A Comprehensive Side-Channel Information Leakage Analysis of an In-Order RISC CPU Microarchitecture

DAVIDE ZONI, ALESSANDRO BARENGHI, GERARDO PELOSI, and WILLIAM FORNACIARI, Politecnico di Milano, ITALY

Side-channel attacks are a prominent threat to the security of embedded systems. To perform them, an adversary evaluates the goodness of fit of a set of key-dependent power consumption models to a collection of side-channel measurements taken from an actual device, identifying the secret key value as the one yielding the best-fitting model. In this work, we analyze for the first time the microarchitectural components of a 32-bit in-order RISC CPU, showing which one of them is accountable for unexpected side-channel information leakage. We classify the leakage sources, identifying the data serialization points in the microarchitecture and providing a set of hints that can be fruitfully exploited to generate implementations resistant against side-channel attacks, either writing or generating proper assembly code.

CCS Concepts: • **Security and privacy → Embedded systems security**; **Side-channel analysis and countermeasures**;

Additional Key Words and Phrases: Applied cryptanalysis, simulation-based side-channel analysis, security by design

ACM Reference format:
Davide Zoni, Alessandro Barenghi, Gerardo Pelosi, and William Fornaciari. 2018. A Comprehensive Side-Channel Information Leakage Analysis of an In-Order RISC CPU Microarchitecture. *ACM Trans. Des. Autom. Electron. Syst.* 23, 5, Article 57 (August 2018), 30 pages.
https://doi.org/10.1145/3212719

1 INTRODUCTION

Side-channel attacks represent one of the most significant threats to the security of embedded systems, which are in charge of an increasing number of tasks in the modern, deeply interconnected era. Indeed, providing confidentiality and data/endpoint authentication on embedded platforms is a widely present and increasing concern, which is also strengthened by the ubiquitous

This work was partially supported by the European Commission under Grant No.: 671668 – H2020 Research and Innovation Programme: "MANGO", and by the European Commission under Grant No.: 688201 – H2020 Research and Innovation Programme: "M^{2}DC".
Authors' addresses: D. Zoni, Politecnico di Milano, Department of Electronics, Information and Bioengineering – DEIB, Via G. Ponzio 34/5, Milano, 20133, Italy; email: davide.zoni@polimi.it; A. Barenghi, Politecnico di Milano, Department of Electronics, Information and Bioengineering – DEIB, Via G. Ponzio 34/5, Milano, 20133, Italy; email: alessandro.barenghi@polimi.it; G. Pelosi, Politecnico di Milano, Department of Electronics, Information and Bioengineering – DEIB, Via G. Ponzio 34/5, Milano, 20133, Italy; email: gerardo.pelosi@polimi.it; W. Fornaciari, Politecnico di Milano, Department of Electronics, Information and Bioengineering – DEIB, Via G. Ponzio 34/5, Milano, 20133, Italy; email: william.fornaciari@polimi.it

https://doi.org/10.1145/3212719

interconnection of everyday objects, including the ones performing critical tasks (e.g., cars and building automation systems) [9, 10, 33, 38, 43, 44, 47]. Cryptographic primitives and protocols have proven to be the prime means to provide the aforementioned security features in an effective and efficient way. However, their use in an embedded scenario calls for a security-oriented design that takes into account both their mathematical strength and their resistance against attacks led by someone having physical access to the computing device. The latter class of attacks, known as side-channel attacks (SCAs), exploit the additional information coming from the measurement of the computing device's environmental parameters to infer information on the data that the device itself is processing. One of the most prominent parameters in this respect is the power consumption of the device, which is proven to be a rich source of information. Indeed, since the pioneering works on extracting secret keys from smart-cards, a significant amount of literature has focused on the improvement of both the understanding of the principles and effectiveness of side-channel attacks [1] and the identification of sound, provably secure countermeasures [46]. A broad spectrum of devices has been successfully attacked via SCAs, ranging from dedicated cryptographic accelerators in Radio-Frequency IDentification (RFID) devices [42] to full-fledged Systems-on-Chip (SoCs), running an operating system and endowed with external DRAM [8]. The reason for such wide success is the fact that the power measurement can be performed either through a minimal modification of the power line to insert a shunt measurement resistor or in a completely tamper-free way measuring the radiated electromagnetic emissions of decoupling capacitors [8]. Open literature classifies the techniques exploiting the information leakage on the power consumption side channel in two sets [34]: *simple power attacks* and *differential power attacks*. The first set encompasses techniques that exploit the changes in power consumption caused by key-dependent divergences in the control flow of the computation, while the second one contains techniques extracting information from the differences in power dissipation caused by discrepancies in the switching activity of a device induced by processing different values. In the following, we will focus on the second set of techniques, i.e., differential power attacks, since it is the one where the architectural characteristics of the underlying CPU play a stronger role. Indeed, while a successful simple power attack exploits flaws in the application control flow design, differential power attacks are tightly coupled to the way data is processed. For this reason they require a combined understanding of the hardware and software platforms implementing the cryptographic primitive to prevent unwanted information leakage.

Counteracting power-consumption-based side-channel attacks requires the designer to break the link between the amount of power consumed during the computation and the data being processed. Depending on whether the issue of side-channel resistance was tackled devising dedicated hardware coprocessors or software implementation of the cryptographic primitives, the proposed solutions target different design layers encompassing the technology and gate levels and the architecture and microarchitecture of the design, up to the software side if any. Ad hoc technology libraries were shown to be highly effective in preventing power-based SCAs [15, 50], although they require a considerable area and power consumption increase, and a significant engineering effort to be interconnected to standard CMOS components. Conversely, the design of balanced logic circuits [50] to provide a data-independent power consumption and the use of circuits that split the computation of Boolean functions into shares [46] to achieve a randomized power consumption have also proven to be successful in hindering SCAs, without the need to resort to a custom technology library. However, such logic level SCA countermeasures impose a performance and/or energy overhead that is far from negligible [46, 50] if they are applied to the whole system without considering that some parts of the design are not actually leaking. These dramatic overheads imposed by technology and logic-level countermeasures make their use not viable to

protect an entire general-purpose CPU based on a Reduced Instruction Set Computing (RISC) design. This, in turn, leaves software implementations of cryptographic primitives as a potential target for side-channel attacks when running on full-fledged CPUs. Indeed, in [8, 21, 22], SCAs to software implementations of standardized ciphers have proven to be successful against both RISC and Complex Instruction Set Computer (CISC) CPU targets.

To this extent, the problem of preventing key retrieval via side channel has been also approached at the software level, either by randomizing the data being computed while preserving the semantic equivalence of the results [5, 29] or by changing continuously the code employed to perform the computation [2–6]. Traditionally, software-based countermeasures rely on the architectural information of the CPU executing the code to prevent unintended side channels. However, as our findings will highlight, the leaked side-channel information depends on the actual CPU microarchitecture. The architectural view observed by the software architect is therefore seen as a viable simplification to implement software that however can be proven to be not sufficient to design SCA-resistant cryptographic libraries. Besides, the software architect designing SCA-resistant cryptographic primitives is willing to be informed on the possible leakage source in the form of a set of hints that emerged as the result of a more accurate microarchitectural side-channel characterization. In particular, those hints should be general enough to be used with all the CPU implementations that fall in the same class of the one from which the hints have been extracted.

This work aims at providing a precise, "clean room" characterization of the effects of the microarchitectural design choices on the side-channel leakage of a general-purpose RISC CPU. The analysis leverages the netlist-level simulation to ensure a "clean room" environment that removes the uncertainty of the measurements while enabling accurate data collection on a per-module granularity.

The intent of such characterization is to precisely pinpoint which portions of the CPU microarchitecture leak information via side channel when a cryptographic primitive is executed, thus demonstrating that the architectural view alone is not enough to deliver SCA-resistant software. An additional effort of our investigation aims at generalizing the obtained results for the class of in-order RISC CPUs, thus serving a twofold objective. First, we take steps from the coarse-grained characterization of the side-channel leakage sources in the CPU microarchitecture [16] and the pioneering works that analyze 8-bit PIC and AVR microcontrollers [24, 47] to carefully identify, for the first time, the side-channel information leakage in a 32-bit in-order RISC CPU. This effort also allows us to extend and complement the current findings in the open literature. Second, we extract a set of practical guidelines from the investigated CPU to improve the design of SCA-resistant software. The application of such guidelines to the benchmarks used in our investigation effectively removes the side-channel leakage that was revealed to lead to the correct key guess. We note that such guidelines can be also included in the back end of the OpenRISC compiler to automatically emit SCA-resistant code.

Contributions. Starting from a precise "clean room" characterization of the side channel of the open-hardware RISC CPU implemented within the *ORPSoCv3* SoC [30], this work encompasses three different contributions:

- **Microarchitectural components inducing side-channel information leakage.** The reported analysis points out a serialization effect concerning sensitive signal values asserted on the same bus in two consecutive clock cycles, thus inducing an easily captured information leakage. Indeed, we note that unintended serialization of sensitive data values is the source of the side-channel leakage arising from the *write-back* stage, the *forwarding paths*, and the *operand dispatch* stage of the pipeline. As a consequence, the reasons for the ineffectiveness of software-based SCA countermeasures (implemented according to the current

best practices) are inferred, while the need to extend the current architectural-level perspective in applying SCA countermeasures to encompass the microarchitectural characteristics of the underlying CPU is also supported. The `load/store` unit (LSU) represents another source of exploitable side-channel leakage due to its typical microarchitectural implementation, which retains the last loaded or stored value to minimize the power consumption through reducing the number of unnecessary signal toggles. Indeed, such a design strategy entails that values fetched by two `load` instructions may leak sensitive information on the power side channel, regardless of the number of non-LSU instructions being processed between them and regardless of register reuse. We verified that also the design strategy of the LSU component is accountable for the ineffectiveness of software-based SCA countermeasures. In addition, we highlight how the signals driving the values into/from the output/input ports of the register file (RF) are accountable for the information leakage arising from the transitions between two values that are consecutively transmitted to/read from the RF, regardless of the specific RF locations addressed by the operations at hand. This observation confirms and extends the results about the RF leakage in [47], allowing us to pinpoint the reasons underlying the information leakage arising from the sequence of instructions with no register reuse (at the level of Instruction Set Architecture). Moreover, it allows us to assess to what extent an automatically performed random renaming of the registers employed by an assembly code snippet [36] may be effective to prevent the information leakage from the RF.

- **Microarchitectural hints to improve the application of SCA countermeasures.** We generalize the findings of our investigation in a set of programming hints that allow us to prevent the microarchitecture-dependent side-channel leakage when applied. Our hints apply to any in-order RISC CPU, with some of them being dependent on specific design choices of the CPU components. In particular, they describe how to modify the architectural-level description of the SCA countermeasure, i.e., the assembly code, to take into account the microarchitectural serialization effects arising across several components of the CPU pipeline (e.g, rescheduling some instructions and/or inserting new dummy instructions). We validate the effectiveness of the proposed hints applying them to our case study microbenchmarks, where their application is shown to prevent serialization-induced leakage. We note that the constraints kept into account to properly apply the SCA countermeasures might be fruitfully employed in the back end of a common `C` compiler tool chain during the optimization passes that precede the binary code emission.
- **Ghost peak characterization.** Our detailed analysis allows us to precisely pinpoint the causes of side-channel behaviors appearing as information leakage, despite not containing any secret key-related information. We clarify the reasons causing the nonspecific SCA robustness test, performed via *t-test* [23, 25], to erroneously report an implementation as potentially leaking information, and we provide a detailed explanation of the causes of this behavior. We also suggest a complementary test that should be paired with the common nonspecific *t*-test to cope with such unwanted false alarms.

Structure of the article. The rest of the article is organized as follows. Section 2 summarizes the fundamentals on the current state of the art of power-based SCA and reports the related work. Section 3 describes the reference CPU architecture employed and the power consumption simulation framework. Section 4 contains the results of our analysis of the side-channel information leakage and the classification of its sources. Section 5 summarizes the findings of the proposed investigation, taking steps from the accurate microarchitectural side-channel analysis. Section 6 draws our conclusions.

2 PRELIMINARIES

In this section, we provide the preliminary notions on power analysis attacks and survey the existing work in the realm of design time assessment of side-channel leakage.

2.1 Power Analysis Attacks and Countermeasures

The typical differential power analysis workflow is an instance of either a *known plaintext attack* or a *known ciphertext attack* against a symmetric cryptographic primitive, aiming at retrieving the secret key being employed by the cipher. The attacker is assumed to know all the details of the implementation of the cipher and is able to measure the power consumption of the device to derive information regarding the secret key from it. The main strength of an SCA lies in the possibility of considering the effect of the secret key bits on the computation of the cipher separately, instead of as a whole, leading to a reduction of the security margin.

