour across those targets. For example, from Table 2 one can identify various classes of difference:

1. equivalent instruction sequences executed on different cores that use different ISAs, e.g., Xilinx MicroBlaze versus ARM,
2. identical instruction sequences executed on different cores (from the same vendor) that use the same ISA, e.g., Xilinx MicroBlaze,
3. identical instruction sequences executed on different cores (from different vendors in different SoCs), that use the same ISA, e.g., NXP- versus ST-based ARM Cortex-M3, and
4. identical instruction sequences executed on the same core (from the same vendor in different SoCs), e.g., ST-based ARM Cortex-M3.

Some experiments show very consistent behaviour across *architectures*. For example, all of the ARM cores behave identically for the MEMORY-BUS/LD-LD experiment. This result is widely reported in the literature. However, when looking at interactions between load and store instructions (such as might occur when spilling registers to the stack) separated by an ALU instruction (MEMORY-BUS/LD-ST and MEMORY-BUS/ST-LD) we see very different results not only between different CPU cores, but even between the same CPU core implemented by the same manufacturer but in different devices. For example, loaded and stored values interact in the $\mathsf{ARM}^{\mathsf{S1}}$ and $\mathsf{ARM}^{\mathsf{S3}}$ devices, both manufactured by ST Microelectronics. However, the $\mathsf{ARM}^{\mathsf{S2}}$ and $\mathsf{ARM}^{\mathsf{S4}}$ devices when running exactly the same code, do not leak in the same way, despite the underlying CPU core, and ISA (and hence program binary) being identical. Closer inspection of the data-sheets for these devices reveals that the $\mathsf{ARM}^{\mathsf{S2}}$ and $\mathsf{ARM}^{\mathsf{S4}}$ are high performance variants, with higher maximum operating frequencies.

A similar comparison can be made between the Xilinx MicroBlaze cores, where the 3-stage $\mathsf{MB}^{\mathsf{X0}}$ had clear interactions between loaded and stored values, but in the longer pipelined $\mathsf{MB}^{\mathsf{X1}}$ and $\mathsf{MB}^{\mathsf{X2}}$, the loaded and stored values do not interact.

For the store-store experiments, we see no Hamming distance leakage between the MEMORY-BUS/ST-ST-1 and MEMORY-BUS/ST-ST-3 variants. From this, we conclude that (for the number of traces we collected), leakage originates exclusively from *registers* in either the CPU core or the memory hierarchy, not from *inside the SRAM*. Indeed, only the MEMORY-BUS/ST-ST-2 experiment causes Hamming distance leakage between the values being stored. This suggests that in some cases, so long as there is an intervening store to another address (thus flushing the store data-path), values with the same mask may be safely overwritten in memory without causing leakage. This avoids the need for expensive countermeasures (some of which are well described in [SSB+20]), but we urge developers to verify that this holds on their own platforms themselves.

5.1.3 Discussion

In explaining some of the differences between targets with the same core but different leakage behaviours ($\mathsf{ARM}^{\mathsf{S2}}$ and $\mathsf{ARM}^{\mathsf{S4}}$), we hypothesise that the need to meet tighter timing requirements drove various design decisions regarding the core and memory interconnect, which apparently have led to totally separate load and store data-paths. Regardless of the actual reason for this difference in behaviour, from a leakage perspective, this is a critical difference between cores which must be accounted for.

Based on our observations, we extend the notion of adjacent instructions to describe different types of instruction:

Definition 10. Two distinct instructions can be described as

- **program-adjacent** if they are executed in consecutive execution cycles (i.e., they appear consecutively in program order),
- **memory-adjacent** if they are both load or store instructions, and no intermediate load or store instructions are executed between them,
- **load-adjacent** if they are both load instructions, and no intermediate load instructions are executed between them,
- **store-adjacent** if they are both store instructions, and no intermediate store instructions are executed between them.

It should be clear that *store-adjacent* and *load-adjacent* instructions are mutually exclusive, and are both subsets of *memory-adjacent* instructions. We can now say that for some cores, *memory-adjacent* instructions will leak the Hamming distance between values written or read from memory, e.g. the $\mathsf{ARM}^{\mathsf{N2}}$. However, in the case of the $\mathsf{ARM}^{\mathsf{S2}}$, only *load-adjacent* and *store-adjacent* instructions will leak as such. We believe that tagging instructions

```
1           .data
2  X:       .byte 0,0,0,A,0,0,0
3
4           .text
5  kernel:  ldr  rX, =X
6           ldrb rA, [rX, I]
```

(a) MEMORY-BUS/WIDTH-LD-BYTE: load from byte array.

```
1           .data
2  X:       .byte 0,0,0,A,0,0,0
3
4           .text
5  kernel:  ldr  rX, =X
6           strb rZ, [rX, I]
```

(b) MEMORY-BUS/WIDTH-ST-BYTE: store zero into byte array.

```
1           .data
2  X:       .hword 0,A,0,0
3
4           .text
5  kernel:  ldr  rX, =X
6           ldrh rA, [rX, I]
```

(c) MEMORY-BUS/WIDTH-LD-HALF: load from half-word array.

```
1           .data
2  X:       .hword 0,A,0,0
3
4           .text
5  kernel:  ldr  rX, =X
6           strh rZ, [rX, I]
```

(d) MEMORY-BUS/WIDTH-ST-HALF: store zero into half-word array.

Figure 4: Pseudo-code for micro-benchmarks described in Section 5.2.1, i.e., those related to the case study on width of the memory access path. Note that for both of these experiments `I` is a parameter rather than a variable: the micro-benchmark is executed 8 separate times (i.e., for $\texttt{I} \in \{0, 1, \ldots, 7\}$) for the byte cases, and 4 separate times (i.e., for $\texttt{I} \in \{0, 1, \ldots, 3\}$) for the half-word cases.

as such will make formal modelling of memory hierarchy related leakage much easier to reason about. There are also obvious similarities between memory consistency models and ordering constraints or fences in various ISAs. We hope that these rules can be easily added to static program checkers, and be used as extra information when looking for interactions between variables.

Based on the leakage taxonomy developed in Section 3, we classify these effects as *sequential, inter-instruction micro-architectural* leakage. Per target device, we can now explain concisely whether, e.g. load-adjacent instructions suffer from inter-instruction sequential micro-architectural leakage, or not.

For the ARM devices, we believe that the range of observed behaviours are much less surprising given a thorough reading of the AMBA-AHB bus standard [ARMa]. This bus standard is used by all of the ARM micro-controllers in the study. It is *explicitly* described as a pipelined bus architecture. Hence, the existence of *some* micro-architectural state should be expected. Observing [ARMa, Figure 1-1], the AHB block diagram, it is clear that there are *opportunities* for registers to be placed in several places, with only a finite number of sensible design choices for engineers to follow. This is shown in our results, because although different cores from the same manufacturer do differ, given that they differ, they differ consistently. This suggests it is possible to separate the leakage modelling of CPUs from the modelling of the memory interconnect.

5.2 Memory: data bus widths

This case study focuses on the interaction between data-type width (e.g., `uint8_t`, `uint16_t`, and `uint32_t`) and memory bus width. The central question is *how does the memory sub-system satisfy an n-byte memory access*, e.g., *are exactly n bytes loaded, or are $m > n$ bytes loaded and then $m - n$ discarded*? The answer is important, because there are different approaches possible and each (potentially) has a different implication

Table 3: A summary of results stemming from the micro-benchmarks in Figure 4a and Figure 4c i.e., cases which explore Hamming distance leakage from 8-bit, byte load (`ldrb`) instructions and 16-bit, half-word load (`ldrh`) instructions respectively. Note that each numbered column refers to the offset `I` into the array `X`, with a bold number used to highlight the offset for `A`; a ✔in the i-th column indicates that the Hamming weight of `A` was leaked.

Device	`ldrb`								`ldrh`			
	0	1	2	**3**	4	5	6	7	0	**1**	2	3
ARM^{N0}				✔	✔	✔	✔			✔	✔	
ARM^{N1}				✔	✔	✔	✔			✔	✔	
ARM^{N2}				✔	✔	✔	✔			✔	✔	
ARM^{N3}				✔	✔	✔	✔			✔	✔	
ARM^{S0}				✔	✔	✔	✔			✔	✔	
ARM^{S1}				✔	✔	✔	✔			✔	✔	
ARM^{S2}				✔	✔	✔	✔			✔	✔	
ARM^{S3}				✔	✔	✔	✔			✔	✔	
ARM^{S4}				✔	✔	✔	✔			✔	✔	
ARM^{S5}				✔	✔	✔	✔			✔	✔	
MB^{X0}				✔	✔	✔	✔			✔	✔	
MB^{X1}				✔	✔		✔	✔		✔		
MB^{X2}				✔	✔	✔				✔		
RV^{PRV}				✔	✔	✔	✔			✔	✔	

Table 4: A summary of results stemming from the micro-benchmarks in Figure 4b and Figure 4d, i.e., cases which explore Hamming distance leakage from 8-bit, byte store (`strb`) instructions and 16-bit, half-word store (`strh`) instructions respectively. Note that each numbered column refers to the offset `I` into the array `X`, with a bold number used to highlight the offset for `A`; a ✔in the i-th column indicates that the Hamming weight of `A` was leaked.

Device	`strb`								`strh`			
	0	1	2	**3**	4	5	6	7	0	**1**	2	3
ARM^{N0}												
ARM^{N1}												
ARM^{N2}												
ARM^{N3}												
ARM^{S0}				✔		✔	✔				✔	
ARM^{S1}				✔		✔	✔				✔	
ARM^{S2}												
ARM^{S3}				✔		✔	✔				✔	
ARM^{S4}												
ARM^{S5}				✔			✔				✔	
MB^{X0}				✔		✔						
MB^{X1}												
MB^{X2}				✔								
RV^{PRV}				✔								

for associated leakage. Shelton et al. [SSB+20, Section IV.E] note that such leakage is evident on an ST-based ARM Cortex-M0 and in the ELMO [MWO16] power model: based on their observations as a starting point, our aim is to then 1) evaluate whether the same leakage effect is evident on other devices, and 2) explain the underlying leakage source(s).

5.2.1 Micro-benchmarks

Figure 4 includes pseudo-code for the micro-benchmarks used; both cases uses an 8-byte, word-aligned array `X`. Each element of `X` is initialised to zero, bar one which is instead initialised to a security-critical value `A`.

For the load (Figure 4a) case, the micro-benchmark loads an 8-bit (`ldrb`) or 16-bit (`ldrh`) value from a given offset `I` within `X`. If only those bytes required are accessed, we expect leakage only for an offset which implies access to the security-critical value; if leakage is observed at other offsets, we infer that bytes *other* than those required are also accessed. For the store (Figure 4b) case, the micro-benchmark stores an 8-bit (`strb`) or 16-bit (`strh`) zero value at a given offset `I` within `X`. Again, the presence (resp. absence) of leakage for a given offset allows us to infer which bytes are accessed.

5.2.2 Results

Table 3 shows consistent sub-word load behaviour for all of the ARM cores in the study. Again, this effect has been noted in the literature, and we are unsurprised to confirm it across multiple devices and manufacturers. We note however the mixed results for MicroBlaze devices.

