Sorbonne Université

Doctoral School of Computer Science, Telecommunication and Electronics (EDITE) of Paris
Computer Science Department

Doctoral Thesis

(On) The Impact of the Micro-architecture on Countermeasures against Side-Channel Attacks

---

Lorenzo CASALINO

Jury composition:

Lilian BOSSUET	University Jean Monnet	Reviewer
Louis GOUBIN	University of Versailles-St-Quentin-en-Yvelines	Reviewer
Sonia BELAÏD	CryptoExperts	Examiner
Erven ROHOU	PACAP, INRIA	Examiner
Arnaud TISSERAND	Lab-STICC, CNRS	Examiner
Nicolas BELLEVILLE	CEA-List	Supervisor
Damien COUROUSSÉ	CEA-List	Supervisor
Karine HEYDEMANN	Thales DIS–LIP6, Sorbonne Université	Supervisor

Audentes Fortuna Juvat

Acknowledgements

30th January 2024. It is in this very day that I conclude another chapter of my life. A chapter began three years ago, the 21st January 2021. On that day, on that train from Milan to Paris, boarded at 06:25 a.m. from the Pt. Garibaldi train station, during the Covid-19 pandemic, I was wondering what I would have lived, who I would have met, how it would have ended. To have lead me to this day, I have to thank my supervisors. Together, we spent three years of intense and fruitful collaboration; although the hard times, you always cared, supported and guided me for the best. For this, I will always be grateful to you. I warmly thank to the members of the "Ufficio del Inferno" (original spelling): an unofficial institution of our laboratory, created by the few Ph.D. students. If I got so far, surviving to the constant threat of infamous *colpo d'aria*, I due it to you. In these Ph.D. journey, my experience hasn't been marked only by the support in the working environment, but also and foremost, by who I met in the everyday life. A huge thank you to Grenoble Swing and all its members: by you, I lived unforgettable days and evenings. Thank to you, I had the possibility to get part, to be part, of something bigger than me, improve myself, do something for the city that have hosted me during these three years. Right in this association, I met who became my nemesis and, likewise, my brother-in-dancing Sami. Together, we spent awesome evenings before a St. Stefanus, throwing each others challenges and dreaming about our future. A not-so-unveil thank you goes to Giulia and Roberto: these three years have been a continuous discovery and evolution of my self; although you didn't notice, you played a central role. But the support I received goes well beyond this three-year-long french chapter. A incredibly big shout-out to my "fixed starts": Silvia, Gianluca, Martina, Federico, Carol, Nipun, Eleonora and Fabrizio. Distant as stars, sometimes in China, Belgium or UK, you have always been—and I hope you always be— my "reference system", my everlasting support, either in the darkest and brightest moments. Really, thank you.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Chapter 1

Introduction

> Don't you realize there's a war on? We can't bring our cryptographic operations to a screeching halt based on a dubious and esoteric laboratory phenomenon. If this is really dangerous, prove it.
>
> U.S. Signal Corps on EM-based Side-Channel [Boa73, p. 90].

In the 10th and last of his lectures, David G. Boak reports one of the first documented traces of what we could define a *side channel*: during World War II, the Bell Telephone discovered that the 132-B2, an encryption device used by the U.S. Signal Corps, emitted an electromagnetic radiation which allowed the reading of the plaintext during its encryption operated by the device. The above quote, directly extracted from the same lecture, documents the scepticism about a phenomenon which, still today, makes raise an eyebrow when heard of it for the first time.

Far from being a dubious and esoteric laboratory phenomenon, side channels represent a concrete and serious threat to any setting concerned with the protection of some sensitive asset: confidential communications and intellectual property just to name a couple.

The study of side channels and their exploitation (*side-channel analysis*) has experimented an exponential development. With this introductory chapter, we frame the context and motivation behind this thesis document and its contributions. We conclude it with an overview of the manuscript organization.

1.1 Context

In a large sense, with *side channel* we refer to any alternative communication channel through which one can gain (partial) information on some fact or on some system. With the term *side-channel analysis* we refer to any analysis methodology relying on the information conveyed through a side-channel to derive some conclusions with a certain probability. For instance, one could measure a room's temperature (the *side-channel*), compare the recorded temperature with respect to the mean room's temperature when empty (the *analysis*) to know whether someone was/is there (the *conclusion*).

This methodology finds wide-spread application in the cryptology field. To ensure confidential communications, cryptography provides mathematically-strong algorithms (*ciphers* or *cryptosystems*), which cannot be broken, in feasible time, by relying solely on the their mathematical structure (the *main* communication channel). We refer to this cryptanalytical approach as *black-box* cryptanalysis: the *attacker* can only interact with the inputs and outputs, exploiting the mathematical structure to recover information on the user key. Instead of resorting solely on this black-box view of the targeted cryptosystem, the attacker can exploit information on the on-going computations to recover the employed cryptographic key. This information stems from the non-idealities of the computing platform which, interacting with the surrounding environment, creates informative side-channels.

The research community widely envisioned and studied effective countermeasures against this kind of analyses. The key idea behind any side-channel countermeasure is to limit the acquisition of exploitable information by the attacker.

Such idea can materialize as a *suppression* of the side-channel or by degrading the quality of the information signal. The first case consists in removing, either physically or logically, the source of the side-channel. The second case, instead, consists in increasing the noise affecting the side-channel.

Concerning the second case, we can categorize the countermeasures preventing the exploitation of side-channel information leakage into *hiding* and *masking*. Hiding aims to *hide* the informative signal behind the noise affective the side-channel. Masking, instead, consists in splitting secret-dependent information in several random values, such that the attacker needs to recover all of them to gain useful information. Masking acquired particular attention from the research community, thanks to its framework to define and prove specific security guarantees.

Despite this formal verification of security, the security proofs often rely on hypotheses that are hardly met in the real world. This discrepancy between theoretical and physical contexts impacts both hardware and software applications of masking. In particular, when employed to provide side-channel protection to software implementations,

one faces a more subtle problematic, as the so-called *Instruction Set Architecture* (ISA) supported by a CPU hides to the masking scheme designer the reasons explaining the discrepancy. According to Hennessy and Patterson, *an ISA consists in the portion of the computer architecture* visible *to the programmer or compiler writer* and *the ISA serves as the boundary between software and hardware* [HP12]. Gao et al. further elaborates on the concept of an ISA, describing it as an *interface* between what the programmer can access to (the *architecture*) and what the programmer cannot interact with (the *micro-architecture*) [Gao+20b]. A micro-architecture encompasses high-level aspects of a CPU, such as the memory system and the CPU's design [HP12]. By acting as a contract between software and hardware, an ISA allows semantics equivalence when executed on different CPUs supporting the same ISA, although these CPUs potentially differ in terms of micro-architecture. As a concrete example, a program described with the ARMv7 ISA is expected to provide the same output if executed on the simple ARM Cortex-M4 CPU, or the more performant ARM Cortex-M7, although their micro-architecture substantially differ. Thus, in the end, the masking scheme designer can directly interact solely with the architectural features of a micro-processor to not convey information through the possible side channels. In the more general sense, due to the *opaque* nature of the micro-architecture, the designer is not aware of the actual security impact that the micro-architecture has on a software implementation.

In the side-channel literature, the role of the micro-architecture on the side-channel security of software is relatively new, with the first publication linking the information leakage to the underlying micro-architecture dating back to 2017 [PV17]. From then, more publications focused on the impact of the micro-architecture in terms of information leakage, exploring and classifying the different sources of leakages encompassed within the micro-architecture. Such sources (registers within the micro-architecture, signal glitches characterizing micro-architecture's combinatorial elements, speculative features and the memory subsystem) have been found as the root cause of informative side-channels [BP18; Gao+20a; Gao+20b; MMT20; GPM21; MPW22].

1.2 Motivation

The impact of the micro-architecture is a problem more and more acknowledged, as witnessed by the increase of bodies of work concerned with the investigation of micro-architecture-induced side-channels [Zon+18; BP18; Gao+20a; Bar+21; MPW22; dHM22]. Due to the hardware nature of the problem, a long-term solution would be to provide micro-architecture designs (and their physical implementation) oriented towards the mitigation of micro-architecture-induced leakages. In this regard, a branch of the side channel literature promotes the conception of a micro-architecture supporting the secure execu-

tion of masked software, either by providing mechanisms to compute masked operations in a secure manner directly in hardware [Kia+19; CP23], or by making the ISA software/hardware contract less opaque through mechanisms to control, from the software layer, the potential side-channel threat [Gao+20b].

However, at the current state of affairs, most of the commercial CPUs do not provide any hardware-based leakage mitigation mechanism. As such, masked software implementations running on these processors are left exposed to the impact of the micro-architectural leakage. A branch of the literature focuses in the identification of short-term solutions. To support this goal, the side channel community developed tools for the verification of masked software implementations [CGD18; Gig+21; KS22; ZMM23] and methodologies to enable the secure execution of these implementations [Gao+20b; She+21b; GD23]. With *methodologies* we mean guidelines for the secure development of masked software, or approaches to automate the development process, taking into account the micro-architecture threat. These guidelines are an important ingredient, as they state general principles to avoid the leakage of sensitive information. In the same vein, approaches for automated generation of micro-architecture-secure implementations constitute an important aspect, as they ease the development of masked software, while meeting more practical aspects, like the time-to-market in the case of an industrial product.

However, such methodologies exhibit some degree of dependency to the targeted micro-architecture, which materializes several challenges. The portability of the implementation's security is not guaranteed in the general case, as the ISA enables behavioural diversity while preserving functional compatibility [Gao+20b]. Furthermore, the development of a micro-architecture-aware implementation strictly relies on the knowledge of the micro-architecture itself. Such knowledge is typically limited to the public available information, if any, as being part of an intellectual property; as a result, the resulting implementation does not cover the whole attack surface provided by the micro-architecture. Another point of difficulty is represented by the *complexity* of the micro-architecture design: as the micro-architecture provides more performance-oriented features, the more increase the potential sources of side-channel leakages. Attempting to address all of them might result in a costly solution in terms of execution time, for instance; or it might result in an impossible task in particular cases. The problematic exacerbates when considering the *types* of leakages: attempting to cover, at the same time, transition-based, glitch-based and coupling-based leakages potentially increase the difficulty of the task. Thus, one might wonder whether there exist approaches more *agnostic* with respect to the underlying micro-architecture.

More generally, one could ask *if* and *how* we can design and develop side-channel masked software *practically* secure in the context of micro-architectural leakage.

1.3 Contributions

With this thesis, we address, along two axes, the issue of developing side-channel masked software *practically* secure in the context of micro-architectural leakage.

The first axis concerns the automated development of masked software resilient to (micro-)architecture-induced transition-based leakages. Indeed, from the state of the art, the existing automated approaches either rely on oversimplified models of the micro-architecture, or only consider leakage effects stemming from the architectural elements of the CPU, or address the problem with a *we fix what we detect* approach. Moreover, part of these approaches work at the machine-code level, losing the opportunities that an approach operating on a higher-level of the compilation process would bring in terms of performance. The first contribution of this thesis describes a methodology for the automated development of masked software resilient against transition-based leakages. This methodology takes advantage of the code-generation phase of optimizing compilers: given in input a software implementation—annotated with side-channel-related information—and a description of the target micro-architecture, we show how to exploit instruction scheduling and register allocation to mitigate transition-based leakages in an automated manner. With respect the current state of the art, this methodology tackles the problem from a different level of the compilation process, acting on an intermediate representation of the masked program. At the same time, by employing code-generation tools, we render the micro-architecture-induced leakage mitigation part of compilation process, acquiring all the benefits of this latter. Last but not least, the approach relies on a micro-architectural description fed as an input to the code-generation tools. This approach provides a separation of concerns, as the design of the code-generation algorithms does not depend on the specific target micro-architecture. The level of security we can reach strictly depends on the quality of the model integrated within this description.

The second axis concerns with a more target-agnostic approach to counteract the micro-architecture-induced leakages. As we remarked right before, relying on the micro-architectural details comes with a huge burden for the masking scheme designer. Not relying on these details potentially improves the situation, in particular in terms of portability of the security of the solution across CPUs supporting the same ISA. If we look at the literature, most of the works focus on protecting Boolean masked software, but Boolean masking is well-known to be sensitive to recombination effects. Some bodies of work show the robustness of other masking schemes against recombination effects, which are likely to occur in a micro-architecture. One might wonder what are the practical security guarantees of these masking schemes in the presence of micro-architecture-induced leakages. At the same time, the state of the art seems to overlook the potential exploitation of the parallelism capabilities of modern CPUs. The design of current processors

have evolved to increase the number of instructions per clock cycle that they might process and, in some cases, complete at the same time. As a consequence, the processor can process different data in parallel. Such parallel capabilities stem from different and orthogonal hardware-oriented techniques. Among these, one of the most wide-spread is the *pipelining*, which we can find also in simple CPUs, like commercial micro-controllers. Motivated by these two observations, we study the practical security offered by first-order Boolean, arithmetic-sum and inner-product masking schemes, considering both transition-based leakages and the leakage induced by data parallelism. At first, we elaborate and practically show how the side-channel leakage induced by data parallelism can manifest in simple CPU designs. We evaluate the leakage resilience of first-order masking encodings when impacted by transition-based leakages and the leakage induced by data parallelism. Finally, we evaluate the leakage resilience of several software implementations of the AES-128 cryptosystem, each masked with one of the first-order masking schemes we study. Such study highlight how, although the better resilience of some of the considered masking schemes, all of them do not withstand the exploitation of the considered sources of leakage.

1.4 Document Organization

We structure the remainder of this thesis manuscript in 5 chapters. In Chapter 2, we introduce the background information essential for the following chapters. In Chapter 3, we put forward a literature review concerning the state-of-the-art methodologies to mitigate micro-architectural leakages. In Chapter 4, we present the first contribution of this thesis: an automated methodology, acting from the software layer, to mitigate transition-based leakages induced by the micro-architecture. In Chapter 5, we present the second contribution of this thesis: the study of the leakage resilience of different first-order masking schemes when impacted by transition-based leakage and the leakage induced by data parallelism, both induced by the micro-architecture. Eventually, with Chapter 6, we provide conclude this thesis and we outline interesting research lines.

Chapter 2

Background

This chapter puts forward the essential backgroung required for the remainder of this thesis manuscript. We first introduce the employed notation. Then, we introduce the concept behind a passive Side-Channel Analysis (or Attack) (SCA), the attacks steps and main elements required to mount such type of attack. We continue presenting the masking countermeasure, its key idea and its formal security model. We conclude with an overview of two evaluation approaches to assess the security of an implementation, either hardware or software, of some algorithm.

Table 2.1: Notations summary for Chapter 2.