The attack workflow starts by choosing an intermediate value in the cipher computation depending on a small portion of the key (usually 8 bits) and a known quantity (usually the plaintext in input or the ciphertext in output). The side channel (e.g., the power consumption) is continuously measured during the execution of the operation computing the said intermediate value, for a large set of different, randomly distributed known inputs. Subsequently, the attacker tries to predict the actual power consumption of the device, according to a chosen *leakage model* [31], relying on the knowledge of the inputs of the cryptographic primitive and constructing a hypothesized power consumption for each of the values of the secret key portion taken into account. Each one of these predictions on the power consumption is compared with the measured side-channel value at each considered time instant, through the use of a statistical test. The correct value of the secret key portion is revealed, as the prediction depending on it will fit best the measurements. From a computational point of view, the attacker is required to compute hypothetical power consumption values for a number of hypotheses, which grows exponentially in the number of key bits involved in the computation of the chosen intermediate value. As a consequence, the attacker will choose as an intermediate value of the computation one that involves the least amount of key bits to be guessed (with the limit case being a single one).

2.2 Leakage Modeling

The choice of which intermediate value of the computation of an algorithm should be predicted and which data-dependent model of power consumption should be chosen to build hypotheses on its side-channel values is typically obtained by exploiting a trial-and-error strategy, especially in the case of large devices [31]. However, one of the main strengths of SCAs, as claimed by [31], is that even a reasonably coarse modeling of the power consumed by the device is sufficient to lead an attack.

The most significant portion of data-dependent power consumption in CMOS devices is constituted by the switching power consumption of the circuit. Following this consideration, it is common to make the assumption that all the logic components of the same kind will consume the same amount of power when switching, and thus the switching power required is proportional to their amount. As a consequence, a popular model for the data-dependent power consumption of memory elements in a circuit is the *Hamming Distance* (HD) between two values held by them in consecutive cycles, which considers the amount of single-bit memory cells switching when the new value is memorized. Such a model requires the attacker to have further information with respect to one obtainable simply predicting an intermediate value of the algorithm: indeed, he or she should be aware of which memory elements are storing it, and what was their previous content. The information obtainable from this model is formalized in [7] according to the notion of *transition leakage*.

Definition 2.1 (Transition Leakage). Given two Boolean values held by a single-bit memory element in subsequent clock cycles, the transition leakage is defined as the portion of the side-channel behavior directly dependent on the difference, i.e., the bitwise eXclusive OR of the two values. The transition leakage of a multibit memory element is defined, by extension, as the portion of the side-channel behavior proportional to the count of single-bit values exhibiting a Boolean difference equal to 1.

However, given the fact that, in practice, the attacker may not be knowing the structure of the device being targeted with a precision sufficient to determine which combinatorial and which sequential elements are present, an alternate power consumption model commonly employed is the *Hamming Weight* (HW) of an intermediate value being computed by the algorithm. This model is intended to capture the power dissipated by logic gates in charging their fan-out, and is defined in literature as *value leakage* [7].

Definition 2.2 (Value Leakage). Given a logic circuit computing a value, its value leakage is defined as the portion of the side-channel behavior depending on the number of signals being set during the aforementioned computation, i.e., the Hamming weight of the computed value.

The HW model represents another popular choice, as it requires extremely limited information on the structure of the computing device [18]. In particular, in [8], the authors were able to perform a successful key retrieval on a 1GHz ARM SoC running Linux, employing the HW model, regardless of the lack of knowledge of the detailed CPU implementation. A straightforward albeit useful observation is that the HW model may be capturing transition leakages should the transitions happen either from or to a fixed all-zeros value.

2.3 Statistical Distinguishers

A large variety of statistical tests have been introduced. In particular, the Difference-of-Means (DoM) test was first employed to derive the secret key of a DES cipher. Subsequent works investigated ways to enhance the accuracy and the efficiency (in terms of number of measurements) of the differential power analysis [19]. Recent work [28] proved that when the leakage model is perfectly known to the attacker, the optimal statistical distinguisher depends only on the statistical distribution of the noise superimposed onto the measurements. Under the assumption of the additive measurement noise being Gaussian, if the leakage model of the targeted sensitive operation arises from a linear relation between the operand values and the power consumption, the optimal statistical distinguisher is the sample Pearson correlation coefficient. The sample Pearson correlation coefficient r is a biased estimator of the actual Pearson correlation coefficient ρ between two random variables when both of them are normally distributed. Let X be the variable representing the instantaneous power consumption values of the device when processing randomly distributed inputs, and let Y be the predicted power consumption according to a model depending on a given key value. Consider X, Y as two random variables, with their expected values being $\mathbb{E}[X], \mathbb{E}[Y]$. When employed as a statistical distinguisher in SCAs, the sample Pearson correlation coefficient is computed between two realizations of the said variables. Recalling that the expression of Pearson's correlation coefficient is

$$\rho = \frac{\mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y]}{\sqrt{\mathbb{E}[X^2] - \mathbb{E}[X]^2}\sqrt{\mathbb{E}[Y^2] - \mathbb{E}[Y]^2}},$$

the expected value of its sample estimator, computed over two sample sets, both taken from normal populations $X = \{x_1, x_2, \ldots\}$, $Y = \{y_1, y_2 \ldots\}$ and of size n, is approximately $\mathbb{E}[r] = \rho(1 - \frac{1-\rho^2}{2n})$, with an even more exact result given by an infinite series containing terms of smaller magnitude.

Elaborating the previous equation, the recommended unbiased estimator for the correlation coefficient is obtained as $\hat{\rho} = r(1 + \frac{1-r^2}{2(n-3)})$ [39]. In the setting of SCAs, n is relatively high (usually greater than 50), and thus the bias is ignored assuming

$$\hat{\rho} = r = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n\sum x_i^2 - (\sum x_i)^2}\sqrt{n\sum y_i^2 - (\sum y_i)^2}}.$$

Concerning the use of statistical tools to extract useful information from a side channel, a separate mention should be made concerning the use of a statistical test to distinguish whether a variation in the circuit inputs causes a measurable change in its side-channel behavior. This approach, pioneered in [18], proposes (as a leakage-model-independent test) to compare two sets of power dissipation measurements obtained by employing a fixed key value and gathering the first set with a uniformly distributed set of input values, while feeding a single fixed input for the second set. For each time instant in which the samples are collected, a t-test is performed to determine whether the set of samples collected with the uniformly distributed inputs has the same expected value of the set collected with a constant input. In case the t-test accepts such a hypothesis, the implementation is not providing sufficient information for a successful (first-order) attack in the time instant to which the compared sample sets are pertaining. However, if the t-test rejects the hypothesis of the expected values being equal, the authors of [18] state that such a result "confirms the probable existence of secret-correlated emanations." Due to the convenience of not requiring one to model the side-channel behavior of a device, the *t-test* was suggested in [25] as a testing methodology to assert the SCA resistance of an implementation of a symmetric cipher.

2.4 Related Work

Framing accurately the form in which the information is leaked on the power consumption side channel is a longstanding issue that has been tackled at different levels, among which are a system view of the device under exam [37], a Register-Transfer Level (RTL) view [41], a gate-level exam [47], and a transistor-level characterization [49]. Depending on the chosen level, an increasing amount of detail about the information leaked by the power consumption is obtained from the logic simulation, at the cost of a corresponding increase in the computational requirements. However, starting the inspection at the lowest possible level, i.e., transistor-level simulations, does not provide insights on whether the SCA vulnerabilities are caused from issues present at the same level of abstraction or in any of the higher ones. We note that applying SCA countermeasures only at the lowest levels incurs significant penalties in terms of efficiency, due to a less structured description of the device, which either forces a blanket application of countermeasures or requires technically challenging solutions to be applied manually to large portions of it. Contrariwise, mitigating side-channel vulnerabilities only at the topmost level may result in information leakage, as the applied countermeasures are not aware of low-level effects [49].

The authors of [37] present a system-level cycle-accurate simulation approach to obtain the side-channel behavior of SoCs. The authors assess the leakage of both a software Advanced Encryption Standard (AES) cipher implementation in ARMv5 assembly and an RTL-described hardware implementation of the same cipher, highlighting the fact that exploitable information is leaked by both of them. While the analysis performed on the simulation results reports coherent leakage for the unprotected implementations, the authors state that the protected ones show a leakage that cannot be explained at system level.

In [41], the authors compare the results of an attack carried out against an ASIC implementation of the AES cipher with one performed against an RTL-level simulation of the same core described in a Hardware Description Language (HDL). The reported results provide good evidence toward

the fact that RTL-level simulation still retains the ability to predict side-channel leakage, albeit underestimating the number of measurements required to recover the secret key by two orders of magnitude with respect to measurements taken on a physical device. Sensible causes for the mismatch are the absence of measurement noise in the simulation and RTL-level simulations not taking into account the propagation delay of the signals.

In [49], the authors analyze the results of SCAs with a circuit-level simulation of an AES core, examining different modeling strategies employed for the capacitive load relative to the interconnection between logic gates. The work shows that choosing a model for such physical factors impacts in a significant way on the predicted effectiveness of the key retrieval procedure.

In [45], the authors describe a methodology to automatically implement ad hoc hardware accelerators employing side-channel-resistant logic, which are employed to compute the sensitive portions of a software-based cryptosystem. Differently from our work, the methodology focuses on the automatic extraction and generation of the hardware accelerators rather than analyzing the side-channel leakage sources in the target CPU. Indeed, the selection of the portion of the software to be hardware accelerated is supposed to be manually pointed out by the software architect.

The work presented in [16] observes that the side-channel leakage from a software computation is stemming from components on the datapaths present between the register file, ALU, and memory, and employs this observation to design a side-channel countermeasure relying on Dual-Rail Precharge logic. In this work, we carefully identify the side-channel leakage sources in the CPU microarchitecture at hand, and we present guidelines to apply software-based SCA countermeasures without resorting to technology-level solutions.

In [11], the authors present a compiler-based approach to insert two software countermeasures (i.e., *Boolean Masking* and *Random Precharging*) to protect cryptographic algorithms against power-based differential side-channel attacks. In particular, the methodology leverages on the data dependencies within the algorithm that can possibly highlight side-channel leakage to drive the application of the countermeasures. The differentiating point from this contribution is the fact that we analyze the microarchitectural structure of a 32-bit CPU, while [11] relies on black-box physical measurements of the power consumption for an 8-bit Atmel AVR ATmega μC to locate the leaking points. Furthermore, our analysis shows that without taking into account the microarchitectural features of a 32-bit CPU, the Boolean masking countermeasure becomes ineffective.

In this work, we employ a gate-level simulation approach, estimating, for the first time, the power dissipated by the circuit on a postsynthesis and map design of a full-fledged pipelined RISC CPU. The methodology trades off the accuracy provided by circuit-level simulations with the ability of providing a faster feedback into the design loop, while retaining the capability to produce a detailed analysis of the portions of the design, pinpointing the sources of exploitable side-channel leakages down to the individual CPU modules. When compared directly with an RTL-simulation approach [37], which exploits the toggle count as a power consumption gauge, our strategy allows us to provide a time-based power trace derived from actual technology library information and take into account the effect of glitches, providing an overall best fit of the actual device. A similar intent in attributing with precision the blame of causing a side-channel information leakage onto a specific processor component is presented in [47], where the authors perform a side-channel leakage evaluation on two implementations of an 8-bit AVR μC, namely, a commercial- grade Atmel ATMega32 and an FPGA implementation of the same core. The focus of their analysis is to infer whether the leakage exhibited by the device can be attributed to the register selection mechanism in the register file. To this end they apply specific tests to the measurements taken on the physical devices. While sharing the intent of investigation with [47], we provide a reproducible and accurate binding between the leakage causes and all the individual CPU components and their interactions, proposing a taxonomy of the leakage sources. Finally, in [24], the authors employ

side-channel information derived from an 8-bit PIC16F687 microcontroller with the purpose of performing reverse engineering of the running code. This work performs an analysis of the leaking portions of the microcontroller architecture to distinguish instructions from their power signature, while our work focuses on the side-channel leakage revealing information on the processed data.

3 ARCHITECTURE AND SIMULATION FLOW

This section overviews the framework used to characterize the side-channel information leakage of the reference CPU microarchitecture according to the popular value leakage and transition leakage models described in Section 2.2. The 32-bit in-order RISC architecture used as a reference platform for the embedded, in-order CPU family is detailed in Section 3.1. The side-channel information leakage assessment workflow is described in Section 3.2, and the description of the power estimation model employed is provided in Section 3.3.

3.1 The OpenRISC Reference Platform System-on-Chip

The employed case study architecture is the OpenRISC Platform System-on-Chip (ORPSoC) Version 3 [30], implementing the royalty-free OpenRISC 1000 architectural specification [40]. Implementations of this architecture specification have been realized in silicon [17], proving its relevance as a test platform, and is currently used by the AR100 power management unit in Allwinner SoCs [32] and supported in the mainline Linux kernel since version 3.1 [12]. Moreover, its free availability allows an easy reproducibility of the results.