For stores, Table 4 shows differing behaviour between different cores, and even the same core implemented in different devices when adjacent bytes in memory are overwritten with zeros. We note that results for $\mathsf{ARM}^{\mathsf{S1}}$, $\mathsf{ARM}^{\mathsf{S3}}$, $\mathsf{ARM}^{\mathsf{S2}}$, and $\mathsf{ARM}^{\mathsf{S4}}$ appear to mirror the results observed in Table 2. Comparatively few devices showed any leakage in this case and it is not immediately clear why Hamming weight leakage should be visible for a byte *in memory* when the word is being written too, but that exact byte is not changing value.

For the store experiments, we also note the clear divide between ARM based targets manufactured by ST-Microelectronics, which show various sources of leakage, and those built by NXP, which show no leakage at all for the same experiments. Recall again that where possible, the binary code running on each device is *identical*, yet still yields very different leakage behaviour.

5.2.3 Discussion

For the differing results of the MicroBlaze devices, we hypothesise this is a side effect of how FPGA BRAMs are grouped together to form different word sizes. The Xilinx 7-Series FPGA BRAMS have port widths[10] of 18, 36, and 72 bits [Xilb][Chapter 1]. Hence two BRAMs must be grouped together to create a 32-bit word. There is no guarantee from the Xilinx IP descriptions, or the synthesis tools, on how sub-word accesses to grouped BRAMs will behave. This is evident from the somewhat inconsistent results for the MicroBlaze cores when loading bytes and half-words. We ran all of the experiments through the Local Memory Bus (LMB) [Xila][Chapter 3, P155] used by the MicroBlaze core "primarily to access on-chip block RAM."

For the stores case, we hypothesise that leakage behaviour here is heavily dependant on the behaviour of the hardened RAMs. For example, it is possible that even when only a single byte is being explicitly accessed, the entire word is implicitly accessed (i.e. it's value is read out of the cell array) within the RAM, whether it is for a load or a store. We

[10] Note that for some power-of-two n, use of an $(n + m)$-bit (e.g., 18) port width is intended to support n bits (e.g., $2^4 = 16$) of data plus m bits (e.g., 2) of error correction meta-data in hardware.

```
1           .data
2  X:       .byte A,B,C,D,E,F,G
3
4           .text
5  kernel:  ldr  rX, =X
6           ldrb rA, [rX, #0]
7           ldrb rB, [rX, #1]
8           ldrb rD, [rX, #2]
9           ldrb rC, [rX, #3]
10          ldrb rD, [rX, #4]
11          ldrb rE, [rX, #5]
12          ldrb rF, [rX, #6]
13          ldrb rG, [rX, #7]
```

(a) MEMORY-BUS/SEQ-LD: sequential load bytes from array.

```
1           .data
2  X:       .byte 0,0,0,0,0,0,0
3
4           .text
5  kernel:  ldr  rX, =X
6           strb rA, [rX, #0]
7           strb rB, [rX, #1]
8           strb rD, [rX, #2]
9           strb rC, [rX, #3]
10          strb rD, [rX, #4]
11          strb rE, [rX, #5]
12          strb rF, [rX, #6]
13          strb rG, [rX, #7]
```

(b) MEMORY-BUS/SEQ-ST: sequential store bytes into array.

Figure 5: Pseudo-code for micro-benchmarks described in Section 5.3.1, i.e., those related to the case study on sequential use of the memory access path.

note that we *never* see Hamming weight leakage when the value in memory is explicitly overwritten with zeros. This might imply forwarding behaviour inside the RAMs, where the RAM *always* reads the word being accessed (regardless of whether it is a load or store), and where the word being read is being written in the same cycle, the written value is forwarded to the RAMs read data register.

As to the difference in stored value leakage behaviour between manufacturers, again we hypothesise this is down to differences in manufacturing approach for the hardened RAMs. We do not believe it is reasonable to conclude that NXP devices are on the whole *less leaky*, but note that this starkly illustrates how families of devices can behave very differently under leakage analysis.

Again, we classify these effects as load/store-adjacent, inter-instruction, sequential micro-architectural.

5.3 Memory: sequential accesses

This case study focuses on the behaviour of sequential accesses (i.e., loads or stores) to memory. Such an access patter can arise, for example, when the nature data-type is `uint8_t`, or when accessing regions of (e.g., an array in) memory with unknown alignment. The central question is *if one loads (resp. stores) bytes from (resp. into) different addresses in memory into (resp. from) different architectural registers, is it possible they interact*? From an architectural perspective the answer should clearly be no, and so any leakage must stem from (micro-architectural) state within the memory sub-system.

5.3.1 Micro-benchmarks

Figure 5 includes pseudo-code for the micro-benchmarks used; both cases uses an 8-byte, word-aligned array `X`.

For the load (Figure 5a) case, the micro-benchmark loads a sequence of bytes from different, consecutive addresses in memory into different architectural registers: note that `ldr rA, [rX, #0]` should be read as "load the 0-th element of `X`, i.e., the variable `A`, into register `rA`". Any Hamming distance leakage between the bytes loaded allows us to infer the presence of shared state, e.g., within 1) the memory sub-system, and/or 2) the pipeline stages used by the core (i.e., between a value being received from memory by the core, and

Table 5: A summary of results stemming from the micro-benchmarks in Figure 5a, i.e., cases which explore Hamming distance leakage from 8-bit, byte load (`ldrb`) instructions using different, consecutive addresses and different architectural registers; the results focus on ST-based devices only. Note that a ✔in the i-th column and j-th row indicates that the Hamming distance between the i-th and j-th bytes loaded was leaked.

	0	1	2	3	4	5	6	7
0		✔	✔	✔	✔			
1						✔		
2							✔	
3								✔
4						✔		
5								
6								

(a) $\mathsf{ARM}^{\mathsf{S0}}$.

	0	1	2	3	4	5	6	7
0		✔		✔	✔			
1			✔			✔		
2				✔			✔	
3								✔
4						✔		
5							✔	
6								✔

(b) $\mathsf{ARM}^{\mathsf{S1}}$.

	0	1	2	3	4	5	6	7
0		✔		✔	✔			
1						✔		
2				✔			✔	
3								✔
4						✔		
5					✔			
6								

(c) $\mathsf{ARM}^{\mathsf{S2}}$.

	0	1	2	3	4	5	6	7
0		✔		✔	✔			
1			✔			✔		
2				✔			✔	
3								✔
4						✔		
5							✔	
6								

(d) $\mathsf{ARM}^{\mathsf{S3}}$.

	0	1	2	3	4	5	6	7
0		✔		✔	✔			
1			✔			✔		
2				✔			✔	
3					✔			✔
4						✔		
5								
6								

(e) $\mathsf{ARM}^{\mathsf{S4}}$.

	0	1	2	3	4	5	6	7
0		✔	✔	✔	✔			
1						✔		
2							✔	
3								✔
4						✔		
5							✔	
6								

(f) $\mathsf{ARM}^{\mathsf{S5}}$.

being written into a GPR). For the load (Figure 5a) case, the micro-benchmark stores a sequence of bytes from different architectural registers into different, consecutive addresses in memory: note that `str A, [rX, #0]` should be read as "store register `rA`, i.e., the variable `A`, into the 0-th element of `X`". Any Hamming distance leakage allows similar inferences to the load case, but could *also* indicate that additional pipeline register(s) exist between the register file and the memory write-port.

5.3.2 Results

The results for a representative subset of the MEMORY-BUS/SEQ-LD experiment can be found in Table 5. We can see two main effects, namely 1) bytes within a word *can* interact but *may* not, and 2) the i-th byte of the array will *often* interact with the $(i + 4)$-th byte of the array.

5.3.3 Discussion

We believe the first effect is due to the necessary multiplexing to select any byte of a loaded memory word, and to place it in the least significant byte of an architectural register. Clearly this multiplexing does not always cause visible leakage for the number of traces used in these experiments.

For the second effect, we believe this is because (as established in prior experiments), even when a byte of memory is requested, in reality, an entire word is loaded. Hence, when we load the first four bytes of the array, we are really repeatedly loading the entire first word. When we load byte 4 of the array, an entire new word is loaded into some micro-architectural state, and we see Hamming distance leakage between all corresponding bytes. Again, none of the instructions shared architectural destination registers, meaning all of the interactions are due to the micro-architectural leakage.

We believe the inconsistency of our results is down to two major factors. First, that if we collected more traces we would see more consistent results. Second, that for a given number of traces, not all intra-cycle leakage (e.g. in multiplexer trees) can manifest, due to subtle differences in the final post-layout silicon design.

For this particular effect, we note the existence of two distinct sources of micro-architectural leakage according to our classification. First, we see inter-instruction sequential micro-architectural leakage, as we have with other memory bus experiments. However, we also see intra-instruction, glitching combinatorial micro-architectural leakage, where bytes of the loaded word interact with each other through the multiplexers, which select which byte of the word is written back to a general purpose register.

5.4 Pipeline register overwrites

Shelton et al. [SSB+20] focus on an ST-based ARM Cortex-M0, which has a 3-stage pipeline and thus 2 sets of pipeline registers. We extend this remit, applying a similar methodology (i.e., that of using a set of micro-benchmarks) to devices which have more diverse micro-architectures and hence different, more complex pipeline structures. In doing so, we demonstrate how to answer an important question, namely *in a scalar pipeline, do consecutive instructions that use different destination registers cause Hamming distance leakage between instruction results*? From an architectural perspective the answer should clearly be no, and so any leakage must stem from (micro-architectural) state within the pipeline structure.

The information generated by this methodology is useful, for example, to 1) reverse engineer details of the (unknown) pipeline structure, and, therefore, 2) model how in-flight instructions proceed through each stage of execution, e.g., whether and how they, and associated intermediate values, interact with each other. Of course, with white-box access

```
1           .text
2  kernel:  eor rA, rA, rB
3           eor rC, rC, rD
```

(a) PIPELINE/EOR-EOR: `eor-eor` interaction.

```
1           .text
2  kernel:  eor rA, rA, rB
3           add rC, rC, rD
```

(b) PIPELINE/EOR-ADD: `eor-add` interaction.

```
1           .text
2  kernel:  eor rA, rA, rB
3           lsl rC, rD, #8
```

(c) PIPELINE/EOR-LSL: `eor-lsl` interaction.

```
1           .text
2  kernel:  eor rA, rA, rB
3           ror rC, rD, #8
```

(d) PIPELINE/EOR-ROR: `eor-ror` interaction.

```
1           .text
2  kernel:  eor rA, rA, rB
3           ldr rC, [rD, #0]
```

(e) PIPELINE/EOR-LDR: `eor-ldr` interaction.

```
1           .text
2  kernel:  eor rA, rA, rB
3           str rC, [rD, #0]
```

(f) PIPELINE/EOR-STR: `eor-str` interaction.

```
1           .text
2  kernel:  eor rA, rA, rB
3           nop
4           eor rC, rC, rD
```

(g) PIPELINE/NOP-EOR: `eor-eor` interaction, with intermediate `nop`.

Figure 6: Pseudo-code for micro-benchmarks described in Section 5.4.1, i.e., those related to the case study on pipeline register use.

