IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received September 8, 2021, accepted September 20, 2021, date of publication November 2, 2021,
date of current version December 2, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3124761

Exploring Cortex-M Microarchitectural Side Channel Information Leakage

ALESSANDRO BARENGHI, LUCA BREVEGLIERI, NICCOLÒ IZZO, AND GERARDO PELOSI, (Member, IEEE)
Department of Electronics, Information and Bioengineering, Politecnico di Milano, 20133 Milan, Italy
Corresponding author: Niccolò Izzo (niccolo.izzo@polimi.it)

ABSTRACT The growing Internet of Things (IoT) market demands side-channel attack resistant, efficient, cryptographic implementations. Such implementations, however, are microarchitecture-specific, and cannot be implemented without an in-depth structural knowledge of the CPU and memory information leakage patterns; a description of such information leakages is presently not disclosed by any processor design company. In this work we propose the first Instruction Set Architecture (ISA) level framework for microarchitectural leakage characterization. Our framework allows to extract a microarchitectural leakage profile from a superscalar in-order processor; we infer detailed pipeline characteristics through the observation of instruction execution timings, and provide an identification of the datapath registers via a side-channel measuring setup. The extracted model can serve as a foundation for building solid countermeasures against side-channel attacks on software cryptographic implementations. We validate the extracted models on the ARM Cortex-M4 and ARM Cortex-M7 CPUs, two of the most widespread ARM microcontroller cores. Finally, as a further validation of the effectiveness of our derived model, we mount a successful attack on unprotected AES implementations for each of the examined platforms.

INDEX TERMS Computer security, correlation power analysis, embedded systems security, microarchitectural reverse engineering, side channel attack countermeasures.

I. INTRODUCTION

Embedded and Internet of Things (IoT) industries use extensively cryptographic algorithms to guarantee authenticity on data retrieved from remote devices, integrity on firmware updates delivered to said devices, and confidentiality on data transmitted via radio channels. Since many of those devices are deployed in publicly accessible environments, where physical protection may be unpractical, their cryptographic implementations are a prime target for side-channel attacks [30] Side-channel attacks rely on extracting information which is unintendedly transmitted by measurable changes in a physical parameter of a computing device. Common physical parameters include the time required by the computation, the energy required to perform it, or the Electro-Magnetic (EM) and acoustic emissions of the device: such parameters are referred to as side-channels. The practicality of extracting information from a variety of side channels is witnessed by many public works; among which, as a sample, we recall Many side-channels such as timing variations [19], acoustic emissions [3], optical emissions [12], power consumption [20] and Electro-Magnetic (EM) emissions [14] have been investigated.

In this work, we will focus on power consumption and EM emission side-channels, due to their significant effectiveness. Although several hardware EM side-channel countermeasures have been developed, such as dual rail logic [22] and virtual secure circuits [37], nonetheless, due to their added cost, they are usually applied to dedicated Integrated Circuits (IC) such as secure enclaves [35], and are usually not present in general purpose Micro Controller Units (MCUs) [34]. Whenever hardware countermeasures are not in place, software protections can be employed. Three general families of countermeasures against power/EM side-channels have been proposed: i) *Masking* [9], [29], which mixes sensitive data with per-execution random values to hamper the use of correlation attacks; ii) *Hiding* [5], [24], [32], which lowers the signal to noise ratio of the side channel either by adding noise to it, or randomizing the order in which computations are made, in turn forcing the attacker to observe

The associate editor coordinating the review of this manuscript and approving it for publication was Gautam Srivastava.

unrelated information on the side channel; iii) *Morphing* [1], which dynamically changes the operations employed to compute the cryptographic primitive, while retaining semantic equivalence.

Masking countermeasures are of particular interest, as they can be proven to be completely blocking side channel attacks, given a chosen measurement model, known as *probing model*. Masking techniques operate combining sensitive values with random, per execution values known as *masks*. The most common combination is performed by adding via Boolean exclusive or $d \geq 1$ random values to the sensitive value itself, and considering the result, together with the d random values, as a redundant representation of the original sensitive value. A protected value is thus also said to be split into $d + 1$ *shares*, which, combined together via Boolean XOR, yield the original value. The masking approach requires to perform the original computation on the $d + 1$ shares of the sensitive value, taking care that the results are also expressed in the same redundant and randomized form. A masking scheme can be thus proven secure against an adversary able to obtain information via side channel on any number of shares up to d of a given sensitive value. This in turn implies that the computation on masked values should never combine together more than d shares of the same value, unless additional fresh randomness is added in the combination process, making the power consumption of the combination once again independent from the combined values. Masking scheme descriptions thus provide both the method to split a value into shares, and the methods to perform computations of arbitrary Boolean functions on share-split values.

While explicit value combinations through operations are avoided in the design of a masking scheme, providing a concrete implementation of the scheme itself which is faithful to this has proven to be challenging. Indeed, setting aside implementation errors, the reuse of shared memory components to store different shares may lead to a violation of the non-combination principle. In dedicated hardware designs, this can be avoided simply by dedicating memory components to the storage of a single share only. Software implementations on the other hand, are forced to employ shared memory resources, e.g. general purpose CPU registers, to store the shares during the computation. This constraint on memory element reuse is known to reduce, or, in critical cases, nullify the security margin of masking schemes. This security reduction, formalized in [4], observes that storing two shares of a given sensitive value in the same memory element, in consecutive clock cycles, will cause a power consumption proportional to the Hamming distance between the two shares, i.e., proportional to the Hamming weight of their xor-combination. In the simple case where $d = 1$, storing the two shares into which the value is split in the same memory element in subsequent clock cycles will transmit on the side channel the Hamming weight of the unprotected value itself. As a consequence of this behaviour, particular attention in avoiding careless architectural register reuse is suggested in [4], to the point of hand-coding the assembly-level description of an algorithm.

While controlling the implementation at an assembly-level description provides the means to a programmer to avoid architectural register reuse, a recent study has shown that significant information leakage is caused by the state transitions of microarchitectural registers [6]. Indeed inter-stage registers in a pipelined processor might leak a combination of values that are unrelated at ISA level, e.g., the result of two consecutive operations performed on orthogonal register sets. Such an addition information leakage can thus invalidate masking scheme implementations which appear to be satisfied at ISA-level, as shown in [32], where the authors detailed a microarchitectural leakage-fragile implementation.

Counteracting microarchitectural leakage can either be done with a hardware approach, e.g. designing an ad-hoc instruction set extension as in [15], or adapting the software to take into account the actual microarchitectural leakage. The second solution is the most flexible in terms of reuse of the existing hardware, but it requires knowledge of the CPU microarchitecture to a point where the microarchitectural leakage can be characterized.

A. CONTRIBUTION

In this work we tackle the problem of characterizing how the CPU and the memory subsystems of an arbitrary superscalar in-order processor leak the processed data due to state change of the memory elements contained in the CPU pipeline and load-store unit. To this end, we propose a microarchitecture-agnostic, ISA-level framework to characterize the power/EM side-channel after inferring the relevant microarchitectural details. The main observation is that the timing side channel provides significant information on the structure of a superscalar CPU, as CPU design aims at performing computation in the least possible amount of time. In lieu of employing timing information to derive the data being processed, as it is common in timing based side channel attack, we employ microbenchmarks with known data and instructions to derive the computation strategy.

Our framework, through the execution of a set of ad-hoc microbenchmarks, and the measurement of software-only parameters such as execution latency, throughput and physical phenomena (i.e., EM emissions), is able to yield a structural characterization of the side-channel features of a target processor and memory subsystem, such as the size and location of all the datapath synchronous registers, and the microarchitectural features necessary to correctly model their leakage behaviors.

We practically demonstrate the soundness of our methodology by applying it to two different processors of the ARM microcontrollers family: a Cortex-M4 and a Cortex-M7. In particular, the Cortex-M4 being a commercially widespread single-issue 4-stage CPU, and the Cortex-M7 a full dual-issue 5-stage superscalar CPU, currently the most powerful CPU of the Cortex-M series.

As an additional confirmation of our characterization work, we demonstrate the feasibility of a CPA attack over an unprotected AES implementation on both platforms, by recovering the correct key with less than 18000 traces. This work represents a first systematic attempt to derive a leakage model that, not only can be practically used to derive CPA-resistant software implementations, but also to gain in-depth knowledge about the causes of the observed leakage, contrarily to data-driven approaches [23], which do not aim at tracking down the causes of the observed leakage phenomena.

II. BACKGROUND

In this section we will start from a recap on the techniques employed to perform a power consumption based side-channel attack, then we recall the relevant microarchiectural pipeline details that will be analyzed in this paper, and sum up the state of the art in side-channel countermeasures and microarchitectural analysis.

A. SIDE-CHANNEL ATTACKS

The computation of hardware and software implementations of cryptographic algorithms, can be considered as an arbitrarily long sequence of states, where the first represents the input of the algorithm and the last, its output. In traditional Complementary Metal-Oxide-Semiconductor (CMOS) based computation, those states are stored as logic values of arrays of memory elements, which embody the sequential part of the processor. Such values accordingly drive the buses and logic lines that feed the combinatorial portion of the CPU. This feature is present in both Application Specific Integrated Circuits (ASIC) and Field Programmable Gate Arrays (FPGA). In the transition between states, we observe an information leakage, on both the power consumption and Electro-Magnetic (EM) emissions side channels, The leakage was characterized to be a change in the power consumption and Electro-Magnetic (EM) emissions correlated with the number of single-bit memory elements changing state [11], [32]. In detail, referring to the leakage categorization by Balasch *et al.* [4], for every transition θ of the set Θ of all the transitions of the target implementation, we can define the sampled leakage function as the sum of a deterministic part $L_{\theta,d}$ and a measurement noise part N_θ: $L_\theta = L_{\theta,d} + N_\theta$. The $L_{\theta,d}$ part itself is correlated with the *transition leakage*, which leaks the values of transitions between states, while the N_θ term takes into account both measurement noise and the contribution of portions of the digital circuit which are not related to the computation at hand.

A typical side channel attack exploits the correlation between $L_{\theta,d}$ and the values involved in the transition θ, considering the transition of a portion of the computing device involving a small part of the secret key. It is possible for the attacker, for each (guessed) value of the part of the secret key, to build a prediction of the value of $L_{\theta,d}$ which would be measured from the device. These prediction must then be checked against the actual behaviour of the device, as it is observed via direct measurement. The best fitting prediction will reveal to the attacker what is the actual value of the key portion.

Since the measurements will obtain the value L_θ, which is affected by noise, statistical comparison tools are employed. In this work, we will adopt the Pearson correlation coefficient; as a consequence the side channel attack performed is denoted as Correlation Power Analysis (CPA). The choice of the Pearson correlation coefficient is backed by the work of Heuser *et al.* [18], which demonstrated that whenever the leakage model is only known to a proportional scale (i.e. $L_{\theta,d} = ax + b$ where both a and b are unknown, and x represents the value being leaked), and the noise is Gaussian, the Pearson coefficient is an optimal distinguisher. Indeed, in our case, the power consumption (and consequent EM emissions) of an memory element is expected to be proportional to the number of elements toggling; i.e. the leakage is proportional to Hamming distance value being leaked.

Operatively, a CPA proceeds as follows. The side channel leakage traces of a large number of encryptions (or decryptions) with random known inputs under the same key are collected. For all the possible values of a small key portion, and for all the cipher inputs, we model the leakage of the value transitions of a portion of the internal state, which can be fully determined by each combination of input and key portion hypothesis. The leakage corresponding to the update operation is modeled as the Hamming distance between the old and the updated value. We compute the Pearson correlation coefficient between the HD leakage model and the captured trace, for each different key portion hypothesis-input pairs. The correct key portion hypothesis will then be the one whose Pearson correlation coefficient value is the highest.

B. STRUCTURE OF A PIPELINED CPU

The computation of a CPU can be logically split in five different execution phases, which in a pipelined architecture become the *stages* of the pipeline itself. The Instruction Fetch (IF) stage loads one or more binary words from the instruction memory (or from the instruction cache if present), and passes them to the Instruction Decode (ID) stage. The ID stage translates each binary opcode into a combination of control signals, which activates the CPU functional units necessary to perform the encoded operation. The ISsue (IS) stage loads the registers from the Register File (RF), checks for eventual hazards and inserts pipeline wait cycles (also called stalls, or bubbles) to solve them. Finally the EXecute (EX) stage performs the actual computation and writes back the computed data to the RF. This last write back operation can be delegated to a fifth optional stage, called WriteBack (WB) [17].

A pipelined architecture needs to store the results of each stage, so that the next stage can read them in the following clock cycle; therefore there exist inter-stage registers between each consecutive pair of stages [17]. We refer to such registers with the names of their adjacent stages, for example an inter-stage register between ISsue and EXecute stages is defined IS/EX. Advanced CPUs might be equipped with

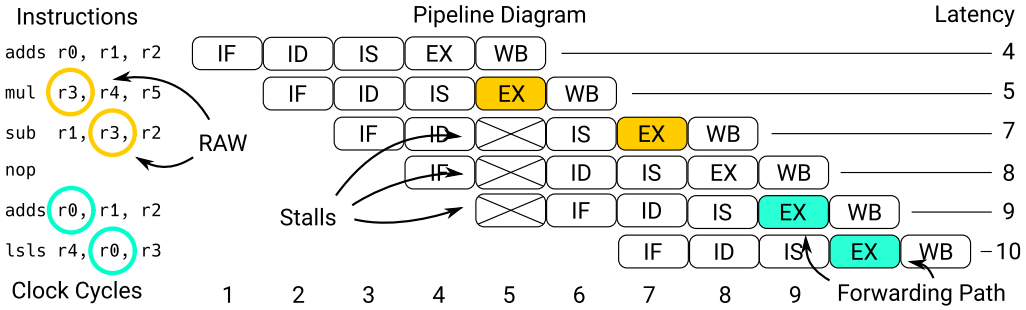


FIGURE 1. Pipeline diagram of a 5-stage in-order CPU executing an instruction sequence reported on the left. The orange highlighting singles out the instructions causing a read-after-write (RAW) conflict, solved by inserting a stall at cycle 5. The teal coloured operations represent a case where a the presence of a forwarding path allows to avoid the insertion of the stall.

several Issue units, which makes them *multiple-issue*, thus able to issue several instructions concurrently [17]. When analyzing a CPU microarchitecture, we need a detailed view of its execution stage, thus we need to distinguish all the functional units that act during this stage. Among them, we have one or more Arithmetic Logic Units (ALU), which are in charge of executing all the algebraic, bitwise and logic manipulations, with some of them equipped with a shifter. The employed shifters usually execute in a single cycle and are built from a matrix of parallel 2×1 multiplexers, called *barrel shifter* [21]. Other relevant functional units are those devoted to memory operations, called Load Unit and Store Unit, together they read and write to a hidden register called Memory Data Register (MDR), which is typically not directly addressable but is implicitly used by all memory operations. Multiplications are usually performed by dedicated multipliers, which will also be analyzed. Analyses concerning floating point ALUs, vector processing and the analysis of other specialized units are left to future explorations due to their comparatively minor role in implementing cryptographic schemes.