Notation	Meaning
$\mathbb{A}$	Set
X	Random variable
x	Realisation of X
$\mathcal{X}$	Sampling space of the random variable X
$\mathbf{X}$	(Row) vector of random variables X_i, $0 \leq i < m$
$\mathbf{x}$	(Row) vector of realisations $(X_i)_j$, $0 \leq i < m$, $0 \leq j < n$
$\mathbf{x}^{\mathrm{T}}$	Transposition of vector $\mathbf{x}$.
$\mathbf{X}_j$	j-th element of m-dimensional vector $\mathbf{X}$, $0 \leq j < m$
$\mathbf{x}_j$	j-th column of the $n \times m$ matrix $\mathbf{x}$, $0 \leq j < m$
$\mathbf{1}$	$n \times m$ matrix of 1s
$\langle \mathbf{A}; \mathbf{B} \rangle$	Inner product between vectors $\mathbf{A}$ and $\mathbf{B}$
$\circ$	Function composition

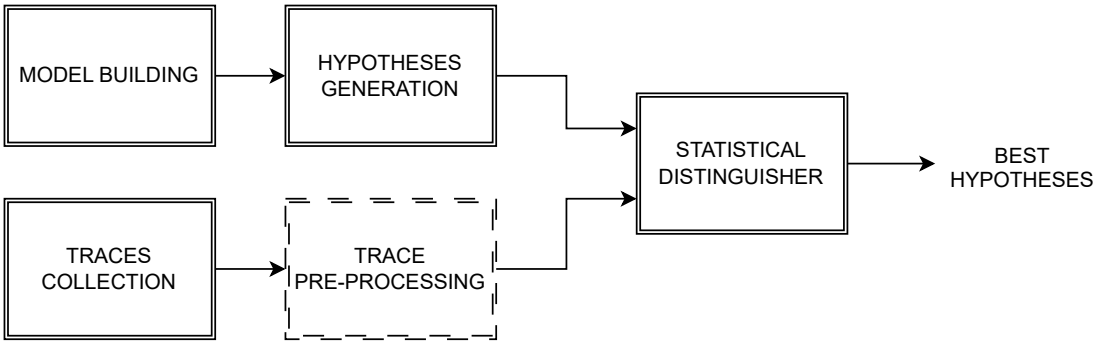


Figure 2.1: Overview of a typical SCA process.

2.1 Notations

In this section we introduce the general notations employed throughout the remainder of this manuscript. Table 2.1 reports the notations in compact form.

With a blackboard bold letter $\mathbb{A}$ we denote a generic set. We refer to random variables with a capital letter X. We denote the sampling space of X with a calligraphic notation $\mathcal{X}$, where the small letter x denotes a realisation of X. With a bold capital letter $\mathbf{X}$ we denote an m-dimensional (row) vector of random variables X_i, whereas the small bold letter $\mathbf{x}$ denotes an n-dimensional (row) vector of realisations of $\mathbf{X}$. Implicitly, the transposition $\mathbf{x}^{\mathrm{T}}$ can be also seen as a $n \times m$ matrix. we denote with $\mathbf{X}_j$ and $\mathbf{x}_j$ the j-th column of the matrix $\mathbf{X}$ and $\mathbf{x}$, respectively. The symbol $\mathbf{1}$ refers to a $n \times m$ matrix full of 1, whose dimensions are inferred by the context. With $\langle \mathbf{A}; \mathbf{B} \rangle$ we denote the *inner product* operation between the two vectors $\mathbf{A}$ and $\mathbf{B}$.

2.2 Passive Side-Channel Attacks

As overviewed in Chapter 1, an SCA is a cryptanalytical tool exploiting additional information on the cryptographic implementation got from an alternative communication channel, the *side* channel. This information can come from a *passive* observation of the implementation's activity, or from its *active* manipulation [MOP07]. The scope of this thesis limits to the former category of SCA, which we overview in this section. For the sake of brevity, in the rest of the manuscript we refer to passive SCA simply as SCA.

2.2.1 Principle

In a classical cryptanalysis (or key-recovery) attack, the attacker attempts to recover the full user key K by solely interacting with the input/output interface of the target

cryptosystem C. On the contrary, an SCA attacker takes further advantage of some measurable quantity M characterizing an implementation I of C. Such quantity M might be the instantaneous power consumption dissipated by I, or the variation of its electromagnetic field [MOP07]. M constitutes the *side channel* through which I *leaks* information.

Figure 2.1 reports the typical flow of an SCA. At first, the attacker measures (or *probes*) the side-channel M, acquiring (partial) knowledge on one or more processed key-dependent intermediate values $V = f(P, K)$, where P is the plaintext employed to compute V. The attacker records the side-channel as a time-dependent signal called *(side-channel) trace*, represented as an s-dimensional vector $\mathbf{T} = [T_i]$, where T_i is the i-th time instant (or *sample*) of the trace. Due to the noisy nature of the measurements, for instance due to physical phenomena such as *thermal noise* [Nyq28], the attacker increases the probability to recover the user key by collecting several traces, represented as an m-dimensional vector $\mathbf{t} = [(t_i)_j]$, where m stands for the number of traces; implicitly, $\mathbf{r} = \mathbf{t}^{\mathrm{T}}$ can be seen as a $m \times s$ matrix.

Traces can be optionally *pre-processed*, in order to improve the efficiency of the analysis.

At the same time, the attacker builds a model L of the expected variation of the side-channel: the *leakage model* (or *function*). The attacker uses the built model to generate an $m \times |\mathcal{K}|$ matrix of *hypotheses* (or *guesses*) $\mathbf{h} = [h_{i,j}]$ for the key-dependent value V, where $h_{i,j} = f(P = p_i, K = k_j)$, p_i is the plaintext employed to collect the i-th trace and k_j is the key guess to test. Finally, the hypotheses $\mathbf{h}$ and the traces $\mathbf{r}$ get analyzed by a statistical tool $D(k)$ called *distinguisher*:

$$D(k) \triangleq [D(\mathbf{h}_k, \mathbf{r}_j)_j] \tag{2.1}$$

such that the (most likely) employed key, at sample j, is $\hat{k}_j = \mathrm{argmax}_k[D(k)_j]$. The key $\hat{k}_j$ does not necessarily match the correct one, but only represents the most likely candidate with a certain probability. Several factors, such as the insufficient number of traces or an inadequate leakage model, potentially compromise the success of the SCA.

We evaluate the interest of an attack by its *complexity*. In the context of SCAs, one bottleneck stems from the traces acquisition, both in terms of acquisition time, storage and analysis. Thus, a first metric to evaluate the attack's complexity is the *number of traces* m required to successfully recover the user key [Pap+22].

But what part of the cryptosystem does the attacker observe? To reduce the complexity of the attack, induced by the dimension of $\mathbf{h}$, the attacker aims at those parts of C where V depends on a small part of K; that is, the first rounds of C. With such approach, the attacker can adopt a *divide-et-impera* approach to get the user key K, as

they can independently target different intermediate values covering a different part of K. As a consequence, the attacker can consider a smaller $\mathbf{h}$.

2.2.2 Leakage Model

As overviewed in Section 2.2.1, to mount an SCA, the attacker has to select a model L to describe the variation of the side channel. The accuracy of the model plays a paramount role in the complexity of the attack, but a too accurate one can prevent its application to more broader contexts, for instance other implementations of the same cryptosystem.

The selection of the leakage model starts from some assumptions on the characteristics of the side channel. A classic assumption is to consider the side-channel signal as the superposition of a deterministic component L_d and a random one representing the Additive Gaussian Noise (AGN) affecting the measured signal [DFS19]. Equation 2.2 reports the described additive model, where σ is the noise' standard deviation.

$$L(X) = L_d(X) + \mathcal{N}(0, \sigma). \tag{2.2}$$

Concerning the deterministic part L_d, different assumptions can be made. For instance, one might assume that the leakage of a variable X is a linear combination of the leakage of its bits. Literature refers to it as a *linear* leakage model:

$$L_d(X) = \langle \mathbf{w}; X \rangle \tag{2.3}$$

where $\mathbf{w} \in \mathbb{R}^n$ expresses the contribution of each bit of X to the side channel. Further assumptions can be made according the particular technology employed. In CMOS-based technology, the update (*transition*) of a memory element is assumed to leak information according to the number of cells changing their state. Literature refers to such linear model as Hamming-Distance (HD) model [MOP07]:

$$L_d(X, Y) = HD(X, Y) \triangleq \langle \mathbf{1}; X \oplus Y \rangle. \tag{2.4}$$

The Hamming-Weight (HW) model (Equation 2.5) represents a particular case of the HD model, where the memory cell is assumed to transition from a constant $X = c$ (usually, $c = 0$).

$$L_d(X, Y) = HD(c, Y) = HW(Y) \triangleq \langle \mathbf{1}; Y \rangle. \tag{2.5}$$

The HD leaks information on the state-transition of the memory elements–hence, revealing information on both X and Y–whereas the HW one leaks only on Y. Literature refers to the former as a *transition-based* leakage model, whereas the latter is referred to as *value-based* leakage model [Bal+14].

2.2.3 Statistical Distinguisher

Once the attacker has generated the hypotheses $\mathbf{h}$ and collected the optionally preprocessed traces $\mathbf{t}$, they provide these elements to the statistical tool employed to *distinguishing* the best hypotheses h_k. Such tool–the *distinguisher* $D(k)$–provides a *score* for each hypothesis h_k, which expresses the likelihood that a given value k of the key is employed. Usually, the computation of D_k requires the knowledge of the true distributions of $\mathbf{h}$ and $\mathbf{t}$. Hence, the attacker computes an *estimation* $\hat{D}(k)$, such that:

$$\lim_{m\to\infty} \mathbb{E}[(\hat{D}(k) - D(k))^2] \to 0 \tag{2.6}$$

where m represents the number of analysed traces [GHR15].

The literature populates with several types of distinguishers, ranging from the simple *Difference of Means*, to more accurate and complex ones, such as the *Mutual Information*. In the following, we briefly present the two most employed distiguishers in literature which we employ in the rest of this manuscript.

Pearson's Correlation Coefficient

The Pearson's Correlation Coefficient (PCC) ρ provides a normalized measure, in the range $[-1, +1]$, of the *linear* correlation existing between two random variables X and Y. In the side-channel context, $X = \mathbf{H}$ and $Y = \mathbf{T}$, respectively the hypotheses and the side-channel traces. Recalling that $\mathbf{r} = \mathbf{t}^{\mathrm{T}}$ can be seen as a $m \times s$ matrix, Equation 2.7 reports the estimator $\hat{\rho}$ of this distinguisher:

$$\hat{\rho}\,(\mathbf{h}, \mathbf{r}) \triangleq \left[\left(\frac{\langle \mathbf{h}_i; \mathbf{r}_j \rangle - m\mu_{\mathbf{h}_i}\mu_{\mathbf{r}_j}}{[(\langle \mathbf{1}; \mathbf{h}_i^2 \rangle - m\mu_{\mathbf{h}_i}^2)(\langle \mathbf{1}; \mathbf{r}_j^2 \rangle - m\mu_{\mathbf{r}_j}^2)]} \right)_{i,j} \right] \tag{2.7}$$

where m is the number of analysed traces.

Heuser et al. show that PCC represents an *optimal* distinguisher when the leakage model L is in the form of Equation 2.2, it is linear and it deviates from the real leakage behaviour for a multiplicative constant factor [HRG14].

Mutual Information

The Mutual Information (MI), an Information-Theoretic (IT) tool, quantifies the information content shared between two random variables:

$$\mathrm{MI}(\mathbf{H}, \mathbf{T}) \triangleq [(H(\mathbf{H}_i) - H(\mathbf{H}_i \,|\, \mathbf{T}_j))_{i,j}] \tag{2.8}$$

where $H(X)$ (respectively, $H(X\,|\,Y)$) is the entropy of X (respectively, the conditional entropy of X given Y), defined as:

$$H(X) \triangleq -\mathbb{E}[\log_2 P(X)] \tag{2.9}$$

$$H(X|Y) \triangleq -\mathbb{E}[\log_2 P(X|Y)] \tag{2.10}$$

The interest in MI lies in its capacity to capture both linear and *non-linear* relationships between the two target variables, providing better distinguishing capability.

Unfortunately, the power of this metric stems from employment of the conditional distribution $P(\mathbf{H}_i|\mathbf{T}_j)$, which, in general, can be only estimated [VS09]. The accuracy of such estimation depends on (1) the number of side-channel traces collected and (2) the estimation technique [VS09; Bro+19].

When computing the MI by means of the estimated conditional entropy, we refer to it as Hypothetical Information (HI), which reflects that the information quantity comes from a hypothetically correct model of the leakage distribution [Bro+19]. Renauld et al. introduced a further approximation of the MI, called Perceived Information (PI) [Ren+11]. This metric quantifies the *perceived* information under the estimated distribution when the real one potentially differs.

Bronchain et al. show that HI and PI are, respectively, an upper bound and lower bound for the true MI [Bro+19].

2.2.4 Further Aspects

In this section, we overviewed the fundamental aspect characterizing SCAs. Still, there are two aspects that we consider worthwhile to mention: the direction of an SCA and the *profiled* SCA.

Vertical and Horizontal Attacks

To succeed in their goal, the attacker can analyse a single, key-dependent variable V working on a portion of the user key K, selecting a suitable sample s of the collected traces. In such case, we say that the attacker performs a *vertical* SCA.

One might remark that the attacker implementation potentially manipulates V across a single execution. Furthermore, another variable W might depend on the same key-portion K too. Therefore, an attacker might take advantage of this additional sources of information in the rest of the trace, by considering more than one sample. In such case, the attacker mounts a *horizontal*, or *multivariated*, SCA [MOP07].

Profiled Attack

The process depicted in Figure 2.1 represents the simplest form of SCA, a so-called *unprofiled* attack. In an unprofiled attack, the attacker relies on an *assumed* leakage

model. As mentioned in Section 2.2.2, the accuracy of the model impacts the complexity of the attack.

Under this observation, and when the attacker has access to an identical copy of the implementation, they may characterize, or *profile*, the probability distribution of the leakage $L(V)$ for each potential value of the key; in other words, they estimate the conditional probability distribution $P(\mathbf{L}(V) \mid K)$. With this *a-priori* knowledge, they retrieve the key hypothesis $\hat{k}$ maximizing the likelihood on the *a-posteriori* knowledge:

$$\hat{k} = \text{argmax}_k \prod \frac{P(\mathbf{L}(V) \mid K = k) \cdot P(K = k)}{\sum_{j=0}^{|\mathcal{K}|} ((\prod P(\mathbf{L}(V) \mid K = k_j)) \cdot P(k_j))} \tag{2.11}$$

Historical examples of such attacks are the *Stochastic Approach* and the *Template Attacks* [SLP05; CRR02].

All the approaches mentioned up to now, both unprofiled and profiled, target intermediate values dependent on a small part of the key. The Algebraic Side-Channel Analysis (ASCA) takes another direction, combining leakage profiling with algebraic attacks [RS09]. The idea is to maximize the probability to recover the full user key by exploiting the information from *all* the intermediate variables. In practice, the attacker builds a system of equations representing the cryptographic algorithm and injects in them the acquired *a-priori* knowledge gained from the leakage profiling of each variable.

Soft-Analytical Side-Channel Analysis (SASCA) represents an evolution of ASCA, which follows the same principle, but improves the efficiency of the latter under several aspects [VGS14].

2.3 The Masking Countermeasure

The power of an SCA attacker stems from their ability to observe the internal state of the running implementation, getting knowledge on the processed intermediate variables. To increase the difficulty of the attack, it is possible to wisely split the key-dependent variables, such that (1) only their observation as a whole is informative, (2) exploitation of the information requires a higher number of measurements. The masking countermeasure satisfies these two properties. Informally, masking amplifies the noise affecting the side-channel measurements. Such amplification effect *masks* the statistical relation between the side-channel leakage and the key-dependent variables, hindering the efficiency of the attack. In this section, we report the basic notions behind masking and its relevant challenges.

2.3.1 Principle

When targeting a key-dependent variable V, an SCA attacker gets information on a portion of the user-key K. In its simplest form, the attacker probes a single sample s of the side-channel trace $\mathbf{T}$.