Table 6: A summary of results stemming from the micro-benchmarks in Figure 6, i.e., during execution of a given instruction pair, was there Hamming distance leakage; N/A indicates the instruction pair (e.g., due to use of `ror`) is not supported by the ISA associated with that device.

Device	EOR-EOR	EOR-ADD	EOR-LSL	EOR-ROR	EOR-LDR	EOR-STR	EOR-NOP
ARM^{N0}					✔	✔	
ARM^{N1}	✔				✔	✔	
ARM^{N2}	✔		✔		✔	✔	
ARM^{N3}			✔		✔	✔	
ARM^{S0}	✔	✔	✔		✔	✔	
ARM^{S1}	✔		✔		✔	✔	
ARM^{S2}	✔						
ARM^{S3}			✔		✔	✔	
ARM^{S4}					✔	✔	
ARM^{S5}	✔	✔					
MB^{X0}	✔			N/A	✔	✔	
MB^{X1}	✔	✔	✔	N/A	✔	✔	
MB^{X2}	✔	✔	✔	N/A	✔	✔	
RV^{PRV}				N/A			

Table 7: A summary of results stemming from the micro-benchmarks in Figure 6, i.e., during execution of a given instruction pair, which operands caused Hamming distance leakage; N/A indicates the instruction pair (e.g., due to use of `ror`) is not supported by the ISA associated with that device. Note that `AC`, for example, indicates that the Hamming distance between variables `A` and `C` was leaked. The Hamming distance between variables `A` and `B` was leaked in *all* cases, so have been omitted.

Device	EOR-EOR	EOR-ADD	EOR-LSL	EOR-ROR	EOR-LDR	EOR-STR	EOR-NOP
ARM^{N0}	BD	BD	AC	AB		BC	
ARM^{N1}	AC, AD, BD	AC, BD	AD, BD	AC		BC	
ARM^{N2}	AC, BD	AC, BD	AD, BD	AC		BC	AC, BD
ARM^{N3}	BD	BD				BC	
ARM^{S0}	AC, BD	AC, BD	AD, BD	AC		BC	
ARM^{S1}	AC, BD	AC, BD, CD	AD, BD	AC		BC	AC, BD
ARM^{S2}	AC, BD	AC, BD	AD, BD			BC	BD
ARM^{S3}	AC, AD	AC, AD, BD	AD, BD	AC, BC		BC	AC, AD
ARM^{S4}	AC, AD, BD	AC, AD, BD	AD, BD	BC		BC	
ARM^{S5}	AC, BD	AC, BD	AD	AC		BC	
MB^{X0}	AC, BD	AC, BD	AD	N/A			
MB^{X1}	AC, BD	AC, BD	AC, AD	N/A	AD	AC	
MB^{X2}	AC, BD	AC, BD	AC, AD	N/A	AD	AC	
RV^{PRV}	AC, BD	AC, BD	AD	N/A		BC	

to the micro-architectural design, e.g., the HDL, per the MAPS simulator of Le Corre et al. [CGD18] for ARM Cortex-M3, one can obtain similar information more directly. However, we argue that a grey-box approach is more scalable: it can cater for cases when said design is unknown and/or inaccessible.

5.4.1 Micro-benchmarks

Figure 6 includes pseudo-code for the micro-benchmarks used, which can be described as follows:

1. PIPELINE/EOR-EOR (Figure 6a): an `eor` instruction followed by another `eor` instruction, neither of which access the same architectural state (i.e., general-purpose registers). The aim is to answer the questions *is there Hamming distance leakage between the operands*, and *is there Hamming distance leakage between the results*?
2. PIPELINE/EOR-ADD (Figure 6b): as PIPELINE/EOR-EOR, except with the second `eor` instruction replaced by an `add` instruction.
3. PIPELINE/EOR-LSL (Figure 6c): as PIPELINE/EOR-EOR, except with the second `eor` instruction replaced by an `lsl` instruction (i.e., a left-shift). The aim is to answer the additional question *which (potential) pipeline register stores the immediate value involved*?
4. PIPELINE/EOR-ROR (Figure 6d): as PIPELINE/EOR-LSL, except with the `lsl` instruction replaced by a `ror` instruction (i.e., a right-rotate).
5. PIPELINE/EOR-LDR (Figure 6e): as PIPELINE/EOR-EOR, except with the second `eor` instruction replaced by an `ldr` instruction. The aim is to answer the question *do results produced by the ALU interact with values loaded from memory*?
6. PIPELINE/EOR-STR (Figure 6f): as PIPELINE/EOR-LDR, except with the `ldr` instruction replaced by an `str` instruction. The aim is to answer the question *do results produced by the ALU interact with values stored into memory*?

7. PIPELINE/NOP-EOR (Figure 6g): as PIPELINE/EOR-EOR, except with an intermediate `eor` (i.e., ALU) instruction separating the `eor` instructions. The aim is to answer the question *does the intermediate* ***`nop`*** *instruction act as an effective barrier*?

5.4.2 Results

Table 6 shows which devices leak the Hamming distance between the *results* of adjacent instructions. Interactions between certain pairs of instructions consistently leak the Hamming distance between instruction results across all devices (e.g. the `eor-ldr`) pair. We were surprised not to see more clear evidence of Hamming distance leakage between consecutive ALU type instructions; particularly the `add`, `lsl` and `ror` instructions. We offer two possible explanations for this. 1) that the interactions are too weak to detect for the number of traces we use, 2) that the results of the instructions make poor targets for Hamming distance analysis anyway. It is also possible that some interactions are purely combinatorial glitching leakage, which we would expect to be much weaker than sequential leakage, which involves state elements.

Table 7 shows which instruction *operands* of the experiments in Figure 6 leak their *Hamming distance*. We find generally very consistent behaviour for interactions between the first and second operand registers of consecutive instructions. As has been reported in the literature we find that two adjacent instructions will usually leak the Hamming distance between their respective `rs1` and `rs2` operands. This was a key finding of [PV17] and [CGD18], and we have replicated their results across many other CPU cores. This is is visible in Table 7 as `AC` and `BD` interactions.

While some very consistent behaviour is easy to see in Table 7, we note two unexpected effects. First, that for some cores, there are unexpected Hamming distance interactions between operands which do not obviously interact. For example, the ARM Cortex-M4 base devices leak the Hamming distance of `rs1` of the `eor`, and `rs2` of the `add`. We were also surprised to consistently see Hamming distance leakage between *both* register operands of the `eor`, and the register operand of the right shift by immediate instruction. This occurred consistently in the ARM Cortex-M3 and Cortex-M4 devices, and the larger MicroBlaze cores.

5.4.3 Discussion

We were able to discern, in the case of instructions with a register and immediate operand (like the load / store word instructions) exactly which operands from the previous instruction interact with the register operand, and which with the immediate operand. For example, all of the ARM cores consistently leak the Hamming distance between `rs2` of an `eor` instruction followed by `rs1` of a store word instruction. It is also useful to see that in the larger MicroBlaze cores, it is actually `rs1` of the `eor` instruction which collides with the store word instruction. Knowing which operand register of the `eor` instruction interacts with the store instruction is essential for leakage modelling.

Comparing Table 6 and Table 7, we can see that generally speaking, Hamming distance leakage between consecutive instruction *operands* is much more consistently detectable than Hamming distance leakage between the *results* of consecutive instructions. Given that the result of one instruction will almost always become the operand of another, we believe that operands are inevitably more effective attack targets. That said, we note that leakage between the results of consecutive instructions, even when the instructions share no architectural registers between them, is likely to be an unexpected source of leakage for some implementations. Even in cores with very short pipelines.

In cores with longer pipelines ($\mathsf{MB}^{\mathsf{X1}}$, $\mathsf{MB}^{\mathsf{X2}}$) we see consistent leakage between the results of adjacent instructions. This is expected, as there are likely multiple pipeline stages between the result being calculated and the write-back of the result to the register

file. In general, we note the very different leakage behaviour across all of the devices (even devices with the same CPU core) mirrors our results for the memory bus related leakage experiments. This would suggest that leakage models, even when they only look to model intra-core leakage effects, cannot be transferable between devices; even if the devices are based on the same CPU core. Indeed, it raises the possibility that a device with multiple identical CPU cores[11], could find that core 0 leaks differently to core 1, despite being implemented in the same piece of silicon.

We note the differing results for whether `nop` acts as a barrier between ALU type instruction operands. Even two ARM Cortex-M4 based devices ($\mathsf{ARM}^{\mathsf{S3}}$ and $\mathsf{ARM}^{\mathsf{S4}}$) behave differently. In the former a `nop` does not alter the pipeline registers which store instruction operands, causing Hamming distance leakage. In the later, the `nop` *does* act as a leakage barrier. Borrowing the notions introduced in Section 5.1, we can say that even though the `eor` instructions are not *program-adjacent*, they still interact. This behaviour, while not exactly expected, is certainly understandable given that the ARMv7-M architecture only guarantees the behaviour of a `nop` in terms of memory alignment. Researchers who use `nop` as a makeshift leakage barrier should beware that this is *not portable*. This has implications for some of the experiments in [SSB+20] (e.g. Listings 1, 2) where `nop` is used to separate instructions in time. We do not believe this had an adverse effect on the experiments in [SSB+20], as our own results confirm `nop` is a reasonable leakage barrier in the ARM Cortex-M0 core. However, this cannot be expected on other, e.g., ARM Cortex-M4 cores. This lends some evidence to the utility of dedicated leakage barrier instructions, such as FENL [GMPP20], which guarantee some level of leakage barrier like behaviour.

Again, it is important to note the differences between different devices (even devices with the same underlying CPU core). That such basic instruction sequences leak visibly on some devices and not on others points to a real challenge for formal and statistical leakage modelling approaches.

In terms of classifying these sources of leakage, we find that program-adjacent, inter-instruction, sequential micro-architectural is the dominant source. This is expected, since we specifically went searching for leakage related to pipeline registers.

5.5 Control-flow instructions

This case study focuses on a relatively complex case for pipelined instruction execution, namely that of conditional and unconditional changes in control-flow. Note that we use the terms branch and jump to refer to the conditional and unconditional cases respectively, mirroring similar terminology used, e.g., in RISC-V and MicroBlaze. The central question is *how does a change in control-flow effect micro-architectural state, such as pipeline registers*, and so, e.g., *does a change in control-flow prevent interaction between instructions before and after it*?