Two kinds of conflicts might occur during the execution of a processor, which are named Read-After-Write (RAW) and structural conflicts [17]. RAW conflicts take place whenever a source operand of an instruction is the destination operand of the previous one. Structural conflicts, instead, occur whenever a needed functional unit is still busy with an ongoing operation, for example in dual-issue architectures when there are single execution units in sequences of identical operations. If no such conflicts occur, e.g., if the two instructions are completely independent from one another and they use different execution units, they usually can be issued together. Instead, if a RAW hazard is present, the dependent instruction should be stalled [17], as it happens in the following ARM assembler snippet, where two instructions cannot be dual-issued because they contain a RAW hazard:

```
DIV r0, r1, r2
ADD r3, r0, r0
```

Each conflict is solved through the insertion of *stalls* [17], which are cycles in which some of the pipeline stages are stopped to let the blocking operation complete its transit through the pipeline. Regarding RAW conflicts, another way to solve them is through the use of forwarding paths [17], which are connections between inter-stage registers that allow to forward output values without storing and retrieving them from the RF. Instead, to limit the insurgence of structural hazards, CPU designers usually add replicas of the various functional units; this is especially beneficial in multiple-issue processors. To maximize their power efficiency, CPUs are designed to run at their maximum capabilities. Therefore, it is reasonable to expect that, whenever no structural or data conflicts take place, the CPU computes all the instructions, multiple-issuing them in the best, i.e., lowest, number of Cycles per Instruction (CPI) [28]. A direct consequence of this fact is that CPI values are a source of information on the inner architecture of a CPU; in particular a decrease in the CPI reveals a conflict. A sample pipelined execution is depicted in Fig. 1, where we can observe from the figure that a RAW data dependency exists between the second and the third instructions, which is resolved by adding a stall on the fifth clock cycle. Furthermore, we can notice that a forwarding path between the fifth and sixth instruction avoids the insertion of an additional stall cycle at clock cycle 10. One additional optimization that can be found in more complex CPUs, is the adoption of pipelined execution units. This optimization is applied on multi-cycle execution units, typically Multiply-And-Accumulate (MAC) or Load/Store Units (LSU). A pipelined unit applies pipelining internally, thus is internally divided into *stages*, one for each execution cycle, and whenever an ongoing computation is passed to the next stage, the current one is able to start a new computation. This approach is used to maintain a sustained throughput of 1 CPI when executing a sequence of multi-cycle operations.

C. EFFECTS OF MICROARCHITECTURAL LEAKAGE ON MASKING PROTECTIONS

Hidden microarchitectural registers pose a threat to software ciphers employing masking CPA countermeasures, as mentioned in Section I, as they can leak a value which is correlated to multiple shares, thus violating the assumption of the masking schemes, as demonstrated by Seuschek *et al.* [32].

We provide a demonstration of the aforementioned case through two running examples of how a microarchitectural leakage on the registers of a XOR operation can be used to compute the original unmasked value, de-facto invalidating the countermeasure. Given two values a and b, each represented using a first order masking scheme, a is split into two shares by using a random value x: $a = (a_0 \oplus a_1), a_0 = (a \oplus x), a_1 = x$, and b is split into two shares by using a random value y: $b = (b_0 \oplus b_1), b_0 = (b \oplus y), b_1 = y$. Their masked XOR operation $c = a \oplus b$ can be computed as $c_0 = a_0 \oplus b_0$ and $c_1 = a_1 \oplus b_1$. The splitting of values into shares can be implemented using the following assembly snippet:

```
XOR a_0, a, x
XOR b_0, b, y
```

If an undisclosed microarchitectural register leaks the linear combination of the subsequent values assigned to the first source register, we would have an inadvertent leakage of the result $c = a \oplus b$. Instead, consider a second assembly snippet, which implements the masked XOR operation that was just described:

```
XOR c_0, a_0, b_0
XOR c_1, a_1, b_1
```

If additionally the microarchitecture leaks the linear combination of subsequent values assigned to the destination registers, we would see a leakage correlated with the unmasked XOR value c as: $c_0 \oplus c_1 = (a_0 \oplus b_0) \oplus (a_1 \oplus b_1) = a \oplus b = c$.

D. RELATED WORK

Since the first pioneering work by Kocher [19], the state of the art in side-channel attack analysis and countermeasures has evolved into different approaches; many of them relying on the creation of leakage models, which can be either opaque, such as the ones generated by machine learning approaches, or explainable, as are the models generated from physical simulation and architectural reverse-engineering. Opaque models can be useful to shuffle the code until a reduced-leakage implementation is found. Instead, explainable models could be used to extend compilers to generate protected code. Orthogonally, the platforms on which such techniques are applied range from being completely known (white-box), like RISC-V compliant open-hardware processors, to being opaque (black-box), like commercial CPUs.

A first notable work to be mentioned, is set in a complete white-box scenario where the CPU is known down to the gate-level description; Sehatbakhsh *et al.*, with their EMSim tool [31], start from a gate-level model of a custom designed RISC-V processor and build a synthetic EM leakage trace generator, able of approximating the real traces of the same CPU, synthesized on an FPGA. Their methodology allows interesting developments, such as software-only verification of leakage in cryptographic algorithms or in the device validation. However such a tool is applicable only when a gate-level description of the target CPU is available, which is not the case of most commercial proprietary processors, like the two we analyse in this work. When access to a CPU gate-level description is available, further countermeasures can be directly embedded into the cores.

Gao *et al.* with their FENL [15] approach, propose an Instruction Set Extension to expose some low-level features of the microarchitecture to the compiler or even to the programmer. For example, they introduce the ability of flushing the pipeline, thus enabling compilers to isolate the computation of the shares from one another by issuing a pipeline flush in between them. Although interesting to consider for future architectures, this is still not applicable to the presently available commercial processors.

Whenever a gate-level description is unavailable, a data-driven approach can be applied, employing a machine learning model to simulate the leakage characteristics of the target architecture. McCann et. al. through their ELMO [23] framework, were able to model the complete microarchitectural leakage using machine learning tools; Shelton et. al., with the ROSITA [33] extensions to ELMO, bring the effort forward, by introducing automatic static rewriting of the cryptographic code to eliminate leakage. Both work however target a simple CPU: the ARM Cortex-M0, for which a detailed microarchitectural model is publicly available [13]. This makes both tools hardly portable to other CPUs whose microarchitecture is unknown. Furthermore, conversely to their data-driven approach, our work aims at producing an explainable model, helpful in assisting security engineers in a transparent and informed way, by overcoming the limitations of randomly changing the code until it stops leaking.

As opposed to a data-driven model, a microarchitectural approach allows for a more structured countermeasure design. Bronchain and Standaert [7] state the necessity of relying on higher-order masking implementations for protecting software ciphers on COTS hardware, and highlight the importance of investigating the actual security level which can be reached by using low-order masking. This security level can be inspected only through a thorough microarchitectural characterization. Furthermore they perform a successful attack on a masked and shuffled implementation published by Agencie Nationale de la Sécurité des Systèmes d'Information (ANSSI) in less than 2000 measurements, furtherly demonstrating the difficulty of implementing secure side-channel countermeasures.

Seuschek *et al.* [32] confirm the importance of considering the characteristics of the underlying architecture. In fact they show how a masked implementation that does not consider the microarchitectural transition leakage is ineffective against an attacker with a detailed knowledge of the architecture. They provide microarchitectural-aware instruction scheduling and register allocation passes, which try to generate code which emits a reduced leakage, upon execution on the analyzed processor. Their leakage model however is still

tailored to the Cortex-M0 and is generated through manual inspection. Thus porting their implementation to a different, eventually more complex CPU, would require a complete manual inspection effort, in some cases simply not possible due to the lack of microarchitectural knowledge.

Chen *et al.* [10] describe what features should be included in a microarchitectural model. They start describing the components of the processor datapath that can generate side-channel leakage: a path from the RF to the Execution Unit, and back into the RF, used for all the arithmetic operations, bitwise and logic manipulations; a second one from the RF to the memory subsystem, used for store operations; and a last one from the memory subsystem to the RF used for loads. Chen et. al then design a custom architecture, that for certain operations, could apply a hiding countermeasure. We strive instead to build a solution for existing COTS platforms.

Barenghi and Pelosi [6] made further progress on the microarchitectural side-channel exploration path, discovering that other features of the CPU are relevant on the side-channels perspective, encompassing the RF read ports, the IS/EX inter-stage register, an eventual inter-stage register between the EXecute, and the WriteBack to the RF (EX/WB), the ALU output registers, the barrel shifter output registers, and the MDR. Finally they consider the LSU to be equipped with eventual registers used to perform eventual sub-word realignments. They apply a CPI-based characterization technique, which will be formalized and expanded in this work.

Finally, whenever an explainable model has been derived, conversely to the data-driven ones, it enables the informed design of software countermeasures. As Agosta *et al.* summarized in their work [1], different compiler-based techniques can be applied to the source code of a cryptographic implementation. A compiler with a detailed knowledge of the leakage of an architecture, such as the knowledge we are deriving in this work, will be able to directly schedule the code in a protected way, without resorting to a manual, thus long and error-prone hand-coding process. Compiler-generated countermeasures could help in protecting cryptographic code, implementing software countermeasures on each of the diverse architectures that are used today in the embedded ecosystem.

III. MICROARCHITECTURAL MODEL EXTRACTION

In this section, we describe our method to obtain a microarchitectural model of the side channel relevant features of an in-order CPU. To this end, we consider the portion of the CPU performing data-dependent activities, i.e., the CPU datapath. Providing a model of the CPU datapath useful to design software side channel countermeasures requires to model the behavior of the synchronous components in the datapath itself.

We consider the case of an in-order CPU, having, as synchronous components of its datapath: *i)* Register File (RF), *ii)* the inter-stage pipeline registers, *iii)* the Memory Address Register (MAR) and Memory Data Register (MDR) contained in the Load-Store Unit (LSU), *iv)* the store buffer, and, *v)* the inter-stage registers of pipelined execution units, if such units are present.

To derive the behavior of the synchronous components of the datapath, we classify all the microarchitectural design choices impacting on them in three categories: *i) data storage features*, including the number of register file read and write ports, *rp* and *wp* respectively, *ii) instruction issuing features*, including the issue width w, the issuing policy, the width of the inter-stage registers, and the presence of forwarding paths, and *iii) instruction execution features* including the number of available execution units, their latencies, and whether they are internally pipelined or not.

More in detail, RF updates are determined by the *data storage* microarchitectural features, since the number of RF ports shapes the updates to the Register File; pipeline inter-stage registers updates are determined by the *instruction issuing* characteristics, such as the issue width w and the issuing policy, which is non-trivial for architectures with $w > 1$, and forwarding paths, which are able to update an inter-stage register with the content of a following inter-stage register. Updates to the LSU synchronous registers, the store buffer and execution unit inter-stage registers, are all determined by the ISA semantics and by the *instruction execution* microarchitectural features. In particular, the presence of unit inter-stage registers is implied by the presence of pipelined execution units. The observation of resource contention allows to infer sharing of synchronous datapath registers among different functional units, which can cause unforeseen transition leakage.

In this work, we assume the target CPU ISA and the number of stages of the pipeline are known, as it is the case with the entire line of ARM Cortex-M CPU designs. For the sake of description clarity, as a case study, we consider a 5-stage pipeline design: Instruction Fetch (IF), Instruction Decode (ID), ISsue (IS), EXecute (EX), WriteBack (WB). If the pipeline length is unknown, an exhaustive search is always within practical reach, since the longest pipeline in a commercial CPU has 31 stages (Intel Pentium 4 Prescott), and architecture independent works demonstrate how shorter pipelines are optimal on modern workloads [16].

A summary of the approach which we will describe in the following is depicted in Figure 2. We assume the target architecture is equipped with the means of count the clock cycles taken by the execution of an arbitrary instruction sequence. This is typically available in the form of a memory mapped, clock cycle counter register which can be read at the user's will. We note that, if this feature is not available, any other means of precisely measuring the number of cycles taken by the execution of an instruction sequence (e.g., asserting and deasserting fast GPIO pins) can be used. We also assume either the knowledge, or an educated guess, on the architecture issue width w. In particular, our approach allows to validate the value of the issue width w before the rest of the microarchitectural inferences are performed.

Once the microarchitectural inferences are complete, we employ a microbenchmark suite to validate that a side

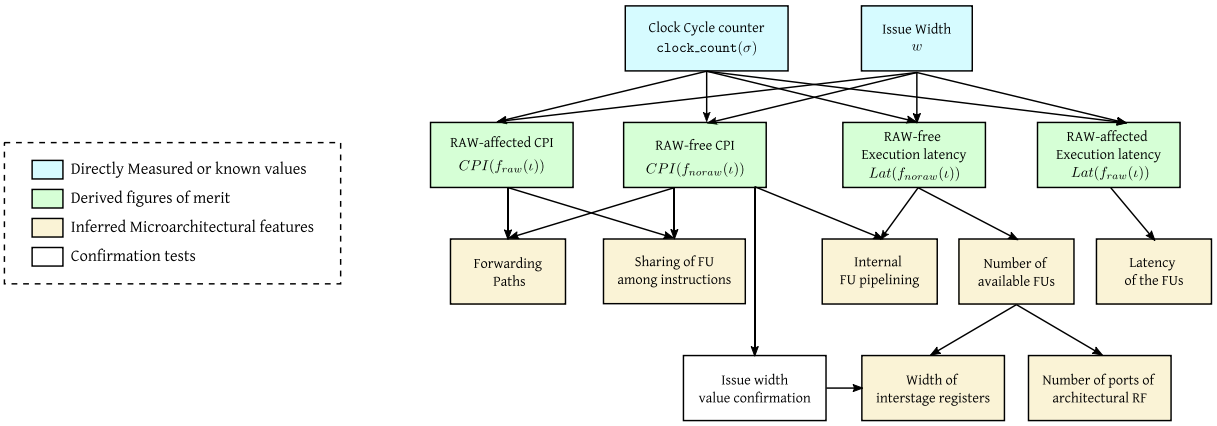


FIGURE 2. Dependency graph showing the proposed flow for deriving all the elicited microarchitectural features. An arrow pointing from one feature to the other implies that the former is used to compute the latter. The numbering of the blocks represents the topological ordering that was followed in this work to extract all the microarchitectural features.

channel information leakage matching the sequential elements we individuate is actually present.