The ORPSoC contains a single 32-bit, single-issue, in-order CPU with a four-stage pipeline, the main memory, and a Wishbone [27] compliant bus connecting the two, implementing a *Single Read/Write* data transfer protocol. The reason for the deviation from the common five-stage pipeline design is that the `load-store` unit of the ORPSoC is able to start executing the `load/store` instruction in the execution stage of the pipeline, as it is endowed with a dedicated adder to compute the memory address, forfeiting the need for receiving it in input from an ALU computation result. The implementation of the reference architecture considered in this work does not implement caches or prefetchers, matching common implementation strategies for several low- and high-end microcontrollers (e.g., Cortex-M0, Cortex-M3, Cortex-M4, and Cortex-M7) [35].

A schematic view of the considered SoC is provided in Figure 1 implementing both an instruction and a data bus. The access to each one of the buses is mediated by a sequential Bus Interface Unit (BIU) manipulating signals coming from the CPU exposing a Wishbone-compliant interface to the on-chip bus fabric. The BIU requires a clock cycle for the signals to traverse it when asserted by the CPU.

The Wishbone bus architecture is capable of managing multiple masters and slaves, and takes two clock cycles to propagate the signals from the BIU to the main memory ports, while a datum requires one clock cycle to be propagated back after memory observes the request. The main memory read ports retain and propagate the contents of the last requested address until the next request is made, saving unneeded signal toggles.

The Instruction Fetch (IF) stage of the CPU fetches a single, fixed-length instruction per clock cycle, save for the requested transfer time, and updates the program counter following the standard MIPS-like architecture approach as described in [26]. Due to the memory access latency, the IF stage receives the fetched instruction after one additional cycle starting from the clock cycle when the memory instruction port observes the request. For the sake of clarity, we point out that all the pipeline stages in the CPU are frozen when the IF stage is stalled waiting for the fetched instruction. The performance impact taken by this architectural choice is not affecting the precision of the side channel leakage assessment, since the datapath transition patterns are preserved. In particular, a

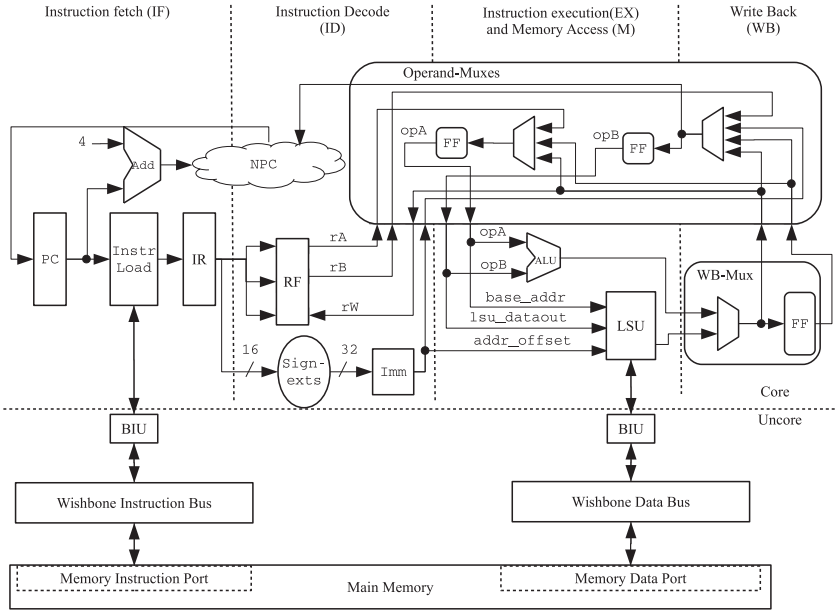


Fig. 1. Overview of the OpenRISC System-on-Chip reference platform, depicting the OpenRISC CPU, Bus Interface Units (BIUs), and two Wishbone-compliant buses toward the two-ported memory. The FF blocks indicate latched modules.

potentially more conservative estimate of the entity of the vulnerabilities can be observed due to the lower amount of temporal superimposition of the instructions in the pipeline.

The Instruction Decode (ID) stage extracts from the instruction loaded into the Instruction Register (IR) the information required to drive the Register File (RF) and the immediate operand logic to set up the operands for the execution stage. Moreover, the ID stage also forwards the signals derived from the instruction *operation code* (opcode) to the Functional Units (FUs), i.e., the Arithmetic-Logic Unit (ALU) and the Load-Store Unit (LSU), present in the pipeline. The operands are not directly forwarded from the register file into the FUs; instead, they are collected side by side to the values forwarded by the pipeline stages and multiplexed in the Operand-Muxes module that actually implements the microarchitectural forwarding paths (see Figure 1). In particular, the Operand-Muxes module presents the required operands to all the available FUs exposing them on the latched signals denoted as opA and opB in Figure 1, which are derived by multiplexing inputs from the RF, the immediate value from the IR, and the forwarding paths.

The EXecution and Memory access (EX/M) stage contains both the ALU and the LSU, which compute the result of the decoded instruction, in turn collected by the subsequent pipeline stage. Both units are always active, regardless of the actual instruction transiting in the EX/M stage. The ALU is designed as a fully combinatorial module taking three primary inputs, i.e., the two operands coming from the Operand-Muxes and the opcode. All the supported instructions are executed in parallel and the opcode is employed as the driving signal of a multiplexer that selects the result meant to be propagated.

The Write Back (WB) stage of the pipeline is composed of two modules. The first one is a combinatorial multiplexer that selects the actual EX result between the ones provided by the ALU and the LSU and propagates it to the Operand-Muxes module. The second module is a latched component that retains the results from the first module to prevent value loss during pipeline stalls.

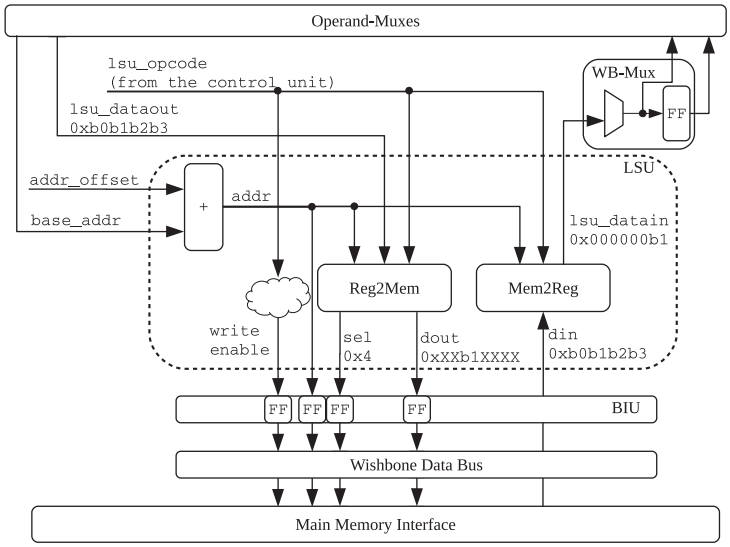


Fig. 2. Detailed view of the Load Store Unit of the ORPSoC. The Mem2Reg and Reg2Mem data alignment modules are used to implement the `load` and `store` instructions for half-word and bytes. The FF blocks indicate latched modules.

The Load-Store Unit (LSU). The OpenRISC CPU follows a strict `load-store` architecture design, and thus the only operations able to access the main memory are `load` and `store` instructions. All memory accesses are mediated by the LSU, detailed in Figure 2, which communicates with the main memory via the BIU. The memory word is 32 bits wide, and memory accesses are made to 32-bit aligned addresses. However, the OpenRISC 1000 architecture specification contains also `load` and `store` instructions allowing it to load half-words and bytes, thus requiring the LSU to handle the proper alignment of the contents to be transmitted, deducing it from the last 2 bits of the memory address. Concerning `load` operations, the LSU needs to extract bytes and half-words from the 32-bit loaded word and perform zero extensions whenever required (e.g., by the `l.lbz` `load` byte with zero extend instruction). To this end, the Mem2Reg module of the LSU receives as input the 32-bit data word from the memory through the BIU (reported in Figure 2 as the `din` signal), the memory address to determine the data alignment, and the `lsu_opcode` that contains the actual `load` operation to be performed (i.e., the amount of data to be loaded). The Mem2Reg module, depicted in Figure 2, is a fully combinatorial module acting on the 32-bit word fetched from the memory to transparently present the requested data to the multiplexer in the WB stage. Figure 2 reports a sample dataflow of a `load` byte instruction requiring the second byte of the data word: the `0xb0b1b2b3` word is loaded from the memory, and the realigned `0xb1` byte is supplied by the Mem2Reg module to the WB stage via the `lsu_datain` signal.

Whenever a `store` operation must be executed, the LSU coordinates the memory transaction while the Reg2Mem module provides the proper alignment of the bytes that should be stored into the main memory. The Reg2Mem module takes as inputs the 32-bit data word to be stored, the destination memory address, and the `lsu_opcode` employed to determine the amount of data to be stored. The Reg2Mem sets the byte to be written back in the correct position within the 32-bit `dout` signal, employing the one-hot encoded `sel` signal to inform the main memory on which actual byte(s) will be written back to the target address location. Both signals are latched into the BIU, and once stable, the data bus is arbitrated and the byte is stored back into the main

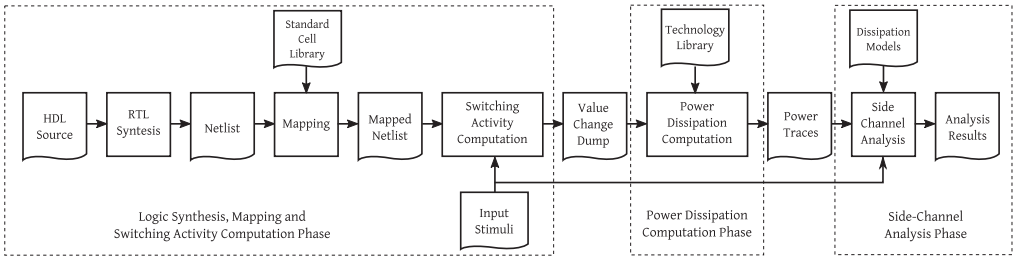


Fig. 3. Overview of the simulation methodology. The HW design is synthesized, mapped onto the standard cell library, and simulated, to obtain its switching activity in the form of a Value Change Dump file. The said switching activity is fed into a power consumption simulator, which generates the power traces required to evaluate the goodness of fit with the SCA prediction models.

memory. It is worth noticing that all the bytes that are not selected for write-back can be assigned to any value depending on the Reg2Mem implementation, as they are specified to be *don't care*s by the OpenRISC 1000 specification. Figure 2 reports an example of the dataflow of a `store` byte instruction, requiring the retention of the second most significant byte of the memory word with value `0xb0b1b2b3`. This results in the Reg2Mem module outputting only the byte to be updated on the `dout` bus and sets the selection signal to $(0100)_2$ to mark it as the one to be stored.

3.2 Simulation Workflow

The proposed workflow[1] (see Figure 3) is general enough to allow an SCA vulnerability assessment of any architecture for which the HDL description is available. In our settings, we employed a Xeon E5-2650 v3 machine, with 64GiB RAM running an `x86_64` Ubuntu 14.04 OS. Starting from the HDL description of the ORPSoC, in the *Logic Synthesis, Mapping, and Switching Computation Phase*, the design is first synthesized and mapped onto the 45nm, standard-cell library FreePDK [48], employing Cadence Encounter RTL Compiler, with a target frequency of 100MHz for the main clock, while not enforcing area constraints. The gate-level netlist activity is simulated by running multiple times the software benchmark on a set of randomly distributed inputs to extract the corresponding Value Change Dump (VCD) data. The VCD contains timing information for all the transitions of each signal of the simulated architecture. To detect the beginning and the end of the simulation of a single execution of the benchmark, we inserted a few ad hoc `nop` instructions at the beginning and end of the benchmark code.

In the *Power Dissipation Computation Phase*, the Cadence Encounter Power System toolchain is used to extract the power traces, starting from the VCD computed in the previous phase and the FreePDK technology library [48]. A power trace for each instance of the simulated benchmark is computed for the CPU, without taking into account the main memory and the bus interconnect. Despite being not directly taken into account, the memory and interconnect indirectly contribute to the computation of the power consumption of the CPU, as their values are sampled at the LSU and IF modules. Focusing on the CPU power consumption allows us to pinpoint the sources of SCA leakage, avoiding the power noise effects introduced by additional components that are not of interest in this work. This choice matches also the typical avenue for an SCA on modern SoCs, where separate power lines for the CPU and the rest of the components are often present, allowing an attacker to single out the power consumption of the CPU. The use of ad hoc `nop` instructions, to mark the beginning and the end of each benchmark instance, ensures that the extracted power

[1] https://github.com/GerardoPe/OpenRISC-Gate-Level-Simulation-SCA.