5.5.1 Micro-benchmarks

Figure 7 includes pseudo-code for the micro-benchmarks used, which can be described as follows:

1. PIPELINE/BRANCH-PRE (Figure 7a): two `eor` instructions, separated in terms of their execution by a `beq` instruction (i.e., a conditional branch). The aim is to answer the questions *is there Hamming distance leakage between the operands*, and *is there Hamming distance leakage between the results*?
2. PIPELINE/BRANCH-POST (Figure 7b): as PIPELINE/BRANCH-PRE, except with the first `eor` instruction moved immediately *after* the `beq` instruction rather than before it. Note

[11] For example, some Xilinx FPGAs include two or more identical hardened ARM CPU cores.

```
1           .text
2  kernel:  cmp rE, rE
3           eor rA, rB
4           beq branch
5           .rept 10
6           eor rZ, rZ
7           .endr
8  branch:  eor rC, rD
```

(a) PIPELINE/BRANCH-PRE: two `eor` instructions operating on security-critical values, separated by a conditional branch instruction.

```
1           .text
2  kernel:  cmp rE, rE
3           beq branch
4           eor rA, rB
5           .rept 10
6           eor rZ, rZ
7           .endr
8  branch:  eor rC, rD
```

(b) PIPELINE/BRANCH-POST: two `eor` instructions operating on security-critical values; only the second is executed, as the result of a conditional branch instruction.

```
1           .text
2  kernel:  eor rA, rB
3           b   branch
4           .rept 10
5           eor rZ, rZ
6           .endr
7  branch:  eor rC, rD
```

(c) PIPELINE/JUMP-PRE: two `eor` instructions operating on security-critical values, separated by an unconditional branch (i.e., jump) instruction.

```
1           .text
2  kernel:  b   branch
3           eor rA, rB
4           .rept 10
5           eor rZ, rZ
6           .endr
7  branch:  eor rC, rD
```

(d) PIPELINE/JUMP-POST: two `eor` instructions operating on security-critical values; only the second is executed, as the result of an unconditional branch (i.e., jump) instruction.

Figure 7: Pseudo-code for micro-benchmarks described in Section 5.4.1, i.e., those related to the case study on conditional and unconditional changes in control-flow.

that if the branch is not taken, said `eor` instruction should not be executed therefore.

3. PIPELINE/JUMP-PRE (Figure 7c): as PIPELINE/BRANCH-PRE, except with the `beq` instruction replaced with a `b` instruction (i.e., an unconditional branch, or jump) and the associated `cmp` instruction removed.
4. PIPELINE/JUMP-POST (Figure 7d): as PIPELINE/BRANCH-POST, except with the `beq` instruction replaced with a `b` instruction (i.e., an unconditional branch, or jump). and the associated `cmp` instruction removed.

5.5.2 Results

There are three broad outcomes from analysing these sequences.

1. There is no interaction between the ALU instructions, whether they are separated by a branch/jump or not.
2. The ALU instructions interact if the branch/jump instruction is absent, but do not interact if it is present. This tells us that any state which the ALU instruction interacts with is somehow cleared or overwritten by the branch/jump instruction. The branch/jump can then be thought of as a weak *leakage barrier* which prevents values manipulated in one loop iteration bleeding through the pipeline into the next iteration.
3. The ALU instructions always interact, regardless of whether a branch/jump is placed between them. In this case, extra care must be taken with masking schemes employing loops, as the "loop branch" instruction cannot be relied upon to stop micro-architectural state from one loop iteration bleeding into the next.

Table 8: A summary of results stemming from the micro-benchmarks in Figure 7, i.e., cases which explore the impact a change in control-flow has on leakage. Note that `AC`, for example, indicates that the Hamming distance between `A` and `C` was leaked; `R` indicates that the Hamming distance between results produced by the two `eor` instructions was leaked. The Hamming distance between variables `A` and `B` and variables `C` and `D` was leaked in *all* cases, so have been omitted.

Device	BRANCH-PRE	BRANCH-POST	JUMP-PRE	JUMP-POST
$\mathsf{ARM}^{\mathsf{N0}}$			R, BD	
$\mathsf{ARM}^{\mathsf{N1}}$			BD	
$\mathsf{ARM}^{\mathsf{N2}}$			BD	
$\mathsf{ARM}^{\mathsf{N3}}$			BD	
$\mathsf{ARM}^{\mathsf{S0}}$			BD	
$\mathsf{ARM}^{\mathsf{S1}}$			BD	
$\mathsf{ARM}^{\mathsf{S2}}$			BD	
$\mathsf{ARM}^{\mathsf{S3}}$		BD	BD	BD
$\mathsf{ARM}^{\mathsf{S4}}$			BD	
$\mathsf{ARM}^{\mathsf{S5}}$			BD	
$\mathsf{MB}^{\mathsf{X0}}$		R, AC, BD		R, AC, BD
$\mathsf{MB}^{\mathsf{X1}}$		R, AC, BD		R, AC, BD
$\mathsf{MB}^{\mathsf{X2}}$		AC, BD	BC	BD
$\mathsf{RV}^{\mathsf{PRV}}$			AD, BD	

This approach is similar to the *dominating instruction* approach taken in [SSB+20] to identify whether instructions share micro-architectural state.

From the results in Table 8, we can see that while no instructions leak in the BRANCH-PRE case, we get scattered Hamming distance leakage in some cases for the other experiments.

For the MicroBlaze based targets ($\mathsf{MB}^{\mathsf{X0}}$, $\mathsf{MB}^{\mathsf{X1}}$, $\mathsf{MB}^{\mathsf{X2}}$) We see leakage between the operands and results of XOR instructions *following* the control-flow change, even when the first XOR instruction *is not executed*.

On all of the ARM cores, we find leakage between one set of operands in the JUMP-PRE case. However, we note that the $\mathsf{ARM}^{\mathsf{S3}}$ also leaks the Hamming distance of one set of operands in the same way, even though ARM has no delay-slot mechanism in the architecture. We investigate this further in Section 5.6.

5.5.3 Discussion

In the case of the MicroBlaze, we believe the leakage is explained by referring to the MicroBlaze Architecture and its use of *branch delay slots*. This is a technique where the instruction immediately following a branch or jump can be executed as a way of hiding the latency of re-filling the pipeline after a control-flow change. MicroBlaze has variants of branch and jump instructions with and without delay slots, and our experiments used the non-delay-slot variants. We believe that on a *micro-architectural* level, the delay-slot instruction is *always* executed, but its result is only thrown away at the last pipeline stage. This is clearly the case for the $\mathsf{MB}^{\mathsf{X0}}$ and $\mathsf{MB}^{\mathsf{X1}}$ cores, which leak operands and results. The $\mathsf{MB}^{\mathsf{X2}}$ does not leak its result, suggesting the delay-slot instruction is killed before its result is computed.

For the ARM JUMP-PRE case, we suggest this is caused by the jump instruction only over-writing a single pipeline register, and leaving the other unmodified from the instruction preceding the jump. The unmodified value will then collide with whatever the next corresponding instruction operand is.

We note the lack of attention paid to branch and jump instructions with respect to

leakage behaviours in the literature. We believe this sort of analysis is important because as we have shown, just because control-flow instructions don't normally read secret or sensitive variables, doesn't mean they can't cause or allow them to interact. Knowing if a "loop back" instruction sequence clears pipeline registers can reduce the need to explicitly clear certain micro-architectural state, or show that it is necessary.

We classify this effect as inter-instruction, sequential micro-architectural leakage between program-non-adjacent instructions. The non-adjacency of instructions is an important distinction, since it is not the branch or jump instructions which *cause* the interaction, rather, they *fail to prevent* interactions between other instructions.

5.6 Speculative execution in short pipelines

Despite being associated with larger, out-of-order and super-scalar cores with deep pipelines, speculative execution still takes place even in very shallow pipelines. In an n stage scalar pipeline, there are in principle n instructions in flight. Depending on the stages where control-flow changes occur, several instructions which are never executed from an *architectural* point of view, still enter the pipeline and interact with architectural and micro-architectural state elements. Further, different types of control-flow change can occur at different points in the pipeline. For example, a conditional branch can only occur after reading and comparing some architectural state, in a decode or execute pipeline stage. An unconditional jump however can occur at the decode or fetch stage, since the destination address can be computed from just the program counter and the instruction. All of this can occur in a scalar, pipelined CPU which may or may not have a branch predictor.

These scenarios are all *micro-architectural* reasons for speculative execution to occur. There is also an *architectural* reason: branch delay slots. These are a somewhat out-of-fashion idea, where the instruction immediately following a control-flow change may also be executed, even when the control-flow change is taken. This was a way of mitigating some of the performance impact of control-flow changes in pipelined CPUs, since one instruction may be executed *for free* while the new instruction stream is fetched. The MIPS architecture made use of branch delay slots, as does the MicroBlaze architecture analysed in this work. They are absent from most new and modern architectures now, as they significantly complicate out-of-order and super-scalar core design.

Here, we present (to our knowledge) the first study of how speculative (and from a software developers point of view, unexecuted) instructions can be identified from power traces. We show how to very simply identify the depth of any speculative behaviour, which guides software developers as to how much this issue might affect a side-channel secure implementation. This is particularly important given the results in Section 5.5, which shows that control-flow changes do not always act as a barrier between ALU instructions.

5.6.1 Micro-benchmarks

Figure 8 includes pseudo-code for the micro-benchmarks used. Although not obvious, each one uses the same structure in which 1 initial `eor` instruction separated from n other `eor` instructions by a branch; the branch is always taken, with the resulting change of control-flow meaning the latter `eor` instructions are not executed from an architectural perspective. The underlying question is *does the initial `eor` instruction before the branch interact with the unexecuted latter `eor` instructions after the branch*, i.e., *are the latter `eor` instructions speculatively executed*, and therefore *does speculatively execution cause leakage in a manner similar to, but more subtle than conventional execution*? More specifically:

- SPECULATION/BRANCH-FWD (Figure 8a): an `eor` instruction (line 3) separated from two other `eor` instructions (lines 5 and 6) by a `beq` instruction (i.e., a conditional branch):

```
1           .text
2  kernel:  cmp rG, rG
3           eor rA, rB
4           beq target
5           eor rC, rD
6           eor rE, rF
7  target:  .rept 10;
8           eor rZ, rZ
9           .endr
10          bx  lr
```

(a) SPECULATION/BRANCH-FWD:

```
1           .text
2  kernel:
3           eor rA, rB
4           b   target
5           eor rC, rD
6           eor rE, rF
7  target:  .rept 10
8           eor rZ, rZ
9           .endr
10          bx  lr
```

(b) SPECULATION/JUMP-FWD:

```
1           .text
2  kernel:  b   branch
3  target:  .rept 10
4           eor rZ, rZ
5           .endr
6           bx  lr
7  branch:  .rept 10
8           eor rZ, rZ
9           .endr
10          cmp rZ, rZ
11          eor rA, rB
12          beq target
13          eor rC, rD
14          eor rE, rF
```

(c) SPECULATION/BRANCH-BWD:

```
1           .text
2  kernel:  b   branch
3  target:  .rept 10
4           eor rZ, rZ
5           .endr
6           bx  lr
7  branch:  .rept 10
8           eor rZ, rZ
9           .endr
10
11          eor rA, rB
12          b   target
13          eor rC, rD
14          eor rE, rF
```

(d) SPECULATION/JUMP-BWD:

```
1           .text
2  kernel:  mov rY, #0
3  loop:    eor rY, rF
4           mov rY, #0
5           add rG, #-1
6           cmp rG, #0
7           bne loop
8  done:    eor rA, rB
9           eor rC, rD
10          eor rE, rF
11          bx  lr
```

(e) SPECULATION/LOOP-0:

Figure 8: Pseudo-code for micro-benchmarks described in Section 5.6.1, i.e., those related to the case study on speculative execution.

Table 9: A summary of results stemming from the micro-benchmarks in Figure 8, i.e., cases which explore the impact speculative execution has on leakage.