We note that our approach, with its black-box nature, can yield a coherent description of the EM leakage, fitting well the scenario of commercial CPUs where the precise design specifications are not available. If, by contrast, the HDL description of the CPU is available a method to backtrack the side channel information leakage up to the single logic lines was described by Buhan *et al.* [8].

A. DEFINITIONS

To infer a microarchitectural model we will measure the CPU performance during the execution of tuples of instructions taken from its ISA, employed as microbenchmarks. However, performing an exhaustive exploration of all the instruction sequences that can fill up the pipeline would prove extremely time consuming. To perform a more efficient analysis, we build our microbenchmarks out of a subset $\mathbf{I}$ of the ISA, containing *representative instructions*.

Definition 1 (Representative Instruction Set, and Representative Instruction Sequence): Let $\mathbf{I}$ be a subset of the ISA, chosen to have one instruction for each functional unit of the processor. We define a representative instruction sequence ι as an arbitrary length sequence of representative instructions $\iota = \langle i_0, \ldots, i_{n-1} \rangle$ with $n \in \mathbb{N}$, where $\forall j \in 0, \ldots n, i_j \in \mathbf{I}$. We denote as $\mathbf{I}^l$ is the set of all the length-l instruction sequences, and as $\mathbf{I}^*$ is set of all the arbitrary length instruction sequences.

We expect all the instructions in the same functional unit to exhibit a similar side-channel behavior, as confirmed for the RISC-V architecture by Sehatbakhsh *et al.* [31]. To this end, we pick a single representative out of the following instruction families: no-operation, move instructions, signed addition with and without immediate, addition with shifting, multiplication, logical shift, load and store operations. Floating point instructions, which can be characterized by employing the same approach, are out of scope for this work, due to their relatively lower occurrence in cryptographic software.

Before a representative instruction sequence ι can be employed as a microbenchmark, its elements need to be specialized to act on actual data registers. Denoting the available general purpose registers as integers in the set $\{0, \ldots, \texttt{max\_reg}\}$, and assuming for the sake of simplicity that all general purpose registers can be used both as source and as destination of an instruction, we define the set of specialized instructions as follows:

Definition 2 (Specialized Instruction Set, and Specialized Instruction Sequence): Let $i \in \mathbf{I}$ be an instruction, we denote as $\texttt{src\_reg}_i$ the amount of its source registers and as $\texttt{dst\_reg}_i$ the amount of its destination registers. The destination registers and source registers of an instruction can be represented as tuples of register indices. The set of specialized instructions $\mathbf{S}$ is the set of all the tuples $s = \langle i, dr_s, sr_s \rangle$, having $i \in \mathbf{I}$, $dr_s \in \{0, \ldots, \texttt{max\_reg}\}^{\texttt{dst\_reg}_i}$, $sr_s \in \{0, \ldots, \texttt{max\_reg}\}^{\texttt{src\_reg}_i}$.

We follow the same convention adopted for the representative instruction sequences, denoting as $\mathbf{S}^*$ the set of all the arbitrarily long sequences of specialized instructions $\sigma \in \mathbf{S}^*$, where $\sigma = \langle s_0, \ldots, s_n \rangle, n \in \mathbb{N}, s_i \in \mathbf{S}$. Similarly, we denote as $\mathbf{S}^l, l \in \mathbb{N}$, the set of all the length-l sequences of specialized instructions.

Both in the case of representative instruction sequences and specialized instruction sequences, we indicate with the juxtaposition of the symbols, the concatenation of two sequences, e.g., $\sigma_0\sigma_1$ denotes the concatenation of the instruction sequences σ_0 and σ_1.

Translating a representative instruction sequence into a specialized one will determine whether or not most structural pipeline conflicts take place. Among the possible causes, one is easily controlled during the microbenchmark design, that is, the Read-After-Write (RAW) conflicts. A RAW conflict takes place whenever, in a specialized instruction sequence, an instruction has among its source registers at least one of the destination registers of a previous instruction. To this end, we define a logical predicate $\text{RAW}(\cdot,\cdot) : \mathbf{S} \times \mathbf{S} \rightarrow \{\texttt{true}, \texttt{false}\}$ over two specialized instructions, which is

true if and only if the two instructions have a RAW conflict, considering the second specialized instruction as executed after the first. We thus have that the RAW predicate is true on two specialized instructions s, r if and only if $\mathrm{RAW}(s, r) := dr_s \cap sr_r \neq \emptyset$.

An *instruction specialization function* $f(\cdot)$ is a function defined over the free monoid of representative instructions, which maps instruction sequences to the specialized instruction sequences. Let $\iota = \langle i_0, \ldots, i_l \rangle, \iota \in \mathbf{I}^*$ be a representative instruction sequence, i_j will be its j-th instruction; function $f(\cdot)$ will map ι to a specialized instruction sequence $\sigma \in \mathbf{S}^*$,i.e., an instruction sequences acting on defined architectural registers, where each member $s_j = \langle i_j, \cdot, \cdot \rangle$ is a specialization of the corresponding instruction i_j. To model the architectural behaviour in case of presence or absence of RAW conflicts, we define two different instruction specialization functions.

Definition 3 (Instruction Specialization Functions): $f_{raw}(\cdot) : \mathbf{I}^* \rightarrow \mathbf{S}^*$ is an instruction specialization function, defined over the free monoid of representative instructions, which maps each representative instruction sequence to a specialized sequence, such that the RAW predicate holds on each pair of consecutive instructions. $f_{noraw}(\cdot) : \mathbf{I}^* \rightarrow \mathbf{S}^*$ is an instruction specialization function, which maps to every instruction sequence, a specialized instruction sequence in which the RAW predicate never holds when evaluated on ordered instruction pairs.

To impose on each sequence of specialized instructions $\sigma = \langle s_0, \ldots, s_j \rangle$ a RAW conflict, every member of the tuple $s_j = \langle i, dr_s, sr_s \rangle, i \in \mathbf{I}, dr_s \in \{0, \ldots, r_{max}\}^{dr_{max}}, sr_s \in \{0, \ldots, r_{max}\}^{sr_{max}}$ must have at least one output register operand $dr_{max} > 0$, and at least one input register operand $sr_{max} > 0$. For tuples which contain instructions where this condition does not hold, the $f_{raw}(\cdot)$ function is not defined. Where $f_{raw}(\cdot)$ is defined, we choose a set of register indices such that one of the output registers of an instruction s_j is also one of the source registers of the following instruction s_{j+1} in the sequence; the resulting register selection causes a RAW conflict between all the consecutive instructions. Let $\iota \in \mathbf{I}^*$, be a representative instruction sequence, given $\sigma = f(\iota) = \langle s_0, \ldots, s_l \rangle$ as its specialized version, if the RAW predicate $RAW(s_j, s_{j+1})$ holds for $j \in \{0, \ldots, l-1\}$, the specialization function can be denoted as $f_{raw}(\cdot)$.

B. MEASURING LATENCY AND DERIVING CPI

In the following, we describe how we exploit the availability of a clock cycle measurement facility and the knowledge of the issue width w (cyan elements in Figure 2) to derive the values of the clock Cycles per Instruction (CPI) $\mathtt{CPI}(\sigma)$ and the execution stage latency $Lat(\sigma)$, for a given specialized instruction sequence σ (green elements in Figure 2). Our strategy is tolerant to possible errors in the known value of w, which can be operatively corrected when the values for $\mathtt{CPI}(\sigma)$ are obtained.

For the sake of explanation, we consider the case of the clock cycle counter of ARM Cortex-M microcontrollers,

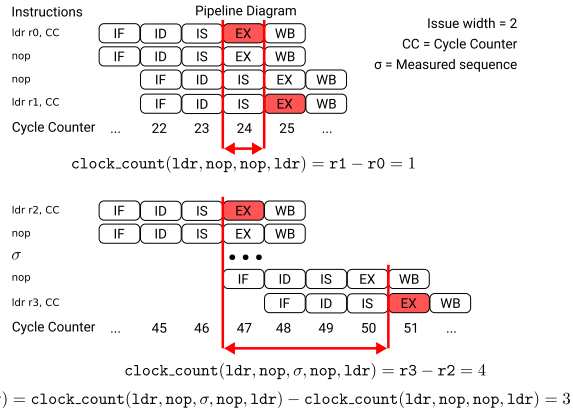


FIGURE 3. Execution latency measuring process in a dual issue ARM architecture. The cycle counter is loaded from the symbolic memory location `CC`, cycle intervals are computed by subtracting subsequent cycle counter values, and the execution cycles of a target instruction sequence σ are measured by subtracting two cycle intervals, one with an empty sequence in between, and another with σ in between.

available as a memory mapped register, which can be read by means of a load (`ldr`) instruction. The net effect of the `ldr` instruction is to store in the target register the value of the clock cycle counter when the `ldr` instruction is in the EX pipeline stage.

We denote as $\mathtt{clock\_count}(\varsigma)$, with $\varsigma = \langle f_{noraw}(\mathtt{ldr}), \sigma, f_{noraw}(\mathtt{ldr}) \rangle$ the result of the difference between the clock cycle counter value read by the latter `ldr` instruction in ς and the one read in the former `ldr` instruction in ς. In single-issue architectures $\mathtt{clock\_count}(\varsigma)$ allows an easy derivation of $Lat(\sigma)$, subtracting the execution latency of the clock cycle counting instruction itself $Lat(f_{noraw}(\mathtt{ldr}))$, obtained as $\mathtt{clock\_count}(f_{noraw}(\mathtt{ldr}, \mathtt{ldr}))$.

Multiple issue architectures require a slightly more complex procedure to obtain the correct value of $Lat(\sigma)$, since the clock cycle counter readout instruction can be multiple-issued together with the first instructions composing σ. If this happens, the value of $\mathtt{clock\_count}(\varsigma)$ will differ in a non-trivial way from $Lat(\sigma)$, since the last clock cycle readout will be performed in parallel to the execution of the last instruction(s) of σ. To handle this issue, we exploit the knowledge of the issue width w of the underlying architecture, and pair the clock cycle readout instructions with $w - 1$ instructions which can be multiple issued together with it. These instructions act as a filler for the unused issue lanes, effectively placing the clock readouts in the appropriate cycle in time. As an example, we report in Figure 3 the case of a dual issue ($w = 2$) architecture. The example assumes that the clock-cycle readout instruction `ldr` takes a single cycle to be executed, and can be dual-issued with others. Note that in this case $\mathtt{clock\_count}(f_{noraw}(\mathtt{ldr}, i_0, i_1, \mathtt{ldr})$, where i_0 and i_1 are two instructions which can be dual issued with `ldr` would match $\mathtt{clock\_count}(f_{noraw}(\mathtt{ldr}, i_0, \mathtt{ldr}))$, since the second clock readout would be performed in parallel to the execution of i_1. In this case, the `ldr` instructions employed to read the clock cycle counter are padded with $w - 1 = 1$ instructions (`nops` in the example) to fill the

remaining issue lanes. We thus obtain $Lat(f_{noraw}(i_0, i_1))$ as $\texttt{clock\_count}(f_{noraw}(\texttt{ldr, nop}), \sigma, f_{noraw}(\texttt{nop, ldr}))-\texttt{clock\_count}(f_{noraw}(\texttt{ldr, nop}), f_{noraw}(\texttt{nop, ldr}))$

1) MEASURING $\texttt{CPI}(\sigma)$

Given the correct value of w and $Lat(\sigma)$ computing $\texttt{CPI}(\sigma)$ can be achieved applying the definition, i.e., $\texttt{CPI}(\sigma) = \frac{Lat(\sigma)}{\texttt{length}(\sigma)}$.

We observe that, even in the case where the value of w is unknown, it is still possible to measure $\texttt{CPI}(\sigma)$, and, as a consequence obtain a sound estimate for the issue width of the architecture at hand, when considering instructions from the relevant instruction set $\mathbf{I}$. Indeed, in a generic w-issue CPU, each cycle counting instruction may be concurrently-issued with at most $w-1$ instructions of σ, thus preventing the straightforward elimination of measurements error in $Lat(\sigma)$ as when w is known. If the clock cycle counting instruction $s_{\texttt{count}}$ is issued with the first $w-1$ instructions of the sequence to be measured, the actual counter readout will be performed after the clock cycle counting instruction will be executed, by subtracting its execution latency from $\texttt{clock\_count}(\sigma)$. A symmetric condition takes place whenever the clock cycle counting istruction executed at the end of the sequence is multiple-issued with the last instructions of σ. In this case, the measured latency of the last w-wide instruction bundle where $s_{\texttt{count}}$ is issued is reduced to $Lat(s_{\texttt{count}})$. As a result, the value of $\texttt{clock\_count}(\sigma)$ may be smaller than $Lat(\sigma)$ by an amount upper limited by $s_{\texttt{count}} + (\ell - s_{\texttt{count}}) = \ell$, where ℓ is the execution latency of the slowest instruction of the architecture. Since the maximum reasonable precision for the $\texttt{CPI}(\sigma)$ figure is $\frac{1}{w}$ (as no smaller fraction can be achieved by a w-issue architecture), a reliable measure of $\texttt{CPI}(\sigma)$ can be obtained as:

$$\texttt{CPI}(\sigma) = \text{RoundNearest}\left(\frac{\texttt{clock\_count}(\sigma^{w\ell})}{w\ell \cdot \texttt{length}(\sigma)}, \frac{1}{w}\right)$$

Note that, when the issue width w is not known a priori, employing a suitable upper bound on its value, w', results in obtaining CPI values which are rounded to a fraction $\frac{1}{w'} < \frac{1}{w}$. Given this fact, it is possible to derive the correct value of the issue width, or verify the one being known, finding the smallest value taken by the CPI over all the relevant instruction sequences executed with no RAW dependencies, i.e. $\min_{\iota \in \mathbf{I}^{w'}} \texttt{CPI}(f_{noraw}(\iota))$, and derive w as:

$$w = \left\lceil \frac{1}{\min_{\iota \in \mathbf{I}^{w'}} \texttt{CPI}(f_{noraw}(\iota))} \right\rceil$$

C. MICROARCHITECTURAL FEATURE INFERENCE

We now describe how to extract the set of microarchitectural features allowing to build a side channel leakage model (orange elements in Figure 2). To this end, we need to identify which synchronous elements compose the datapath, and how their contents change during a computation.