An intuitive approach to defeat them is to employ a *secret sharing scheme*: the key-dependent intermediate V gets split in two or more intermediate variables V_i called *shares*, such that the information on V is shared among them [Sha79]. As such, the attacker cannot recover V by probing a single sample. We call the tuple $\mathbf{V} = (V_i)_{i=0}^{n}$ an encoding of V:

Definition 2.1. ***Encoding.*** *Given a random variable $X \in \mathcal{X}$ the tuple $\boldsymbol{X} = (X_i)_{i=0}^{n} \in \mathcal{X}^{(n+1)}$ is an encoding of X. The random variables $\boldsymbol{X}_i \in \mathcal{X}$ are called shares.*

The encoding of V is built from an *nth-order masking scheme* M, where n represents the number of shares (the *masking order*). Informally, an nth-order masking scheme is an invertible vector-valued function $\mathrm{M} : \mathcal{X} \mapsto \mathcal{X}^{(n+1)}$, such that it satisfies *correctness* (Definition 2.2) and *dth-order security* (Definition 2.3).

Definition 2.2. ***Correctness.*** *Let M be an nth-order masking scheme, and $\boldsymbol{X}$ an encoding of X. Then, $X = M^{-1}(\boldsymbol{X})$.*

Definition 2.3. ***dth-Order Security.*** *Let M be an nth-order masking scheme. M satisfies dth-order security if and only if, for each $X \in \mathcal{X}$, any subset of (at most) d shares of $\boldsymbol{X} = (X_i)_{i=0}^{n} = M(X)$ does not statistically depend on X. $d \leq n$ defines the* security *order of M.*

According to the sharing strategy, we obtain different kinds of encodings:

Definition 2.4. ***Boolean Encoding.*** *Let us consider $X \in \mathcal{X}$, where $k \geq 1$ and $\boldsymbol{X} = (X_i)_{i=0}^{n} = \mathrm{BM}(X)$ the boolean encoding of X. Then $X = \bigoplus_{i=0}^{n} \boldsymbol{X}_i$, where $\oplus$ is the eXclusive OR.*

Definition 2.5. ***Arithmetic-Sum Encoding.*** *Let us consider $X \in \mathcal{X}$, where $k \geq 1$ and $\boldsymbol{X} = (X_i)_{i=0}^{n} = \mathrm{ASM}(X)$ the arithmetic-sum encoding of X. Then $X = \boxplus_{i=0}^{n} \boldsymbol{X}_i$, where $\boxplus$ is the arithmetic sum.*

Definition 2.6. ***Inner-Product Encoding.*** *Let us consider $X \in \mathcal{X}$, where $k \geq 1$ and $\boldsymbol{X} = (X_i)_{i=0}^{n} = \mathrm{IPM}(X)$ the inner-product encoding of X. Then $X = \langle L; X \rangle$. $\boldsymbol{L} = (1, L_i)_{i=1}^{n} \in \mathcal{X}^{(n+1)}$ is a* public *random vector, and $\langle \cdot ; \cdot \rangle$ is the inner-product operator.*

Once the key-dependent intermediates get shared, the original algorithm C has to be *compiled* into a semantically equivalent algorithm G able to process the encodings. We call this G a *gadget* of C:

Definition 2.7. ***Gadget.*** *Let us consider an algorithm $Y = C(X)$ and a masking scheme M. We call* gadget *of C an algorithm G such that $(M^{-1} \circ G)(\boldsymbol{X}) = C(X)$.*

The transformation of a variable into an encoding (and vice-versa) is operated by an *encoding* (*decoding*) gadget.

The chosen masking scheme provides a set of transformations to mask (or *compile*) C to G. Hence, we can define a masking scheme as a strategy specifying (1) the encoding transformation and (2) the set of rules to compile the algorithm C.

The usual strategy to compile an algorithm C is by *decomposition*: considering C as the composition of m smaller algorithms $C_0, C_1, \ldots, C_{m-1}$, we can mask each of them, such that:

$$G = M(C) = M(C_{m-1}) \circ \ldots \circ M(C_0).$$

The masking scheme M defines a *base* B of atomic algorithms in which C can be decomposed, as well as the set of transformations to mask each element of B. Other approaches, instead, attempt to find ad hoc solutions for a specific algorithm C. For instance, in the case of lookup-based Substitution Box (SBox), it is possible to follow a *recomputation* approach, where the whole lookup table gets masked [Cha+99; Cor14].

When adopting a decomposition approach, the masking of an element $b \in B$ has a different impacts on the *complexity* according to the nature of b. According to the employed masking scheme, we can distinguish two types of algorithms: linear and non-linear. When dealing with, for instance, a binary linear algorithm $C(A, B)$, a sound strategy is to apply C directly on the shares of A and B, such that:

$$G(\mathbf{A}, \mathbf{B}) = (C(\mathbf{A}_0, \mathbf{B}_0), C(\mathbf{A}_1, \mathbf{B}_1))$$

Hence the *time* and *randomness* complexity increase *linearly* in the number of shares. Concerning non-linear computation, the matter becomes more complex, as non-linear terms, called *cross-terms*, appear in the equation. For illustration purposes, let us consider the masking of the finite field multiplication $C = A \odot B$. According to Rivain et al. [RP10], we can expand the multiplication as:

$$C = A \odot B = \bigoplus_{0 \leq i,j \leq n} \mathbf{A}_i \odot \mathbf{B}_j.$$

Thus, the masked multiplication (Algorithm 1) requires a time complexity quadratic in the masking order. Such results holds for every non-linear computation. In general, masking non-linear computations comes with an *expansion* phase, where all the n^2 cross-terms are computed, and a *compression* phase, where the cross-terms are recombined and the n shares of the output are generated [Rep+15].

The compression phase intrinsically leads to information leakage. To counteract this, the usual strategy is to add to the cross-terms a new fresh random variable. As we get n^2 cross-terms from the expansion phase, the new randomness required is asymptotically $\mathcal{O}(n^2)$.

The generation of randomness is a costly task, which impact on the time complexity (and area, in the case of hardware implementations) of the final gadget G. Hence, also the *randomness* complexity has to be considered.

Algorithm 1: Gadget SecMult

Input: $\mathbf{A}$, $\mathbf{B}$
Output: $\mathbf{C} = \mathbf{A} \odot \mathbf{B}$

```
1  begin
2      for i from 0 to n do
3          C_i ← A_i ⊙ B_i;
4      for i from 0 to n do
5          for j from i + 1 to n do
6              R_{i,j} ← U_{2^k};
7              C_i ← C_i ⊕ R_{i,j};
8              T_i ← A_i ⊙ B_j;
9              R'_{i,j} ← R_{i,j} ⊕ T_i;
10             T'_i ← A_j ⊙ B_i;
11             C_j ← C_j ⊕ (R'_{i,j} ⊕ T'_i);
12     return C
```

2.3.2 Security Models

The masking countermeasure finds its appeal in the formal framework where its security guarantees can be proven. We call this framework a *security model*. Within its scope, the masking scheme designer defines the model of the leakage and of the attacker against which they want to prove the security of the masking design.

Several security models have been proposed in literature. Chari et al. introduced the *noisy security model*, in which they prove that an n-th order masking exponentially increases the number of traces m–that is, the difficulty of a successful attack–for an

attacker observing the leakage of the whole encoding, where each share *independently* leaks a *sufficiently noisy* function of their value [Cha+99].

Isahi et al. introduced the so-called *t-probing* security model. The t-probing model considers an implementation leaking a noise-free function of each intermediate computation of the algorithm C and the attacker is limited to observe, at most, t-out-of-$(t+1)$ of these computations. Under such model, an implementation is said to be *t-probing secure* if and only if an SCA attacker cannot retrieve any portion of the user-key K from the t observations [ISW03].

The t-probing model is quite simple, as the attacker is assumed to have limited knowledge and the leakage model is noise-free. The simplicity of the t-probing model makes the security evaluation of masking implementations simpler and amenable to automation. Yet, it was proven that an implementation secure in the t-probing model implies security in the more realistic setting captured by the noisy model [DDF14].

As said, the security model provides specific assumptions on how the implementation leaks and how the attacker observes the leaked information. When one of these assumptions do not hold, the proven security guarantees do not too. As we will discuss in Section 2.3.3, a typical example is represented by the leakage model, which does not capture certain physical effects. To bridge this gap, newer models have been developed, adapting and extending more abstract ones.

2.3.3 Leakage Independence and Physical Effects

Masked algorithms are proven secure under a specific model of security. In reality, these models hardly comply with the real behaviour of the implementation. A recurrent violated assumption is the so-called Independent Leakage Assumption (ILA) [Bal+14]. Under this assumption, each probe provides information on (at most) one share. Unfortunately, hardware and software implementations are characterized by physical effects which violate the ILA. That is, these effects *recombine* the shares, allowing the attacker to learn, through a single probe, information on multiple shares at once.

These effects can stem from three different sources, which are the mirror of three different physical phenomena characterizing a circuit design: memory transitions, signal glitches and coupling (or wire cross-talking) [MPW22; GPM21; De +17].

Memory transitions (or just transitions) leaks an amount of information related to the older and new state of the involved memory element (for instance, a register). Signal glitches (or just glitches) is a well-known phenomenon affecting combinational hardware: the relative delay between the input signals of a combinational circuit let latter's output go through several temporary states before stabilizing, leaking information on the input

signals. Coupling effects emerge when two hardware elements (for instance, two wires) are close enough to interact with each other, or due to power supply noise.

Balasch et al. formalized the impact of recombination effects through the *Security Order Reduction* theorem [Bal+14]:

Theorem 2.1. ***Security Order Reduction.*** *A d-th order secure implementation against value-based leakage functions is $\frac{d}{2}$-th order secure against transition-based leakage functions.*

Yet, such theorem considers recombination effects affecting two shares, for instance memory updates. Gigerl et al. practically proved that signal glitches potentially recombine more than two shares, further reducing the security order claimed by the above theorem [GPM21].

To counteract these effects, several masking techniques have been developed, such as Threshold Implementation (TI) [Bil+14] and Domain-Oriented Masking (DOM) [GMK16], two design paradigms for the development of masking schemes resilient to glitch-based leakages. In order to provide a formal analysis and design process of leakage-resilient implementations, new security models extending the original d-probing model have been proposed [Fau+18].

Yet, the design of masked implementations resilient against recombination effects can be challenging in software as the hardware design of a CPU is *fixed*. If we think about a software implementation as a virtualisation of a hardware design, the only way to avoid recombination effects is to carefully craft the software implementation, such that it properly selects which hardware component employ to perform the computations.

2.4 Security Evaluation of Implementations

When developing cryptographic implementations, the designer (or the evaluator thereof) naturally wants to understand what are the security guarantees of such implementation. A straightforward way to assess the security level is to attempt a key-recovery attack against it. Yet, such approach is quite naïve, as the security level obtained is linked to the specific attack employed. In other words, a *leakage-exploitation* approach does not provide the derivation of generic results concerning the security of the implementation [Pap+22]. Indeed, an evaluator is likely interested to understand the implementation' security level independently of a specific attacker model [SMY09].

A natural observation is that an SCA attacker can recover the key if and only if the implementation leaks key-dependent information. A first approach is to check whether the implementation *leaks* or not. Such approach takes the form of *statistical hypothesis testing*. A second methodology relies on the *quantification* of the amount of information

the worst-case attacker might exploit; dually, this translates to quantify the amount of information the implementation leaks.

2.4.1 Statistical Hypothesis Testing

When performing security evaluations of a cryptographic implementation, the evaluator aims to certify whether an implementation leaks or not in the most general setting (i.e., independently of the attacker's capabilities).

To this end, the evaluator can employ *statistical hypothesis testing*: given a *null* hypothesis H_0 we would like to verify and an *alternative* one H_1, the procedure computes a measure called *statistic* and a *threshold*, through which the evaluator can decide whether reject or not the null hypothesis. In terms of SCA security evaluation, usually we have that $H_0 :=$ I does not leak.

Among the different hypothesis testing procedures, the *Test Vector Leakage Assessment* (TVLA) is extensively used in literature [SM15]. In its simplest form, the TVLA certifies whether an implementation leaks by testing whether two sets of side-channel traces $\mathbb{S}_{\text{fixed}}$ and $\mathbb{S}_{\text{random}}$ can be distinguished by their means. $\mathbb{S}_{\text{random}}$ refers to side-channel traces collected while the implementation processes a different plaintext for each trace, whereas $\mathbb{S}_{\text{fixed}}$ refers to the usage of the same plaintext for each trace. Both sets are collected employing the same cryptographic key, kept fixed across the measurements.

The evaluator computes the t-statistic t:

$$t = \frac{\hat{\mu}_{\text{fixed}} - \hat{\mu}_{\text{random}}}{\sqrt{\frac{\hat{\sigma}^2{}_{\text{fixed}}}{n_{\text{fixed}}} + \frac{\hat{\sigma}^2{}_{\text{random}}}{n_{\text{random}}}}} \tag{2.12}$$

where $\hat{\mu}_{\text{fixed}}, \hat{\mu}_{\text{random}}$ refer to the sample mean, $\hat{\sigma}^2{}_{\text{fixed}}, \hat{\sigma}^2{}_{\text{random}}$ to the sample variance and $n_{\text{fixed}}, n_{\text{random}}$ to the number of traces of the fixed and random set, respectively.

The rejection of H_0 happens when the t-statistic overcomes a given *t-threshold*. Such threshold tells the evaluator that the H_0 can be rejected with a certain probability. Usually, evaluators set it to ± 4.5, meaning that they can reject the null hypothesis with a probability confidence of 99.999%.

As said, the TVLA relies is an hypothesis testing procedure. As all this types of procedures, it is affected by statistical errors too. We distinguish between Type-I errors (or *false positives*) and Type-II errors (or *false negatives*) [Ros10]. Type-I errors refer to the cases where the test fails (null hypothesis rejected), although the implementation does not leak. Type-II errors, on the other hand, refer to the acceptance of the null hypothesis, although the implementation actually leaks. Type-II errors are the most troublesome, as they would report an implementation as leakage-free when it is not. As

a mitigation technique against these types of errors, a strategy is to repeat the TVLA several times, each with a distinct fixed key [SM15].

Eventually, TVLA, and hypothesis testing in general, can only answer to whether a given implementation leaks or not, given a specific measurement setup. Even in the case of a leaking device, they cannot provide any insight on the difficulty of an SCA [Pap+23].

As we present in the next section, TI tools, in particular the MI, have been connected to metrics quantifying the SCA difficulty for a worst-case attacker, providing security projections of an implementation in the worst case.

2.4.2 Information Leakage Quantification

As informally elaborated at the beginning of this section, another approach to evaluate the security of an implementation is to understand how much information does the implementation leak. The idea is that an SCA attacker cannot exploit more information than the one leaked by the implementation. In Chapter 5, we will rely on this quantification approach to evaluate the leakage resilience of different masking schemes.

Chapter 3

State of the Art and Research Question

Although masking constitute a provably-secure methodology against passive side-channel attacks, in practice certain physical effects, such as memory transitions, hinder its efficacy. In the software context, these effects stem from the underlying hardware of the platform executing the masked implementation; in particular, from the so-called *micro-architecture* of CPUs. Different works attempt to address such problematic, re-establishing the theoretical security guarantees of masking. This chapter provides a literature review of the existing leakage mitigation approaches.

3.1 Preliminaries

In this section, we overview some key concepts paramount for the remainder of the chapter. We cover the concept of Instruction Set Architecture (ISA), micro-architecture and the enstablished *software/hardware* contract, the recombination effects issued from the micro-architecture and the variability of the leakage behaviour across micro-architecture designs.