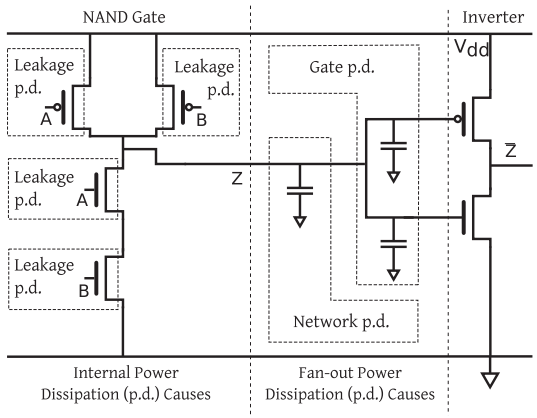


Fig. 4. Depiction of the power consumption model employed by the Cadence RTL Compiler Power Engine for a logical-and equivalent circuit made out of a NAND and an inverter.

traces are perfectly aligned in time, a crucial factor to obtain accurate results when evaluating side-channel information leakage. The power traces are computed considering a 5ns time precision, and thus, two power samples per each simulated clock cycle are collected. We note that each power sample is constituted by the sum of the dissipated power over the entire half-clock cycle, thus also including the contribution due to the glitches.

Each power trace is then stored paired with the corresponding input data, to be employed in the *Side-Channel Analysis Phase* of the workflow (see Figure 3). The SCA leakage characterization stage is made of a C++ of the Pearson-correlation-based side-channel analysis [31]. The implementation computes the sample correlation coefficient for each one of the collected power sample sets and reports the results to be analyzed. To the end of providing a sound statistical evaluation of the results, we collected 2,000 simulated power traces (employing, for all our benchmarks, ≈1 hour, on average, for the VCD generation, and ≈10 hours for the power trace generation), resulting in a confidence interval of around ±0.03 for the sample Pearson correlation coefficient r, with a 95% confidence level. Given the complete absence of measurement noise, we deem the width of the confidence interval to be narrow enough for the purpose of our analysis. Computing the Pearson correlation coefficient for 2,000 traces, employing 256 key-dependent models takes tens of milliseconds with a Matlab implementation. We note that, in case multiple models need to be evaluated, an optimized implementation of the same computation is able to obtain the results of all the tests prescribed in [23] in around 20 minutes.

3.3 Power Dissipation Computation

Starting from the gate-level netlist of the OpenRISC CPU and the VCD, power traces are computed at a module granularity. This choice allows a balanced tradeoff between the possibility to observe a portion of the circuit that has a well-defined architectural functionality and the need to investigate the information leakage sources among the CPU submodules.

The Cadence RTL Compiler Power Engine is the element of the Cadence Encounter Power System that computes the total power consumption accounting for all the transitions of each signal in the design for which the power has to be estimated within the specified timeframe (i.e., 5ns in the reported scenario). The implemented power model takes into account four separate contributions to estimate the power consumption of the considered logic circuit as depicted in Figure 4, where

a NAND gate and a not cell are considered. First of all, the *internal power* dissipation of a logic gate is the sum of the power consumed by the short circuit taking place at switching time and the one dissipated when charging or discharging the gate-to-substrate capacitance. The data to compute the *internal power* are obtained from the technology library and are a function of the input slew rate, the drive strength of the transistors, and the load driven by the output of the gate. The second contribution to the circuit power dissipation is the *leakage power* dissipation, i.e., the power dissipated due to the leakage current between source and drain in modern transistors that, due to a reduction of the threshold voltage, cannot be completely turned off. The dissipated leakage power is totally technology dependent and thus fully determined by the characteristics of the exploited technology library. The third and fourth contributions are a consequence of the *fanout power* dissipation, i.e., the power dissipated when charging or discharging the capacitive load of the networks connected to the output of the logic gate at hand. Such a power consumption includes the energy required to charge the capacitance of both the input lines of the connected gates (`Gate p.d.` in Figure 4) and the wires linking the output of the logic gate with the inputs of each gate on its fanout (`Network p.d.` in Figure 4).

The power dissipation estimation of a module is computed as a natural extension of the gate-level one, i.e., the sum of all the power consumption of its gates considering the instances of cells of the technology library present in it. The power dissipation of all said gates is part of the considered module, with the exception of the *fanout power* on the primary output signals. The *fanout network power* is usually accounted for in the signals that are directly connected to the logic that is downstream with respect to the primary output signals. Conversely, a change in the primary inputs of a module causes a power consumption that is attributed to the module itself, due to the leakage and internal power consumed by the logic gates driven directly or indirectly by those input signals.

4 SIDE-CHANNEL LEAKAGE ANALYSIS

This section describes the side-channel information leakage of the reference CPU presented in Section 3.1. In particular, the section characterizes the said SCA leakage with two main objectives: first, to motivate the importance of using the microarchitectural side-channel information to design SCA resistant software, and second, to precisely characterize the side-channel information leakage evaluated on an open hardware microarchitecture to later generalize some of the findings to the class of in-order RISC CPUs.

The CPU is initially analyzed as a whole and, to increase the precision in pinpointing the information leakage, different CPU modules are investigated in isolation: the Arithmetic-Logic Unit (ALU), the Register File (RF), and the Datapath, as made of the LSU, the Operand-Mux, and the WB stage modules. The side-channel leakage analyses have also been performed on the remaining modules, namely, the IF pipeline stage and the control logic unit, but no leakage is reported, in accordance with their behavior being data independent, and thus they are omitted in the following description. To the end of exposing the side-channel leakage caused by microarchitectural features of the CPU at hand, we devised ad hoc assembly benchmarks to activate the datapaths according to the observation presented in [16], i.e., the paths connecting the memory to and from the RF, and the RF to itself via the pipeline. Among the viable instruction sequences fitting this purpose, we chose some that are ubiquitous in the software implementations of several cryptographic libraries.

Benchmark 1 loads two operands from the main memory, a known input and a secret key; combines them via `xor`; and stores back the result into the main memory. This execution pattern matches the key addition present in symmetric block ciphers, where the round key is added to the cipher state via bitwise eXclusive OR (`xor`). We note that this is a widely employed approach, as seven out of the nine ISO standardized block ciphers [20] perform key addition only with this

strategy, and the remaining two mix this approach with an integer-addition-based one. We note that, while alternative instruction schedules are possible for a key addition, in particular ones reducing the amount of transfer to/from the main memory, the effects observed in *Benchmark 1* are still relevant. Moreover, even in the fortunate case of a schedule free from interactions with the main memory, the side-channel leakage we report from other CPU components is still matching the behavior of the device.

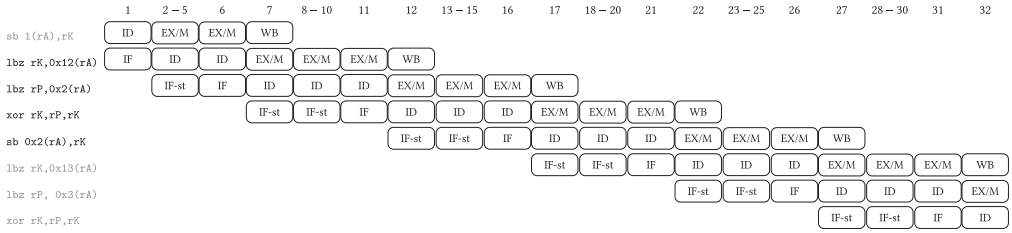
The proposed investigation highlights the side-channel information leakage as a consequence of the *data serialization* effect imposed by the microarchitecture on the two operands and not due to the specific instruction executed to combine them, thus allowing us to generalize the result to any instruction that combines the secret key and the known input.

Benchmark 2 combines two values, known input and secret key, contained in two registers with two copies of the same random value held in a separate register. Similarly to the scenario in *Benchmark 1*, different schedules for the instructions are possible, depending on the number of available registers and the order of the masking employed. However, we note that in this case it is even less likely to have a schedule free from memory interactions due to the increased register pressure caused by masking schemes. This instruction sequence is the one computing the randomized encoding required by all the Boolean masking countermeasures [31], which are the ones providing provable security guarantees on symmetric ciphers. Our results show that a possible reduction in the security margin of a masking scheme may happen if the known input and secret key are processed by two subsequent, unrelated operations both as first operands or both as second operands.

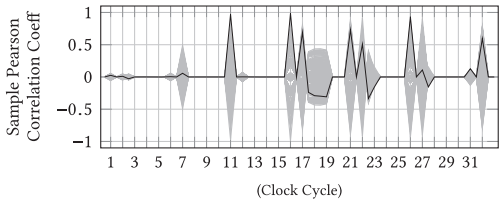
The Pearson's sample correlation coefficient r is used as the statistical tool of choice to quantify the fitness of the power model to the simulated power consumption. We employ the *Hamming Weight* of the xor between the input and secret key as our power consumption prediction, and we observe that, while it correlates successfully with a significant amount of instantaneous power consumption values, the motivations for this correlations are diverse. The choice of Pearson's r was made in accordance with the results on the optimality of the statistical distinguisher presented in [28] (see Section 2).

Figure 5 and Figure 6 report the results for *Benchmark 1* and *Benchmark 2*, respectively, obtained by computing r between the power consumption of either the entire CPU (Figure 5(b) and Figure 6(b)), the ALU (Figure 5(c) and Figure 6(c)), the RF (Figure 5(d) and Figure 6(d)), or the datapath (Figure 5(e) and Figure 6(e)) and the key-dependent prediction as a function of time. For the sake of representation, the execution of boilerplate code at the beginning and at the end of the benchmark is omitted. Figure 5(a) also reports the state of the CPU pipeline, starting from the clock cycle when the first instruction of an intermediate iteration of *Benchmark 1* pattern is fetched, as the benchmark pattern is iterated multiple times in real-world ciphers. Each line of Figure 5(a) depicts the progress of a CPU instruction (represented without its fixed syntactic decorator prefix `l.` for the sake of clarity) in the pipeline for *Benchmark 1*, while Figure 6(a) displays the same information for *Benchmark 2* for which the correlation obtained with the power consumption of the entire CPU is also reported. It is also noteworthy that, despite the fact that the correlation values are bound to the ones of the results for the CPU modules, the values of r for the entire CPU cannot be obtained simply by adding together the ones of the separate modules due to the non-additive nature of Pearson's sample correlation coefficient. To provide the complete picture of the instructions being processed by the ORPSoC, we also report some of the instructions preceding and following the ones in an iteration of the benchmark.

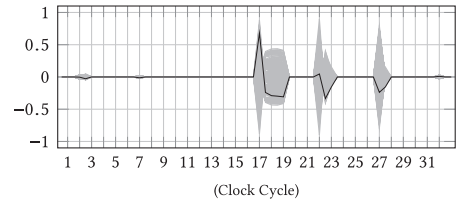
In the code reported for *Benchmark 1*, `rK` is the register into which a byte of the secret key is loaded and it acts as a storage for the result, while `rP` is the storage for the known input. `rA` contains the base address in memory from which the position of both the known input and the key is



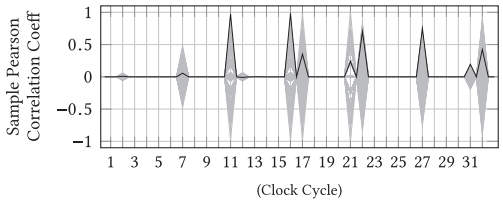
(a) State of the pipeline during the computation of the instructions on the left side



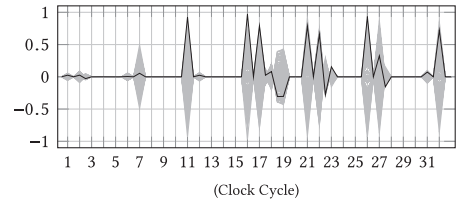
(b) Time-wise Pearson sample correlation coefficient for the power consumption of the CPU executing the code in (a)



(c) Time-wise Pearson sample correlation coefficient for the power consumption of the ALU executing code in (a)



(d) Time-wise Pearson sample correlation coefficient for the power consumption of the RF executing the code in (a)

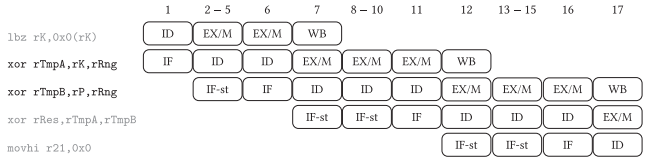


(e) Time-wise Pearson sample correlation coefficient for the Datapath power consumption executing the code in (a)

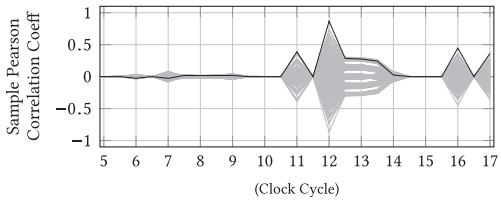
Fig. 5. Pipeline state (a) and side channels (b)–(e) of the execution of *Benchmark 1* on the reference platform. Both the known input and the secret key are loaded from the main memory of the SoC, computing their address starting from the base address value stored in register rA. The known input is stored in register rP and the secret key in register rK before being xor-ed. The result is stored back into main memory.

computed. In the second benchmark, rP and rK contain the known input and key values, respectively, while rRng contains a random number, different for each execution of the code. rTmpA and rTmpB are supposed to store constant values at the beginning of the benchmark code. Before detailing the results obtained from the leakage analysis of the benchmarks, a classification of the different types of information-leaking transitions that emerged from our investigation is highlighted in Table 1 also binding each type of leakage to the affected portions of the microarchitecture.