We derive the aforementioned features by examining the values of $\texttt{CPI}(f_{noraw}(\iota))$ and $\texttt{CPI}(f_{raw}(\iota))$, for all the admissible length-w representative instruction sequences ι, and the values $Lat(f_{noraw}(\iota))$, for all the representative instructions i.

1) FORWARDING PATHS

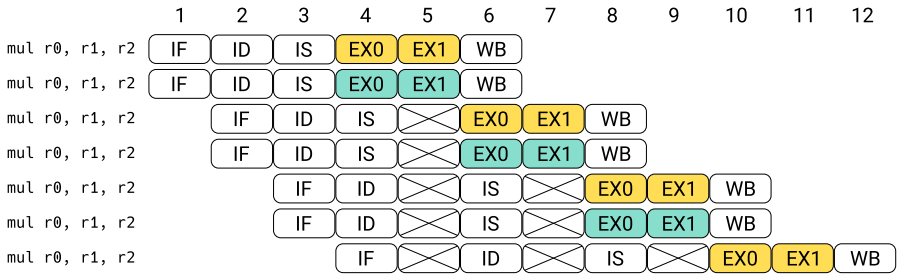
We derive the presence of a forwarding path from the output of the EX stage or the MEMory (MEM) stage to the input to the EX stage, by comparing the execution latency of $2w$ long sequences of representative instructions, constituted by the concatenation of two instances of $\iota = \langle i_0, \texttt{nop}^{w-1}\rangle$, where i_0 is either a computational or a load-store instruction, depending on the nature of the forwarding path to be investigated. The reason for inserting a $w-1$ nop padding sequence between the two representative instructions, is to simplify the forwarding path analysis, uniforming the behavior of a multiple-issue architecture to the one of a single-issue one. Comparing the execution latency of $\sigma_{noraw} = \texttt{CPI}(f_{noraw}(\iota^2))$ and $\sigma_{raw} = \texttt{CPI}(f_{raw}(\iota^2))$, we detect the presence of a forwarding path in case the two execution latencies match; instead, in case $Lat(\sigma_{raw})$ is higher than $Lat(\sigma_{noraw})$, we detect the insertion of pipeline stalls as a result of the absence of the forwarding paths.

2) NUMBER OF AVAILABLE FUs, THEIR LATENCY AND THEIR PIPELINING

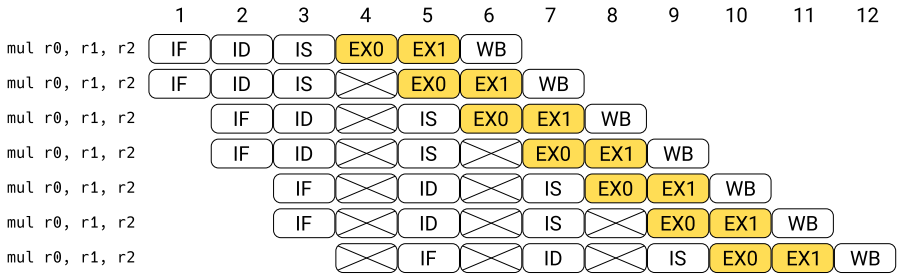
The number of functional units present for a given operation will not exceed the one which can be employed when the w issue architecture is actually issuing w concurrent instructions. Indeed, any extra FU would simply be unused, even under the best possible computation conditions, leading to a waste of resources. As a consequence, we detect the maximum number of functional units for a given operation from the execution latency of representative instruction sequences obtained from $1 \leq l \leq w$ repetitions of a single representative instruction without RAW hazards, $\sigma = f_{noraw}(i^l \texttt{nop}^{w-l})$. The padding with nop instructions is added in order to saturate the remaining lanes of the issue unit, assuming that the nop operation is found by the instruction issuing features investigation to be amenable to multiple issuing with any other instruction. We specialize the said representative instruction sequences, so that no RAW conflicts are present, in order to prevent data-hazards from allowing the simultaneous issuing of instructions on the available functional units. The execution latency $Lat(\sigma)$ of the sequence is expected to increase when the number of representative instruction repetitions l rises above the available functional units. Therefore, we determine the number of functional units for the said instruction as the maximum value l for which $Lat(f_{noraw}(i^l \texttt{nop}^{w-l})) = Lat(f_{noraw}(\langle i, \texttt{nop}^{w-1}\rangle))$ holds.

The latency of a given FU is simply detected as either $Lat(f_{noraw}(\langle i \rangle))$ or $Lat(f_{raw}(\langle i \rangle))$, where i is a relevant instruction computed by the FU under analysis.

In order to fully characterize the FUs themselves, beyond their latency, we should ascertain their design as either combinatorial circuits taking more than one clock cycle to stabilize, or pipelined circuits. To do so, we observe whether



(a) Two combinatorial multi-cycle multipliers



(b) Single pipelined multiplier

FIGURE 4. Pipeline timeline schemes of a dual non-pipelined and a single fully-pipelined multi-cycle multiplicative execution units. Although both implementations achieve a CPI value of 1 on a sequence of `mul` instructions, the execution latency allows to tell the designs apart. In this scheme the dual-issue architecture is equipped with EX-EX forwarding paths.

$\texttt{CPI}(f_{noraw}(\langle i^l \rangle))$ equals $\frac{1}{\texttt{number\_of\_FUs}}$, in which case we deem the FUs to have a pipelined design, or whether it exceeds the said value, in which case the unit design is deemed to be combinatorial. Indeed, in order to be able to sustain a CPI of $\frac{1}{\texttt{number\_of\_FUs}}$, the functional units must be designed with synchronous elements, so as to be able to accept a fresh input set at each clock cycle. Exploiting the combination of the execution latency and CPI metrics allows us to distinguish clearly between two possible cases, which would be impossible to tell apart from the CPI metric alone. As a running example, consider the two different implementation choices shown in Fig. 4 to achieve a unitary CPI on a sequence of multiplication instructions. This can be achieved with either two combinatorial multipliers, taking two clock cycles each, or a single pipelined multiplier. However, the single pipelined multiplier design will exhibit an increase in the execution latency when the number of multiplications in a sequence $\iota = \texttt{mul}^l\texttt{nop}^{2-l}$ increases from one to two, therefore allowing us to detect the presence of a single multiplier. Combining this inference with the unitary CPI, we are able to deem the unit as pipelined, as expected.

3) FU CONTENTION AMONG DIFFERENT INSTRUCTIONS

The last microarchitectural feature to be analyzed in the instruction execution feature category is the potential FU resource contention by different representative instructions. While it is usually possible to cluster representative instructions by the functional unit expected to compute them all, it is possible for different FU to share resources. The typical case is the one of a Load-Store Unit (LSU) sharing the use of the adder circuit to compute destination addresses with the Arithmetic-Logic Unit (ALU). We detect this microarchitectural resource sharing, which has the potential to cause data serialization in the pipeline interstage stage buffers as it causes a structural hazard. Therefore, considering two representative instructions i_0, i_1, belonging to different FUs, we detect a resource sharing whenever the said instructions cannot be multiple-issued together, despite the absence of RAW conflicts. This can be inferred comparing $\texttt{CPI}(f_{noraw}(\langle i_0, i_1 \rangle))$ and $\texttt{CPI}(f_{raw}(\langle i_0, i_1 \rangle))$: if no multiple issuing is possible, the two figures will match; instead if dual issuing is possible, the CPI obtained when no RAW conflicts are present will be lower.

4) NUMBER OF ARCHITECTURAL RF PORTS AND INTERSTAGE REGISTER WIDTHS

The minimum width of inter-stage registers is bound by the issue width w of the architecture and the operand requirements of the FU present in the architecture. In particular, the Issue to Execute (IS/EX) inter-stage register must be wide enough to accommodate all the data of the most demanding instruction bundle in terms of inputs. The inter-stage register between the Execute and WriteBack stages (EX/WB) the contents to be written back into the register file. As a consequence, the number of architectural words stored in each interstage register can be determined finding the maximum

amount of read operands and written operands among all the possible instruction bundles issued simultaneously. We note that the maximum amount of read/written registers may be achieved in an instruction bundle with less than w instructions, in case of particularly demanding ones such as the fused multiply and accumulate instruction.

Register File (RF) ports are an expensive resource, and thus tend to be designed in a number which is the strict minimum required to exploit the FU to their utmost. We therefore determine the number of read ports and write ports of the RF by counting the maximum number architecture words which should be read from and written to the architectural register file during the computation of an instruction bundle.

D. EVALUATING THE LEAKAGE OF THE SYNCHRONOUS COMPONENTS

After describing our microarchitectural inference procedure, we now provide the means to cross-validate the inferred synchronous components and their behaviour by means of a power consumption/EM emissions side channel analysis. To do this, we turn on its head the common power consumption/EM emissions side channel attacks, where the architecture is assumed to be known, and the processed data are inferred. Indeed, through the use of carefully crafted benchmarks, where the instructions and data being processed are known, we validate the presence and behaviour of the synchronous components of the CPU.

To this end, we consider the information leakage on the power consumption and EM emissions side channels caused by the switching activity of the synchronous components. Such an information leakage is essentially proportional to the amount of single-bit synchronous memory elements switching in a given clock cycle. We therefore design microbenchmarks crafting stimuli which elicit the switching activity of a single, or a small group, of synchronous elements. We note that this can be achieved measuring the behaviour of the device in a specific time interval, and excluding the eventual instructions required to prepare the pipeline in the desired state. To detect if the side channel behaviour of the device matches the expectations suggested by a given synchronous component behaviour, we compare a sequence of predicted side channel behaviours, obtained varying the input data in a fixed instruction sequence with the actual measurements. Such an evaluation method exploits the theoretical result of modeling optimality reported in [18], where the authors state that, under the hypothesis that the component being modeled has a leakage proportional to its switching activity, employing the Pearson correlation coefficient between the toggle-count and the measured power consumption allows to extract the full information concerning how well the consumption is modeled. We now describe in detail the microbenchmarks for the specific components.

1) REGISTER FILE

To obtain a leakage behaviour related to the switching activity of the register file, we fill, outside of the side-channel measured computation, two registers with input values, and we measure the power consumption of the entire lifecycle of a `mov` instruction, through the pipeline. The said `mov` instruction is expected to generate a leakage proportional to the Hamming distance between the contents of the two registers, due to the switching activity of the write port, and the destination register. Such a leakage is expected to be present in the clock cycle when the actual write operation takes place. We note that this microbenchmark also elicits side channel leakage from the interstage registers, therefore its results should be analyzed together with them to single out the register file activity.

2) IS/EX PIPELINE REGISTER

The IS/EX inter-stage register is overwritten by the input values of the instructions issued in each slot. To single out its activity, we load $2w$ distinct inputs in the architectural register file. We then prepare the pipeline state copying the said $2w$ values into $2w$ destination registers distinct from both the source ones, and among themselves. This operation is performed so that no switching activity in the architectural register file will take place when measuring the one of the IS/EX pipeline register. We subsequently clear the interstage registers with a sequence of `nop` instructions and measure the switching activity of the pipeline re-executing the $2w$ move instructions. We employ as power model the Hamming distance of the values being stored in the IS/EX inter-stage register, that is, we compute the Hamming distance between the source operands of moves being issued in the same slot of the two w wide move instruction bundles.

3) EX/WB PIPELINE REGISTER

The EX/WB inter-stage register is updated with the destination register values of all the retired instructions. We expect this buffer to leak the HD of the computation results of instruction pairs taken from two consecutive sequences of $2w$ instructions. Our leakage testing strategy for the EX/WB interstage registers follows an approach similar to the one of the IS/EX register with a single difference. To reduce the potential unintended, and matching leakage of the IS/EX register, we employ a sequence of instructions where the result differs from the operands. In particular, in our tests, we employ additions, picking randomly both addends as our representative instructions for this test. Similarly to the strategy for the IS/EX leakage evaluation, we precharge the contents of both the source and destination registers in the RF to the inputs and outputs of $2w$ distinct addition operations. Once the pipeline state is initialized as described, we measure the power consumption of the target device during a sequence of $2w$ additions, and correlate the power consumption of the device with the Hamming distance of the results of additions being issued in the same slot as the instruction bundle.

4) INTERNAL DATA STORAGE IN THE LSU

The Memory Address Register (MAR) Memory Data Register (MDR), and the store buffer, if present, are expected to

TABLE 1. Number of source and destination 32 bit registers for the representative instruction set of the ARMv7-M Thumb ISA.

Set of Instructions	No. of Source Registers	No. of Destination Registers
`nop`	0	0
`mov, addilsli`	1	1
`ldr, str`	1	1
`add, addsh, mul, lsl`	2	1
`ldrd,strd` (load,store double word)	2	2
`smlal` (32 bit equivalent registers)	4	2

exhibit an information leakage proportional to their switching activity. As a consequence of our pipeline characterization, we are also able to validate such a fact. To this end, prepare the pipeline and memory state taking care of storing a copy of $2x$ random values both in $2x$ distinct registers, and in $2x$ memory locations, where x is the number of LSUs present in the CPU, assuming that `load`/`store` operations can be multiple-issued. Subsequently, we measure the power consumption of x load operations that fetch from the main memory the first x values, storing them in the registers that have been precharged to the said values during the pipeline preparation phase. We then execute w or more non-memory instructions, followed by another run of x load operations, loading the remaining x values onto the registers containing a copy of them, placed there in the preparation phase. This sequence of instruction is expected to exhibit a power consumption proportional to the Hamming distance between the values loaded by load instructions computed by the same LSU, regardless of the execution of purely computational instructions in between.

5) RESOURCE CONTENTION ON SYNCHRONOUS ELEMENTS

Sharing portions of the execution stage among different FUs may lead to have also shared synchronous elements. To this end, if resource sharing among FUs is detected, employing a sequence of instructions, interleaving instructions which are computed by different FUs will elicit specifically the use of the shared buffers. Since in case no resource sharing is present, no interaction among the units should happen, correlating the power consumption of the circuit during the interleaved instruction sequence with the Hamming distance between the values taken by the expected shared intermediate result (e.g. the target address of the LSU and the sum of two register values), will lead to the confirmation or denial of the presence of the shared synchronous element.

IV. EXPERIMENTAL EVALUATION

In this section we describe the adopted measurement setup, and the results of the application of the microarchitectural inference techniques exposed in Section III to two real-world in-order CPUs: the ARM Cortex-M4 and the ARM Cortex-M7. We validate the presence of the information leakage on the EM emissions side channel stemming from the synchronous components we infer as described in Section III. Finally, we compare the results of our microarchitectural inference procedure against the publicly available information on the two microarchitectures.