3.1.1 ISA and Micro-architecture

When developing software, the implementation is encoded in a machine-code specification compliant with the ISA supported by the target platform. Informally, the ISA serves as an interface between what the software can access to (the *architecture*) and what it cannot interact with (the *micro-architecture*) [Gao+20b]. With micro-architecture, literature refers to the high-level aspects of the CPU's organization, as its logical design [HP12]. In practice, the ISA provides a *software/hardware* contract: given the same ISA, the

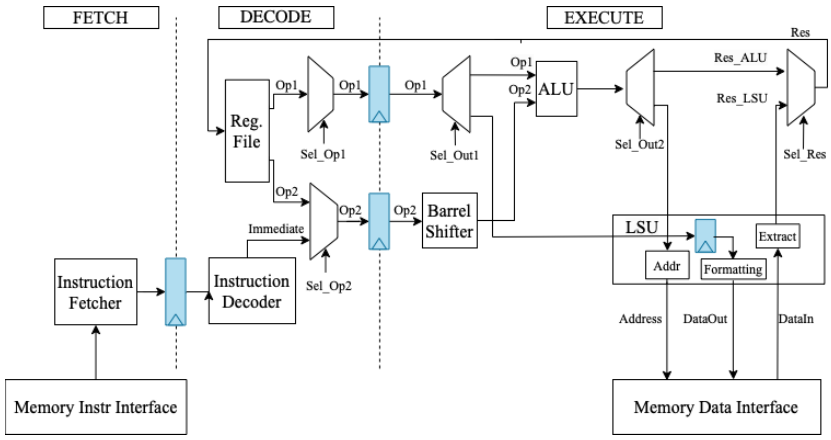


Figure 3.1: Simplified model of a 3-stage, in-order pipelined micro-architecture.

underlying micro-architecture provides the specified functionalities [MPW22]. In such way, the same piece of code will execute in the same way from a semantics point of view, regardless of the particular micro-architecture design.

According to the number of completed instructions per clock cycle, micro-architecture designs can be distinguished into *scalar* and *super-scalar*. A scalar micro-architecture can issue and complete at most 1 instruction per clock cycle, whereas super-scalar ones can issue and complete multiple instructions per clock cycle. Also, we can describe designs as *in-order* (instructions are issued and completed according to the order specified by the machine-code) or *out-of-order* (instruction issuing and completion does not follow the machine-code's order).

One hardware-oriented technique extensively employed to reach a throughput of 1 instruction per clock cycle is *instruction pipelining* [HP12]. According to this technique, the micro-architecture is partitioned into several *stages*, where each stage takes care of a part of the instruction life cycle. We refer to this sequence of stages as *execution pipeline*. Figure 3.1 depicts a simplified 3-stage, in-order, micro-architecture. In such example, the *Instruction Fetch* (`IF`) stage fetches the next instruction to be executed, the *Instruction Decode* (`DE`) interprets the instruction (e.g., selecting operands from the Register File), whereas the *Instruction Execute* (`EXE`) executes the instruction.

In the next section, we explain how micro-architectures can induce information leakage sources reducing the security of masked software.

3.1.2 Micro-architectural Leakage

As described in Section 3.1.1, an ISA defines an interface between the architecture and the micro-architecture of a CPU. Both of them encompass combinatorial and sequential (or

Table 3.1: Summary of selected works concerning micro-architectural leakage investigations. For each combination recombination effect/interaction type and origin, we report the work who report the observed effect. Concerning the coupling-based leakages, the only work we are aware of does not precise the origin of the leakage.

Leakage Source	Interaction	Origin					
		RF	Memory	Pipeline	Speculation	FU	Instr. Enc.
Transition	Intra	-	-	-	-	[BP18]	-
	Inter	[PV17]	[BP18; MPW22]	[BP18]	[MPW22]	[BP18]	[dHM22]
Glitch	Intra	[GPM21]	-	-	-	[Gao+20a]	-
	Inter	-	[She+21b]	[GPM21]	-	-	-
Coupling				[LBS19]			

memory) resources (e.g., physical registers, reported in blue in Figure 3.1). In particular, due to their sequential nature, memory resources can preserve a *state* across several clock cycles. We define *(micro-)architectural state* the set of states of the (micro-)architectural memory resources at given clock cycle.

Architectural and micro-architectural resources (both combinatorial and sequential) have been witnessed as the sources of several and different sources of leakage hindering the theoretical security of masked software implementations. According to the resource (architectural or micro-architectural) from which the information leakage originates, we define the leakage as *architectural* (or *architecture-induced*) or *micro-architectural* (or *micro-architecture-induced*).

Literature distinguishes recombination effects in:

- Transition-based: the recombination stems from the update of sequential elements (for instance, general-purpose registers) [Bal+14].
- Glitch-based: the recombination stems from the signal instability in combinatorial elements due to signal glitches [MPW22].
- Coupling-based: the recombination stems from the proximity of physical components (for instance, wires) or power supply noise [De +17].

Leakages can be further classified according to the type of *interaction*: a leakage causes *inter-bit* interaction when the recombination effects involves the bits of two distinct intermediate values; a leakage causes *intra-bit* interaction when the recombination effects involves the bits of the same intermediate value [GD23].

In this section, we overview the main leakage sources which stem from the *micro-architecture*. Yet, the goal is not to perform a review of all the works investigating

micro-architecture-induced leakage; rather, to present, for each recombination effect, a work that investigated it and found a new physical origin. When more than one work investigate the same effect, we tie the break by choosing the one that provides more insights on the said effect. For instance, both Papagiannopoulos et al. [PV17] and Marshall et al. [MPW22] observed transition-based leakages stemming from the memory; yet, the latter evaluates this effect in more depth and across different target micro-architectures.

Table 3.1 resumes the selected investigation works and explored leakages according to the above classification.

Transition-based Leakages Papagiannopoulos et al. empirically observed the presence of inter-bit recombination effects when the same architectural register is overwritten [PV17]. Marshall et al. systematically analysed the recombination effects stemming from memory accesses [MPW22]. In particular, they observed inter-bit interactions between consecutive and non-consecutive memory accesses. Barenghi et al. further observed inter-bit interaction on micro-architectures endowed with store buffers [BP18]. Barenghi et al. pointed out that inter-stage pipeline registers induced inter-bit interaction between the operands of consecutively issued instructions [BP18]. They also observed that functional units can originate inter and intra-bit interaction, for instance from the re-alignment buffer of the Load-Store unit and output register of the barrel-shifter, respectively. Marshall et al. observed inter-bit interaction due to speculative execution [MPW22]. Finally, de Grandmaison et al. showed that different encodings of the same instruction can lead to different leakage behaviours, and so different inter-bit interactions [dHM22].

Glitch-based Leakages Gigerl et al. observed that glitches in the address decode logic of the register file can lead to inter-bit interaction [GPM21]. The same authors also witnessed how inter-bit interaction can take place due to signal glitching in the forwarding mechanism of the execution pipeline. Shelton et al. observed an inter-bit interaction potentially stemming from the memory bus [She+21b]. Finally, Gao et al. show how glitches in the barrel-shifter potentially lead to intra-bit interactions [Gao+20a].

Coupling-based Leakages By means of acquisition setup manipulation, Levi et al. detected coupling-based leakages, although they didn't identify their specific origin [LBS19].

Table 3.2: Synthesis of works related to the exploration of micro-architectural leakage. We use n to indicate that the solution supports masking (up to) a fixed order. (✓): potentially handled. S.Scal: Super-Scalar, O.o.O: Out-of-Order. NC: Non-completeness. B: boolean; A: arithmetic.

Work	Method	Source		Bit Inter.		μ–Arch			Order	Masking
		Trans.	Glitch	Intra	Inter	Scalar	S.Scal.	O.o.O.		
[SSG17]	ISA	✓	-	-	✓	✓	-	-	1	B
[WSW19]	ISA	✓	-	-	✓	✓	-	-	1	B
[Ath+20]	ISA	✓	-	-	✓	✓	-	-	1	B
[Abr+21]	ISA	(✓)	(✓)	(✓)	(✓)	✓	-	-	1	B
[She+21b]	ISA	✓	-	✓	✓	✓	-	-	1	B
[She+21a]	ISA	✓	-	✓	✓	✓	-	-	Any	B
[Tso+23]	ISA	✓	-	-	✓	✓	-	-	1	B
[Kia+19]	ISE	✓	✓	✓	✓	✓	-	-	1,3	B
[KS20]	ISE	✓	✓	✓	✓	✓	-	-	1	B, A
[Gao+20b]	ISE	✓	-	-	✓	✓	-	-	Any	Any
[MP21]	ISE	✓	✓	✓	✓	✓	-	-	n	B, A
[Gao+21]	ISE	-	✓	✓	✓	✓	-	-	1	B, A
[CP23]	ISE	✓	-	-	✓	✓	-	-	Any	B, A
[MMT20]	NC	✓	✓	✓	✓	✓	✓	✓	Any	B
[GPM21]	NC	✓	✓	-	✓	✓	✓	-	Any	B
[GD23]	NC	✓	✓	✓	✓	✓	-	-	1	B

3.1.3 On the Variability of the Leakage Behaviour

The leakage behaviour of a given piece of code is not consistent across different platforms and across platforms supporting the same ISA [MMT20; MP21]. The reason is found on the transparency of the hardware/software contract implemented by the given ISA. According to such contract, the logical design of the micro-architecture can take any physical form, as long as it supports the ISA [Aro+21]. From a functional point of view, such transparency provides code portability (although it does not guarantee that the code exploits at best the underlying platform hardware). On the other hand, from a security point of view, it forbids a transfer of the security guarantees verified on a given platform. Furthermore, Arora et al. empirically show that the physical implementation of two instances of the same CPU can exhibit different leakage behaviours, worsening the leakage transferability problem [Aro+21].

3.2 Mitigation of Recombination Effects in Software

In Section 2.3.3 we discussed the security reduction of masked algorithms when executed on real platforms. The root cause of the discrepancy between the proven security and the practical one are physical effects which recombine the shares of a masked value. A natural way to *bridge* the gap between the theory and practice is to restore the violated ILA; in other words, enforcing *data isolation* [Bel+23].

A way to enforce data-isolation to mitigate transition-based leakage is through a *flushing* mechanism. With flushing, we refer to any hardware or software mechanism that overwrites (*flush*) the content of a memory resource (e.g., a register) with some value not related to any masked variable. By interleaving the consecutive storing of two shares in a memory resource with a flush of the latter, we mitigate the transition-based leakage.

In this section, we present a systematic review of the literature concerning methodologies to mitigate the recombination effects in software, narrowing the gap between the theoretic-proven security and the practical one.

3.2.1 Scope of the Review

The scope of this literature review focuses on the methodologies which aim to mitigate the impact of (micro)-architectural leakage on masked software implementations. we define the following research criteria to lead the literature investigation:

- Masking-specific Approach: the methodology must specifically target masked software implementations. In literature, there are works providing general protection, regardless of the software executed [Gro15; Gro+16; MGH19; SLP+19; ABP21; BJ22].
- Active Mitigation: the methodology must not only produce or support the execution of masked software, but also attempt to mitigate the security degradations implied by (micro-)architectural leakage. For instance, in literature there are works providing automated approaches for the generation of masked software, but they do not tackle micro-architectural leakage [Bel+18].

From the literature investigation, we identified 13 works matching the research criteria (Section 3.2.1). From their analysis, we identified 6 attributes of interest:

1. Method: it describes what method the work adopts to mitigate (micro-)architectural leakages: *ISA-based, Instruction Set Extension (ISE)-based, non-completeness-based.*
2. Source: it describes what leakage sources the work addresses: *transition-based, glitch-based leakages.*
3. Bit Interaction: it describes whether the work addresses the interaction between bit of the same machine word (*intra*) and/or of different machine words (*inter*).
4. μ-Architecture: it describes the type of micro-architecture the work considers: *scalar, super-scalar (s.scalar), out-of-order (o.o.o.).*

Table 3.3: Synthesis of works exploring the mitigation of micro-architectural leakages by means of ISA-based approaches.

Work	Approach	Scope		Input	Output		Order	Detect.
		Arch.	μArch.	Masked	Determ.	Converge		
[WSW19]	Pro-Act.	✓	-	✓	✓	-	1	Model
[Tso+23]	Pro-Act.	✓	✓	✓	✓	-	1	Model
[SSG17]	Pro-Act.	✓	✓	✓	-	-	1	Model
[Abr+21]	React.	✓	✓	-	✓	✓	1	None
[Ath+20]	React.	✓	-	✓	✓	✓	1	Model
[She+21b]	React.	✓	✓	✓	✓	✓	1	Profiled
[She+21a]	React.	✓	✓	✓	✓	✓	Any	Profiled

5. Order: it describes the order of the masked software executed on the target platform.

6. Masking: it describes the type of masking applied to the executed software.

Table 3.2 resumes the identified works. For the classification of these works, we chose the *method* attribute. The reason stems from the fact that, as we discuss in Section 3.3, each different approach has a different degree of *portability*, *efficacy* and *invasiveness*.

3.2.2 ISA-based Methodology

To deal with the leakage effects stemming from the micro-architecture, the developer can employ the instructions provided by the target ISA to mitigate the impact of such leakages by hand [PV17; GPM21]. Although possible to proceed via a by-hand approach, the hardening of software implementations is time-consuming and error prone. Therefore, in this section we focus solely on approaches relying on *automated* strategies for the generation of leakage-resilient masked software.

From the literature exploration, we identified 7 distinct works and 6 attributes of interest:

1. Approach: it describes whether the work *reactively* mitigate the sources of leakage (hence, directly on the binary code) or it *pro-actively* generates leakage-resilient code.

2. Scope: it describes whether the work mitigates architectural and/or micro-architectural leakages.

3. Input Masked: it describes whether the input program must be *masked* or not.

4. Output: it describes whether the approach acts *deterministically* (given the same input, produces the same hardened code) and whether the approach is *guaranteed* to generated a hardened masked implementation.

5. Order: it describes the masking order of the output. In those instances where the input must be masked, it matches also the input's masking order.

6. Detection: it describes whether the detection of leakages comes from the use of a *model* of the micro-architecture, or it originates from a *profiling* of the leakage.

We classify the identified works according to the *approach* attribute. The reason stems from the fact that pro-active approaches allow the direct generation of protected implementations from high-level specifications (hence, they can take advantage of optimizations made during the whole compilation process), whereas reactive ones can only deal with already generated code but do not have to deal with programs optimizations (which might alter the applied masking).

Pro-Active Approaches

Seuschek et al. attempt to automate the elimination of micro-architecture-induced transition-based leakages by programming an instruction scheduling and a register allocation engine to, respectively, re-organize the machine code of 1^{st}-order masked implementations, such that sequences of instructions do not leak on micro-controllers, and provide a leakage-free assignment of registers [SSG17]. The tool employs a pre-compiled list of some instruction pairs that leak on the given target micro-architecture. On the downside, the tool converges probabilistically towards a leakage-free implementation.

Wang et al. handle the transition-based leakages stemming from architectural register overwrites at the register-allocation level [WSW19]: given in input a high-level specification of a 1-st order masked implementation, they model the register allocation as an optimization problem with a leakage constraint to identify a leakage-free allocation of registers to intermediate variables. Their solution does not require any flush of the architectural registers.

Tsoupidi et al. mitigate the transition-based leakages stemming from architectural registers overwrites and from consecutive memory accesses [Tso+23]. To achieve their goal, they formulate the register allocation and the instruction scheduling as optimization problems, which are solved by means of an SMT solver. In contrast to the work of Wang et al., the potential lies in the exploration of all possible solutions to the formulated problem, keeping only the ones with minimum cost *and* which prevent any leakage.