Device	JUMP-FWD	JUMP-BWD	BRANCH-FWD	BRANCH-BWD	LOOP-0
ARM^{N0}					
ARM^{N1}		AF			
ARM^{N2}	BD	BD	BD		AC, AD, AF, DF
ARM^{N3}					
ARM^{S0}					
ARM^{S1}					
ARM^{S2}	AC, AD	AD			AE, AF
ARM^{S3}	CD	CD		CD	AF, BF
ARM^{S4}					AD, AF, BF
ARM^{S5}					
MB^{X0}	BC, CD	BC, CD, DF	BC, CD, DF	BC, CD	AB, BF
MB^{X1}	CD, DE, DF	CD, DE, DF	CD, DE, DF	CD, DE, DF	AB, AC, BF, DF
MB^{X2}	BC, DE, DF	BC, DE, DF	AC, BC, DE, DF	BC, DE, DF	AC, BF, DF
RV^{PRV}					

(a) Instances of Hamming distance leakage: AC, for example, indicates that the Hamming distance between A and C was leaked, noting that only cases that involve speculatively executed instructions are shown.

Device	JUMP-FWD	JUMP-BWD	BRANCH-FWD	BRANCH-BWD	LOOP-0
ARM^{N0}					
ARM^{N1}					
ARM^{N2}	C, D	C, D	C, D	C, D	
ARM^{N3}					
ARM^{S0}					
ARM^{S1}	C, D	C, D	C, D		A, C, D
ARM^{S2}	C	C	C	C	A, C, D, E
ARM^{S3}	C, D	C, D	C, D		A, B
ARM^{S4}	D		C, D		A
ARM^{S5}					
MB^{X0}	C, D	C, D	C, D		A, B, D, E
MB^{X1}	C, D	C, D	C, D	C, D	A, B
MB^{X2}		C, D, E	C, D, E	C, D, E	C, D
RV^{PRV}					

(b) Instances of Hamming weight leakage: A, for example, indicates that the Hamming weight of A was leaked, noting that only cases that involve speculatively executed instructions are shown.

this causes a forward (versus backward) change in control-flow, i.e., using a target which is at a higher (versus lower) address.

- SPECULATION/JUMP-FWD (Figure 8b): as BRANCH-FWD, except with the `beq` instruction replaced with a `b` instruction (i.e., an unconditional branch, or jump) and the associated `cmp` instruction removed.
- SPECULATION/BRANCH-BWD (Figure 8c): an `eor` instruction (line 11) separated from two other `eor` instructions (lines 13 and 14) by a `beq` instruction (i.e., a conditional branch): this causes a backward (versus forward) change in control-flow, i.e., using a target which is at a lower (versus higher) address.
- SPECULATION/JUMP-BWD (Figure 8d): as BRANCH-BWD, except with the `beq` instruction replaced with a `b` instruction (i.e., an unconditional branch, or jump) and the associated `cmp` instruction removed.
- SPECULATION/LOOP-0 (Figure 8e): a `for` loop structure in which an `eor` instruction (line 3: at the beginning of the loop body) separated from three other `eor` instructions (lines 8, 9, and 10: after the end of the loop terminates) by a `beq` instruction (i.e., a conditional branch): this causes a backward (versus forward) change in control-flow, i.e., using a target which is at a lower (versus higher) address.

Note that we carefully distinguish between conditional and unconditional, and forward and backward changes in control-flow. This allows us to deal with different static branch prediction strategies, e.g. predict taken for a conditional backward branch (which is indicative of a loop).

5.6.2 Results

Table 9 shows the results of our experiments. Each row represents a single device and shows which variables leaked, and how, for each speculation experiment listed in Figure 8. We identify this by using correlation analysis to search for the Hamming weight of operands, and the Hamming distance between consecutively accessed operands, and consecutive instruction results.

In terms of the Hamming distance results (Table 9a), there are a number of devices which exhibit this effect. We believe that the MicroBlaze targets ($\mathsf{MB}^{\mathsf{X0}}$, $\mathsf{MB}^{\mathsf{X1}}$ and $\mathsf{MB}^{\mathsf{X2}}$) exhibit speculative leakage mainly because of the architectural branch delay slot. This is explained in the MicroBlaze Processor Reference Manual [Xila, Page 44]:

> *When executing a taken branch with delay slot, only the fetch pipeline stage in MicroBlaze is flushed. The instruction in the decode stage (branch delay slot) is allowed to complete. This technique effectively reduces the branch penalty from two clock cycles to one.*

Looking at this part of the manual, we see clearly that this behaviour is built into the micro-architecture. It explains why for the micro-blaze targets, we clearly see Hamming weight leakage in the XOR instruction operands immediately following all taken control-flow changes and Hamming distance leakage between the operands - i.e. the result of the XOR.

We also see this behaviour in the $\mathsf{ARM}^{\mathsf{N2}}$, $\mathsf{ARM}^{\mathsf{S2}}$, $\mathsf{ARM}^{\mathsf{S3}}$, and $\mathsf{ARM}^{\mathsf{S4}}$ devices. Here, we consistently see Hamming weight leakage from the operands of the instruction following a control-flow change, and the Hamming weight of the instruction result. This is a purely micro-architectural effect, since the ARM architecture has no notion of a branch delay slot. This result was not expected, given the shallow pipeline of the ARM cores. We suggest two possible reasons for seeing these effects in some cores and not others. First, there may be implementer visible design choices around configuring the core for high performance, or low power. Some of these choices many affect when and if different registers are cleared,

or how multiplexer trees are implemented, all of which will affect the leakage. Second, it is possible that EDA tooling plays a part, and that these effects are simply unintentional side effects of synthesis and place and route tools optimising for various metrics other than side channel resilience or energy efficiency. For example EDA tooling quirks might explain Hamming distance leakage in the $\mathsf{ARM}^{\tilde{\mathsf{N}}\mathsf{1}}$ device for the jump-bwd experiment, since no other ARM Cortex-M0 based core leaks this way.

5.6.3 Discussion

Ultimately, one may view the architectural and micro-architectural speculation shown here as essentially caused by the same thing: pipelined instruction execution. The key difference for a software developer being that it is only *documented* (however partially) for the MicroBlaze architecture. In the ARM cores, that leakage effects are visible for *unexecuted* instructions surprised us greatly, and to our knowledge, this is the first study to detail the effect.

For first-order masking schemes, clearly these effects represent significant dangers that software developers and leakage model builders need to be aware of. For higher order schemes, they may naturally be more difficult to exploit owing to the small number of share re-combinations they result in. That said, there is clearly potential for unexpected combinations of all of the effects detailed in this and other works. In any case, they must still be accurately modelled if the absence of leakage must be demonstrated or proved. We believe simply knowing these effects exist is useful, as they are often unexpected. Who would guess that unexecuted instructions after the end of a loop body could interact with the first instruction of the *next* loop iteration?

We note that being able to detect the Hamming weight of the results of unexecuted instructions could be thought of as comparable to a fault attack, since the values do not represent algorithmically correct values, and may be used to further inform attacks.

Going forward, we believe that this effect breaks several assumptions which leakage models (formal and statistical) have until now relied upon. Namely, that *only computation leaks* (which for numerous reasons shown in this and past works, is clearly broken) and that only *architecturally executed* instructions leak. This will make modelling of leakage more complicated, and mean that all models will be much more dependant on the exact micro-architecture of the CPU, rather than the ISA.

In terms of classification, we see these effects as program-non-adjacent, inter-instruction, sequential micro-architectural leakages. To help with further classification, we suggest using *inter* or *intra-block* (as in, basic block) as an additional distinguisher. Previous effects have all been intra-block, whereas this effect shows how leakage can occur across basic blocks. We can also tag this as *speculative* leakage, as opposed to *non-speculative* leakage, which is the case for all prior experiments.

6 Discussion

This section frames the lower level findings presented in Section 5 at a higher level, and so in a more directly usable form. That is, we aim to equip readers with an intuition for micro-architectural leakage in general, and therefore the capacity to reason about it; we then discuss the implications of our work for two specific fields, namely 1) the design, implementation, and evaluation of masking schemes, and 2) construction of accurate fine-grained leakage models.

6.1 A mental model of micro-architectural leakage

Having investigated leakage across different target devices, we now try to help researchers and engineers develop an intuition for reasoning about them. We use a contrived CPU

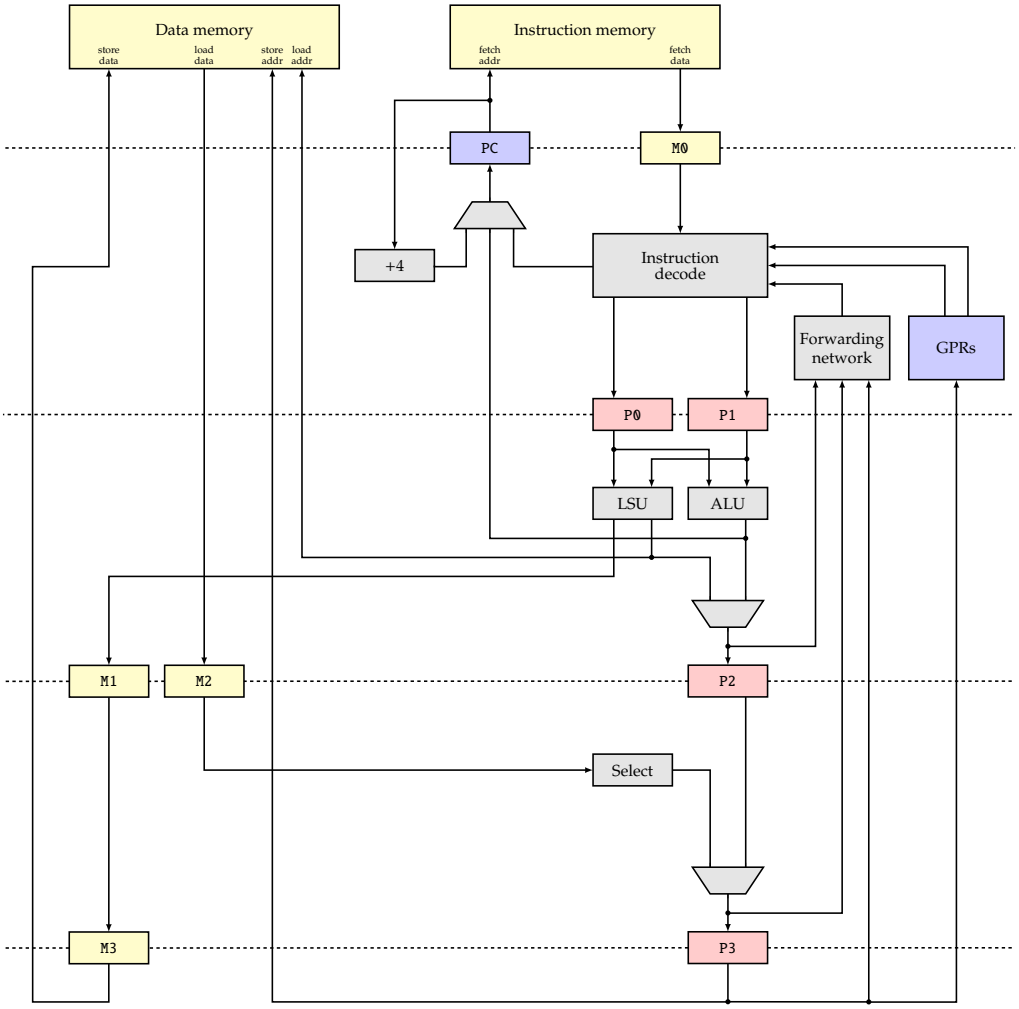


Figure 9: A contrived 5-stage, pipelined processor micro-architecture with architectural (blue), micro-architectural (red), and memory sub-system (yellow) components annotated as appropriate. Note that, we deliberately 1) exclude some components related to the control path, because they are usually irrelevant with respect to leakage, and 2) include some components related to the memory sub-system, despite normally being outside the micro-architectural remit, because they are highly relevant with respect to leakage.

and memory sub-system Figure 9 block diagram, and describe the *journey* of instructions through the pipeline. At each stage, we note the different ways instructions *could* be implemented in a reasonable micro-architecture, and link to examples of these options in our case study results.