A. CASE STUDY MICROARCHITECTURES

ARM Cortex-M4 and ARM Cortex-M7 are two microcontroller-grade CPUs, compliant with the same ARMv7-M Thumb ISA [2]. Our first step is to define the representative instruction set for the ARMv7-M Thumb ISA, selecting a representative for each of the instruction families listed in Subsection III-A. As a result, our microbenchmarks representative instruction set $\mathbf{I}$ is composed as follows:

- No-Operation: `nop`
- Move operation: `mov rA, rB`
- Signed addition with and without immediate operands: `adds rA, rB, rC` and `adds rA, rB, #imm`, which we denote, for the sake of compactness as `add` and `addi`. Whenever their behaviour does not differ, we indicate both of them as `add[i]`
- Addition with operand shifting: `add rA, rB, rC, lsl #immediate`, denoted in short as `addsh`
- Multiplication: `mul rA, rB, rC`
- Logical Shift with and without immediate operands `lsls rA, rB, rC lsls rA, rB, #immediate`. Analogously to the `add` case we denote the logical shift without immediate as `lsl`, the one with an immediate as `lsli`, and any of the two instructions, as `lsl[i]`
- Load and store instructions of single 32b words: `ldr rA,[rB], str rA,[rB]`
- Load and store instructions of double 32b words: `ldrd rA,[rB], strd rA,[rB]`
- Signed Long Multiply, with optional Accumulate `smlal rClow, rChigh, rA, rB`.

Given the chosen representative instructions, we can compute the number of RF read and write ports required by each one of them, deriving it from their number of source registers and destination registers. This information, which is functional to the derivation of the number of read and write ports of the register file for both the Cortex-M4 and Cortex-M7 CPUs, is reported in Table 1.

B. MEASUREMENT SETUP

We chose as the Cortex-M4 and Cortex-M7 implementations for our exploration two microcontrollers produced by STMicroelectronics, namely the STM32F401 microcontroller, mounted on a Nucleo-F401RE board, for the Cortex-M4 and the STM32F746ZG, mounted on a Nucleo-F746ZG board, for the Cortex-M7.

On both microcontrollers, the `clock_count`$(\cdot)$ measurement is done employing the Data Watchpoint and Trace (DWT) debugging subsystem, which contains a clock cycle counter accessible via a single-cycle load (`ldr`) operation. Cortex-M architectures allow for the presence of both instruction and data caches, at the designer's will. In order to cope with the potential non-determinism introduced by cached load and store operations, we performed all our `clock_count`$(\cdot)$ measurements taking care of preheating the instruction cache with the specialized instruction sequence we want to measure, and the data cache with the

in-memory data that are handled by the said specialized instruction sequence. Employing the pre-heating strategy, we were able to remove the effects of the variable latency coming from storing the executable code segment in Flash, as all the measured instructions are fetched from the CPU instruction cache. Additionally, we ensure that the instruction sequence to be measured can fit into a single instruction cache line, condition which can be empirically verified by observing how the variability of the execution latency decreases as the size of the measured code is reduced below a platform-dependent threshold.

The firmware that executes the described measuring algorithm is based on the Miosix kernel [36], which natively supports both boards. The Operating System (OS) is used to facilitate access to Input/Output facilities, such as logging functions and the Universal Asynchronous Receiver-Transmitter (UART) communication, but is not a prerequisite for the implementation of our framework. To zero out possible interferences of the Operating System (OS) kernel, the interrupts are kept disabled during the execution of the microbenchmarks.

The side channel measurement setup employed to validate our microarchitectural inferences is the same for both target boards. In particular, we measured the Electro-Magnetic (EM) emissions of the microcontrollers under analysis by means of a custom loop probe derived from a 50Ω coaxial cable, connected to two cascaded Agilent INA-10386 amplifiers with a gain of 26 dB each. The amplified probe is connected to a PicoScope 5244D digital sampling oscilloscope, sampling at 500Msamples/s, with an 8-bit per sample vertical resolution. We collect 200k measurements of the EM emissions over the microcontroller package, for each one of the microbenchmark sequences described in Section III. The operations of the microbenchmarks are performed on randomized inputs, obtained as the output of an AES-128-CTR based PseudoRandom Number Generator (PRNG). We correlate the measured EM emissions with the switching activity of the synchronous component whose leakage we want to evaluate, taking as the model the Hamming distance of its contents in two subsequent clock cycles. We employ the Pearson sample correlation coefficient as a measure of the correlation, deeming a sample correlation to be significant whenever its confidence interval for $\alpha = 0.1$ is disjoint from the confidence interval for null correlation, with the same significance level.

The employed CPA methodology could lead to false positives, as the hypothesized component could not be the only one leaking the expected value. However, a model containing more registers than the ones actually present, will actually lead to the implementation of a masking scheme employing more shares than necessary, but still will not violate the masking countermeasure assumptions.

C. MICROARCHITECTURAL INFERENCE ON CORTEX-M4

The first step in analyzing the Cortex-M4 pipeline was to confirm the fact that the CPU is has indeed issue width $w = 1$. To this end we measured the values of $\texttt{CPI}(\sigma)$ for instruction sequences of 200 instructions, to obtain a negligible error in the CPI measurement as specified in Section III. We observed that the minimum CPI value is achieved with a sequence of 2 instructions, and its value is 1, thus indicating a single-issue CPU. We also know from the publicly available information [25] that the Cortex-M4 CPU has a three-stages pipeline; we therefore do not need to infer the pipeline depth.

Having confirmed the issue width value, we proceeded to the measurement of the values of $\texttt{CPI}(f_{raw}(\langle \iota_0 \iota_1 \rangle))$ and $\texttt{CPI}(f_{raw}(\langle \iota_0 \iota_1 \rangle))$ for any two instruction pairs of the representative instruction set, which are reported in Table 2. For the sake of a more effective readout, we highlight in green the specialized instruction pairs attaining the minimum CPI, and in increasingly hotter colours the others.

Observing that, for all computational instructions (i.e., non-load/store instructions), the Cortex-M4 is able to maintain a CPI of 1 even in case of RAW hazards between them, we infer the presence of an EX/EX forwarding path. We note that, as the CPU is a single issue one, we deduce that at most one functional unit per operation is present in the CPU, and no sharing of functional units among different instructions takes place.

After analyzing the CPI data of Table 2 alone, we proceeded from left to right in the inferences allowed by our framework as per depiction in Figure 2. We therefore measured the value of $Lat(f_{raw}(\iota)^n)$ for $1 \leq n \leq 9$ for each instruction in our relevant instruction set. The obtained latency values are reported in Table 3: as it can be observed, all computational instructions are executed in a single cycle, while load and store operations take either two cycles (for single word ones), or three (for double precision ones). This is coherent with the ARM documentation on Cortex-M4 which states that the CPU has a 32 bit wide bus connecting it to the main memory. As a consequence it is expected that a double 32 word load/store operation takes an extra cycle with respect to its single precision counterpart.

Examining the instruction latency values and the corresponding RAW-free CPI for sequences of the same instruction, we are able to ascertain when functional units are internally pipelined, i.e., when an instruction with a given latency l achieves a CPI lower than l itself. The first case of this fact taking place is the `ldr` instruction, which has an execution latency of two cycles and an asymptotic CPI of 1 for long instruction sequences. This indicates that the load-store unit is indeed internally pipelined. We note that, by contrast, the `str` instruction only achieves a CPI of 2, indicating that the instruction execution is considered complete by the pipeline only when the memory writeback (taking an extra clock cycle) is done. It is worth observing that the asymptotic CPI of 1.5 which is measured whenever computational and `ldr` instructions are interleaved can be caused by a structural hazard on the write port of the register file. Indeed, interleaving `ldrs` and single cycle computational instructions causes a contention on the RF write port when the results of the `ldr` (which had its execution started one cycle earlier)

TABLE 2. Cortex-M4 CPI measurement results. The left side of the table represents the measured CPI values for instruction sequences satisfying the $f\_raw(\cdot)$ specialization function, while the right side of the table represents the measured CPI values of the sequences chosen to satisfy the $f\_noraw(\cdot)$ specialization function. The color coding of the values is the following: in green are represented the minimum achievable CPI values, corresponding to $1/w$, in orange are represented values which reach a CPI greater than the minimum and in red are represented values which achieve more than the double of the minimum achievable CPI.

	$\mathtt{CPI}(f_{raw}(\langle \iota_0 \iota_1 \rangle))$				$\mathtt{CPI}(f_{noraw}(\langle \iota_0 \iota_1 \rangle))$				
$\iota_1 \rightarrow$ / $\iota_0 \downarrow$	nop	mov/add[i]/addsh mul/smlal/lsl[i]	ldr str	ldrd strd	nop	mov/add[i]/addsh mul/smlal/lsl[i]	ldr	str	ldrd strd
nop	-	-	-	-	1.0	1.0	1.0	1.0	2.0
mov	-	1.0	2.0	-	1.0	1.0	1.5	1.5	2.0
add[i]	-	1.0	2.0	-	1.0	1.0	1.5	1.5	2.0
addsh	-	1.0	2.0	-	1.0	1.0	1.5	1.5	2.0
mul	-	1.0	2.0	-	1.0	1.0	1.5	1.5	2.0
smlal	-	1.0	2.0	-	1.0	1.0	1.5	1.5	2.0
lsli	-	1.0	2.0	-	1.0	1.0	1.5	1.5	2.0
lsl	-	1.0	2.0	-	1.0	1.0	1.5	1.5	2.0
ldr	-	2.0	2.0	-	1.0	1.5	1.0	1.5	2.5
str	-	2.0	2.0	-	1.0	1.5	1.5	2.0	2.5
ldrd	-	-	-	-	2.0	2.0	2.5	2.5	3.0
strd	-	-	-	-	2.0	2.0	2.5	2.5	3.0

TABLE 3. Cortex-M4 execution latencies for increasing length instruction sequences, composed as a sequence of a single type of instruction.

Instruction	$Lat(f_{raw}(\iota)^n)$ for $1 \leq n \leq 9$								
ldr	2	4	6	8	10	12	14	16	18
str	2	4	6	8	10	12	14	16	18
ldrd	3	6	9	12	15	18	21	24	27
strd	3	6	9	12	15	18	21	24	27
All others	1	2	3	4	5	6	7	8	9

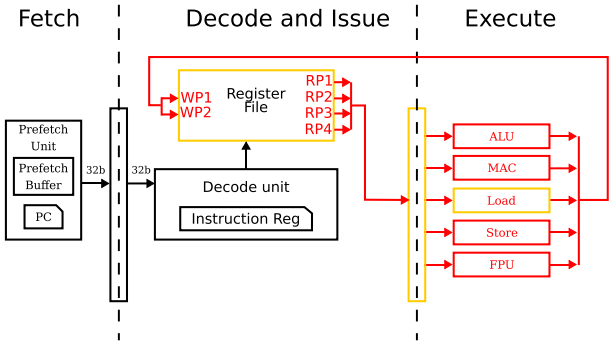


FIGURE 5. Block diagram of the inferred information of the Cortex-M4 processor datapath architecture. In red are highlighted the components that were correctly identified through our feature inference. In yellow are highlighted the components that contain synchronous datapath registers, whose leakage behavior has been confirmed.

TABLE 4. Deduced features of the Cortex-M4 functional units.

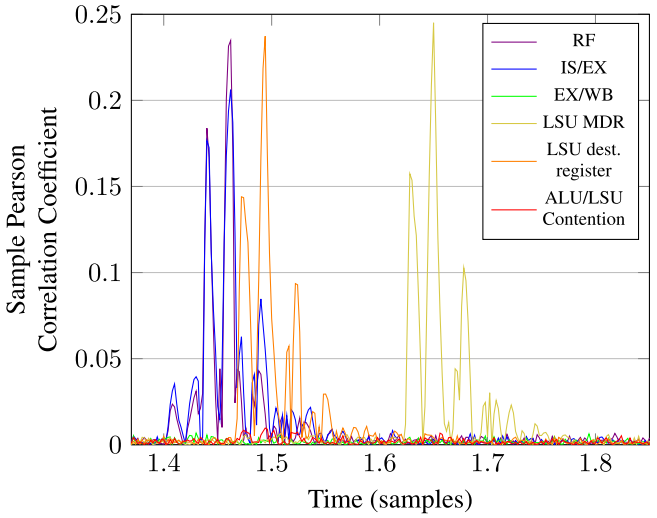
ι	$Lat(f_{noraw}(\iota))$	Pipelined
nop	1	False
mov	1	False
add[i]	1	False
addshf	1	False
mul	1	False
smlal	1	False
lsl[i]	1	False
ldr	2	True
str	2	False
ldrd	3	False
strd	3	False

and of the computational instruction have to be written back.

Given the single issue nature of the Cortex-M4, the number of read ports and write ports is the one of the most demanding operation, namely the double precision Multiply-and-ACcumulate, that requires 4 read ports and 2 write ports to be completed in a single cycle. The inter-stage IS/EX register is thus expected to be 4, 32-bit words wide, while the EX/WB register is not expected to be present, due to the Cortex-M4 3-stages pipeline.

Having completed our microarchitectural inference procedure we can thus draft the alleged Cortex-M4 pipeline structure we have been able to infer. Figure 5 depicts the said pipeline, highlighting in red the inferred components, and in yellow the ones containing synchronous elements about which we can verify the presence of side channel leakage.

We report the results of the side-channel analysis confirmation of the presence of the leakage of the inferred synchronous pipeline elements in Figure 6. In particular, Figure 6 reports the sample Pearson correlation coefficient between the instantaneous power consumption of the device, proportional to its EM emissions, and the Hamming weight of a set of modeled values reported in the right table of Figure 6. The first point to be observed is the confirmation of the leakage related to the RF switching activity, and the IS/EX interstage register, which matches the Hamming distance between two subsequent stored values in the said memory elements. Subsequently, we confirm the absence of leakage from the EX/WB register as no side channel leakage compatible with the Hamming distance between two subsequent add operation results storing results in different registers is present. This matches the public information on the



Synchronous component	Microbenchmark	Power Model	Verified
RF	`mov r0, r1`	$r0 \oplus r1$	✓
IS/EX	`mov r0, r1` `mov r2, r3`	$r1 \oplus r3$	✓
EX/WB	`add r0, r1, r2` `add r3, r4, r5`	$(r1+r2) \oplus (r4+r5)$	×

LSU MDR	`ldr r0, [x]`
	`3 × (ALU Ops)`
	`ldr r2, [y]`
		$[x] \oplus [y]$	✓

LSU dest. register	`ldr r0, [x]` `ldr r2, [y]`	$r0 \oplus r2$	✓
ALU/LSU contention	`ldr r0, [r1]` `mov r2, r3`	$r1 \oplus r3$	×

FIGURE 6. On the left, it is depicted the result of the synchronous datapath registers verification process, applied on the Cortex-M4. The result shows the value of the Pearson correlation coefficient, computed between the HD of the registers value updates and the captured EM traces. Correlation peaks represent a confirmed leakage behaviour. On the right a summary of the microbenchmarks eliciting the said information leakages.