Reactive Approaches

Athanasiou et al. develop BATTL, a tool for the automated identification and mitigation of transition-based leakages stemming from architectural registers overwrite [Ath+20]. Their tool takes in input a 1-st order masked binary, perform symbolic execution to identify transition-based leakages and removes them by flushing the leaking registers.

Abromeit et al. pursue a modular approach for the protection of an unmasked machine code specification of an algorithm, which integrates the application of 1-st order Boolean masking with the protection against micro-architectural leakages [Abr+21]. Their approach consists in porting the concept of *compositional reasoning* [Bar+16] in the software domain: (1) leaking computations are replaced by semantically equivalent 1-st order leakage-resilient gadgets; (2) glue-code is inserted before and after each gadget invocation for ensuring their secure composition. As an hypothesis, the authors assume the existence of a set of 1-st order masked gadgets secure against micro-architectural leakages.

Shelton et al. envision ROSITA, a tool for the automated application of code patches to 1^{st} order masked machine-code [She+21b]. The tool acts in two iterative phases: (1) identification of code patterns which leak information; (2) replacement of leaking code patterns via hand-made patches specific for the target micro-architecture. A target-tailored *leakage emulator* provides the simulated side-channel traces for the identification of leaking code patterns. The iterative process terminates once no leakage is detected from the simulated traces.

Shelton et al. develop ROSITA++, an enhanced version of ROSITA able to handle also higher-order masked implementations [She+21a].

Discussion Almost all the known tools deal with already-masked implementations; only the instance from Abromeit et al. is able to generate a hardened masked implementation starting from an unprotected machine code specification [Abr+21]. Resorting on tools which are able to handle unmasked inputs comes at the advantage of the developer, which (1) does not require any knowledge on masking and (2) does not bother with potential errors in the application of the masking scheme on the high-level specification.

Concerning the masking order, almost every tool limits to 1-st order masking: avoiding share recombinations without incurring in high performance penalties is a difficult task. To the best of our knowledge, only ROSITA++ handles higher-order masked implementations [She+21a].

Concerning on how the transition-based leakages are mitigated, most of the approaches rely on instruction sequences to flush the (micro-)architectural state. This can easily increase the performance penalties of the implementation in terms of code-size

and execution time [She+21b; She+21a]. On the other hand, Wang et al. and Tsoupidi et al. provide solutions which do not require flushing the micro-architectural state: their solutions rely on modelling register allocation and instruction scheduling as an optimization problem and choosing a solution which does not leak [WSW19; Tso+23]. Although proved on different small/medium-sized use-cases, it is unclear whether they approach can always converge to a solution. As a consequence, their approach potentially does not guarantee the identification of a solution in the general case. The work of Seuschek et al. share the same convergence problem [SSG17]. In their particular case, due to the nature of their approach, the convergence can probabilistically fail, requiring several call to the tool to convergence.

To detect the leakage induced by the micro-architecture, Seuschek et al. [SSG17] and Tsoupidi et al. [Tso+23] rely on a model of the target micro-architecture. In principle, the knowledge of the micro-architectural details allows a precise mitigation of the leakages. Still, these two works rely on incomplete models, either missing certain features that induce leakages (such as inter-stage registers) or do not consider the execution activity of the micro-architecture, which potentially impact on the observed leakage.

The tools ROSITA and ROSITA++ are interesting as they do not require knowledge of micro-architectural details. Indeed, their strength stands in the on-the-fly *emulation* of the leakage of the input machine code, which allows to iteratively test and patch the implementation until fixing all leakage points. Yet, they are characterized by a low degree of portability: the emulation requires a model of the target's leakage behaviour extracted by profiling the target platform. Such profiling is extremely dependent on the actual physical implementation of the profiled target architecture [Aro+21], the measured side channel and the employed experimental setup.

Finally, none of the tools *officially* support glitch-based leakage mitigation: the work of Abromeit et al. only specifies the presence of a library of leakage-free gadgets; thus, the mitigation capabilities of their approach strictly depends on such library. Although ROSITA and ROSITA++ can detect glitch-based leakages, none of the hand-made patches handle them.

3.2.3 ISE-based Methodology

ISE are a mean to enrich the properties of an ISA, usually by introducing new functional features (for instance, SIMD paradigm) or non-functional ones (for instance, capabilities [Wat+20]). In the context of leakage-resilient cryptography, they found application to support masked software. Specifically, the idea behind the usage of ISEs is to expose a more friendly software/hardware contract, such that the software can take advantage of some hardware mechanism to mitigate the potential security impacts due to micro-

Table 3.4: Synthesis of works exploring the mitigation of micro-architectural leakages by means of ISE-based approaches. We use n to indicate the solution supports masking (up to) a fixed order. B: Boolean; A: Arithmetic.

Work	Deleg.	Source		Scope			Order	Masking
		Transition	Glitch	Pline	Cache	Mem.		
[KS20]	HW	✓	✓	✓	✓	-	1	B, A
[Gao+20b]	HW	✓	-	✓	-	✓	Any	Any
[Kia+19]	HW/SW	✓	✓	✓	-	-	1, 3	B
[MP21]	HW/SW	✓	✓	✓	-	-	n	B, A
[Gao+21]	HW/SW	✓	✓	✓	-	-	1	B, A
[CP23]	HW/SW	✓	-	✓	-	-	Any	B, A

architectural leakages [Gao+20b].

We identified 6 distinct works dealing with a leakage-resilient execution of masked software implementations through the aid of ISEs.

We identified 5 attributes of interest concerning these works:

1. Leakage Mitigation Delegation: it denotes whether the mitigation of the leakage happens at *software* level, *hardware* level or a mix of theme.

2. Leakage Source: it denotes the sources of leakage handled: *transition-based* and/or *glitch-based*.

3. Scope: it denotes what part of the core are handled: *pipeline*, *cache* and/or *memory*.

4. Masking Order: it denotes the supported masking order.

5. Masking Type: it denotes the supported masking type: *Boolean* and/or *arithmetic*.

Table 3.4 summarizes the attributes associated to each the 6 works. Among these 5, we perform the classification of these works according to the *Leakage Mitigation Delegation*.

The reason stems from the fact that the first attribute can provide a first qualitative metric to understand where does the complexity lies (on the software developer or on the hardware designer), as well as the potential cost in terms of area, code size and execution time.

Hardware Delegation

Kiaei et al. develop a pure ISE-based approach to allow the execution of leakage-free 1-st order masked software [KS20]. Their strategy bases on two principles: (1) the introduction of dedicated instructions for computing masked data, (2) isolated data path.

The first point is addressed by extending the RISC-V ISA. From the hardware side, the new instructions are processed by a DOM-based *masked* ALU, which handles, in a glitch-free manner, both Boolean and arithmetic 1-st order masked operations. The isolated data path hosts the masked ALU and supports the secure manipulation of masked data by (1) duplicating the register file and data caches, where each copy stores a given share domain, (2) transparently refresh load and store instruction's operands to mitigate transition-based leakages.

Gao et al. follow a different approach, by preventing transition-based leakages by *flushing* the memory elements that might leak [Gao+20b]. Such refreshing is hinted by the software by means of dedicated RISC-V *fence* instructions. The modification brought to the whole system concerns also the memory interface, allowing prevention of transition-based leakages stemming from elements outside the CPU core. The solution provides flexibility to the developer on deciding which elements the hardware should reset. The solution transparently support any type of masking and masking order.

Hardware/Software Delegation

Kiaei et al. introduced SKIVA, a RISC-V extension to support transition-based leakage-free 1-st and 3-rd order Boolean masking *and* fault detection [Kia+19]. Concerning masking, the approach relies on a *share-sliced* representation of data to avoid transition-based leakages [Gao+20a]. For such purpose, the ISE introduces dedicated instructions to convert from and to share-sliced representation of masked data, *plus* a specialized rotate instruction for the in-place rotation of the encodings. The share-sliced representation transparently mitigates the transition-based leakages stemming from memory accesses. Yet, this comes at the cost of implementing the algorithm such that it can work on share-sliced representation of data.

Marshall and Page mitigate glitch and transition-based leakages via an overloaded extension of the RISC-V ISA, where the security semantics of the instructions is chosen according to the input registers and implemented through DOM-based gadgets [MP21]. The register file is replicated in n register banks, each for a given share domain, where the n is set via software at configuration time by the software designer. Concerning transition-based leakages stemming from load and store instructions, the authors delegate such operation to the software developer to schedule such instructions in order to avoid shares recombination.

Gao et al. provide an ISE for the RISC-V supporting 1-st order Boolean and arithmetic-sum masking [Gao+21]. The approach bases on dedicated instructions, which are implemented by a masked DOM-based masked ALU, mitigating glitch-based leakages. The micro-architectural data path is modified in order to accommodate the masked ALU and

handle the secure the manipulation of masked data (for instance, map each share domain to a dedicated inter-stage pipeline register), mitigating transition-based leakages. Concerning the transition-based leakage from memory accesses, it is up to the software designer to properly schedule them or use instructions to flush the micro-architectural state.

Cheng and Page extend the RISC-V ISA to deal with transition-based leakages: new instructions, copying the semantics of already existing ones, are introduced and used to communicate to the micro-architecture to mitigate the leakage [CP23]. The authors provide two proof-of-concepts: a *latency-optimized* version, which mitigates all the transition-based leakages at hardware level; an *area-optimized* version, which mitigates the architecture-induced transition-base leakages (for instance, register overwrites) via hardware, whereas the ones stemming from loads and stores are removed decoding the instructions in a specialized sequence of general-purpose instructions. The latter solution modifies at the minimum the underlying decoding logic, following a CISC approach: the micro-architecture decodes the instruction in a specific instruction sequence to remove the leakage.

Discussion

A fair comparison between the ISE-based approaches represent a hard task, as they are more proofs of concept to show the ISE's potential to achieve leakage-resilient masked software [MP21]. Even willing, it is not possible to compare in terms of security, as evaluation are not performed [MP21], are done on small code sequences [CP23] or the modified architectures are different [Kia+19; Gao+20b] or unspecified [KS20]. The usage of different/unspecified modified architectures forbids a fair comparison in terms of occupied silicon area and latency.

Nonetheless, it is still possible to perform some general observations concerning their mitigation capabilities and their invasiveness (that is, the degree of hardware changes). In general, pure hardware delegation enables better leakage mitigation, as the hardware can precisely flush micro-architectural elements (and not only) and/or handle masked data with hardware-secured operations. On the downside, these solutions potentially can come with non-negligible hardware cost in terms of silicon area and require a substantial modification of the core.

On the other hand, hardware/software delegation approaches represent a compromise, where the part of the leakage mitigation is delegated to the software: for instance, instead of having a hardware module to protect memory accesses, implementation's code has to be properly scheduled to avoid information leakage [MP21; Gao+21].

All in all, both approaches have the potential to mitigate transition-based leakages

and glitch-based leakages to a *local* extent: indeed, the implementation of masked operations with glitch-free gadgets make the single computation secure against both types of leakages [KS20; MP21; Gao+21]. Yet, there is still a lack of mitigation for more subtle sources, as for the forwarding path in the execution pipeline [Gao+20b].

3.2.4 Non-Completeness-based Methodology

The non-completeness property was introduced with the TI masking paradigm [Bil+14]. In essence, a non-complete operation (or function) operates on a strict subset of the shares, such that whatever recombination takes place, the attacker will always miss a piece of information to recover the secret.

Non-completeness can be viewed as a weak form of data isolation, according to which only a strict subset of shares can leak. At the time of writing, few works put forward the idea to mitigate shares recombinations by enforcing this property in software.

Meyer et al. propose to decompose a high-level implementation in sub-functions, each working on a strict subset of the shares [MMT20]. Gigerl et al. propose to mitigate glitch-based leakages by applying a stricter version of non-completeness at pipeline-level [GPM21]: at each clock cycle, one and only one share of a masked variable can be processed within the pipeline. Gaspoz and Dhooghe mitigate the effects of shares recombinations applying the non-completeness at the register file level [GD23].

Discussion Enforcing the non-completeness property at software level is an interesting approach, which does not prevent the implementation to leak, but let it do it in such a way the attacker cannot exploit recombinations. The approach is similar to the original idea behind masking.

This property potentially minimizes the efforts to mitigate certain leakage effects by relying on few, public information concerning the micro-architecture (e.g., the depth of the execution pipeline). Furthermore, the employment of such properties potentially promote an increased portability of the implementation, as long as the running CPUs share a similar micro-architecture design.

On the other hand, as remarked by Gaspoz and Dhooghe, non-completeness provides a necessary but not sufficient property to bridge the gap between the theoretically-proven security and the witnessed one in practice, as certain micro-architectural features (e.g., micro-architectural registers) can still reduce the implementation's security [GD23].

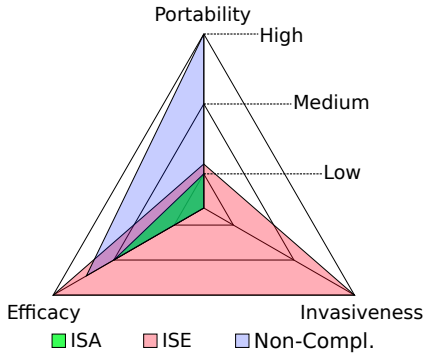


Figure 3.2: Qualitative Comparison of ISA-based, ISE-based and Non-Completeness-based Approaches.

3.3 Discussion

In this literature review, we investigated the different approaches to mitigate the effect of micro-architectural leakages on the security of masked software implementations. From the analysis of the works, we identified three attributes of interest:

- Efficacy: it qualitatively describes the mitigated leakages sources.
- Invasiveness: it qualitatively describes the degree of hardware changes required.
- Portability: it qualitatively describes the easiness with which the security guarantees of a software implementation can be transferred across micro-architectures without modification.

Figure 3.2 reports a qualitative radar-plot of the different approaches with respect to efficacy, invasiveness and portability.

Concerning the type of covered leakage sources (or *efficacy*), ISA-based approaches focus on transition-based ones. ISE-based solutions provide better mitigation capabilities, as they can cover also glitch-based leakages. Non-completeness-based approaches show some instances able to cover both leakage sources.

From an *invasiveness* point of view, ISA and non-completeness-based solutions do not require any hardware modification, whereas ISE ones might require non-negligible modifications to the hardware [CP23].

In terms of portability, ISA-based approaches show the lowest degree of portability, as an implementation potentially has to be modified to transfer the same security guarantees to different micro-architectures. ISE-based ones show slightly better portability, as architectures supporting the same ISE contract will provide the same security guarantees for the same implementation; yet, this is granted as long as the ISEs implement

the same hardware/software contract. Non-completeness-based approaches guarantee the highest degree of portability, since the software specification (potentially, a high-level one) has just to ensure the non-completeness property. Looking at the supported micro-architectures, ISA-based and ISE-based approaches have applied only on scalar micro-architectures, whereas the non-completeness method have also been explored on more complex ones.

3.4 Open Problems and Research Question

In this section, we present the identified open problems and research question that this thesis addresses.

3.4.1 Open Problems

From the literature investigation we identified the following open problems:

Micro-architecture Knowledge and Security Guarantees The security guarantees provided by ISA-based approaches strictly depend on the knowledge of the micro-architecture. For instance, to protect an implementation against transition-based leakages, the solution requires a precise knowledge of all the micro-architectural elements that might leak the distance between two intermediate values. Such a piece of information is usually not disclosed, being part of the intellectual property covering the micro-architecture. Thus, the employment of ISA-based to close the gap between theory and practice is prevented by the lack of public information concerning the micro-architecture.