Intuitively, we can investigate if the micro-architecture will cause share combinations by looking at *convergence* and *divergence* points in the data-flow of the system. *Convergence* points are where data from multiple places meet, and the micro-architecture selects *one* of the data items to progress. *Divergence* points are the opposite. Each can be further classified using the scheme developed in Section 3.4 as ISA, sequential or glitching-combinatorial.

We discuss types of leakage which *can* occur, some of which we have observed in this work, some of which are described in other works, and some of which are hypothetical. Our intention is to provide a starting point for engineers and researchers to identify and isolate causes of leakage in their own systems.

6.1.1 ALU-type instructions

An XOR (or similar) instruction might have two operand registers, or an operand register and an immediate. Decoding of the immediate or register variant may cause glitches in the register address decode logic, causing unexpected toggling between registers which are *not explicitly addressed* by the instruction. This is described in [GHP+20] and can be mitigated by structuring register file data ports as AND-OR trees, rather than multiplexer trees. The same effect can occur in the forwarding network, where results from uncommitted instructions are routed back to the decode stage to prevent stalls. This makes the decode stage a *convergence* point for instruction operands and results, which may be in-flight or at rest in the GPRs. The most common kind of leakage observed here is the sequential micro-architectural leakage, where instruction operands are loaded into pipeline registers `P0` and `P1`, as seen in Section 5.4. Knowing which operands are loaded to which pipeline register for each instruction is essential for leakage modelling. Convergences can also be glitching-combinatorial, due to multiplexer switching, though in our experience, this is much harder to detect. One concrete example of this is the results in Section 5.1. Due to the nature of glitches, their occurrence can even be data dependant.

In the execute stage, we can see combinatorial and glitching-combinatorial leakage depending on how the ALU is structured. For example, if different functions in the ALU are not gated, we might see Hamming Weight leakage for the result of an XOR, the result of and ADD, *and* the result of a shift. We may even see Hamming Distance leakage between these results, as they converge through a multiplexer tree to the `P2` register. Loading of the `P2` pipeline register will also cause sequential Hamming Distance leakage between the results of the current instruction, and the previous instruction which loaded `P2`. Note this is not always the previous instruction in program order.

Registers `P2` and `P3` are important as they force results of adjacent instructions to interact before being written to a GPR. This implies that adjacent instructions which share no architectural registers as operands or destinations will still interact from a leakage perspective. We can see this clearly in, Section 5.4 where in most CPUs the results of subsequent ALU instructions *do* interact, for ARM Cortex-M0+ and PicoRV32 based systems, which have only 2 and 0 pipeline stages respectively, there is no such leakage.

Between `P2` and `P3`, there is a convergence point between ALU instruction results and data loaded from memory. We observed this sort of interaction between loaded data and ALU results in the 5 and 8 stage Xilinx MicroBlaze cores in Section 5.4, but not in the shorter pipelined cores.

Finally, there is a divergence point where the result is written to the GPRs, where there will be Hamming Distance leakage between the old and new GPR values. This is the *only* form of ISA leakage which occurs.

6.1.2 Control-flow instructions

When first entering the pipeline, a jump instruction *can* change the program counter immediately, as it is not dependant on any architectural state other than the program counter. Hence in many cases, it *need not* alter any of the data-carrying pipeline registers `P0`, `P1`, `P2`, and `P3`. We found in Section 5.5 that for all of the ARM cores, a jump instruction placed between two XOR instructions *does not* prevent Hamming distance leakage between some of the *operands* of the XOR instructions. This suggests that *one* of the pipeline registers is updated (likely with the branch target), but the other is left unchanged as an energy efficiency optimisation. We also found this was the case in the $\mathsf{MB}^{\mathsf{X2}}$ and $\mathsf{RV}^{\mathsf{PRV}}$ based systems.

A programmer must consider if they rely on control-flow changes to act as implicit barriers between instructions to prevent share combinations, and whether that assumption holds for their system. This effect is also apparent in the *loop* speculation experiment in Section 5.6.

Branch instructions, being conditional on architectural state, tend to act as barriers between ALU-type instructions in our experiments in Section 5.5. In our hypothetical system, this would mean loading `P0` and `P1` with the operands to the comparison. Note this does not *necessarily* change the values of `P2` and `P3`, meaning that we could potentially see Hamming distance leakage between the *results* of ALU instructions which are separated by a conditional branch. This is observed the case of the MicroBlaze based $\mathsf{MB}^{\mathsf{x0}}$ and $\mathsf{MB}^{\mathsf{x1}}$ systems.

On ARM systems, conditional branches are evaluated over two instructions: one does the *comparison* (`cmp`) and sets a condition flag, another tests a condition flag and performs the control flow change. Our experiments in Section 5.5 deliberately placed an ALU instruction between the `cmp` and branch instructions. It would be reasonable to reorder the `cmp` instruction to act as a barrier for the *operands* of the ALU instructions if the branch does not already provide this. However, we did not a have access to an ARM core with a longer pipeline (e.g. Cortex-M7), so we could not check if, as in the case of the MicroBlaze cores, we still get Hamming distance leakage from pipeline registers which only store instruction results.

For both branch and jump instructions, the actual control flow change may occur some way down the pipeline. In our hypothetical system, branches feed the target address to the PC from the execute stage. This allows for subsequent instructions in the decode stage to *speculatively* read architectural state, either purely combinatorially, or sequentially into the operand registers `P0` and `P1`. Indeed, we observed exactly this behaviour in Section 5.6, where the Hamming weight of operands were often detectable in unexecuted instructions. In some cases, we even observed the Hamming weight of the instruction result, even in ARM cores with a short pipeline.

A different approach would be to prevent updates to `P0` and `P1` whenever a branch or jump is occurring in the execute stage. This is a recommendation supported by our results, since it results in fewer register updates (thus saving energy) and make reasoning about leakage much easier.

6.1.3 Load and store instructions

As shown in Section 5.1 and other works, memory accesses can have a large and counter-intuitive effect on leakage. Here, we focus on explanations for differences in behaviour shown in Table 2.

A load instruction with a base address operand register, an immediate offset and destination register enters the pipeline in the decode stage. It is subject to the same sources of leakage described for the ALU type instructions. Note that unless the *address* (either base or offset) is dependant on a secret value, a load instruction is unlikely to

explicitly cause leakage in the decode stage of our hypothetical system.

In the execute stage, the load address is sent to the memory, which on the next cycle will load the `M2` register with the requested data. The relevant bytes of `M2` are then selected (load word/half/byte) and converge with ALU results in the `P3` register, before being written back to the GPRs. Memory load data *could* have been put into `P2`, and observations in Table 2 show that this is *sometimes* the case. Note that the same CPU core, implemented by the same vendor in different devices, can exhibit different behaviours here. We believe that some cores have a separate *load-data* register as part of the memory bus for performance or timing reasons, while some merge this with a pipeline register to reduce area. This design choice has implications for leakage behaviour. If the `M2` and `P2` are separate, then loads which are potentially many instructions apart can interact. If they are represented by the same physical register, then ALU type instructions can act as a barrier between load data instructions.

As with ALU instructions, the existence of `P3` in our hypothetical system can mean that load instructions which do not have the same architectural destination register can still leak the Hamming Distance between their loaded values, and the result of an adjacent ALU instruction. This is because the two streams of data converge before reaching the architectural register file. The see this in many devices even with short pipelines, in Table 6. For short pipelines, where `P3` would not exist, this suggests that loaded data is actually routed to `P2` in our hypothetical system.

The *bytes select* block, between `M2` and `P3` is the reason we see the kind of leakage patterns shown in Figure 5. Here, 1, 2 or 4 contiguous bytes of data are selected from the loaded 4-byte memory word. This multiplexing can cause potentially *all* of the bytes to leak their Hamming distance. This sort of leakage is expected in systems using a word-orientated memory interconnect, such as AHB-Lite or AXI-Lite. This also explains the effect noted in past works, where requesting a byte from memory can actually load the entire word which contains the byte, leading to unexpected leakage. Our results in table Table 3 show that this effect is common across most of the devices studies here.

For store instructions, the potential for leakage exists in the decode stage, as it must read an extra architectural register as the data to be stored. Knowing which pipeline register (`P0` or `P1` in our system) is overwritten with the data-to-be-stored, rather than the address, is important to avoid leakage collisions or the need to add barrier instructions which reduce performance. This need for barrier instructions to flush shared state is detailed in [SSB+20]. Our results in Table 2 repeat similar experiments for a range of cores, showing how diverse the actual behaviour of cores can be in terms of memory hierarchy leakage. A programmer who implemented all of the [SSB+20] countermeasures on a core which does not need them would have an unnecessarily inefficient program.

In the execute stage of our hypothetical system, the data to be stored *and* the address must progress all the way to the final stage of the pipeline before being sent to memory. Any earlier than this would cause incorrect program behaviour, as the store could be perceived to have occurred before the instruction retires. In our design, the store data travels down a *separate* pipeline (a dedicated store buffer, `M1` and `M3`) rather than the normal ALU pipeline. This separation or sharing of load, store and ALU data pipelines is a design trade-off between performance and area, and we believe explains why in Table 2, adjacently loaded and stored data interact in some cases but not others.

6.2 Implications for evaluation of masking schemes

Differences in leakage behaviour between devices can have a *considerable* impact on the complexity, performance and code size of leakage resistant code. For example, the $\mathsf{ARM}^{\mathsf{S2}}$ and $\mathsf{ARM}^{\mathsf{S4}}$ devices do not cause collisions between consecutively loaded and stored values (see Table 2), while other similar devices do. Adding "flush" instructions for more leaky devices may add a significant performance and energy efficiency penalty, which is

unnecessary on the non-leaking devices. Likewise, code which is secure on the less leaky devices may not be secure on others.

We did not survey the literature to discover which devices were used to derive overhead claims for different masking schemes and implementations, but suspect our results imply considerable margin for variance across devices. If such devices are chosen as benchmarking platforms for standardisation processes, it may be impossible to get a clear picture of the overhead required to implement certain masking schemes due to the underlying micro-architecture of the device.

As a single example, in [BDM+20], the authors apply their tool to several NIST Lightweight Cryptography candidates, using an ARM Cortex-M4 based Nucleo STM32F401RE device. Our experiments clearly show that one ARM Cortex-M4 does not leak like another. This poses the question of how portable the code generated by TORNADO or other tools like is. This is especially important, given its reliance on GCC to perform register assignment and low level instruction ordering, which may leave it open to some of the effects described in this and other works.