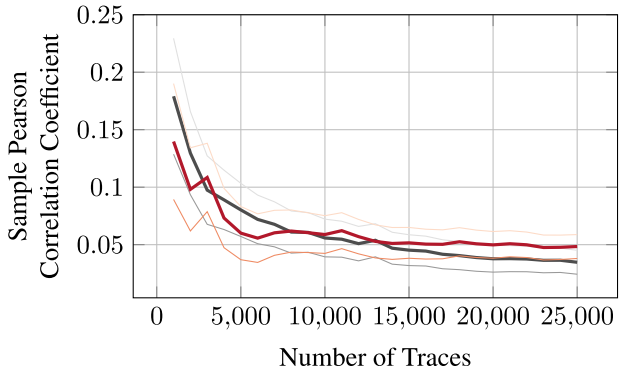


FIGURE 7. Result of a CPA attack against an unprotected AES implementation, we compare the Pearson correlation coefficient of the correct key (colored in thick red), with the second best candidate (colored in grey). The right key byte becomes statistically distinguishable after ~ 10000 traces.

Cortex-M4 pipeline lacking being a 3 stages pipeline. We then observe the presence of an information leakage attributable to the MDR present in the LSU, employing as a microbenchmark a pair of load operations between which three ALU operations are run. Coherently with the timing delay induced by the three additional, single cycle, operations, the peaks in the value of the Pearson correlation coefficients appears 4 cycles later than the one of computational instructions (3 cycles for the added instructions plus one as the load operation writes the contents of the MDR in the second cycle). The leakage from the LSU destination register, which is contained in the RF, can be observed to take place one cycle after the one caused by computational instructions. Finally, we observe the absence of side channel leakage modeled by the Hamming distance between the target address of the `ldr` operation and a subsequent `mov` operation, suggesting the operand paths are distinct.

To validate the practical impact of the identified leakage, we mounted a CPA attack over an unprotected software AES implementation, executed on the Cortex-M4 core, thus confirming how the EM leakage enables successful key-retrieval attacks. We executed a CPA attack on the first byte of the AES key, by computing 25000 encryptions of random plaintext on the target, capturing for each execution an EM leakage trace, and computing the Pearson correlation coefficient between the traces and the values extracted from a model of the power consumption. The model is built with the Hamming weights of the values computed at the SubBytes stage of the second AES round. The attack can be independently applied on all the subsequent bytes of the key, leading to a full key recovery. The results of the attack are shown in Figure 7.

D. MICROARCHITECTURAL INFERENCE ON CORTEX-M7

We start our inference procedure on the Cortex-M7 confirming the value of the issue width, i.e. $w = 2$, reported in the component datasheet [26]. To this end, we derive all the values of $\texttt{CPI}(f_{noraw}(\langle \iota_0 \iota_1 \rangle))$, and report them in Table 5. In order to avoid issues from caching mechanisms, which would provide a non deterministic access to the main memory, we run our microbenchmark code and place our microbenchmark data in the Tightly Coupled Memory (TCM) provided by Cortex-M7, and implemented in the STM32F746ZG. The TCM is designed to provide deterministic, single cycle access to its contents. As we observe, the minimum achieved CPI value is indeed $\frac{1}{2}$, confirming the dual-issue nature of Cortex-M7.

Before analyzing the results obtained in terms of CPI and latency, it is useful to recall some facts on the TCM as reported by ARM [27]. The TCM is split in two portions, an Instruction TCM (ITCM) and a Data TCM (DTCM). To clearly interpret the results we obtain, it is useful to know that the Data TCM is split onto two banks, D0TCM and D1TCM, and the bank selection is made using the third least significant bit of the accessed address [27]. As a consequence,

TABLE 5. Cortex-M7 $\texttt{CPI}(f_{noraw}(\langle \iota_0 \iota_1 \rangle))$ measurement results. The color coding is as follows: in green are the minimum achievable CPI values, corresponding to $1/w$, in orange are the values which reach a CPI greater than the minimum and in red are the values which achieve more than the double of the minimum CPI. Load/store instructions marked with a $*$ make use of an address offset equal to 4, while unmarked instructions in which the address offset is set to zero. Base addresses are 32 bit aligned.

	$\texttt{CPI}(f_{noraw}(\langle \iota_0 \iota_1 \rangle))$													
$\iota_1 \rightarrow$ / $\iota_0 \downarrow$	nop/lsli mov/add[i]	addsh	mul/smlal	lsl	ldr ldrb	ldr* ldrb*	ldrd	ldrd*	str	str*	strb	strb*	strd	strd*
nop/lsli	0.5	0.5	0.5	0.5	0.5	0.5	1.0	1.0	0.5	0.5	1.5	1.5	1.0	1.5
mov/add[i]	0.5	0.5	0.5	0.5	0.5	0.5	1.0	1.0	0.5	0.5	1.5	1.5	1.0	1.5
addsh	0.5	1.0	0.5	0.5	0.5	0.5	1.0	1.0	0.5	0.5	1.5	1.5	1.0	1.5
mul/smlal	0.5	0.5	1.0	0.5	0.5	0.5	1.0	1.0	1.0	1.0	1.5	1.5	1.0	1.5
lsl	0.5	0.5	0.5	0.5	0.5	0.5	1.0	1.0	0.5	0.5	1.5	1.5	1.0	1.5
ldr/ldrb	0.5	0.5	0.5	0.5	1.0	0.5	1.0	1.0	2.5	0.5	3.5	1.5	3.0	1.5
ldr*/ldrb*	0.5	0.5	0.5	0.5	0.5	1.0	1.0	1.0	0.5	2.5	1.5	3.5	3.0	3.0
ldrd	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	3.0	3.0	4.0	4.0	3.0	3.0
ldrd*	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	3.0	1.5	4.0	3.0	3.5
str	0.5	0.5	1.0	0.5	2.5	0.5	3.0	1.0	1.0	1.0	2.0	1.5	1.0	1.5
str*	0.5	0.5	1.0	0.5	0.5	2.5	3.0	3.0	1.0	1.0	1.5	2.0	1.0	1.5
strb	1.5	1.5	1.5	1.5	3.5	1.5	4.0	1.5	2.0	1.5	3.0	1.5	2.0	2.0
strb*	1.5	1.5	1.5	1.5	1.5	3.5	4.0	4.0	1.5	2.0	1.5	3.0	2.0	2.0
strd	1.0	1.0	1.0	1.0	3.0	3.0	3.0	3.0	1.0	1.0	2.0	2.0	1.0	1.5
strd*	1.5	1.5	1.5	1.5	1.5	3.0	3.0	3.5	1.5	1.5	2.0	2.0	1.5	2.0

TABLE 6. Cortex-M7 execution latencies for increasing instruction sequences lengths, composed as a sequence of a single type of instruction. We do not include store instructions in this table since they have no output register operand, therefore it's impossible to compose an instruction sequence satisfying $f_{raw}(\cdot)$. In the add with shift instructions marked with a †, the RAW conflict is applied on the shifted register, conversely, in the unmarked add with shift, the RAW is placed on the non-shifted register. In load instructions marked with a $\diamond$, the RAW conflict is applied on the offset operands; RAW conflicts are applied to the address operand otherwise.

Instruction	$Lat(f_{raw}(\iota)^n)$ for $1 \leq n \leq 9$								
mov,add[i]	1	1	2	3	4	5	6	7	8
lsl[i]	1	1	2	3	4	5	6	7	8
addsh	1	2	3	4	5	6	7	8	9
addsh†	1	3	5	7	9	11	13	15	17
mul/smlal	1	3	5	7	9	11	13	15	17
ldr	1	3	5	7	9	11	13	15	17
ldrd	1	3	5	7	9	11	13	15	17
ldr$^\diamond$	1	4	7	10	13	16	19	22	25
ldrb,ldrb$^\diamond$	1	4	7	10	13	16	19	22	25

it is only possible to read two 32-bit words in parallel from the DTCM if they are stored in different banks, as DTCM banks have a single read port, while the CPU is endowed with 2 32 bit read ports towards DTCM [26].

After obtaining the results on the CPI achieved by instruction pairs with (Table 7) and without RAW (Table 5) hazards between them, and the value of the instruction execution latencies $Lat(f_{raw}(\iota)^n)$ for $1 \leq n \leq 9$ (Table 6), we move onto the identification of the number of available functional units, and their latency. We note that Table 7 does not report the same number of cases as Table 5, since it is not possible to cause the RAW predicate to hold for all pairs of instructions in an instruction sequence where store instructions are present. Indeed, since store instructions (str/strb/strd) have no output register, they cannot cause a RAW conflict with the instruction following them. First of all, we observe that the

TABLE 7. Cortex-M7 $\texttt{CPI}(f_{raw}(\langle \iota_0 \iota_1 \rangle))$ measurement results. The color coding is as follows: in green are the minimum achievable CPI values, corresponding to $1/w$, in orange are the values which reach a CPI greater than the minimum and in red are the values which achieve more than the double of the minimum CPI. In the addsh instructions marked with a †, the RAW conflict is applied on the shifted register, in the unmarked add with shift, the RAW is placed on the non-shifted register. Instructions marked with a $*$ make use of an address offset, conversely to unmarked instructions in which the address offset is set to zero. In load instructions marked with a $\diamond$, the RAW conflict is applied on the offset operands. RAW conflicts are applied to the address operand otherwise. Base addresses are 32 bit aligned.

	$\texttt{CPI}(f_{raw}(\langle \iota_0 \iota_1 \rangle))$						
$\iota_1 \rightarrow$ / $\iota_0 \downarrow$	mov add lsl[i]	addi	addsh	addsh†	mul smlal	ldr$^\diamond$	ldrb$^\diamond$
mov/add	1.0	1.0	1.0	1.5	1.5	1.5	2.0
lsl[i]	1.0	1.0	1.0	1.5	1.5	1.5	2.0
addi	1.0	1.0	1.0	1.5	1.5	1.5	2.0
addsh	1.0	1.0	1.0	1.5	1.5	2.0	2.0
addsh†	1.5	1.5	1.5	2.0	2.0	2.0	2.5
mul/smlal	1.5	1.5	1.5	2.0	2.0	3.0	3.0
ldr*$^\diamond$	1.5	1.5	2.0	2.0	3.0	3.0	3.0
ldrb*$^\diamond$	2.0	2.0	2.0	2.5	3.0	3.0	3.0

Cortex-M7 is able to reach a CPI of $\frac{1}{2}$ for any combination of computational instructions in our relevant instruction set, save for the case of add instructions with a shifted operand (addsh) and mul/smlal instructions. From this we can infer the presence of a single ALU able to compute addsh (ALU0 from now on) and a single one capable of mul/smlal (ALU1). The second fact is consistent with what is described in the Cortex-M7 datasheet [26], where the presence of a single MAC capable unit is described. We also infer the presence of independent ALUs to be able to sustain a CPI of $\frac{1}{2}$, plus an independent LSU, as all the computational instructions can be dual-issued with a load instruction. We now analyze the latencies of the functional units, and the presence of forwarding paths. Starting from

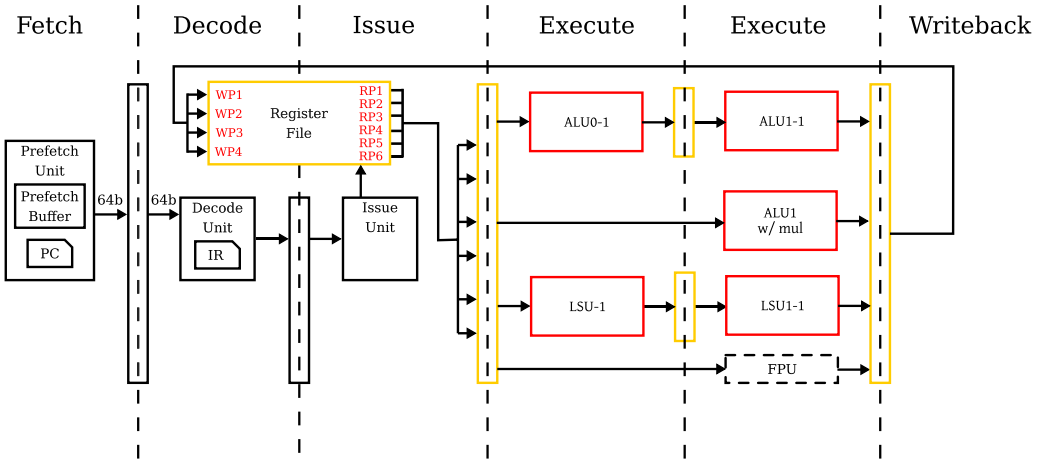


FIGURE 8. Inferred structure of the Cortex-M7 pipeline. In red there are highlighted the datapath components that were correctly inferred in this work, in yellow there are highlighted the synchronous components whose leakage has been experimentally confirmed.