Lack of Portable Security Implementations generated by ISA-based approaches are generally not portable from a security point of view. Indeed, as explained at the beginning of this chapter, several physical implementations can satisfy the contract described by a given ISA. By consequence, since two physical implementations potentially exhibit different leakage behaviours, a piece of code secure on a given micro-architecture potentially is not on another one.

This problem is partially handled by ISE-based approaches: by extending the ISA contract with specific security guarantees, the same piece of code shows the same security guarantees on two micro-architectures implementing the ISE. Nonetheless, such outcome is restricted only to implementations satisfying the very same extended contract.

Simple Models of the Micro-architecture Some of the ISA-based methodologies consider a simple model of the micro-architecture, either excluding well-known leakage

sources, such as inter-stage pipeline registers [Tso+23], or not considering the actual activity of the micro-architecture which influences the observed leakage [SSG17]. As such, the usage of a simplified model (or the lack thereof) forbid the generation of leakage-resilient masked software implementations.

No Convergence Guarantees Certain ISA-based methodologies do not guarantee the generation of a machine-code implementation free of any (micro-)architecture-induced leakage [SSG17; Tso+23].

Performance Overheads Part of the ISA-based methodologies mitigate micro-architecture-induced leakages on the machine-code version of the implementation [Abr+21; She+21b; She+21a]. Acting on this code representation, these methods apply hand-made patches designed to properly mitigate the leakages *while* preserving the semantics of the program. The design of these patches to also account for performance faces two major challenges:

- the designer must know the execution state of the micro-architecture when the code in the patch is executed
- the designer must cope with a code already generated: a given modification potentially implies an avalanche of modifications, requiring further patching. As such, it is in the interest of the designer to *localize* the impact of the patch, potentially generating a less performant code.

Practical Higher-order Security is Unexplored Except for the works of Shelton et al. [She+21a] and Gigerl et al. [GPM21], all the reviewed approaches focus on achieved practical 1-st order security.

Lack of Super-Scalar and Out-of-Order Micro-Architectures The complexity of simple micro-architectures and the lack of public complete knowledge on them makes challenging the task of achieving practical security. For this, only few works attempt to provide approaches to reach practical security even on such types of micro-architectures [MMT20; GPM21].

Glitches: a yet-to-be explored case A large bodies of work concern with recombination effects stemming from transition-based leakages. Although highlighted as a real problem [Gao+20a; GPM21], software-based mitigation of glitch-based leakages is practically unexplored, except for works promoting the non-completeness property [MMT20; GPM21; GD23].

3.4.2 Research Question

From this literature review, it clearly emerges how achieving practical security of masked software in a general sense–that is, providing a masked implementation secure against worst-case attackers on any micro-architecture–is still an open problem. Reducing the scope, it is still challenging to achieve 1-st order security on simple scalar architectures against attackers exploiting only transition-based leakages. As highlighted in Section 3.4.1, the main reason is the lack of more complete micro-architectural models.

Two orthogonal challenges are the containment of the performance overhead and the guarantee of an approach always converging towards a leakage-free solution.

At the same time, it emerges that most of the works focus on Boolean masking and the risk implied by transition-based leakages. Meyer et al. remarked that Boolean masking and transition-based leakages share the same algebraic structure [MMT20]. From this observation, they suggest to employ masking schemes with a different algebraic structure to mitigate these recombination effects. However, few works investigated their employment in software [Che+21; Wu+22; Bec+22] and none studied the impact of the micro-architecture on their practical security.

Eventually, within these thesis, we follow four research directions:

1. What mitigation capabilities can we reach by considering more complete models of the micro-architecture
2. Can we contain the performance overhead while mitigating (micro-)architectural leakage
3. Can we ensure the convergence towards a leakage-free solution
4. What is the impact of the micro-architecture on the practical security masking schemes with a different algebraic structure from the Boolean one.

Chapter 4

An Automated Methodology to Mitigate Transition-based Leakages at Software Level

This chapter bases on an on-going joint work with the Politecnico di Milano.

As it emerged from the previous chapter, preserving the security guarantees of first-order masked software implementations is an open problem. Current ISA-based methodologies (Section 3.2.2) addressing both architectural and micro-architectural transition-based leakages, exhibit two shortcomings: performance overhead [She+21b; She+21a], partial leakage mitigation and no convergence guarantees [SSG17; Tso+23].

With this chapter, we aim to address these two drawbacks. We propose an ISA-based *pro-active* methodology (Section 3.2.2) for the automated generation of first-order masked software implementation resilient to both architectural and micro-architectural transition-based leakages.

Given an *intermediate representation* of the masked program, the approach relies on two code generation algorithms—*instruction scheduling* and *register allocation*—to generate a machine code implementation *while* mitigating transition-based leakages. These algorithms mitigate the leakages through a careful assignment of physical registers and code re-organization. These two operations are driven by an accurate simulation of the micro-architectural state evolution, which bases on a description of the target micro-architecture we provide to the algorithms. This description encompasses public information concerning both the micro-architectural resources inducing transition-based leakages *and* the information related to structure of the micro-architecture and the execution latencies of each instruction. In the unfortunate case where the algorithms cannot mitigate

the occurring leakage, preventing the convergence towards a solution, a *flush* mechanism removes the leakage.

By mitigating the leakages during the machine code generation, we potentially reduce the performance overhead (*shortcoming* #1). The employment of an accurate micro-architectural model not only helps to reduce the performance overhead of the implementation (*shortcoming* #1), but also supports a more complete mitigation of the leakages (*shortcoming #2*). At the same time, with the flushing mechanism we guarantee the convergence of the approach towards a leakage-free solution (*shortcoming* #3), as long as we verify the mitigation capabilities of the mechanism for the target micro-architecture.

We exemplify the core idea of our mitigation strategy through some use cases and provide the rationale supporting it (Section 4.2).

By viewing the register allocation and instruction scheduling (Section 4.3) as optimization problems, we show how to extend existing algorithms to integrate transition-based leakages as a further problem's constraint (Section 4.4).

We implement our approach as part of the `LLVM` Core Libraries (Section 4.3.5, Section 4.5) and we experimentally evaluate it along two axes: a security evaluation and an overhead evaluation (Section 4.6). For these evaluations, carried out on the scalar, in-order pipelined ARM Cortex-M4 CPU, we generate several implementations of the SIMON128/128 cryptosystem: an unmasked one, a first-order masked, a micro-architecturally protected first-order masked and a second-order masked. In the security evaluation, accordingly with the TVLA methodology, we assess the information leaked from each of these implementations. In the overhead evaluation, we assess the impact of our approach in terms of execution time, code size and required randomness measuring these metrics from these SIMON128/128 implementations.

At the time of writing, the current state of the works does not allow fair comparisons with the second-order masked implementation and the other ISA-based methodologies. Yet, we show that our automated approach contributes to the mitigation of leakages while nullifying the overhead on the required randomness; a cost with a high impact on the implementation's execution time (Section 4.6.3).

4.1 Notations

We start the chapter by extending the notations presented in Section 2.1. We use these extended notations to describe the algorithms presented in the incoming sections. Table 4.1 reports the notations in compact form.

With the capital letter Q, we refer to a generic queue of elements. With the capital letter P, we refer to a generic program. For simplicity, we model a program as a queue

Table 4.1: Notations summary for Chapter 4.

Notation	Meaning
Q	Queue
P	Program
M	Association (or map)
$\mapsto$	Left-to-right association operator
$[\cdot]$	Access operator for generic map M
$[]$	Empty queue or map
$\varnothing$	Empty set
$\leftarrow$	Assignment, enqueue, dequeue, set insertion and extraction operator
$\bigcup$	Union of two sets or two queues
$\bigcap$	Intersection of two sets
a	A generic element
$=, \neq$	Equality/inequality boolean predicate
$\perp$	No value assigned, e.g., $a = \perp$
`fun`	A procedure named *fun*
P	A generic program

of instructions. With the capital letter M, we refer to an association (or map) $\mathbb{A} \mapsto \mathbb{B}$ between two generic sets, with the direction of the map flowing from left to right.

The notation $[\cdot]$ represents the *access* operator for the map data-structures. According to the employed data-structure, with $[]$ we refer to the empty queue or the empty map. With $\varnothing$, we refer to the empty set. The $\leftarrow$ represents the assignment operator. When a queue appears on the left-hand side (right-hand side) of $\leftarrow$, it also enqueues (dequeues) an element in (from) the queue. When a set appears on the left-hand side (right-hand side) of $\leftarrow$, it also inserts (extracts) an element in (from) the set. The $\bigcup$ ($\bigcap$) represents the union (intersection) between two sets. When we apply $\bigcup$ to queues, it creates a set containing the elements of two queues.

With an italic, small letter we refer to a generic element. For instance, we use i to refer to a generic instruction. If not explicitly stated, the context defines the actual element represented by this notation.

The $=$ and $\neq$ represent, respectively, the *equality* and *inequality* boolean predicate. For the map data-structure, given an element a, we signal the absence of a mapped value with $\perp$, i.e., $M[a] = \perp$. We use the same notation to indicate that a generic element a has not value assigned, i.e., $a = \perp$.

Finally, we use the `truetype` font family to refer to a procedure (or function). Within an algorithm, the name of a procedure implicitly trigger its execution at that point of the program.

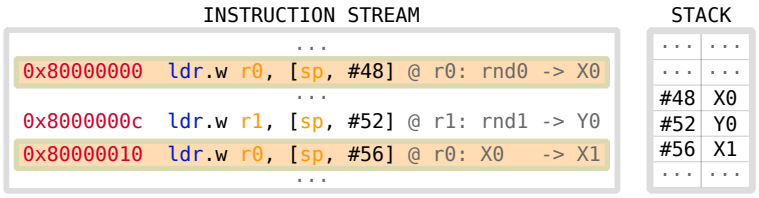


Figure 4.1: Architectural Register Overwrite.

4.2 Transition-based Leakage Mitigation

When we talk about *transition-based leakage mitigation*, we mean the act of reducing or suppressing the impact that such types of leakages have on implementations proven secure under the t-probing model. We recall that, according to the t-probing model, the masked implementation leaks a noise-free function of the handled input, intermediate and output variables, while the attacker can observe (at most) t of such variables.

Before presenting our methodology, we provide some examples to explain how instruction scheduling and register allocation algorithm can mitigate transition-based leakages. Specifically, for each example, we illustrate how either (1) a different register allocation or code organization or (2) a software-based flush mechanism can mitigate the observed transition-based leakage. The following list of examples is not exhaustive, and covers only a subset of known sources of leakages. Without lack of generalization, we report each example in the ARM `Thumb-2` assembly language, which manipulates some shares of two first-order masked intermediate variables X and Y. We denote these shares with `X0`, `X1` for X, and `Y0`, `Y1` for Y. Some of these examples also handle values statistically independent from X and Y. We refer to them with `rnd0`, `rnd1` and `rnd2`. In the rest of this section, we assume the case of a scalar, in-order, potentially pipelined micro-architecture.

We conclude the section by providing a rationale behind the correctness (i.e., we mitigate the leakage) of the adopted mitigation strategies.

Cautionary note. Although the *principle* behind our mitigation approach holds in general, its actual *materialization* strictly depends on the target micro-architecture. By consequence, in order to provide a sound implementation of our methodology, the mitigation applied via register allocation, instruction scheduling *and* the flushing mechanism must undergo a thorough side-channel analysis, e.g., according to the methodology of Marshall et al. [MPW22] or de Grandmaison et al. [dHM22].

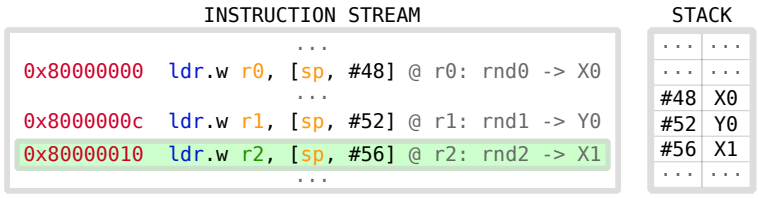


Figure 4.2: Architectural Register Overwrite: Register Re-allocation.

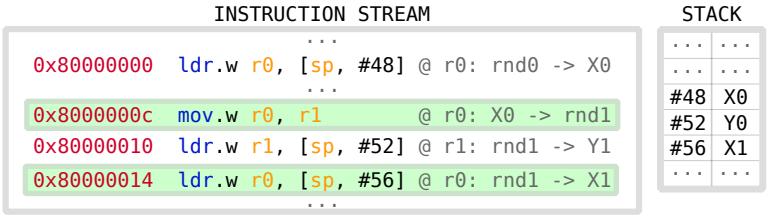


Figure 4.3: Architectural Register Overwrite: Register Flushing.

4.2.1 Architectural Register Overwrite

A typical case of leaking transition stems from the consecutive overwrite of the same architectural register. Figure 4.1 exemplifies this case. Before this code snippet, the architectural registers `r0`, `r1` and `r2` contain, respectively, the random value `rnd0`, `rnd1` and `rnd2`. The load instructions at `0x80000000` and `0x80000010` reads the two shares $\mathbf{X}_0, \mathbf{X}_1$, respectively, and write them in the same register `r0`. By re-using the same register, we induce a transition-based leakage, which provides information on the masked variable X.

As depicted in Figure 4.2, a different register allocation—assigning `r2` to the load at address `0x80000010`—can mitigate the problem.

In case such re-allocation might not possible, it possible to flush the guilty register. Figure 4.3 exemplifies such case: before storing $\mathbf{X}_1$ in register `r0`, we overwrite the latter with the random value `rnd1` stored in register `r1`. In this manner, the transition-based leakages do not involved any more the two shares at the same time, and we successfully prevent leaking information on X.

4.2.2 Micro-architectural Register Overwrite—Pipeline Registers

In pipelined micro-architectures, another typical example of leaking transition happens in the registers between the pipeline stages. Figure 4.4 reports a code snippet exhibiting this case. We assume the following content for the involved architectural registers, before the execution of the reported snippet: `r0` and `r1` contain, respectively, the random

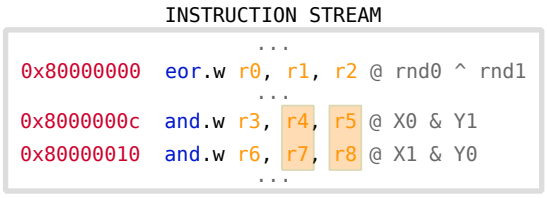


Figure 4.4: Micro-architectural Registers Overwrite–Pipeline Registers.

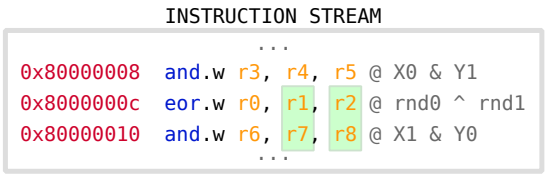


Figure 4.5: Micro-architectural Registers Overwrite–Pipeline Registers: Instruction Rescheduling.