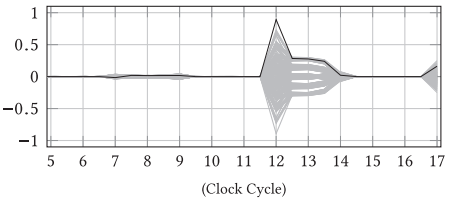
- **Register (Over-)Write.** This type of transition is the one taking place whenever a new value is written into a register; i.e., the value to be written is asserted on the write port of the RF, causing the input latch of the register to toggle if the previously held value is different from the new one. Such an operation also involves the datapath as the carrier of information toward the RF write port.
- **LSU Data Remanence.** In the ORPSoC, a common choice on the management of the data bus is made; i.e., the bus does not toggle unless a memory access is required. Such a design decision implies that transitions between the values involved in subsequent memory



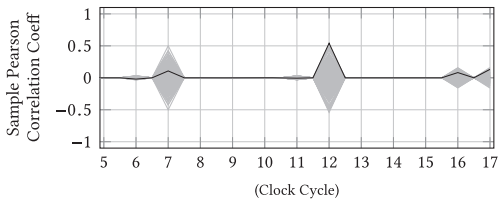
(a) State of the pipeline during the computation of the instructions on the left side (benchmark 2,boilerplate code not reported)



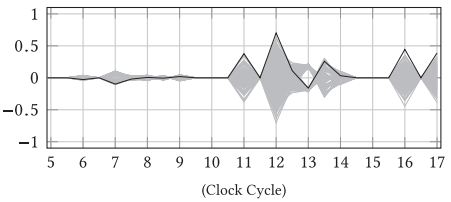
(b) Time-wise Pearson sample correlation coefficient for the power consumption of the entire CPU executing the assembly in (a)



(c) Time-wise Pearson sample correlation coefficient for the power consumption of the ALU executing the assembly in (a)



(d) Time-wise Pearson sample correlation coefficient for the power consumption of the RF executing the assembly in (a)



(e) Time-wise Pearson sample correlation coefficient for the power consumption of the Datapath executing the assembly in (a)

Fig. 6. Pipeline state (a) and side channel analysis (b)–(e) of the execution of *Benchmark 2* (randomized encoding of input and secret key) on the ORPSoC platform. The known input and the secret key are held in registers rP and rK, respectively, while rRng contains a random value.

Table 1. Taxonomy of Transitions Causing Potential Information Leaking Power Dissipation on the ORPSoC Platform, Together with the Involved Modules

Transition Type	CPU Component			Event in Time					
		Register	Datapath	ALU		Register File		Datapath and LSU	
	ALU	File	and LSU	Figure 5(c)	Figure 6(c)	Figure 5(d)	Figure 6(d)	Figure 5(e)	Figure 6(e)
Register (Over-)Write		✓	✓			21, 32	12	11, 16, 22, 27	
LSU Data Remanence		✓	✓			11, 16		11, 16, 22, 27	
WB Buffer (Over-)Write			✓					17	16
ALU Computation	✓			17, 18, 19					
EX Stage Operand Assertion	✓	✓	✓	22, 27	12		12	22, 32	11, 12
Signal Glitches		✓	✓			17, 22, 27	7	21, 26	

For each pair of transition type and CPU component we report the instant in time when the leakage is observed according to Figure 5 and Figure 6.

accesses will take place regardless of the amount of purely computational operations taking place in between them. Since such values are propagated by the WB forwarding path of the pipeline to the RF, such transitions involve both the datapath and the RF.

- **WB Buffer (Over-)Write.** The WB buffer employs as the *enable* signal the negated *freeze signal* of the pipeline and latches the value coming from the previous stage at each clock cycle to forward it in case of a pipeline stall.
- **ALU Computation.** Since the ALU in the ORPSoC design is fully combinatorial, any change of its primary inputs, either its operands or the opcode, will trigger a computation that is operand dependent, and thus may be leaking information.
- **EX Stage Operand Assertion.** At the beginning of the EX stage, the operands are asserted on both the ALU and the LSU primary inputs by the sequential elements in the Operand-Muxes module of the datapath. Such a transition may be a cause for information-leaking power dissipation between operands sharing the ALU as well as the LSU input port in subsequent clock cycles.
- **Signal Glitches.** Unbalanced combinatorial paths give rise to signal glitches before the setup time of a clock cycle. Whenever such glitches occur on signals carrying known input and key-dependent values, an information-leaking power consumption may take place.

4.1 The ALU Information Leakage

A widespread assumption in open literature is that the portion of a digital circuit performing the computation of a result provides a significant information leakage on the side channel [31], and it is believed that such value leakage is well modeled by the Hamming weight of the said result. Our investigation examines the information leakage from the ALU in accordance to the previously presented classification appearing in Table 1 and further refines the validity of the open literature assumption.

ALU Computation. Concerning *Benchmark 1*, Figure 5(c) shows how, and in contrast with the open literature assumption, the said consumption model fails to capture the behavior of the ALU at hand, as the key-dependent model exhibiting the highest correlation at cycle 17 is not the one depending on the correct key hypothesis. We ascribe such a behavior to the fully combinatorial nature of the ORPSoC ALU, which computes simultaneously all the available operations on its inputs, to propagate the required result only to its primary output, thus superimposing the power consumption of all the arithmetic-logic computations to the one of the bitwise eXclusive OR (`xor`). The presence of key-dependent models exhibiting nonnegligible correlation suggests that employing a model different from the value leakage applied to the output of the operation being computed might allow one to successfully capture the information contained in the side channel. Such a behavior depends on the specific ALU design strategy, which is not investigated in further detail, as it exceeds the scope of the current work, i.e., relate microarchitectural design features to side-channel leakage. We note that investigating the most appropriate leakage models given specific ALU design strategies may provide interesting future research directions.

A second valuable observation is devoted again to the fully combinatorial nature of the ALU that provokes stray information leakage in the two cycles, i.e., 18 and 19, following the beginning of an EX stage computation. The investigation verified that such a behavior is the result of the combinatorial paths forwarding a spurious opcode to the ALU, despite the pipeline being frozen. This fact causes an input-dependent power consumption that, however, is not well fit by the value leakage of the computed value consumption model.

EX Stage Operand Assertion. The assertion of the operands to the functional units at the beginning of the EX stage is a cause for significant information-leaking power dissipation that can be

practically exploited, as shown by executing *Benchmark 2*. Indeed, in Figure 6(c), at cycle 12, a clear correlation between the Hamming weight of the `xor` combination of known input and key and the power consumption emerges. This suggests that the Hamming distance between the operands of two subsequent operations is indeed a good model for the power dissipation of the ALU. The correlation is due to the fact that both the known input and the secret key appear as the first operand of the two consecutive `xor` instructions that combine the two operands with the random value, with the second one starting its EX phase at cycle 12. We note that such a behavior is particularly detrimental, as it effectively removes the protection that masking schemes are supposed to provide, despite the fact that the code fragment respects the best practices in terms of avoiding careless register reuse [7]. In particular, the masking security is reduced by one order, as the microarchitecture-induced leakage matches the share recombination function of the masking scheme at hand (i.e., Boolean masking). Different masking schemes may equally be affected, although the relation with their share recombination function must be considered on a per-scheme basis. In addition to the presence of nonnegligible values of r when a computational instruction is in the EX stage, we note that two other points in time are characterized by a significant correlation with the ALU power dissipation (cycles 22 and 27 in *Benchmark 1*). In both cases, the root cause of the correlation is the fact that the Operand-Muxes propagates the operands to both the ALU and the LSU (see Figure 1) regardless of which instruction is in the execution phase. As a consequence, whenever an operand of the ALU depends on both the known input and the secret key, the ALU will dissipate power, providing an information leakage even if a non-ALU instruction is in the execution stage.

The first point in time when this happens in *Benchmark 1* (Figure 5(c), cycle 22) is when the currently asserted ALU inputs, i.e., the input and key values used by the `xor` operation, are replaced by the base address and the result of the `xor` itself when the store byte operation (`sb`) enters EX, while the second time instant (Figure 5(c), cycle 27) sees the second operand of the aforementioned `sb` instruction being replaced by the fixed offset of the address of the subsequent `load` byte with zero extension instruction (`lbz`). In both cases, the value leakage model does not fit the instantaneous power consumption, thus resulting in an incorrect key value being the one employed by the prediction with the highest correlation. However, this still does not exclude the possibility of extracting the correct key value with a different model of the ALU power consumption.

Finally, we are able to justify the fact that, despite the transitions between the values of operands of subsequent instructions taking place also at other cycles, substantially no correlation is present. Indeed, in these cases, the starting and ending values of the transitions are statistically independent from the known input, and will thus result in an independent power consumption.

4.2 The Register File Information Leakage

Considering a `load-store` architecture, the register file is known to be a significant source of exploitable information leakage [14, 31, 47], due to the fact that it stores both operands and results of each instruction. In particular, the most common assumption is that a register will provide an information-rich power leakage, which is well modeled by the Hamming distance of its previous and next contents. Such a transition is typically associated with the write-back of the result of an instruction.

Register (Over-)Write. The common transition model from open literature matches three different time instants in the benchmarks, namely, cycles 21 and 32 in *Benchmark 1* (Figure 5(d)) and cycle 12 in *Benchmark 2* (Figure 6(d)). In particular, the former information leakage is caused by the `xor rK,rP,rK` completing its EX transit and asserting the result up to the sense portion of the FFs of the destination register, which contains the value of the key. This results in a signal transition having a number of toggles equal to the input value Hamming weight, and consequentially

fitting the employed power consumption model, i.e., the Hamming weight of the `xor` combination of input and key with an apparent key value of zero. A consequence of this fact is that, despite the fact that it is possible to obtain an effective power model for the transition taking place, it is not possible to retrieve the actual value of the key from it, giving rise to the first so-called *ghost peak* [13] we observed in our evaluation. A spike in the sample Pearson correlation coefficient is deemed to be a ghost peak according to [13] whenever it is related to a computation from which it is not possible to extract information regarding the secret key. A similar ghost peak is present at cycle 32 when the value of the key byte used in the subsequent benchmark iteration and loaded from the `lbz rK,0x13(ra)` instruction is written into the `rK` register, which contains the `xor` combination of the actual known input byte and the actual key byte. We had practical confirmation of this *ghost peak* noting that the best power model for clock cycle 32 in *Benchmark 1* is the one with the key value equal to the `xor`-combination of the actual and next key bytes (namely, 1).

EX Stage Operand Assertion. The assertion of the operands to the functional units in *Benchmark 2* causes a practically exploitable information-leaking power dissipation on the Register File. Indeed, Figure 6(d), at cycle 12, shows a clear correlation between the Hamming weight of the `xor` combination of known input and key, suggesting that the Hamming distance between the operands of two subsequent operations is indeed a good model for the power dissipation of the RF. The correlation is due to the fact that both the known input and the secret key are encoded in the assembly to appear as first operands of the two consecutive `xor` instructions that combine the two operands with the random value, with the second one starting its EX phase at cycle 12. In particular, we note that in this case the detrimental effect reduces the masking security by one order, as the microarchitecturally induced leakage matches the share recombination function of the masking scheme at hand (i.e., Boolean masking). Different masking schemes may equally be affected, although the relation with their share recombination function must be considered on a per-scheme basis.

LSU Data Remanence. Besides the leakage due to the write-back of the result of an instruction, our analysis of the causes of information-supplying power dissipation identified a source of vulnerability in the `load`/`store` operations acting on single bytes, in combination with the last-transmitted-value retention policy of the data bus (see Section 3). This security issue produces information leakages that are far from intuitive from an assembly programmer point of view, such as the one appearing at cycle 11 of *Benchmark 1* (Figure 5(d)), where the transition leakage between a key and an input byte models perfectly ($r = 1$) the power dissipation of the RF. The aforementioned leakage can be explained considering the evolution of the data port of the memory, and which portion of it is propagated to the write port of the register file by the LSU between cycles 6 and 11. We note that the analyzed iteration of the benchmark manipulates the third byte in the 32-bit word. In particular, at cycle 6 the `sb` instruction storing the second byte of the known input word leaves on the data bus a value containing the first and second bytes obtained as a combination of known input and key, and the third and fourth being simply the unmodified known input bytes. When the subsequent `lbz rK,0x12(rA)` instruction enters EX at cycle 7, the LSU is instructed to forward the contents of the third byte obtained from the data bus to the RF write port, and does so asserting the unmodified third byte of the known input. We note that, while such a value is not actually memorized by the destination register, the data port of the FFs constituting the register are charged. At cycle 11, the value being loaded by `lbz rK,0x12(rA)`, i.e., the actual key byte, is made available from the memory and is asserted on the lines connected to the destination register, effectively causing a known-input-to-key transition, which is perfectly modeled by a transition leakage. An analogous transition takes place at cycle 16 of *Benchmark 1*, when the value requested by the `lbz rP,0x2(rA)` instruction, i.e., the known input, is provided on the data bus, replacing the key value and thus producing a key-to-known-input transition (see Figure 5(d)).