We suggest evaluations of software masked algorithms be evaluated on *at least* two devices which have different micro-architectural leakage characteristics. This will enable researchers to separate the overhead from a masking scheme, and the overhead from a particular micro-architecture. This has particular relevance for ongoing standardisation processes, such as the NIST Lightweight Cryptography project. We hope this work can inform the choice of devices.

Our results suggest tools which *generate* side-channel secure code will benefit considerably from being more micro-architecture-aware. Indeed, we believe they *must* be micro-architecture-aware in many cases in order to avoid generating leaking code. However, knowing which mitigations they must apply, or when they can omit them because a particular micro-architectural effect is not present in the target device, will likely lead to considerable performance and security improvements. Adding more shares to a masking scheme may remove the risk of the interactions seen in this and prior works, but this is unsatisfactory where performance and code size are an issue. We still want to be able to use the minimal number of shares possible for a given security requirement.

Finally, while the portability of side-channel resilient code has always been dubious, our results emphasise just how challenging *efficient* and portable side-channel resilient code is to build.

6.3 Implications for leakage modelling

Here, we discuss how our experimental results inform formal and statistical different approaches to leakage modelling and tooling. Note that when talking about formal modelling of leakage, we specifically mean in the context of CPU pipelines, not hardware implementations of cryptographic primitives.

Capturing types of leakage. Using our list of leakage types in Section 3.4 and based on past work, we know both formal and statistical models cope well with ISA leakage. This is not sufficient to provide a usable leakage model for the real world, given the micro-architectural effects described in this and other works.

We next consider sequential micro-architectural modelling. Again, we believe there is no *inherent* difficulty in using formal or statistical approaches, but the relative merits of each approach now become apparent. Formal models must have *perfect* information about the system being modelled. Based on our results in Section 5.6, we also know that a trace of *retired* instructions is not enough, one must also model the pipeline of the target device explicitly. This is considerably more effort to build for a formal model than a statistical model, since a statistical model Elmo will capture all of this information as a side effect of the initial device profiling. Statistical approaches are hence able to treat target devices as

a black box, and so build usable models with comparatively little information about the micro-architecture. Formal models however *must* have access to this information, requiring either an NDA with the device manufacturer, or exhaustive reverse engineering of the kind we describe in this work. [BGG+20] is a good example of this, where the quality of the security guarantee is heavily dependant on the user-supplied micro-architectural model of the target device. We believe the kinds of leakage-benchmarks developed in this work can play an important role in validating and building formal models of the kind described in [BGG+20].

Once we move to combinatorial and glitching combinatorial leakage however, we see formal modelling approaches to software masking begin to struggle, since the number of variables and possible interactions explodes, even in the presence of perfect information about the system.

Separating methodology, models, and devices. Given the variety of results we find, including wildly different leakage behaviours even for devices with the same CPU core, we believe that the community must focus heavily on building *replicable methodologies* for leakage modelling, rather than very accurate leakage models for particular devices.

For formal models in particular, we believe that a machine readable description of micro-architectural state elements, and when they are updated by instructions, will be an essential part of their methods. This allows for manufacturers to re-use the model building tool without sharing sensitive details about their device. In this respect, we believe the Domain Specific Language (DSL) based approach to describing device micro-architectures in [BGG+20] is very promising. Indeed, the authors even demonstrate the necessity of their tool being micro-architecture aware by showing how a secure implementation in one ARM Cortex-M0+ based device is not secure in a ARM Cortex-M4 based device. This is well supported by our empirical results.

Temporal locality. In Section 5.1, Section 5.2 and Section 5.3, we saw how loads and stores in many devices are the only instructions to interact with micro-architectural state elements. As discussed in [SSB+20], this is a problem for statistical models, which only consider N adjacent instructions in a sliding window as contributing to leakage at any one moment. Clearly, if a load instruction outside the window previously altered a piece of state, then this will be missed if there are greater than N instructions before the next load.

Based on our results, we believe one solution to this could be a hybrid model. Specifically, ALU style instructions could be modelled statistically. We include in this interactions between ALU style instructions and memory accesses, as documented in Section 5.4. This has the clear benefit of not needing a very detailed model of the execution pipeline within a CPU core, and being able to capture combinatorial and glitching leakage inherently within the statistical profiling step.

For temporally distant interactions, like loads and stores, which affect pieces of micro-architectural state which no other instructions interact with, we suggest formal modelling (of the kind described in [BGG+20]) is a good approach. One would need only to model ISA state, architectural data values and a very small set of micro-architectural state elements - perhaps only two additional registers, as indicated by some of our results. The update rules for these pieces of micro-architectural state are very easy to reverse engineer, because there are only a small number (of the order of 6 or 12, depending on the addressing modes supported by the ISA) of instructions which alter them.

The formal model would be able to *prove* that no load or store instruction cause leakage. The statistical model would then be used to evaluate the rest of the instruction stream for leakage.

Benchmarking for model quality. Anyone building, buying or using a leakage model needs to verify its accuracy. Historically it has been hard to judge model accuracy outside the results of a paper. We believe the set of leakage micro-benchmarks in this paper provide an important route to improving the confidence people have in a model which they did not personally develop. By isolating critical corner case behaviours into individual experiments, we can realise something similar to unit testing for leakage models. By publishing all of our micro-benchmarks and the analysis flow, we hope they can be used to augment the development of new models.

We recognise that it may always be possible to identify pathological cases of micro-architectural leakage which are not identified by leakage models, and these tools should focus on cases which make them the most useful to software engineers.

7 Conclusion

Summary. In this paper, we first presented and then used MIRACLE to study leakage from 32-bit micro-controllers which span a range of different architectures (or ISAs), micro-architectures, and vendors. Although we expect the infrastructure presented in Section 4 to be useful in the long term, the case studies in Section 5 provide evidence which leads to or reinforces some important conclusions:

- The assumption that only computation leaks can be invalid. For example, Section 5.5 demonstrates that even instructions which are not retired, and thus not executed from an architectural perspective, can cause leakage as a result of speculation: the attack surface enabled by transient execution, which has been so well explored from the perspective of *discrete* side-channels (see, e.g., [CBS+19]), also extends *analogue* side-channels.
- The assumption that instructions leak independently can be invalid. For example, Section 5.4 replicates known leakage effects stemming from adjacent instructions; Section 5.1 extends this, proposing some more formal notions of adjacency, and demonstrates that even architecturally independent instructions can interact under some circumstances.
- Given the above, for example, any claim regarding the portability of security properties between different devices can be invalid. For example, Section 5.4 demonstrates differences in efficacy of ad hoc leakage barriers based on the use of `nop`-based; this was also noted in [GMPP20, Section 4.1]. A corollary of this fact is that it is insufficient to make a security claim based simply on an *architecture* (e.g., "ARMv7-M") or even a *core* (e.g., "ARM Cortex-M3"): throughout Section 5, we have demonstrated that it one *must* quote and evaluate a given instruction sequence on a specific *device* (e.g., "STM32F100RBT6B"). In our view, this has (at least) two implications: 1) authors should adopt this approach in writing, alongside publication of associated artefacts to support reproducibility, and 2) care must be taken when selecting standard evaluation platforms, e.g., to support the standardisation.

Future work. In terms of future directions, we believe a more formal and quantitative way of reasoning about the different types of leakage described in Section 3, and how they contribute to leakage models would be of clear benefit. This includes insights into when certain kinds of leakage may be safely ignored. We also believe investigation into building hybrid leakage models, as described in Section 6 is a pragmatic way to get the best of both worlds from different modelling approaches. Finally, we have only investigated micro-architecture effects in terms of documenting their existence. Further research into effective countermeasures (as described in [SSB+20]) and exploitation is needed.

Acknowledgements

We extend our thanks to Si Gao and Elisabeth Oswald for their invaluable insight, feedback, questions, and encouragement.

This work has been supported in part by EPSRC via grant EP/R012288/1, under the RISE (http://www.ukrise.org) programme.

References

[AARR02] D. Agrawal, B. Archambeault, J.R. Rao, and P. Rohatgi. The EM side-channel(s). In *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 2523, pages 29–45. Springer-Verlag, 2002.

[ABB64] G.M. Amdahl, G.A. Blaauw, and F.P. Brooks. Architecture of the IBM System/360. *IBM Journal of Research and Development*, 8:87–101, 1964.

[AR19] A. Abel and J. Reineke. nanoBench: A low-overhead tool for running microbenchmarks on x86 systems, 2019.

[ARMa] ARM. https://static.docs.arm.com/ihi0033/bb/IHI0033B_B_amba_5_ahb_protocol_spec.pdf.

[ARMb] ARM. Data-sheet. https://developer.arm.com/ip-products/processors/cortex-m/cortex-m0-plus.

[ARMc] ARM. Data-sheet. https://developer.arm.com/ip-products/processors/cortex-m/cortex-m0.

[ARMd] ARM. Data-sheet. https://developer.arm.com/ip-products/processors/cortex-m/cortex-m3.

[ARMe] ARM. Data-sheet. https://developer.arm.com/ip-products/processors/cortex-m/cortex-m4.

[ARM18] ARM. *ARMv7-M Architecture Reference Manual*, DDI0403E.d edition, 2018. https://developer.arm.com/documentation/ddi0403.

[BBC+19] G. Barthe, S. Belaïd, G. Cassiers, P.-A. Fouque, B. Grégoire, and F.-X. Standaert. maskverif: Automated verification of higher-order masking in presence of physical defaults. In *European Symposium on Research in Computer Security (ESORICS)*, LNCS 11735, pages 300–318. Springer-Verlag, 2019.

[BDM+20] S. Belaïd, P.-É. Dagand, D. Mercadier, M. Rivain, and R. Wintersdorff. Tornado: Automatic generation of probing-secure masked bitsliced implementations. In *Advances in Cryptology (EUROCRYPT)*, LNCS 12107, pages 311–341. Springer-Verlag, 2020.

[BGG+14] J. Balasch, B. Gierlichs, V. Grosso, O. Reparaz, and F.-X. Standaert. On the cost of lazy engineering for masked software implementations. In *Smart Card Research and Advanced Applications (CARDIS)*, LNCS 8968, pages 64–81. Springer-Verlag, 2014.

[BGG+20] G. Barthe, M. Gourjon, B. Gregoire, M. Orlt, C. Paglialonga, and L. Porth. Masking in fine-grained leakage models: Construction, implementation and verification. Cryptology ePrint Archive, Report 2020/603, 2020.

[BGI+18] R. Bloem, H. Groß, R. Iusupov, B. Könighofer, S. Mangard, and J. Winter. Formal verification of masked hardware implementations in the presence of glitches. In *Advances in Cryptology (EUROCRYPT)*, LNCS, pages 321–353. Springer-Verlag, 2018.

[BGR18] S. Belaïd, D. Goudarzi, and M. Rivain. Tight private circuits: Achieving probing security with the least refreshing. In *Advances in Cryptology (ASIACRYPT)*, LNCS 11272, pages 343–372. Springer-Verlag, 2018.