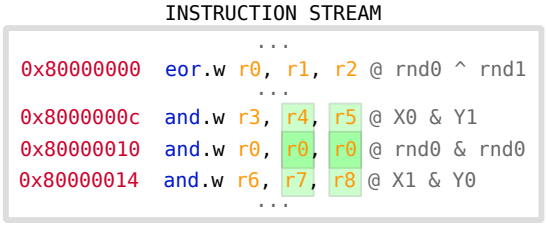


Figure 4.6: Micro-architectural Registers Overwrite–Pipeline Registers: Micro-architectural Registers Flushing.

values `rnd0` and `rnd1`; `r4` and `r7` contain, respectively, the two shares $\mathbf{X}_0$ and $\mathbf{X}_1$ of the variable X; `r5` and `r8` contain, respectively, the two shares $\mathbf{Y}_1$ and $\mathbf{Y}_0$ of the variable Y. Furthermore, we assume that the architecture registers `r0`, `r3` and `r6` contain random values not related to the variables X and Y.

In simple pipelined micro-architectures, the content of inputs registers at the same position (for instance, `r4` and `r7`) might employ the same pipeline registers (e.g., the ones associated to ALU's input operands). As such, when executed *back-to-back* (that is, one after the other), the two `and.w` instructions will generate two leaking transitions: the first one due to `r4` and `r7`, the second one due to `r5` and `r8`.

Since the back-to-back execution of the two instructions represents the root cause of the leaking transition, we can provide a different code organization to remove the leaking transition. Figure 4.5 provides an example of this approach. For the involved architectural registers, we assume the same initial reported for the previous example. In this code snippet, the `eor.w` instruction, handling random values, is scheduled between

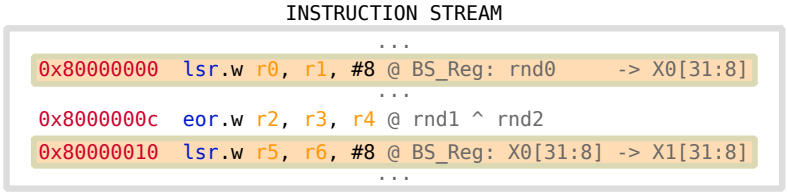


Figure 4.7: Micro-architectural Registers Overwrite–Separated Data-paths.

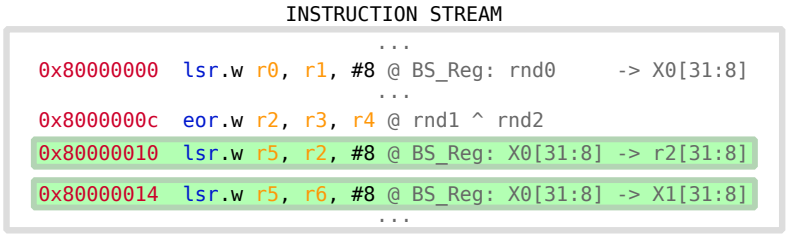


Figure 4.8: Micro-architectural Registers Overwrite–Separated Data-paths: data path flushing.

the two `and.w` instructions. In this example, we assume that changing the position of the `eor.w` (1) does not change the semantics of the program, (2) mitigates the observed transition-based leakages and (3) it does not originate further transitions-based leakages.

In case one cannot satisfy one of these three conditions, we can get rid of the leakage by introducing a *flushing* instruction between the two leaking instructions. Figure 4.6 exemplifies this strategy. Again, we assume the same initial register content as in the example of Figure 4.4. In this snippet, between the two leaking `and.w`, we interleave a third crafted `and.w` instruction, which handles random values. In this particular example, the random values are sequentially stored in register `r0`. Although simple in the concept, this later strategy has to satisfy constraints similar to the first solution: there has to be *at least* one random value available; it must not change the program semantics (that is, the crafted instruction must write the results in an unused register).

In the example of Figure 4.5, we employed an `eor.w` instruction to mitigate the transition-based leakage generated by the `and.w`. As we will show in the next example, under the hood the micro-architecture might employ two different data paths for the two instructions, implying *partial* mitigation of the leakage. Thus, in general, to mitigate the transition-based leakages, we must apply the flushing mechanism employing an instruction using the same data path; conservatively, the very same instruction.

4.2.3 Micro-architectural Register Overwrite—Separated Data-paths

A further example of micro-architecture-induced leaking transition stems from the overwrite of registers hosted on different data paths. Let us assume that shift instructions employ a distinct data path with respect to all the other instructions. This data path hosts a dedicated functional unit, called *barrel-shifter*; furthermore, this unit temporarily saves its output in a dedicated register. Figure 4.7 reports an example: two logical shift right instructions, interleaved by an `eor.w` instruction. We assume the following content for the involved (micro-)architectural registers, before the execution of the reported snippet: `BS_Reg`, `r3` and `r4` contain, respectively, the random values `rnd0`, `rnd1` and `rnd2`; `r1` and `r6` contain, respectively, the two shares $\mathbf{X}_0$ and $\mathbf{X}_1$ of the variable X; `r0`, `r2` and `r5` contain random values unrelated to the variable X. When executed, we observe a transition-based leakage occurring from the `BS_Reg` update, which locates on a data path separated from the one used by the `eor.w` instruction.

As for the pipeline register case, we can mitigate the problem by re-scheduling the code or crafting a flushing instruction. In both cases, the flushing instruction must (1) operate on random inputs and (2) employ (at least) all the data paths of the instruction originating the leakage (Figure 4.8).

4.2.4 Rationale Behind the Mitigation Strategies

With the previous examples, we exemplified potential occurrences of transition-based leakages and the approaches we can employ to mitigate them on a scalar, in-order, potentially pipelined micro-architecture. One might question the correctness of such strategies, where with *correctness* we intend that the strategy mitigates the targeted transition-based leakage. Whereas the mitigation carried out by the register allocation can be easily understood (i.e., use two distinct architectural registers to store two shares $\mathbf{X}_i, \mathbf{X}_j$), the case of mitigation by means of instruction scheduling and flushing requires a bit more of elaboration. Before proving the correctness of these two last strategies, we put forward some concepts for the proof.

(Micro-)Architectural State First of all, we observe that a CPU contains several resources that might preserve a *state* across several clock cycles, such as architectural and micro-architectural registers. We refer to such resources as *memory elements* (or *resources*), which we denote with m. With $[\![m]\!]$, we refer to the *(memory) state* (or *content*) of a memory resource m.

Such memory elements can be visible from the ISA or not; respectively, we define them as *architectural* and *micro-architectural* memory resources. Thus, we can see the *architecture* of a CPU as the set of its architectural memory elements. In the same vein,

we can see a CPU *micro-architecture* as the set of the CPU's micro-architectural memory resources.

Since each memory element carries a state, so does a *set* of memory elements. Thus, we can informally define a CPU's *architectural state* as the ensemble of the states of each memory element in the CPU's architecture. Likewise, we define a CPU's *micro-architectural state* as the set of states of each memory element in the CPU's micro-architecture.

Instructions and (Micro-)Architectural State Modification We observe that, when processed, an instruction modifies a (non-strict) sub-set of the architectural and micro-architectural resources of the CPU. For instance, on the ARM Cortex-M4 CPU, the `Thumb-2` instruction `eor.w` (1) modifies the content of an architectural register to save the result and (2) modifies some micro-architectural registers devoted to temporarily store its input operands or the intermediate results obtained by the processing of these input operands. Thus, we can assign to each instruction i in an ISA a set of architectural and micro-architectural memory resources that it modifies when executed by the CPU.

(Micro-)Architectural Transition-based Leakage A transition-based leakage might stem from *the change of state* of an architectural memory resource (*architecture-induced*), from a micro-architectural memory resource (*micro-architecture-induced*), or from both. We remark that a (micro-)architectural transition-based leakage can be traced back to the processing of a specific instruction i: this leakage originates from the modification of some memory resource which state, in turns, gets modified by an instruction i. For instance, considering the example in Figure 4.7, the micro-architecture-induced leakage stems from the change of state of the barrel-shifter's output register; a change caused by the `lsr.w` instruction at address `0x80000010`.

Flushing: a Generic Approach to Mitigate Transition-based Leakages To mitigate the security degradations implied by transition-based leakages, one needs to forbid the implication of shares in such phenomena. In the rest of this section, otherwise stated, when we refer to a *transition-based leakage*, we imply the involvement of shares masking the same variable.

Recalling that a transition-based leakage originates from a change of state of a memory resource, the generic strategy we employ—the *flushing*—is to cause a change in the (micro-)architectural state by means of a random or constant value. That is, considering a generic memory resource m, two shares $\mathbf{X}_i, \mathbf{X}_j$ and a random or constant value \$, we cause the following *transitions* of state ($\Rightarrow$):

$$[\![m]\!] = \mathbf{X}_i \Rightarrow [\![m]\!] = \$ \Rightarrow [\![m]\!] = \mathbf{X}_j.$$

As such, each transition does not involve the two shares.

In practice, we apply such strategy by means of some sequence of instructions, each with random or constant input operands, which modifies the set of (micro-)architectural memory resources originating the transition-based leakage.

To show that this flushing approach is correct, we need to prove that (1) we modify all the (micro-)architectural memory resources from which the leakage stems and (2) show that the modification by a random or constant value is a sufficient condition to mitigate the transition-based leakage.

As already stated at the beginning of this section, the principle behind flushing applies in general. Yet, in the following elaborations we strictly consider scalar, in-order, potentially pipelined architectures.

Concerning the first point, we observe that when the CPU starts the execution of an instruction i, it modifies the set of (micro-)architectural memory resources assigned to i. Hence, by executing some sequence of instructions modifying (at least) the same set of (micro-)architectural memory resources employed by i, we are sure to have some impact on the transition-based leakages. Note that such instruction sequence must be executed *before* i to prevent all the occurring transition-based leakages.

Concerning the second point, we can prove it by contradiction.

Theorem. *Let P be a $(t+1)$-order masked implementation proven secure at order t in the t-probing security model. Then, P is secure at order t in the t-probing security model extended to transition-based leakages, if we mitigate all the transition-based leakages by flushing the memory resources originating the information leakage.*

Proof. The proof works by contradiction: we mitigate all the transition-based leakages occurring with the execution of P by a random or constant value \$, but P is $(t-k)$ order secure in the t-probing model extended with a transition-based leakage model, where $0 < k < t$.

Let us represent the i-th observation, performed by an attacker, with o_i. Each observation o_i provides information on one of the following elements:

$$\{(\$, \mathbf{X}_i), \mathbf{X}_i, \$\}$$

where $\mathbf{X}_i$ represents a share of a variable X of interest for the attacker, and \$ represents a random or constant value independent on X, such that, given a *strict* subset of shares $\mathbb{S} = \{\mathbf{X}_i\}$:

$$\$ \cup \mathbb{S} \not\sim X.$$

The element $(\$, \mathbf{X}_i)$ represents the information got by a transition-based leakage involving \$ and $\mathbf{X}_i$. Since we mitigate all transition-based leakages by \$, the case $(\mathbf{X}_i, \mathbf{X}_j)$ cannot occur.

By assumption, an attacker can observe $\mathbb{O} = \{o_i\} \wedge |\mathbb{O}| \leq t - k$ to mount a $t - k$ order attack to recover information on X. We remark that each observation o_i provides information on (at most) one share of X; thus, the set of observations $\mathbb{O}$ provides information on (at most) $t - k$ shares. This is a contradiction: recalling that P is secure in the t-probing model, we have that $\mathbb{O} \not\sim X$.

Thus, when flushing the memory resources originating the transition-based leakages, P is secure also in the t-probing model extended to transition-based leakages. □

Equivalence of Flushing and Instruction Scheduling based Mitigation We remark that the mitigation carried out by the instruction scheduler is equivalent to the flushing mechanism. This can be easily understood by observing that, to mitigate the micro-architecture-induced leakages, the scheduler has to select an instruction:

1. which employs those resources leaking information

2. the instruction's input operands do not induce other transition-based leakages

The flushing mechanism needs to satisfy the same requirement. As such, we can see the instruction selected by the scheduler as flushing instruction *already* available in the program to protect; thus, we can apply the same reasoning to prove the correctness of the instruction scheduling based mitigation.

4.3 Compilers and Compilation

In the previous section, we illustrated how register allocation and code re-organization can mitigate transition-based leakages stemming from a CPU (micro-)architecture. In modern compilers, we can find algorithms designed specifically to handle these two tasks. With this section, we overview the general organization of such tools and the process of *compiling* the high-level specification of a program into a low-level one, amenable for the execution on a given target machine. We conclude by describing the code generation module provided by the `LLVM` core libraries, a set of libraries through which we provide an implementation of our automated methodology. Specifically, we describe the enhanced instruction scheduling and register allocation algorithms.

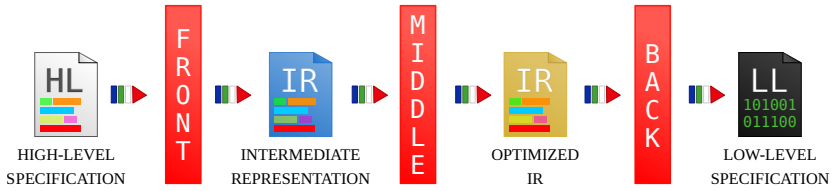


Figure 4.9: Compiler Organization and Compilation Process.

4.3.1 Compiler Organization

In its simplest form, we define a compiler as a tool for translating a program P, written in some high-level language, into a semantically equivalent program P^*, expressed in terms of a low-level language, such that it is amenable for the execution on some execution device.

Modern compilers do not take only the challenge of language translation, but also embed several steps (or *passes*) to *optimize* the translated program. These steps work on an Intermediate Representation (IR) of the original high-level specification. Usually, the IR *lifts* the high-level specification, moving language-specific constructs to abstract ones. In such representation, each program's variables is assigned to a virtual representation of a physical registers, which we call *virtual register*. Working on the IR version of a program has a two-fold advantage: first, it allows the compiler to support the compilation of programs described in different high-level languages; second, it allows to operate architecture-agnostic code-optimizations.

Figure 4.9 provides a high-level overview of the *modular* organisation of a modern compiler. The *Front(-End)* module translates the high-level specification to an IR. The *Middle(-End)* module carries out several target-independent optimizations on the IR. Finally, the *Back(-End)* module translates the IR to an IR closer the low-level language supported by the target machine, and runs target-dependent optimization on this low-level IR. Eventually, the compiler's back-end produces the optimized low-level specification of the input program P.

The back-end module embeds several passes to optimize the IR input. Among these, we find the *code generation* passes. A code generation step first converts the input IR to an instance of the optimization problem they try to solve. Then, it goes through the solving process, until it converges to a solution. We remark that, in most cases, these optimization problems belong to the class of NP-complete problems. As such, most of the code generation algorithms implement heuristic approaches, which provide, in general, sub-optimal solutions.

Among these, typically we find *Register Allocation* and *Instruction Scheduling*. For brevity, we will refer to these two algorithms also as *allocator* and *scheduler*, respec-

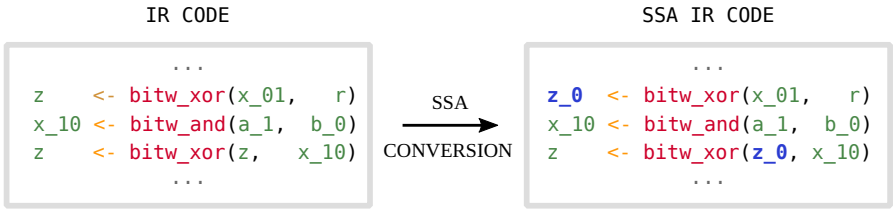


Figure 4.10: Conversion to SSA Form of an Example Code.