Signal Glitches. The last among the information leakage causes we detected in our analysis concerns the power dissipation due to logic glitches during the computation. Inspecting the VCD, we confirmed that the presence of correlation at cycles 17, 22, and 27 of *Benchmark 1* and at cycle 7 of *Benchmark 2* is caused by multiple transition glitches that occur before the result is stable. We note that, given our simulation environments, the presence of a glitch in a synthesized design is deterministic, given the known input triggering it. This in turn allows the power dissipation caused by glitches detected at this simulation level to appear clearly as information leaking whenever it is the case. In particular, at cycles 22 and 27 of *Benchmark 1*, when the result of the `xor rK,rP,rK` operation is being propagated in the datapath, a glitch causes the carrying signals to transition to zero before asserting the said value. This in turn results in a power dissipation matching perfectly the one predicted by the consumption model employing the correct key. A similar glitching issue is the root cause of the power model with a zero-valued key having the best correlation at cycle 17 of *Benchmark 1* (Figure 5(d)). In particular, a transition of the write port of the register file glitches to zero before the final value, i.e., known input `xor` key, is asserted. We thus observe that also glitches may be one of the causes of *ghost peaks* during a side-channel attack. A similar ghost peak is present at cycle 7 of the *Benchmark 2*, where a power dissipation well fit by the same zero-key model is caused by the assertion of the known input value by the RF, which stabilizes only after having a glitch to zero.

4.3 The Datapath Information Leakage

We complete the analysis of the information-leaking power dissipation of our architecture considering the contribution arising from the datapath, as made of the LSU, the logic contained in the WB stage of the pipeline, and the Operand-Mux module. The power consumption of the datapath does not include the ALU and the RF that have been analyzed in precedence, nor does it include the power dissipated by non-data-processing CPU modules, such as the IF stage.

Register (Over-)Write and LSU Data Remanence. Two instances of leakage arising from the value persistence on the data bus are the ones allowing a correct key retrieval at cycles 11 and 16 of *Benchmark 1*. In both cases, the exploitable leakage is the result of the data memory bus asserting the last value transmitted on it, before switching to the next required one. It is worth noticing once more that the considered iteration of both benchmarks manipulates the third byte in the 32-bit data word. In particular, at cycle 11 the third byte of the known input word, a remanence of the `sb 1(rA),rK` instruction, is substituted by the third key byte requested by `lbz rK,18(rA)`, while at cycle 16 the third key byte loaded by the aforementioned instruction is again substituted on the data bus by the third one of the known input, as requested by `lbz rP,0x2(rA)`. A similar information leakage, which instead is leading to an incorrect key hypothesis given the employed power model, is present at cycle 27 of *Benchmark 1*. In that clock cycle, the `opB` of the LSU toggles from the value of the known input combined with the key via `xor`, required to complete the EX phase of `sb 2(rA),rK`, to the value of the offset of the address required to perform `lbz rK,19(rA)`. Such a transition leads to a transition leakage model with a wrong key hypothesis being the one with the highest correlation. A last point in time exhibiting leakage in *Benchmark 1* is cycle 22, where the realignment logic contained in the Reg2Mem module asserts the byte to be stored twice in an internal buffer, previously zero-filled. Since such a value is the `xor` combination of known input and key, an exploitable power dissipation is present.

WB Buffer (Over-)Write. Instances of power dissipation due to the memorization of a value in the WB buffer of the ORPSoC pipeline, leading to information leakage, are present in both benchmarks. In particular, at cycle 66 of *Benchmark 1*, the WB buffer memorizes the value being output by the EX stage, i.e., the key value retrieved by the `lbz` operation, due to the incoming pipeline stall.

Further on in the computation, namely, at cycle 17, the key value is replaced by the one being produced by the EX stage, i.e., the input-value-loaded `lbz` operation, in turn giving rise to a power dissipation that is well modeled by the Hamming distance between the input value and the secret key, thus allowing key value retrieval. A similar transition takes place at cycle 16 of *Benchmark 2*, where the WB buffer is replaced with the value of the `xor` combination of the key value and the random value its previous contents, i.e., the `xor` combination of the known input and the same random value. Such a transition is once again well modeled by the Hamming distance between the input value and the secret key and undermines the effectiveness of the masking technique, should it take place in a real-world implementation.

EX Stage Operand Assertion. Typical conditions causing information-revealing power dissipation are the ones at cycles 11 and 12 of *Benchmark 2*. At cycle 11, the sequential portion of the Operand-Mux latches both operands due to a pipeline stall taking place in the next clock cycle. The latched value for `opA`, i.e., the input byte at hand, replaces the previously stored value memorized at cycle 7, i.e., the secret key byte. Such a transition will in turn cause a power dissipation that is perfectly modeled by the prediction depending on the correct key value. In the subsequent cycle (cycle 12), the effects of the aforementioned transition are replicated on the entire datapath, as the flip-flop toggles its output, resulting in a considerable information leakage. A more sophisticated leakage due to the operand assertion into the ALU, despite the face that the result is not needed, is the one taking place at cycle 32 of *Benchmark 1*. The root cause of such a leakage is the accidental assertion of the combination of known input and key as an operand and an offset value as the second operand, taking place at cycle 27. An accidental ALU opcode (namely, an inclusive `or`) is also asserted at cycle 27, despite an `sb` instruction being actually in the EX stage. The ALU accidentally computes the `or` of its operands, which gets stored into the WB buffer as a result of the oncoming stall. The contents of the WB buffer are replaced at cycle 32 by the loaded value, namely, the key, in turn exhibiting a ghost peak due to the known input and key-dependent transition. A last case of leakage due to the assertion of the EX operands is the one of cycle 22 in *Benchmark 1*. During cycle 22, the base address of the `sb` instruction is asserted as an operand to both the LSU and the ALU, in turn replacing the previous value, which was indeed the input byte. Since the actual value of the byte of the address being asserted differs only by a single bit from the correct key value (namely, `0x14` instead of `0x15`), the transition results in a ghost peak, which is completely unrelated with the actual key value, despite being remarkably similar in appearance. We note that an address ending in `0x15` would have caused a ghost peak to look perfectly like a correct key extraction, given the secret key value considered in our test.

Signal Glitches. We report the fact that leakage coming from glitching behavior at cycles 21 and 26 is caused by the very same glitching behavior as the one reported in precedence, i.e., due to both the structure of the synthesized design and the critical information contained in the datapath. In particular, the value equal to the bitwise eXclusive OR between the key value and the known input value is at the output of the ALU (cycle 21) and propagated from the RF to the LSU at cycle 26 ready to be stored while the main memory continuously forwards the old stored value until the new one is written.

5 DISCUSSION

This section summarizes the findings that emerged in Section 4 and classifies them according to the contributions detailed in Section 1, providing a set of programming hints aimed at preventing information leakage due to data serialization. Such hints can be applied to the assembly of any in-order single-issue RISC CPU.

5.1 Microarchitectural Components Inducing Side-Channel Information Leakage

The *data serialization* effect arises when two values are updated on the same wire of the microarchitecture in two consecutive clock cycles, thus yielding a power consumption that correlates with the Hamming distance between the two values. Our investigation marks this microarchitectural effect as the cause of the *WB Buffer (Over-)Write*, the *Register (Over-)Write*, and the *EX Stage Operand Assertion* information leakage types (see Section 4) and shows the possibility to exploit the corresponding leakage to recover the secret key. The leakages classified as either *WB Buffer (Over-)Write* or *Register (Over-)Write* arise from the WB pipeline stage and the forwarding paths of the CPU, respectively. The two components are present in any modern in-order RISC CPU microarchitecture to provide precise exception handling support and to minimize pipeline stalls, respectively. We note that both the WB and the Forwarding Path components are also present in the design of superscalar and out-of-order CPUs, with the same end. Indeed, in out-of-order CPUs, the WB stage embodies the data serialization point needed to perform an in-order update of the architectural register file, while the forwarding paths are implemented as a mechanism to notify about the availability of the intermediate results to the *reservation stations*. The leakage classified as *EX Stage Operand Assertion* arises from the serialization of operand values in input to the functional units. Our investigation highlights how the effectiveness of a side-channel countermeasure (i.e., Boolean masking) may be forfeit due to the serialization of two shares of the same value on the same input of a functional unit, despite the fact that they are never combined, from an Instruction Set Architecture (ISA) point of view.

The *data serialization* effect also induces an exploitable information leakage in the RF and LSU components. The information leakage in the RF arises from the signals that drive the values into/from the input/output ports of the RF itself. This observation confirms and extends the results about the RF leakage in [47], allowing us to clarify the reasons underlying the information leakage happening in a sequence of instructions with no register reuse (at ISA level). In contrast, the leakage classified as *LSU data remanence* arises from the transition between two consecutive loaded or stored values in the LSU, regardless of the number of non-LSU instructions being computed in between.

Open literature states that the ALU is a further source of information leakage that is fairly captured employing the Hamming weight of the output of a sensitive operation as a power consumption model. Our analysis shows that the said consumption model fails to match the information leakage of the fully combinatorial design of the ALU at hand. While employing a dedicated model for the combinatorial circuits in the ALU chosen as a case study may reveal an exploitable information leakage, the study of ALU leakage and side-channel-resistant ALU designs is out of the scope of this work.

Finally, signal glitches represent a critical issue when designing SCA-resistant hardware and/or software cryptographic primitives. The proposed investigation examines all the glitches that emerged for which the employed information leakage model shows a nonnegligible correlation, even in cases where a wrong key is guessed. Coherently with the open literature, we state that a necessary condition for a glitch to induce exploitable information leakage is that it happens on a signal carrying both a known input and a secret key-dependent value. Whenever that is not the case, a glitch inducing a nonnegligible correlation with a wrong key guess results in a so-called *ghost correlation peak*.

5.2 Microarchitectural Hints to Improve the Application of SCA Countermeasures

This section presents four practical programming hints to allow the assembly programmer to prevent a realization of an SCA-protected cryptographic implementation from leaking due to

data serialization effects. These hints consist of adopting a given programming style or inserting *dummy* instructions in specific locations of the (assembly) source code. We denote as a *dummy* instruction an ISA instruction that processes, fetches, or stores random values, without any effect on the original program semantics.

Hint 1: Prevent LSU remanence with dummy `load/store` instructions. The leakage classified as *LSU data remanence* leverages the transition in the LSU between the values involved in a consecutive pair of `load/store` instructions regardless the number of non-`load/store` instructions being processed in between. Whenever a transition between the aforementioned values may imply an information leakage, a dummy `load/store` instruction should be inserted between them.

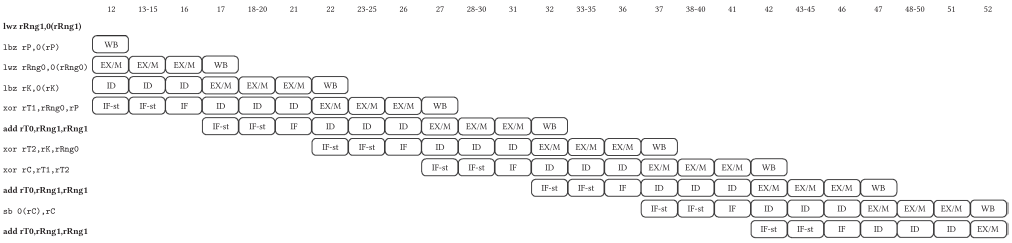
Hint 2: Remove LSU intrinsic leakage avoiding in-place computation. We recall that our analysis highlighted how, in case an in-place computation is performed, the LSU observes a transition between the precomputation value, which is still being forwarded to the LSU by the memory, and the one that should be stored. If such a design strategy for the LSU is adopted (which is reasonable from a power consumption reduction standpoint), we suggest that in-place computation is avoided altogether.

Hint 3: Leakage-aware operand order choice. The leakage labeled as *EX Stage Operand Assertion* can induce an exploitable information leakage due to the assertion of two data values on the input operand of the functional unit in two consecutive clock cycles. In these cases, either a swap of the source operand order of the instruction should be performed or a dummy computational instruction should be inserted to prevent the said unwanted information leakage.