ALU0, we observe that the latency of `addsh` instructions for a sequence of two or more of them raises by 2 clock cycles per instruction. In contrast with this, the latency of a single `addsh` is of a single clock cycle. This fact, in conjunction with the capability of the Cortex-M7 to sustain a unitary CPI for `addsh` whenever no RAW conflicts are present points to ALU0 being a 2 stage pipelined ALU, with its first stage being able of performing a shift operation in a single cycle. This description matches the one of the first ALU described in the described in the Cortex-M7 datasheet [26] (Main/ALU #1 Pipeline in the datasheet). Observing that $\texttt{CPI}(f_{raw}(\langle \texttt{addsh}, \texttt{addsh} \rangle))$ changes between 1 and 1.5 depending on whether the RAW conflict is present among the shifted operands or the non-shifted operands we are able to detect the presence of an EX-EX forwarding path for ALU0. We now analyze the behaviour of the `mul`/`smlal` capable ALU1. Considering the unitary CPI in case of a sequence of `mul`/`smlal` and the fact that a sequence of `mul`/`smlal` operation behaves in the same fashion as one of `addsh` for what the latency of n operations with RAW conflict concerns, two possible architectures are possible. Either ALU1 matches the structure of ALU0, or it is a single-cycle combinatorial ALU, but no EX-EX forwarding path is present. Considering the description of ALU1 in the Cortex-M7 datasheet [26] (ALU #2 Pipeline in the datasheet) we are able to directly know that ALU1 is a single cycle `mul`/`smlal` unit with and thus resolve the point. In the hypothetical case where this point could not have been resolved via public information, we would have proceeded to perform side channel leakage tests for both cases, and confirm the correct one. Finally we analyze in detail the behaviour of the LSU. Starting from the data available in Table 5, we observe that 32 bit load operations (`ldr`) can be dual-issued with other `ldr` operations, only when they employ as a target an address belonging to a different DTCM bank (namely, a `ldr` operation with the third LSB of the address set and a `ldr` operation with the third LSB of the address clear). By contrast, a sequentialization of the operations, due to a structural hazard in the access to the DTCM causes the CPI to reach 1 in case both loaded addresses belong to the same DTCM bank. An analogous behaviour is reported for the 32 bit store (`str`) instructions. The load instruction loading a single byte `ldrb` behave in the same fashion as the `ldr` instructions and have no structural conflicts with them. The store instruction storing a single byte requires an additional load operation of the word into which the byte should be stored, as the CPU-DTCM bus is 32 bits wide, and does not allow individual byte transfers. This in turn raises the CPI of the `strb` instructions as a (non parallelizable) hidden load must be performed. Double-word load and store instructions (`ldrd`/`strd`) achieve a CPI of 1, which is consistent with the ability of the Cortex-M7 to perform two 32 bit word memory operations toward the DTCM, provided that they reside in different banks. Indeed, loading or storing 64 bits from a 32 bit word-aligned address as in our benchmarks will always result in two memory operations on different DTCM banks.

Finally, we derive the number of read and write ports of the register file, and the interstage register size. Given that the Cortex-M7 is able to issue concurrently a `smlal` and an `add` instruction, the total number of 32 bit read operands from the register file is 6, while, to cope with the needs of storing concurrently the outcome of a `smlal` and a `strd` (as it happens analyzing a sequence of such operations unaffected by RAW conflicts) 4 write ports are needed. We note that our inference matches the feature description of the Cortex-M7 datasheet [26].

We provide a graphical depiction of the alleged structure of the pipeline of the ARM Cortex-M7, according to our microarchitectural inferences in Figure 8. We note that our deductions are aligned with the publicly available information available in the datasheet, i.e., the Cortex-M7 is a dual issue, in-order CPU, with two ALUs, and an independent

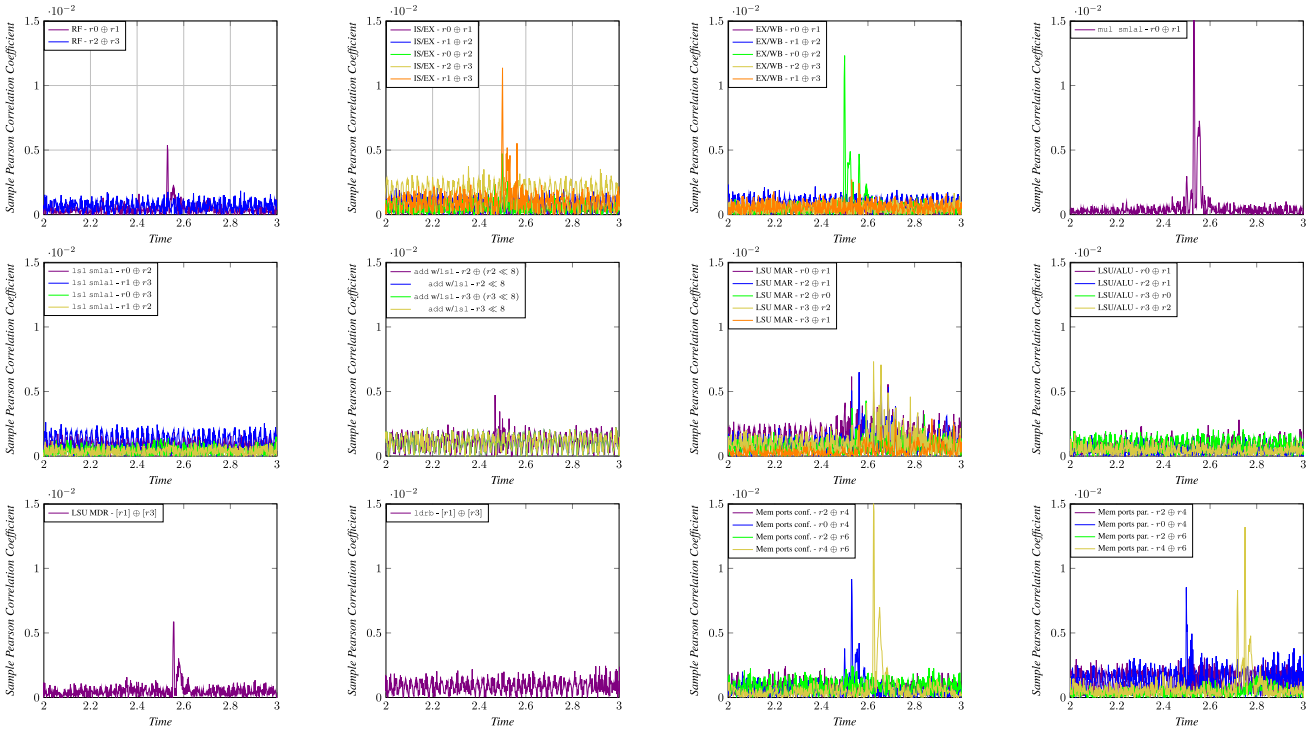


FIGURE 9. Complete report of the detected correlation between the synchronous component switching activity predicted from our inferred microarchitecture, and the benchmark contents. Benchmarks depicted in the same order as listed in Table 8.

LSU and a 6 stages pipeline, of which 2 stages are devoted to EX. We note that all the instructions have $Lat(f_{noraw}(\iota))$ since, in case no RAW conflicts are present, pipelined units allow for an instruction issue per clock cycle.

We now proceed to validate the structure of the pipeline reported in Figure 8 through dedicated microbenchmarks. Table 8 reports a summary of the synchronous component target of the benchmark, the values of which the Hamming weight is employed as the power consumption model and whether or not the model matches actual side channel leakage. Figure 9 reports the quantitative values of the correlation as a function of time; in particular a plot is reported for each synchronous component being benchmarked. Each plot superimposes the correlation of the power consumption models of Table 8. All the correlation values were obtained sampling 1 million traces from the device under examination.

The first three microbenchmarks concern the information leakage coming from the switching activity of the Register File (RF), the IS/EX interstage register and the EX/WB interstage register. The benchmark on the RF has effectively detected correlation between the switching activity of the RF and the power consumption itself, although the correlation with the register move between `r2` and `r3` appears to be quantitatively smaller. The benchmark detecting the information leakage coming from the IS/EX interstage register shows a clear correlation between the (source) operands of the instruction which are being issued in the same pipeline, while no correlation between the (source) operands of different pipelines. This is coherent with the IS/EX register holding them separately. We note a persistent, small but not statistically negligible, information leakage correlated to the distance between `r2` and `r3` is present even when the pipeline is executing only `nops`. An analogous situation takes place when detecting the leakage of the EX/WB interstage register, where the switching activity related to the destination operands, `r0,r2`, of the instructions issued in the first execution lane shows a clear correlation, while the switching activity due to the destination operands of the second execution pipeline, `r1,r3` provides far less side channel leakage (the maximum correlation value is coherent in timing and about three times higher than the threshold for statistical artifacts, $\frac{4}{\sqrt{10^6}} \approx 0.001$). In the `mul smlal` benchmark we test that the multiplier unit is performing both operations is indeed the same, through testing the switching activity due to a single operand change. The corresponding benchmark shows significant correlation. The `lsl smlal` is similar in spirit, willing to test the orthogonality of the two execution pipelines, the one for shifts and the one for multiply-and-accumulate instructions. Indeed, no significant spike in the correlation values is detected in correspondence with the time instant where the operations are performed. The `add` with `lsl` benchmark aims at detecting the leakage stemming from the interstage register present in the 2 stages, pipelined ALU0. Indeed, we have that the device provides statistically significant information leakage related to the distance between the shifted operand, and its original value in an `add`

TABLE 8. Results of the validation of the presence of synchronous components in the Cortex-M7 pipeline via side channel correlation.

Buffer	Stimulus	Power Model	Verif.
RF	`mov r0,r1` `mov r2,r3`	$r0 \oplus r1$ $r2 \oplus r3$	✓ ×

IS/EX	`mov r4,r0`
	`mov r5,r1`
	`mov r6,r2`
	`mov r7,r3`
		$r0 \oplus r1$
		$r1 \oplus r2; r0 \oplus r2$
		$r2 \oplus r3; r1 \oplus r3$
			×
			×;✓
			×;✓

EX/WB	`add r0,r0,r4`
	`add r1,r4,r1`
	`add r2,r2,r4`
	`add r3,r4,r3`
		$r0 \oplus r1$
		$r1 \oplus r2; r0 \oplus r2$
		$r2 \oplus r3; r1 \oplus r3$
			×
			×;✓
			×;✓

`mul smlal`	`mul r0,r0,#1` `smlal #0,#0,r1,#1`	$r0 \oplus r1$	✓
`lsl smlal`	`lsl r4,r0,r1` `smlal r5,r6,r2,r3`	$r0 \oplus r2; r1 \oplus r3$ $r0 \oplus r3; r1 \oplus r2$	×;× ×;×
`add` w/`lsl`	`add r0,r1,r2,lsl#8` `add r0,r1,r3,lsl#8`	$r2 \oplus (r2 \ll \#8)$; $(r2 \ll \#8)$ $r3 \oplus (r2 \ll \#8)$; $(r3 \ll \#8)$	✓;× ×;×

LSU MAR	`ldr r4,[r0]`
	`ldr r5,[r1]`
	`ldr r6,[r2]`
	`ldr r7,[r3]`
		$r0 \oplus r1$
		$r2 \oplus r1; r2 \oplus r0$
		$r3 \oplus r2; r3 \oplus r1$
			✓
			✓;✓
			✓;✓

LSU/ALU	`add r4,r8,r0`
	`ldr r5,[r8, r1]`
	`add r6, r2, r0`
	`ldr r7,[r3, r8]`
		$r0 \oplus r1; r2 \oplus r1$
		$r3 \oplus r0; r3 \oplus r2$
			×;×
			×;×

LSU MDR	`ldr r0,[r1]`
	3 × (ALU Ops)
	`ldr r2,[r3]`
		$[r1] \oplus [r3]$	✓

`ldrb`	`ldrb r0,[r1, #1]` `ldrb r2,[r3, #1]`	$([r1]$ & (0x00FFFFFF)) $\oplus$ $([r3]$ & (0x00FFFFFF))	×

Mem conf.	`ldr r0,[r1, #0]`
	`str r2,[r3, #0]`
	`ldr r4,[r5, #0]`
	`str r6,[r7, #0]`
		$r2 \oplus r4; r0 \oplus r4$
		$r2 \oplus r6; r4 \oplus r6$
			×;✓
			×;✓

Mem par.	`ldr r0,[r1, #0]`
	`str r2,[r3, #4]`
	`ldr r4,[r5, #0]`
	`str r6,[r7, #4]`
		$r2 \oplus r4; r0 \oplus r4$
		$r2 \oplus r6; r4 \oplus r6$
			×;✓
			×;✓

operation with left shift of an operand. This in turn confirms the presence of an internal register.

We finally turn to the analysis of the information leakage coming from the load-store unit. The LSU MAR benchmark aims at detecting if the switching activity of the LSU MAR(s) provides information on the side channel. Figure 9 reports that a significant correlation exists between the power consumption and the Hamming weight of addresses employed in consecutive memory operations, while less evident, but still statistically significant correlation exists between addresses of load operations issued either both as the first or both as the second of an issuing bundle. From this result we can infer that there is a logic component whose switching activity depends on the sequence of addresses transiting through the LSU, providing more significant information leakage than the MAR themselves.

Subsequently, the LSU ALU benchmark tests if the adder available in the ALU is also employed to perform address-offset computations. From our microarchitectural

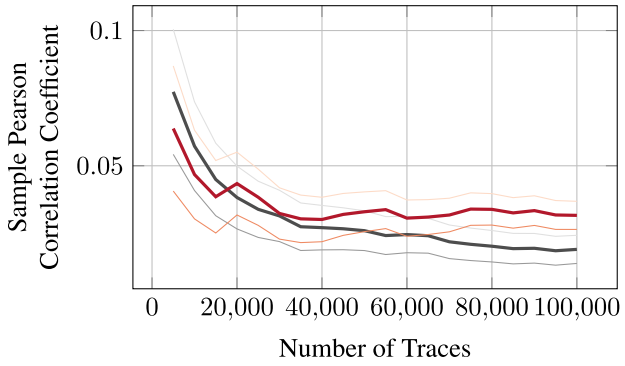


FIGURE 10. Result of a CPA attack against an unprotected AES implementation, running on our Cortex-M7 platform. We compare the Pearson correlation coefficient of the correct key (colored in thick red), with the second best candidate (colored in grey). The right key byte becomes statistically distinguishable after ≈ .75k traces.

inference, we expect it not to be so, and the experimental benchmarks confirm our inferences. In the LSU MDR benchmark we aim at detecting if the MDR of the LSU leaks, per se, information via side channel. To single out its switching activity from the rest of the pipeline we insert three ALU operations acting on orthogonal registers between two `ldr` operations and test the correlation with the Hamming distance of the loaded values. Figure 9 indeed reports correlation with the said Hamming distance in the time instant corresponding to the execution of the second stage of the LSU, i.e. the loading of the word from the memory. Willing to test if the realignment of loaded values is performed by a combinatorial circuit, and thus less likely to provide a punctual leakage in time, we test the correlation with the Hamming distance between two bytes loaded with two `ldrb` operations, acting on addresses containing random 32b values. The corresponding benchmark provides no correlation, suggesting the absence of a synchronous component storing the actual single byte. Finally, we discern if the two MDR, bound to the two data memory interfaces of Cortex-M7 are orthogonal. To this end, we devise a microbenchmark performing a parallel load/store employing the two data memory ports (*memory par.* in Table 8). The benchmark tests for the correlation between all the possible pair of loaded values residing at two addresses belonging to different DTCM banks. As reported in Figure 9, we obtain a significant correlation when employing as a model the Hamming distance between the words stored in the same DTCM bank, in turn pointing to the fact that the four involved loads/stores operations employ two separate MDRs. Finally, we consider the case where a structural hazard on the access to the DTCM banks prevents the execution of two load/store operations at a time (*mem conf.* in Table 8). Two possible outcomes are present: the bank conflict detection is done at issue time, and only a single load/store is issued, or the delay is inserted by the LSU, which accepts the parallel operations, resolves the bank conflict, and delays them. In the former case, we expect to detect correlation between the Hamming weights of the loaded/stored values of adjacent operations, as they are serialized onto the same MDR. In the

latter case, we expect the same leakage pattern as *memory par.*, but with the second relevant leakage taking place later in time. As it can be seen comparing the last two bottom right plots in Figure 9, we can deduce that the bank conflict is detected by the LSU, and two independent MDRs are used also in this case.