[BMT16] W. Burleson, O. Mutlu, and M. Tiwari. Who is the major threat to tomorrow's security? You, the hardware designer. In *Design Automation Conference (DAC)*, pages 145:1–145:5, 2016.

[BP18] A. Barenghi and G. Pelosi. Side-channel security of superscalar CPUs: evaluating the impact of micro-architectural features. In *Design Automation Conference (DAC)*, pages 120:1–120:6, 2018.

[CBG+17] T. De Cnudde, B. Bilgin, B. Gierlichs, V. Nikov, S. Nikova, and V. Rijmen. Does coupling affect the security of masked implementations? In *Constructive Side-Channel Analysis and Secure Design (COSADE)*, LNCS 10348, pages 1–18. Springer-Verlag, 2017.

[CBS+19] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security Symposium*, pages 249–266, 2019.

[CEM18] T. De Cnudde, M. Ender, and A. Moradi. Hardware masking, revisited. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2018(2):123–148, 2018.

[CGD18] Y. Le Corre, J. Großschädl, and D. Dinu. Micro-architectural power simulator for leakage assessment of cryptographic software on ARM Cortex-M3 processors. In *Constructive Side-Channel Analysis and Secure Design (COSADE)*, LNCS 10815, pages 82–98. Springer-Verlag, 2018.

[CGMA+15] C. Cernazanu-Glavan, M. Marcu, A. Amaricai, S. Fedeac, M. Ghenea, Z. Wang, A. Chattopadhyay, J. Weinstock, and R. Leupers. Direct FPGA-based power profiling for a RISC processor. In *IEEE International Instrumentation and Measurement Technology Conference (I2MTC)*, pages 1578–1583, 2015.

[DAK19] W. Diehl, A. Abdulgadir, and J.-P. Kaps. Vulnerability analysis of a soft core processor through fine-grain power profiling. Cryptology ePrint Archive, Report 2019/742, 2019.

[GHP+20] B. Gigerl, V. Hadzic, R. Primas, S. Mangard, and R. Bloem. Coco: Co-design and co-verification of masked software implementations on CPUs. Cryptology ePrint Archive, Report 2020/1294, 2020.

[GJJR11] G. Goodwill, B. Jun, J. Jaffe, and P. Rohatgi. A testing methodology for side-channel resistance validation. In *NIST Non-Invasive Attack Testing Workshop*, 2011.

[GMO01] K. Gandolfi, C. Mourtel, and F. Olivier. Electromagnetic analysis: Concrete results. In *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 2162, pages 251–261. Springer-Verlag, 2001.

[GMPP20] S. Gao, B. Marshall, D. Page, and T. Pham. FENL: an ISE to mitigate analogue micro-architectural leakage. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2020(2):73–98, 2020.

[GYCH18] Q. Ge, Y. Yarom, D. Cock, and G. Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering (JCEN)*, 8:1–27, 2018.

[GYH18] Q. Ge, Y. Yarom, and G. Heiser. No security without time protection: we need a new hardware-software contract. In *Asia-Pacific Workshop on Systems (APSys)*, 2018.

[HKSS12] Y. Hori, T. Katashita, A. Sasaki, and A. Satoh. SASEBO-GIII: A hardware security evaluation board equipped with a 28-nm FPGA. In *IEEE Global Conference on Consumer Electronics*, pages 657–660, 2012.

[KGBR19] M. Arsath KF, V. Ganesan, R. Bodduna, and C. Rebeiro. PARAM: A microprocessor hardened for power side-channel attack resistance. 2019.

[KJJ99] P.C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in Cryptology (CRYPTO)*, LNCS 1666, pages 388–397. Springer-Verlag, 1999.

[KSM20] D. Knichel, P. Sasdrich, and A. Moradi. SILVER – Statistical Independence and Leakage Verification. In *Advances in Cryptology (ASIACRYPT)*, LNCS 12491, pages 787–816. Springer-Verlag, 2020.

[LBS19] I. Levi, D. Bellizia, and F.-X. Standaert. Reducing a masked implementation's effective security order with setup manipulations. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2019(2):293–317, 2019.

[MDLG18] D. Mercadier, P.-É. Dagand, L. Lacassagne, and G. Gilles. Usuba: optimizing & trustworthy bitslicing compiler. In *Workshop on Programming Models for SIMD/Vector Processing (WPMVP)*, pages 1–8, 2018.

[Mea] ST Micro-electronics. Data-sheet. https://www.st.com/resource/en/datasheet/stm32f071cb.pdf.

[Meb] ST Micro-electronics. Data-sheet. https://www.st.com/resource/en/datasheet/stm32f100cb.pdf.

[Mec] ST Micro-electronics. Data-sheet. https://www.st.com/resource/en/datasheet/stm32f215re.pdf.

[Med] ST Micro-electronics. Data-sheet. https://www.st.com/resource/en/datasheet/stm32f303cb.pdf.

[Mee] ST Micro-electronics. Data-sheet. https://www.st.com/resource/en/datasheet/dm00037051.pdf.

[Mef] ST Micro-electronics. Data-sheet. https://www.st.com/resource/en/datasheet/dm00039193.pdf.

[MGH19] E. De Mulder, S. Gummalla, and M. Hutter. Protecting RISC-V against side-channel attacks. In *Design Automation Conference (DAC)*, pages 45:1–45:4, 2019.

[MMT20] L. De Meyer, E. De Mulder, and M. Tunstall. On the effect of the (micro)architecture on the development of side-channel resistant software. Cryptology ePrint Archive, Report 2020/1297, 2020.

[MOP07] S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007.

[MR04] S. Micali and L. Reyzin. Physically observable cryptography. In *Theory of Cryptography (TCC)*, LNCS 2951, pages 278–296. Springer-Verlag, 2004.

[MWO16] D. McCann, C. Whitnall, and E. Oswald. ELMO: Emulating leaks for the ARM Cortex-M0 without access to a side channel lab. Cryptology ePrint Archive, Report 2016/517, 2016.

[New] NewAE. https://wiki.newae.com/CW308T-STM32F.

[NXPa] NXP. Data-sheet. https://www.nxp.com/docs/en/data-sheet/LPC81XM.pdf.

[NXPb] NXP. Data-sheet. https://www.nxp.com/docs/en/data-sheet/LPC111X.pdf.

[NXPc] NXP. Data-sheet. https://www.nxp.com/docs/en/data-sheet/LPC1311_13_42_43.pdf.

[PV17] K. Papagiannopoulos and N. Veshchikov. Mind the gap: Towards secure 1st-order masking in software. In *Constructive Side-Channel Analysis and Secure Design (COSADE)*, LNCS 10348, pages 282–297. Springer-Verlag, 2017.

[RKL+04] S. Ravi, P.C. Kocher, R.B. Lee, G. McGraw, and A. Raghunathan. Security as a new dimension in embedded system design. In *Design Automation Conference (DAC)*, pages 753–760, 2004.

[RP10] M. Rivain and E. Prouff. Provably secure higher-order masking of AES. In *Cryptographic Hardware and Embedded Systems (CHES)*, LNCS 6225, pages 413–427. Springer-Verlag, 2010.

[RRKH04] S. Ravi, A. Raghunathan, P.C. Kocher, and S. Hattangady. Security in embedded systems: Design challenges. *ACM Transactions on Embedded Computer Systems*, 3(3):461–491, 2004.

[RSVC+11] M. Renauld, F.-X. Standaert, N. Veyrat-Charvillon, D. Kamel, and D. Flandre. A formal study of power variability issues and sidechannel attacks for nanoscale devices. In *Advances in Cryptology (EUROCRYPT)*, LNCS 6632, pages 109–128. Springer-Verlag, 2011.

[RV:19] The RISC-V instruction set manual. Technical Report Volume I: User-Level ISA (Version 20190608-Base-Ratified), 2019. http://riscv.org/specifications/.

[SSB+20] M.A. Shelton, N. Samwel, L. Batina, F. Regazzoni, M. Wagner, and Y. Yarom. Rosita: Towards automatic elimination of power-analysis leakage in ciphers, 2020.

[Sze19] J. Szefer. Survey of microarchitectural side and covert channels, attacks, and defences. *Journal of Hardware Systems Security*, 3(3):219–234, 2019.

[Wel47] B.L. Welch. The generalization of "student's" problem when several different population variances are involved. *Biometrika*, 34(1-2):28–35, 1947.

[Wol] C. Wolf. https://github.com/cliffordwolf/picorv32.

[Xila] Xilinx. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug984-vivado-microblaze-ref.pdf.

[Xilb] Xilinx. https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf.

A Links for web-based interface

Table 10 captures the correspondence between micro-benchmarks listed in this paper, and those accessible via the web-based interface described in Section 4.4: differences between the two stem from use of a shorter or more meaningful alias, and, in either case, improved readability, within the paper.

Table 10: Correspondence between micro-benchmarks listed in this paper, and those accessible via the web-based interface.

Paper	Online
PIPELINE/BRANCH-PRE (Figure 7a)	PIPELINE/BRANCH-FLUSH-REGS-PRE
PIPELINE/BRANCH-POST (Figure 7b)	PIPELINE/BRANCH-FLUSH-REGS-POST
PIPELINE/JUMP-PRE (Figure 7c)	PIPELINE/JUMP-FLUSH-REGS-PRE
PIPELINE/JUMP-POST (Figure 7d)	PIPELINE/JUMP-FLUSH-REGS-POST
SPECULATION/BRANCH-FWD (Figure 8a)	SPECULATION/BRANCH-FWD
SPECULATION/JUMP-FWD (Figure 8b)	SPECULATION/JUMP-FWD
SPECULATION/BRANCH-BWD (Figure 8c)	SPECULATION/BRANCH-BWD
SPECULATION/JUMP-BWD (Figure 8d)	SPECULATION/JUMP-BWD
SPECULATION/LOOP-0 (Figure 8e)	SPECULATION/LOOP-0
PIPELINE/EOR-ADD (Figure 6b)	PIPELINE/ISEQ-XOR-ADD
PIPELINE/EOR-LSL (Figure 6c)	PIPELINE/ISEQ-XOR-SRLI
PIPELINE/EOR-ROR (Figure 6d)	PIPELINE/ISEQ-XOR-ROR
PIPELINE/EOR-LDR (Figure 6e)	PIPELINE/ISEQ-XOR-LW
PIPELINE/EOR-STR (Figure 6f)	PIPELINE/ISEQ-XOR-SW
PIPELINE/NOP-EOR (Figure 6g)	PIPELINE/ISEQ-NOP-XOR
MEMORY-BUS/LD-LD (Figure 3a)	MEMORY-BUS/IMPLICIT-REGS-LD-LD
MEMORY-BUS/LD-ST (Figure 3b)	MEMORY-BUS/IMPLICIT-REGS-LD-ST
MEMORY-BUS/ST-LD (Figure 3c)	MEMORY-BUS/IMPLICIT-REGS-ST-LD
MEMORY-BUS/ST-ST-1 (Figure 3d)	MEMORY-BUS/IMPLICIT-REGS-ST-ST-1
MEMORY-BUS/ST-ST-2 (Figure 3e)	MEMORY-BUS/IMPLICIT-REGS-ST-ST-2
MEMORY-BUS/ST-ST-3 (Figure 3f)	MEMORY-BUS/IMPLICIT-REGS-ST-ST-3