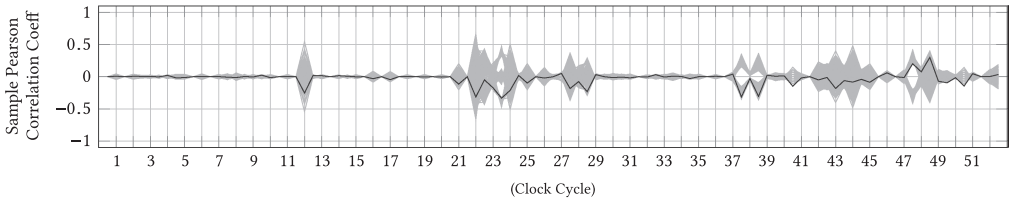
Hint 4: Avoid unwanted write-back serialization with dummy computational instructions. The two leakages labeled as *WB Buffer (Over-)Write* and *Register (Over-)Write* can induce an exploitable leakage at the write-back stage and the CPU forwarding paths due to the serialization of the two values computed and/or fetched during the EX/M stages. We suggest inserting a *dummy* computational instruction anywhere between the two consecutive instructions that see their outputs serialized in the WB buffer or the forwarding paths.

We now show how applying the aforementioned hints to the code of *Benchmark 1* and *Benchmark 2* on our reference CPU actually prevents leakage stemming from *data serialization* effects.

Benchmark 3 is built from the combination of the instructions of *Benchmark 1* and *Benchmark 2*. It combines two values, a known input and secret key contained in `rP` and `rK`, respectively, with two copies of the same random value held in `rRng0`. The two intermediate results are then combined together and the result is stored back into memory. The instruction sequence is the one computing a single `xor` employing a Boolean masking countermeasure [31]. To derive *Benchmark 3*, we apply the presented programming hints as follows. We prevent the information leakage stemming from the operand serialization due to the line of code 2 and 3 in Figure 6(b) by inserting a dummy computational instruction between the two `xor` operations according to *Hint 3*. We note that, according to *Hint 4*, such dummy instruction also breaks the serialization of the results of the random value additions into the WB buffer. We also apply *Hint 2* by choosing a storing address for the result (contained in `rC`) that is different from the addresses of both the inputs (namely, the values loaded into `rK` and `rP`). In this instruction sequence, an alternative to the avoidance of in-place computation as per *Hint 2* is to insert a dummy `load/store` following *Hint 1*. Indeed, if an in-place computation is performed here, *Hint 1* breaks the data serialization between the secret key value, which is fetched by the last LSU instruction before the `store` one, and the known input, which is forwarded by the memory until the `store` completes.



(a) State of the pipeline during the computation of the instructions on the left side. *Dummy* instructions are reported in bold font



(b) Time-wise Pearson sample correlation coefficient for the power consumption of the CPU executing the code in (a)

Fig. 7. Pipeline state (a) and side channels (b) of the execution of *Benchmark 3* on the reference platform. The known input and the secret key are held in registers `rP` and `rK`. Registers `rRng0` and `rRng1` contain two different random values. Moreover, registers `rT0`, `rT1`, and `rT2` store intermediate values, while the result of the computation is held in `rC`. `(rX)` defines the memory location of the `X` value, where $X \in \{P, K, Rng0, Rng1, T0, T1, T2\}$. We note that the pipeline state (a) reports the *dummy* instructions in bold.

We report in Figure 7 the results of the side-channel analysis performed on *Benchmark 3*. Figure 7(a) reports the state of the CPU pipeline starting from the clock cycle when the first instruction of the *Benchmark 3* pattern is executed. Figure 7(b) reports the results of the Pearson's sample correlation coefficient between the power consumption of the entire CPU and the key-dependent prediction as a function of time, during *Benchmark 3*.

EX Stage Operand Assertion. *Benchmark 2* shows an exploitable information leakage due to the serialization of known input and secret key on the input operand of the ALU (see clock cycles 11 and 12 in Figure 6(b)). *Benchmark 3* solves the information leakage issue by introducing a dummy instruction, i.e., `add rT0, rRng1, rRng1`, according to *Hint 3* (see cycles 26 and 27 in Figure 7(b)). As a side effect, we note that the additional dummy instruction also solves the exploitable information leakage that affects *Benchmark 2* at cycle 16 due to the data serialization at the WB (see Figure 6(b)).

Benchmark 1 shows an information leakage at cycles 22 and 27 due to the serialization of the immediate zero value used and the result of the computation (see Figure 5(b)). The result of the computation is materialized between cycles 22 and 27. The immediate value is used to compose the address of the `store` and the subsequent `load` and is asserted on the same wire before cycle 22 and at cycle 27. The introduction of two dummy instructions before and after the `store` in *Benchmark 3* solves the two information leakage points (see cycles 42–47 and 52 in Figure 7(b)). *Benchmark 1* also shows an exploitable information leakage at cycle 26, due to a glitching behavior caused by data serialization happening in the LSU when the `store` enters the EX stage (see Figure 5(a)). In particular, we observe a data serialization between the actual value stored in memory, i.e., the secret key, that is retained before the `store` instruction completes and the last loaded value, i.e.,

the known input. By avoiding the in-place computation according to *Hint 3*, *Benchmark 3* resolves such information leakage points (see cycles 47–51 in Figure 7(b)).

WB Buffer (Over-)Write. The exploitable information leakage reported at cycle 16 of *Benchmark 2* due to the serialization effect in the WB of the two intermediate results that combine the known input and the secret key with the same random value disappears in *Benchmark 3*. This is due to the insertion of the dummy `add rT0, rRng1, rRng1`, between the two `xor` instructions that combine known input and secret key with the random value, i.e., `xor rT1, rRng0, rP` and `xor rT2, rK, rRng0`, which follows *Hint 4*. We remove the exploitable information leakage of *Benchmark 1* at cycle 17 due to the serialization of the secret key and the known input values that are loaded one after the other (see Figure 5(b)). We note that *Benchmark 3* does not suffer the leakage due to the `load` instruction that fetches the random value to be used in the Boolean masking scheme (see cycle 17 in Figure 7(b)).

Register (Over-)Write and LSU Data Remanence. *Benchmark 1* shows information leakage at cycles 21, 22, 26, and 27 due to the data serialization on the sense portion of the RF (see Figure 5). The introduction of two dummy instructions, following *Hint 4* before and after the `store` in *Benchmark 3*, solves the four information leakage points (see cycle 42–47 and 52 in Figure 7(b)). Moreover, the consecutive fetch of the known input and the secret key at cycles 11 and 16 leads to a correct key guess (see Figure 5). We note that cycle 11 represents the beginning of the iteration of the benchmark pattern and the information leakage is due to the data serialization between the last value processed by the LSU instruction in the previous loop iteration and the first one loaded in the current iteration. In accordance with *Hint 1*, we fetch a random value as the first instruction in *Benchmark 3* to break such data serialization effect. Figure 7(b) reports no information leakage until cycle 12. In particular, the information leakage shown at cycle 12 of *Benchmark 3* is independent from the secret key and always suggests 0 as a secret key guess. The information leakage is due to the sequence of two `load` instructions, where the first instruction zero-extends a single byte of the known input fetched from memory. At cycle 12 the second `load` forces the LSU to show the entire 32-bit word to the CPU. However, the data at the address of the previous `load`, i.e., the known input, is asserted until the second `load` completes. To this extent, we note a data serialization between a zero-extended byte of known input and the entire work of the same known input. Moreover, we note that *Benchmark 3* does not suffer the leakage due to the `load` instruction that fetches the random value to be used in the Boolean masking scheme (see cycle 17 in Figure 7(b)).

The wrong operand encoding in *Benchmark 2* produces an exploitable leakage at cycle 12 (see Figure 6(b)). According to *Hint 3*, *Benchmark 3* solves the information leakage by introducing a dummy instruction, i.e., `add rT0, rRng1, rRng1`, in between the two `xors` that process the known input and the plain key with a random value (see cycle 26 and 27 in Figure 7(b)).

ALU Computation and Signal Glitches. According to the results obtained for *Benchmark 1* and *Benchmark 2*, the employed consumption model fails to capture the behavior of the ALU at hand during the execution of *Benchmark 3*. However, the presence of a key-related, nonnegligible correlation at cycles 37 to 41 in Figure 7(b) suggests that the use of a different model might successfully exploit the side-channel information leakage coming from the combinational portion of the ALU. The investigation of an accurate, nonlinear model depending on the ALU design is out of the scope of this work.

We also report a glitching behavior leading to the correct key guess in the second half of cycles 40 and 50 of *Benchmark 3*. The glitching behavior is similar in nature to the other glitches discussed in Section 4 and it is due to both the structure of the synthesized design and the presence of critical information in the datapath.

5.3 Ghost Peak Characterization

Our accurate analysis environment allows us to shed light on the nature of *ghost peaks* appearing in the results of a side-channel attack. Recalling that SCAs leverage an input data-dependent variation of the power consumption, we can classify ghost peaks into two sets according to whether the corresponding power dissipation is also influenced by the key value or not. In case the influence from the key is present but the key hypothesis yielding the best-fitting model is incorrect, the design may appear safe to a less accurate analysis while it is not. It is possible to ascertain if a ghost peak belongs to this set employing a variant of the nonspecific t-test [25] where the value of the known input is not changed while collecting the measurements, and the value of the key is kept fixed for the first set of measures and randomly changed in the second set. If the t-test reports the instant of the ghost peak to have a different power consumption profile in the second set with respect to the first one, the ghost peak is an actual source of leakage, although the model employed to fit its power consumption is inadequate.

Whenever a *ghost peak* is present in correspondence of a time instant where the power dissipation is not related with the key, the resulting best-fitting power model will point to an unrelated value as the candidate key. Note that such a value may either be different from the correct one (e.g., the zero-valued key extracted in cycle 21, *Benchmark 1*) or quite similar (the off-by-1-bit one present at cycle 22, *Benchmark 1*). The trickiest case is the one where the algorithmic constant causing the leakage matches the value of a portion of the key employed in the tests. Indeed, in such a case, the system designer may be tricked into considering a ghost peak evidence of an information-leaking power consumption. Employing the same modified t-test strategy described before, it is possible for a system designer to rule out ghost peaks of this kind as a source of possible information leakage. Indeed, collecting the two traces' sets, differing only by the employed key values, any moment in time where the computation is not dependent on the key will exhibit the same behavior, and thus be deemed as not leaking by the modified t-test.

6 CONCLUDING REMARKS

This work provides a precise, "clean room" characterization of the effects of the microarchitectural side-channel leakage that are traditionally exploited to lead passive SCAs against a software cryptographic primitive running on a CPU. This scenario, which is increasingly common due to the widespread availability of full-fledged 32-bit CPUs that execute open cryptographic software implementations, is characterized by a significantly higher degree of complexity with respect to the analysis of 8-bit microcontrollers that have been previously considered in the open literature. To this end, we employed two benchmarks obtained from instruction patterns that are ubiquitous in the symmetric block ciphers standardized by ISO. Our investigation pinpoints which portions of the CPU are sensitive to the different information leakage patterns identified, thus serving a threefold objective. First, the correlation between the analyzed parts of the microarchitecture and the observed information leakage patterns allows us to extend the validity of our findings to a broader class of CPU implementations falling within the RISC CPU family. Moreover, the in-depth microarchitectural investigation confirms several findings of the open literature and allows us to motivate more precisely those whose a sound explanation is missing or incomplete. Furthermore, we presented a set of programming hints to cope with the side-channel leakage induced by different instances of the *data serialization* effect. The application of such recommendations to our case study benchmarks effectively suppresses the unintended leakage that our investigation demonstrates to lead to the correct key guess on the CPU at hand. We also note that a fruitful future direction is the definition of a set of formal constraints stemming from the side-channel analysis of the microarchitecture of a CPU, which can be practically integrated in the back

end of a common C compiler toolchain to the end of preserving the security properties of SCA countermeasures during code emission.

Finally, the accurate temporal match obtained in our simulation environment allows us for the first time to show when the input dependence of the side-channel behavior pointed out by a non-specific t-test is indeed caused by a switching activity that is not dependent on the secret key, and thus not exploitable by an attacker. We note that such an analysis is also amenable to being automated and integrated in an EDA toolchain, to reduce the engineering effort in evaluating the security of an implementation.

REFERENCES

[1] Giovanni Agosta, Alessandro Barenghi, Massimo Maggi, and Gerardo Pelosi. 2013. Compiler-based side channel vulnerability analysis and optimized countermeasures application. In *The 50th Annual Design Automation Conference 2013 (DAC'13)*. ACM, 81:1–81:6.

[2] Giovanni Agosta, Alessandro Barenghi, and Gerardo Pelosi. 2012. A code morphing methodology to automate power analysis countermeasures. In *The 49th Annual Design Automation Conference 2012 (DAC'12)*, P. Groeneveld, D. Sciuto, and S. Hassoun (Eds.). ACM, 77–82.

[3] Giovanni Agosta, Alessandro Barenghi, Gerardo Pelosi, and Michele Scandale. 2014. A multiple equivalent execution trace approach to secure cryptographic embedded software. In *The 51st Annual Design Automation Conference 2014 (DAC'14)*. ACM, 210:1–210:6.