Finally, we assess the effectiveness of our leakage model by performing a side-channel attack on an unprotected AES implementation, running on the Cortex-M7. The results of the said attack is depicted in Figure 10. We performed a CPA attack on the first byte of the AES key, by running 100k executions of the cipher on the target, and computing the Pearson correlation coefficient with a model of the power consumption. The model, as in the Cortex-M4 case, is built with the Hamming weights of the values computed at the SubBytes stage of the second AES round. We confirmed how the side-channel leakage enables the retrieval of the first key byte with $\approx$ 75k traces.

V. CONCLUSION

In this work we introduce a complete framework for extracting a side-channel centric microarchitectural leakage model from any unknown in-order CPU. The only prerequisites to our technique are the knowledge of the ISA, the means to obtain the cycle count between two instructions, and a side channel measurement setup to evaluate the hypothesized EM leakage. We validated the proposed methodology on two CPUs of the Cortex-M class, obtaining concrete leakage models, which can be employed in a software only pre-screening for side channel vulnerabilities. Our models highlight the pressing need to consider a microarchitectural view on the side channel leakage, as it highlights potential avenues for attacks, which are not easily caught by examining the ISA-level (i.e., assembly-level) implementation of a cryptographic primitive.

REFERENCES

[1] G. Agosta, A. Barenghi, and G. Pelosi, "Compiler-based techniques to secure cryptographic embedded software against side-channel attacks," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 8, pp. 1550–1554, Aug. 2020, doi: 10.1109/TCAD.2019.2912924.

[2] ARM Limited. (2019). *Cortex-M4 Instruction Set Summary*. Accessed: Sep. 9, 2021. [Online]. Available: https://developer.arm.com/docs/dui0553/latest/the-cortex-m4-instruction-set/instruction-set-summary

[3] M. Backes, M. Dürmuth, S. Gerling, M. Pinkal, and C. Sporleder, "Acoustic side-channel attacks on printers," in *Proc. 19th USENIX Secur. Symp.*, Washington, DC, USA, Aug. 2010, pp. 307–322. [Online]. Available: http://www.usenix.org/events/sec10/tech/full_papers/Backes.pdf

[4] J. Balasch, B. Gierlichs, V. Grosso, O. Reparaz, and F. Standaert, "On the cost of lazy engineering for masked software implementations," IACR Cryptol. ePrint Arch., Tech. Rep. 2014/413, 2014. [Online]. Available: http://eprint.iacr.org/2014/413

[5] A. Barenghi, W. Fornaciari, G. Pelosi, and D. Zoni, "Scramble suit: A profile differentiation countermeasure to prevent template attacks," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 9, pp. 1778–1791, Sep. 2020, doi: 10.1109/TCAD.2019.2926389.

[6] A. Barenghi and G. Pelosi, "Side-channel security of superscalar CPUs: Evaluating the impact of micro-architectural features," in *Proc. 55th Annu. Design Automat. Conf.*, San Francisco, CA, USA, Jun. 2018, p. 120, doi: 10.1145/3195970.3196112.

[7] O. Bronchain and F. Standaert, "Side-channel countermeasures' dissection and the limits of closed source security evaluations," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2020, no. 2, pp. 1–25, 2020, doi: 10.13154/tches.v2020.i2.1-25.

[8] I. Buhan, L. Batina, Y. Yarom, and P. Schaumont, "SoK: Design tools for side-channel-aware implementations," Cryptol. ePrint Arch., Tech. Rep. 2021/497, 2021. Accessed: Sep. 30, 2021. [Online]. Available: https://ia.cr/2021/497

[9] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi, "Towards sound approaches to counteract power-analysis attacks," in *Proc. Annu. Int. Cryptol. Conf.* in Lecture Notes in Computer Science, vol. 1666, M. J. Wiener, Ed. Santa Barbara, CA, USA: Springer, 1999, pp. 398–412, doi: 10.1007/3-540-48405-1_26.

[10] Z. Chen, A. Sinha, and P. Schaumont, "Using virtual secure circuit to protect embedded software from side-channel attacks," *IEEE Trans. Comput.*, vol. 62, no. 1, pp. 124–136, Jan. 2013, doi: 10.1109/TC.2011.225.

[11] D. Das, M. Nath, B. Chatterjee, S. Ghosh, and S. Sen, "STELLAR: A generic EM side-channel attack protection through ground-up root-cause analysis," in *Proc. IEEE Int. Symp. Hardw. Oriented Secur. Trust (HOST)*, McLean, VA, USA, May 2019, pp. 11–20, doi: 10.1109/HST.2019.8740839.

[12] J. Ferrigno and M. Hlavac, "When AES blinks: Introducing optical side channel," *IET Inf. Secur.*, vol. 2, no. 3, pp. 94–98, Sep. 2008, doi: 10.1049/iet-ifs:20080038.

[13] S. Furber, *ARM System-on-Chip Architecture*, 2nd ed. Reading, MA, USA: Addison-Wesley, 2000.

[14] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic analysis: Concrete results," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst.*, in Lecture Notes in Computer Science, Ç. K. Koç, D. Naccache, and C. Paar, Eds., vol. 2162. Paris, France: Springer, 2001, pp. 251–261, doi: 10.1007/3-540-44709-1_21.

[15] S. Gao, B. Marshall, D. Page, and T. H. Pham, "FENL: An ISE to mitigate analogue micro-architectural leakage," *IACR Trans. Cryptogr. Hardw. Embedded Syst.*, vol. 2020, no. 2, pp. 73–98, 2020, doi: 10.13154/tches.v2020.i2.73-98.

[16] A. Hartstein and T. R. Puzak, "The optimum pipeline depth for a microprocessor," in *Proc. 29th Annu. Int. Symp. Comput. Archit.*, Anchorage, AK, USA, Y. N. Patt, D. Grunwald, and K. Skadron, Eds., 2002, pp. 7–13, doi: 10.1109/ISCA.2002.1003557.

[17] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. Amsterdam, The Netherlands: Elsevier, 2017.

[18] A. Heuser, O. Rioul, and S. Guilley, "Good is not good enough—Deriving optimal distinguishers from communication theory," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst.*, in Lecture Notes in Computer Science, vol. 8731, L. Batina and M. Robshaw, Eds. Busan, South Korea: Springer, 2014, pp. 55–74, doi: 10.1007/978-3-662-44709-3_4.

[19] P. C. Kocher, "Timing attacks on implementations of Diffie–Hellman, RSA, DSS, and other systems," in *Proc. Annu. Int. Cryptol. Conf.*, in Lecture Notes in Computer Science, vol. 1109, N. Koblitz, Ed. Santa Barbara, CA, USA: Springer, 1996, pp. 104–113, doi: 10.1007/3-540-68697-5_9.

[20] P. C. Kocher, J. Jaffe, and B. Jun., "Differential power analysis," in *Proc. Annu. Int. Cryptol. Conf.*, in Lecture Notes in Computer Science, vol. 1666, M. J. Wiener, Ed. Santa Barbara, CA, USA: Springer, 1999, pp. 388–397, doi: 10.1007/3-540-48405-1_25.

[21] D. Kroening, R. Bryant, and O. Strichman, *Decision Procedures: An Algorithmic Point of View* (Texts in Theoretical Computer Science. An EATCS Series). Berlin, Germany: Springer, 2008.

[22] V. Lomné, P. Maurine, L. Torres, M. Robert, R. Soares, and N. Calazans, "Evaluation on FPGA of triple rail logic robustness against DPA and DEMA," in *Proc. Design, Automat. Test Eur. Conf. Exhib.*, L. Benini, G. D. Micheli, B. M. Al-Hashimi, and W. Müller, Eds., Nice, France, 2009, pp. 634–639, doi: 10.1109/DATE.2009.5090744.

[23] D. McCann, E. Oswald, and C. Whitnall, "Towards practical tools for side channel aware software engineering: 'Grey box' modelling for instruction leakages," in *Proc. 26th USENIX Secur. Symp.*, E. Kirda and T. Ristenpart, Eds. Vancouver, BC, Canada: USENIX Association, Aug. 2017, pp. 199–216. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/mccann

[24] T. S. Messerges, "Power analysis attacks and countermeasures for cryptographic algorithms," Ph.D. dissertation, Univ. Illinois, Chicago, IL, USA, Jan. 2000.

[25] (2020). *ARM Cortex-M4 Datasheet*. Accessed: Sep. 30, 2021. [Online]. Available: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Processor%20Datasheets/Arm%20Cortex-M4%20Processor%20Datasheet.pdf?revision=904a9fc1-9c66-4816-80bf-ff8c76420e5a&hash=C7B6353BCA7831C236A5509DD62CE8A2
[26] (2021). *ARM Cortex-M7 Datasheet*. Accessed: Sep. 30, 2021. [Online]. Available: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Processor%20Datasheets/Arm-Cortex-M7-Processor-Datasheet.pdf
[27] (2021). *ARM Cortex-M7 Datasheet*. Accessed: Sep. 30, 2021. [Online]. Available: https://developer.arm.com/documentation/ddi0489/d/Chdeajag
[28] D. Patterson and J. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*. Amsterdam, The Netherlands: Elsevier, 2011.
[29] M. Rivain and E. Prouff, "Provably secure higher-order masking of AES," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst.*, in Lecture Notes in Computer Science, vol. 6225, S. Mangard and F. Standaert, Eds. Santa Barbara, CA, USA: Springer, 2010, pp. 413–427, doi: 10.1007/978-3-642-15031-9_28.
[30] E. Ronen, A. Shamir, A. Weingarten, and C. O'Flynn, "IoT Goes nuclear: Creating a Zigbee chain reaction," *IEEE Security Privacy*, vol. 16, no. 1, pp. 54–62, 2018, doi: 10.1109/MSP.2018.1331033.
[31] N. Sehatbakhsh, B. B. Yilmaz, A. Zajic, and M. Prvulovic, "EMSim: A microarchitecture-level simulation tool for modeling electromagnetic side-channel signals," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2020, pp. 71–85, doi: 10.1109/HPCA47549.2020.00016.
[32] H. Seuschek, F. De Santis, and O. M. Guillen, "Side-channel leakage aware instruction scheduling," in *Proc. 4th Workshop Cryptogr. Secur. Comput. Syst.*, M. Brorsson, Z. Lu, G. Agosta, A. Barenghi, and G. Pelosi, Eds., Stockholm, Sweden, Jan. 2017, pp. 7–12, doi: 10.1145/3031836.3031838.
[33] M. A. Shelton, N. Samwel, L. Batina, F. Regazzoni, M. Wagner, and Y. Yarom, "Rosita: Towards automatic elimination of power-analysis leakage in ciphers," 2019, *arXiv:1912.05183*.
[34] STMicroelectronics. (2018). *Introduction to STM32 Microcontrollers Security*. Accessed: Sep. 30, 2021. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337e/DDI0337E_cortex_m3_r1p1_trm.pdf
[35] (2020). *STSAFE-A100 Authentication & Brand Protection Secure Solution*. Accessed: Sep. 30, 2021. [Online]. Available: https://www.st.com/en/secure-mcus/stsafe-a100.html
[36] F. Terraneo. (2019). *Miosix*. Accessed: Sep. 30, 2021. [Online]. Available: https://miosix.org/
[37] P. Yu and P. Schaumont, "Secure FPGA circuits using controlled placement and routing," in *Proc. 5th IEEE/ACM Int. Conf. Hardw./Softw. Codesign Syst. Synth.*, S. Ha, K. Choi, N. D. Dutt, and J. Teich, Eds., Salzburg, Austria, 2007, pp. 45–50, doi: 10.1145/1289816.1289831.

ALESSANDRO BARENGHI is an Associate Professor with the Politecnico di Milano, Italy. He has published more than 80 papers in international peer-reviewed venues. His research interests include computer and network security, formal languages, and compilers.

LUCA BREVEGLIERI was born in Italy, in 1962. He received the M.Sc. degree in electronic engineering and the Ph.D. degree in electronic engineering of information and systems from the Politecnico di Milano, Milan, Italy, in 1986 and 1991, respectively. From 1991 to 1998, he worked as the Information and Communications Technology (ICT) Manager and a Network Administrator. In 1998, he was appointed as an Associate Professor at the School of ICT, Politecnico di Milano. He researched the design of dedicated architectures for computer arithmetic and signal and image processing. Since he was appointed as an Associate Professor, he has been researching mainly the fields of security of digital devices, both dedicated and programmable, of efficient applied cryptographic algorithms, and of the protection against attacks (mathematical and side channel); and secondarily the field of theoretical computer science (formal languages and automata). He has participated in several European Union (EU) and national research projects (MEDEA+, ENIAC, and PRIN), mostly on the security of digital devices. He collaborated with CERN, Geneva, on the design of a digital to analog converter (DAC) for the particle detectors for large hadron collider (LHC), and with STMicroelectronics and Micron on applied cryptography topics. He is a coauthor of over 100 international peer-reviewed journals and conference papers on his research topics. He supervises the supercomputing activities by his university.

NICCOLÒ IZZO is currently pursuing the Ph.D. degree in information technology with the Politecnico di Milano, Italy. He collaborated with Micron on the application of cryptography on emerging memory devices and co-authoring one patent. His research interests include security in traditional and emerging memory subsystems, and microarchitectural side channel attacks.

GERARDO PELOSI (Member, IEEE) is an Associate Professor with the Politecnico di Milano, Italy. He has published more than 90 papers in international peer-reviewed journals and conference proceedings and is a co-inventor of ten patents concerning the design of cryptographic systems. His main research interests include in computer security, cryptography, security in hardware, and in the area of data security and privacy.