[4] Giovanni Agosta, Alessandro Barenghi, Gerardo Pelosi, and Michele Scandale. 2015. Information leakage chaff: Feeding red herrings to side channel attackers. In *Proceedings of the 52nd Annual Design Automation Conference*. ACM, 33:1–33:6.

[5] Giovanni Agosta, Alessandro Barenghi, Gerardo Pelosi, and Michele Scandale. 2015. The MEET approach: Securing cryptographic embedded software against side channel attacks. *IEEE Transactions on CAD of Integrated Circuits and Systems* 34, 8 (2015), 1320–1333.

[6] Giovanni Agosta, Alessandro Barenghi, Gerardo Pelosi, and Michele Scandale. 2015. Trace-based schedulability analysis to enhance passive side-channel attack resilience of embedded software. *Information Processing Letters* 115, 2 (2015), 292–297.

[7] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. 2014. On the cost of lazy engineering for masked software implementations. In *Smart Card Research and Advanced Applications - 13th International Conference (CARDIS'14). Revised Selected Papers (LNCS)*, M. Joye and A. Moradi (Eds.), Vol. 8968. Springer, 64–81.

[8] Josep Balasch, Benedikt Gierlichs, Oscar Reparaz, and Ingrid Verbauwhede. 2015. DPA, bitslicing and masking at 1 GHz. In *Cryptographic Hardware and Embedded Systems (CHES'15) (LNCS)*, T. Güneysu and H. Handschuh (Eds.), Vol. 9293. Springer, 599–619.

[9] Josep Balasch, Benedikt Gierlichs, Roel Verdult, Lejla Batina, and Ingrid Verbauwhede. 2012. Power analysis of Atmel cryptomemory - Recovering keys from secure EEPROMs. In *Proceedings of Topics in Cryptology - The Cryptographers' Track at the RSA Conference 2012 (CT-RSA'12) (LNCS)*, O. Dunkelman (Ed.), Vol. 7178. Springer, 19–34.

[10] Alessandro Barenghi, Guido Marco Bertoni, Luca Breveglieri, and Gerardo Pelosi. 2013. A fault induction technique based on voltage underfeeding with application to attacks against AES and RSA. *Journal of Systems and Software* 86, 7 (2013), 1864–1878.

[11] Ali Galip Bayrak, Francesco Regazzoni, David Novo, Philip Brisk, Francois-Xavier Standaert, and Paolo Ienne. 2015. Automatic application of power analysis countermeasures. *IEEE Transactions on Computing* 64, 2 (Feb. 2015), 329–341.

[12] Jeremy Bennet. 2008. The OpenCores OpenRISC 1000 Simulator and Tool Chain Installation Guide. Retrieved April 2018, from https://www.embecosm.com/appnotes/ean2/embecosm-or1k-setup-ean2-issue-3.html.

[13] Eric Brier, Christophe Clavier, and Francis Olivier. 2004. Correlation power analysis with a leakage model. In *Cryptographic Hardware and Embedded Systems (CHES'04) (LNCS)*, M. Joye and J. Quisquater (Eds.), Vol. 3156. Springer, 16–29.

[14] Marco Bucci, Raimondo Luzzi, Francesco Menichelli, Renato Menicocci, Mauro Olivieri, and Alessandro Trifiletti. 2007. Testing power-analysis attack susceptibility in register-transfer level designs. *IET Information Security* 1, 3 (2007), 128–133.

[15] Alessandro Cevrero, Francesco Regazzoni, Micheal Schwander, Stéphane Badel, Paolo Ienne, and Yusuf Leblebici. 2011. Power-gated MOS current mode logic (PG-MCML): A power aware DPA-resistant standard cell library. In *Proceedings of the 48th Design Automation Conference (DAC'11)*, L. Stok, N. D. Dutt, and S. Hassoun (Eds.). ACM, 1014–1019.

[16] Zhimin Chen, Ambuj Sinha, and Patrick Schaumont. 2013. Using virtual secure circuit to protect embedded software from side-channel attacks. *IEEE Transactions on Computing* 62, 1 (Jan. 2013), 124–136.

[17] Francesco Conti, Davide Rossi, Antonio Pullini, Igor Loi, and Luca Benini. 2016. PULP: A ultra-low power parallel accelerator for energy-efficient and flexible embedded vision. *Journal on Signal Processing Systems* 84, 3 (2016), 339–354.

[18] Jean-Sébastien Coron, David Naccache, and Paul C. Kocher. 2004. Statistics and secret leakage. *ACM Transactions on Embedded Computing Systems* 3, 3 (2004), 492–508.

[19] Julien Doget, Emmanuel Prouff, Matthieu Rivain, and François-Xavier Standaert. 2011. Univariate side channel attacks and leakage modeling. *Journal on Cryptographic Engineering* 1, 2 (2011), 123–144.

[20] Walter Fumy, Limei Yu, and Bastien Gavoille. 2012. IT Security techniques. ISO/IEC 18033-3:2010 and 29192-2:2012. Retrieved April 2018, from https://www.iso.org/standard/54531.html.

[21] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Adi Shamir, and Eran Tromer. 2016. Physical key extraction attacks on PCs. *Communications of the ACM* 59, 6 (2016), 70–79.

[22] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. 2016. ECDSA key extraction from mobile devices via nonintrusive physical side channels. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and Shai Halevi (Eds.). ACM, 1626–1638.

[23] George Becker, J. Cooper, Elke DeMulder, Gilbert Goodwill, Joshua Jaffe, G. Kenworthy, T. Kouzminov, A. Leiserson, M. Marson, Pankaj Rohatgi, and S. Saab. 2013. Test vector leakage assessment (TVLA) methodology in practice. In *International Cryptographic Module Conference*.

[24] Martin Goldack. 2008. *Side-Channel Based Reverse Engineering for Microcontrollers*. Technical Report. Ruhr University Bochum. Retrieved April 2018, from https://www.emsec.rub.de/media/attachments/files/2012/10/da_goldack.pdf.

[25] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. 2011. A testing methodology for side-channel resistance validation. In *NIST Non-Invasive Attack Testing Workshop (NIAT'11)*.

[26] John L. Hennessy and David A. Patterson. 2011. *Computer Architecture: A Quantitative Approach* (5th ed.). Morgan Kaufmann Publishers, San Francisco.

[27] Richard Herveille, OpenCores Organization. 2010. Wishbone System-on-Chip (SoC) interconnection architecture for portable IP cores. Retrieved April 2018, from https://www.ohwr.org/attachments/179/wbspec_b4.pdf.

[28] Annelie Heuser, Olivier Rioul, and Sylvain Guilley. 2014. Good is not good enough - Deriving optimal distinguishers from communication theory. In *Cryptographic Hardware and Embedded Systems (CHES'14) (LNCS)*, L. Batina and M. Robshaw (Eds.), Vol. 8731. Springer, 55–74.

[29] Yuval Ishai, Amit Sahai, and David Wagner. 2003. Private circuits: Securing hardware against probing attacks. In *Advances in Cryptology (CRYPTO'03) (LNCS)*, D. Boneh (Ed.), Vol. 2729. Springer, 463–481.

[30] Franck Jullien, Jeremy Bennett, Jonas Bonn, Julius Baxter, Michael Gielda, Olof Kindgren, Peter Gavin, Sebastian Macke, Simon Cook, Stefan Kristiansson, Stafford Horne, and Stefan Wallentowitz. 2016. OpenRISC Reference Platform SoC version 3. Retrieved April 2018, from https://github.com/openrisc?page=1.

[31] Paul C. Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. 2011. Introduction to differential power analysis. *Journal of Cryptographic Engineering* 1, 1 (2011), 5–27.

[32] Stefan Kristiansson. 2016. Bare-metal introspection application for the AR100 controller of Allwinner A31 SoCs. Retrieved April 2018, from https://github.com/skristiansson/ar100-info.

[33] Pei Luo, Yunsi Fei, Xin Fang, A. Adam Ding, David R. Kaeli, and Miriam Leeser. 2015. Side-channel analysis of MAC-Keccak hardware implementations. In *Proceedings of the 4th Workshop on Hardware and Architectural Support for Security and Privacy (HASP@ISCA'15)*, R. B. Lee, W. Shi, and J. Szefer (Eds.). ACM, 1:1–1:8.

[34] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. 2007. *Power Analysis Attacks - Revealing the Secrets of Smart Cards*. Springer.

[35] Trevor Martin. 2016. *The Designer's Guide to the Cortex-M Processor Family* (2nd ed.). Elsevier.

[36] David May, Henk L. Muller, and Nigel P. Smart. 2001. Random register renaming to foil DPA. In *Cryptographic Hardware and Embedded Systems (CHES'01) (LNCS)*, Ç. K. Koç, D. Naccache, and C. Paar (Eds.), Vol. 2162. Springer, 28–38.

[37] Francesco Menichelli, Renato Menicocci, Mauro Olivieri, and Alessandro Trifiletti. 2008. High-level side-channel attack modeling and simulation for security-critical systems on chips. *IEEE Transactions on Dependable and Secure Computing* 5, 3 (2008), 164–176.

[38] Amir Moradi, Alessandro Barenghi, Timo Kasper, and Christof Paar. 2011. On the vulnerability of FPGA bitstream encryption against power analysis attacks: extracting keys from Xilinx Virtex-II FPGAs. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*, Y. Chen, G. Danezis, and V. Shmatikov (Eds.). ACM, 111–124.

[39] Ingram Olkin and John W. Pratt. 1958. Unbiased estimation of certain correlation coefficients. *Annals of Mathematical Statistics* 29, 1 (1958), 201–211.

[40] OpenRISC Community. 2014. OpenRISC 1000 architectural manual. Retrieved April 2018, from https://raw.githubusercontent.com/openrisc/doc/master/openrisc-arch-1.1-rev0.pdf.

[41] Siddika Berna Örs, Frank K. Gürkaynak, Elisabeth Oswald, and Bart Preneel. 2004. Power-analysis attack on an ASIC AES implementation. In *International Conference on Information Technology: Coding and Computing (ITCC'04)*, Shahram Latifi (Ed.), Vol. 2. IEEE Computer Society, 546–552.

[42] David Oswald and Christof Paar. 2011. Breaking Mifare DESFire MF3ICD40: Power analysis and templates in the real world. In *Cryptographic Hardware and Embedded Systems (CHES'11) (LNCS)*, B. Preneel and T. Takagi (Eds.), Vol. 6917. Springer, 207–222.

[43] Christof Paar, Thomas Eisenbarth, Markus Kasper, Timo Kasper, and Amir Moradi. 2009. Keeloq and side-channel analysis-evolution of an attack. In *6th International Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC'09)*, L. Breveglieri, I. Koren, D. Naccache, E. Oswald, and J. Seifert (Eds.). IEEE Computer Society, 65–69.

[44] Olivier Pereira, François-Xavier Standaert, and Srinivas Vivek. 2015. Leakage-resilient authentication and encryption from symmetric cryptographic primitives. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, I. Ray, N. Li, and C. Kruegel (Eds.). ACM, 96–108.

[45] Francesco Regazzoni, Alessandro Cevrero, François-Xavier Standaert, Stéphane Badel, Theo Kluter, Philip Brisk, Yusuf Leblebici, and Paolo Ienne. 2009. A design flow and evaluation framework for DPA-resistant instruction set extensions. In *Cryptographic Hardware and Embedded Systems (CHES'09) (LNCS)*, C. Clavier and K. Gaj (Eds.), Vol. 5747. Springer, 205–219.

[46] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. 2015. Consolidating masking schemes. In *Advances in Cryptology (CRYPTO'15)–Part I (LNCS)*, R. Gennaro and M. Robshaw (Eds.), Vol. 9215. Springer, 764–783.

[47] Hermann Seuschek and Stefan Rass. 2016. Side-channel leakage models for RISC instruction set architectures from empirical data. *Microprocessors and Microsystems* 47, Part A (2016), 74–81.

[48] James E. Stine, Jun Chen, Ivan D. Castellanos, Gopal Sundararajan, Mohammad Qayam, Praveen Kumar, Justin Remington, and Sohum Sohoni. 2009. FreePDK v2.0: Transitioning VLSI education towards nanometer variation-aware designs. In *IEEE International Conference on Microelectronic Systems Education (MSE'09)*. IEEE Computer Society, 100–103.

[49] Kris Tiri and Ingrid Verbauwhede. 2005. Simulation models for side-channel information leaks. In *Proceedings of the 42nd Design Automation Conference (DAC'05)*, W. H. Joyner Jr., G. Martin, and A. B. Kahng (Eds.). ACM, 228–233.

[50] Kris Tiri and Ingrid Verbauwhede. 2006. A digital design flow for secure integrated circuits. *IEEE Transactions on CAD of Integrated Circuits and Systems* 25, 7 (2006), 1197–1208.

Received September 2017; revised March 2018; accepted April 2018