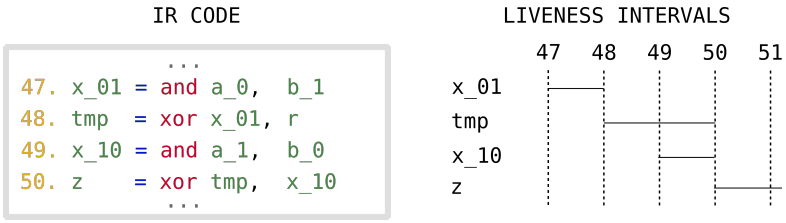


Figure 4.11: Liveness Intervals of Variables in an Example Code.

tively. Before detailing these two algorithms, we overview a particular IR form called Static Single-Assignment (SSA) form, fundamental for the efficiency of several code optimizations.

4.3.2 Static Single Assignment Form

To simplify the application of a large class of code optimizations, modern optimizing compilers rely on a particular form of IR known as SSA form [MP02]. Under an SSA-based IR, each variable definition is unique; in other words, re-definitions of a variable are forbidden. By consequence, each virtual register can be defined only ones. Figure 4.10 reports a simple example of conversion of a piece of IR code from non-SSA to SSA form. As depicted in the left listing, the code snippet defines the variable `z` twice. In this example, when converting the code snippet to SSA form, we change the first definition of `z` to the definition of a new variable `z_0`. To preserve the semantics of the program, we replace all the uses of `z`, prior to its re-definition, with the use of `z_0`. The conversion of code to SSA involves the handling of more complex cases (e.g., loops), whose description we omit as not fundamental for the remainder of the chapter. In the rest of the chapter, we refer to the SSA form only to explain certain simplifications or passages in the text. Otherwise stated, we will always refer to IR code in SSA form.

4.3.3 Register Allocation

In its simplest form, the register allocation pass maps (also *assigns* or *allocates*) the intermediate variables employed by the program to the finite set of architectural physical registers available on the target architecture. In general, a register allocation algorithm works on the concept of *liveness interval* of a variable.

Let us consider an enumeration of all the instruction in a generic program, such that we uniquely identify an instruction by the positive integer number associated. Each of these number identifies a *point* in the program.

Let us consider a variable X and two instructions $i_{\text{def}}, i_{\text{use}}$ defining and using X, respectively, and their enumeration $i_{\text{def}} \mapsto h, i_{\text{use}} \mapsto k$. Informally, the liveness interval of X represents the interval of points $l = [h, k)$ in the program such that i_{use} represents the last use of X. We recall that the we consider the IR input in SSA form. Thus, instructions using X can only appear between i_{def} and i_{use}.

Figure 4.11 reports an example of liveness intervals in an example code. On the left side of the listing, we report an hypothetical enumeration of the example code. On the right of the figure, we report a graphical representation of the liveness intervals for each intermediate variable in the example code. For instance, we associate to the intermediate variable `tmp` the liveness interval $[48, 51)$. In the rest of the chapter, we employ a left-closed, right-opened interval notation to denote a liveness interval.

We say that two variables X and Y *interfere* if and only if their liveness intervals *overlap*; that is, the two variables needs a physical register at the same time. We call this overlap a *liveness interference*. From the example in Figure 4.11, the variables `tmp` and `x_10` interfere as their liveness intervals overlap. Also, given two intervals $l = [i, j), l' = [h, k)$, l precedes l' $(<)$ if and only if $j \leq h$.

With this notion of liveness interval and interference, the register allocation algorithm looks for an allocation of the physical registers which does not raise any liveness interference. Due to the difficulty of the underlying optimization problem [Cha+81], existing algorithms rely on heuristic approaches. These heuristics do not guarantee the identification of a solution free of interferences. In such worst case, they put in place a *spilling* procedure. To spill an already allocated variable means to de-allocate it from its physical register and temporally store it in memory. Such procedure makes a physical register available to another live variable. The problem with spilling is that it implies the insertion of code (1) to write to memory (*spill code*) and (2) to read from memory (*reload code*) before any instruction using the spilled variable. Spill and reload code potentially imply an overhead. Thus, register allocation algorithms usually attempt to minimize the spilling of frequently used variables.

4.3.4 Instruction Scheduling

The instruction scheduling pass modifies the order of the instructions of the IR input, such that the re-organized code minimizes (or maximizes) a given cost metric. The execution time of a piece of code represents a typical example of cost metric the instruction scheduler attempts to minimize. At the same time, the instruction scheduler has to respect some constraints during the re-organization; in particular, it must avoid any re-organization that modifies the semantics of the program. Usually, instruction scheduling algorithms resort on a description of the target micro-architecture, which provides useful information to provide a solution near the optimal one. As an example, when we want to minimize the execution time, the micro-architecture description might provide information about the execution time of each instruction, about the functional units available and the multiple-issue capability of the micro-architecture. Instruction scheduling can take place both *before* and *after* register allocation. In the former case, we talk about Pre-Register-Allocation (Pre-RA) instruction scheduling; in the latter, we talk about Post-Register-Allocation (Post-RA) instruction scheduling. The Pre-RA instruction scheduling works on the IR of the program in SSA form, in which intermediate variables do not have an associated physical register. Hence, we reduce the risk of violating the semantics of the program, since the algorithm needs only to preserve the definition order of each variable. On the other hand, the Post-RA instruction scheduling works on programs with allocated physical registers. As such, it has to preserve the definition order of each physical registers. By consequence, the Pre-RA version has more freedom to re-organize the code. Still, the Post-RA version can reduce the overhead introduced by the register allocator due to spill code.

4.3.5 Code-Generation in `LLVM` Core Libraries

The `LLVM` Core Libraries (from now on, simply `LLVM`) is a set of algorithms and data structures for the development of compilers according to the modular organization described in Section 4.3. In the scope of this thesis, we are interested in the code generation tools provided by `LLVM`; in particular, in the instruction scheduling and register allocation algorithms. Before describing the instruction scheduling and register allocation algorithms we enhance, we overview the code generation infrastructure that `LLVM` provides.

`LLVM` provides a *Target-Independent* code generation infrastructure: the algorithms operate in a target-independent fashion, but rely on an interface to query the infrastructure for target-specific information. As an example, a particular code generation algorithm might require to know whether a given instruction reads or writes to memory. Thanks to the target-independent interface, the algorithm can use a single function to retrieve the information, whatever is the target machine for which we generate the code.

LLVM-IR CODE

```
                ...
%x_01 = i32 and %a_0,  %b_1
%tmp  = i32 xor %x_01, %r
%x_10 = i32 and %a_1,  %b_0
%z    = i32 xor %tmp,  %x_10
                ...
```

DATA-DEPENDECY GRAPH

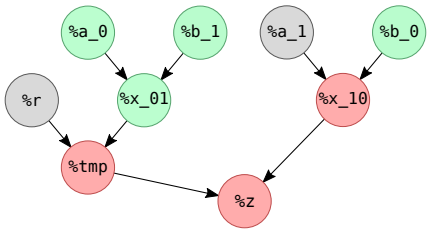


Figure 4.12: (Partial) Data-Dependency Graph of an Example `LLVM-IR` Code.

`LLVM` encodes the target-specific information, stored in Target Description (`.td`) files, by means of a domain-specific language implemented through the `TableGen` language. For the register allocation, dedicated `.td` files encode information concerning the available physical registers, the physical register banks and calling convention related information. For the instruction scheduling, dedicated `.td` files provide information concerning the micro-architecture: number of instructions that can be executed in parallel, misprediction penalty, hosted functional units (with execution latency and whether they are pipelined or not) and presence of forwarding paths. Furthermore, these files maps each instruction i to a subset of micro-architectural resources $\mathbb{R}_i = \mathbb{R}_i^{\text{in}} \cup \mathbb{R}_i^{\text{out}}$, where $\mathbb{R}_i^{\text{in}}$ and $\mathbb{R}_i^{\text{out}}$ contain the resources employed by i's inputs and output, respectively.

The `.td` files are processed and converted into `C++` files, which the target-independent interface queries when asked to.

Now, we describe the instruction scheduling and register allocation algorithms provided by `LLVM`. Specifically, we consider the algorithms `LLVM` version 9.0.1 provides (commit `a10a70238ac`) and on which we implement our automated approach. This version of `LLVM` provides one instruction scheduling algorithm, the Machine Scheduler (MS), and four different register allocation algorithms: the *fast*, the *basic*, the *greedy* and the *PBQP* allocators. Since we primarily aim to evaluate the methodology from a security point-of-view, we decide to enhance the simple allocation strategy which the Basic Register Allocator (BasicRA) implements. We remark that `LLVM` supports *Pre-Register-Allocation* and a *Post-Register-Allocation* instruction scheduling phases, both relying on the same MS algorithm.

Machine Scheduler (MS)

The MS works on a Direct Acyclic Graph (DAG) representation of the input program P. This graph—the Data-Dependency Graph (DDG)—encodes the data-dependencies between variables: a node represents a variable, an edge the data dependency between two variables. Since an instruction computes a variable in the DDG, a node equivalently

Algorithm 2: Machine Scheduler

Input: P, program to schedule
Data: M_μ, micro-architectural state
Output: P^*, scheduled program

```
1  begin
2      Q_Ready ← []; Q_Pending ← []; P* ← [];
3      for i ← P do                                   // Initialize ready queue
4          if inputsReady(i, M_μ) then
5              Q_Ready ← i
6          else
7              P ← i
8      while Q_Ready ≠ [] do
9          c ← pickBest(Q_Ready, M_μ) ;                    // Get best candidate
10         P* ← c ;                                  // Schedule best candidate
11         Q_Pending ← successors(c);
12         for i ← Q_Pending do                    // Release ready instructions
13             if inputsReady(i, M_μ) then
14                 Q_Ready ← i
15             else
16                 Q_Pending ← i
17     return P*;
```

represents an instruction and an edge the data dependency between two instructions.

This algorithm performs a *micro-architecture-accurate* scheduling of the instructions: it simulates and tracks the evolution of the micro-architectural state. With *micro-architectural state*, we refer to the state of each micro-architectural feature, for instance, if the Arithmetic-Logic Unit (the feature) is busy and in how many clock cycles it will be freed (the state). To schedule an instruction, it checks the data dependencies in the DDG and it verifies if the involved micro-architectural features computed the input operands.

The algorithm classifies a DDG node (thus, the variables) as:

Ready : the algorithm scheduled all the parent nodes and they have been computed.

Pending : the algorithm either (1) scheduled at least one parent node, but not all of them, or (2) scheduled all the parent nodes, but some of them are still in computation

Unvisited : the algorithm didn't schedule any of the parent nodes.

Since we can associate a variable to the instruction computing it, we classify instructions in the same way.

Figure 4.12 reports the (partial) DDG of an `LLVM-IR` example code. We report ready, pending and unvisited nodes in green, red and grey, respectively.

Algorithm 2 reports the pseudo-code description of the MS algorithm. Internally, the algorithm tracks *ready* and *pending* instructions by means of the Q_{Ready} and Q_{Pending} queues, respectively. The algorithm tracks the evolution of the micro-architectural state with the map M_μ. This map associates micro-architectural resources to their state. The scheduler gets the set of available resources in the micro-architecture by querying the target-independent code generation interface. At the beginning, the algorithm marks each micro-architectural resource as *free*.

The algorithm resorts on three helper procedures:

`inputsReady` : given an instruction i and a micro-architectural state M_μ, it checks whether all i's input operands have been scheduled and computed or not.

`pickBest` : given a queue of instructions Q and a micro-architectural state M_μ, it returns the best instruction to schedule, according to some internal heuristics. These heuristics take into account the current micro-architectural state M_μ and its potential evolution. The procedure simulates the evolution of M_μ following the micro-architectural description accessible through the Target-Independent interface.

`successors` : given an instruction i, it returns the set of instructions depending on i's output.

At the beginning, the algorithm scans the DDG and populates Q_{Ready} with ready instructions (Line 3–7). We remark that, since working with a queue-based representation of the program P, by dequeuing and re-enqueueing we preserve the original program's order (Line 7). Then, the algorithm starts the re-organization of the input program P. The re-organization happens by selecting the best instruction c from Q_{Ready} (Line 9), which the scheduler enqueues in P^*, the scheduled program (Line 10). Once moved the successors of c to Q_{Pending} (Line 11), the scheduler checks whether any new instruction changed from *pending* to *ready* state and moves them to the ready queue Q_{Ready} (Line 12–16). Eventually, the scheduler returns the scheduled program P^* (Line 17).

Basic Register Allocation (BasicRA)

The BasicRA implements an enhanced version of the *linear scan algorithm* [MP02]. Roughly, the BasicRA prioritizes the allocation of physical registers to liveness intervals with the highest *spill* weight. The spill weight quantifies the impact on the execution

Algorithm 3: Basic Register Allocator (BasicRA)

Input: Q_{LIs}, queue of live intervals
Data: $M_{\text{LI,W}}$, map liveness intervals to spill weights
Q_{PhysRegs}, list of physical registers
Output: $M_{\text{LI,PhysRegs}}$, map live intervals to physical registers

```
1  begin
2      M_PhysReg,LIs ← []; M_LI,PhysRegs ← [];
3      for l ← Q_LIs do
4          L_PhysCands ← [];
5          for p ← Q_PhysRegs do          // Assign PhysReg or track it for spilling
6              if collectInterfs(M_PhysReg,LIs[p],l) = ∅ then
7                  assign(M_PhysReg,LIs, M_LI,PhysRegs, p, l);
8                  break;
9              else L_PhysCands ← p ;
10         if M_LI,PhysRegs[l] ≠ ⊥ then
11             continue ;                                  // Register assigned
12         for c ← L_PhysCands do                      // Spill cheaper interval
13             Q_Evict ← []; canEvict ← true;
14             for l' ← collectInterfs(M_PhysReg,LIs[c],l) do
15                 if M_LI,W[l'] > M_LI,W[l] then
16                     canEvict ← false;
17                     break;
18                 Q_Evict ← l';
19             if canEvict then
20                 for e ← Q_Evict do
21                     Q_LIs ← evict(M_PhysReg,LIs, M_LI,PhysRegs, c, e);
22                     break;
23                 assign(M_PhysReg,LIs, M_LI,PhysRegs, c, l);
24         if M_LI,PhysRegs[l] ≠ ⊥ then
25             continue ;                                  // Register assigned
26         Q_LIs ← spill(l) ;                          // Spill analyzed interval
27     return M_LI,PhysRegs;
```

time when spilling a given variable. In case a liveness interference occurs, the algorithm applies a spilling procedure. For completeness, we remark that this spilling procedure not only inserts spill and reload code, but also *splits* the spilled liveness interval. Indeed, we recall that the BasicRA works on an SSA form of the IR.

To understand why, let us consider a generic liveness interval $l = [i, j)$, where $i < j$. When adding spill code, the allocator terminates the original liveness interval at a point $i < h < j$ in the program, creating a new liveness interval $l_0 = [i, h)$. When inserting

reload code, the allocator defines a new variable at a point $h \leq k < j$, with a new liveness interval associated $l_1 = [k, j)$. As such, spilling l implies splitting it in two new intervals l_0 and l_1. We refer to this operation with the term *splitting*. Once created, the new intervals get the highest possible spill weight. In this way, the allocator will allocate them once visited. Yet, for the sake of simplicity, we just refer to spilling and reloading the original interval l.

Algorithm 3 reports the pseudo-code description of the BasicRA algorithm. Internally, the algorithm employs three maps: $M_{\text{PhysReg,LIs}}$, $M_{\text{LI,PhysRegs}}$ and $M_{\text{LI,W}}$. $M_{\text{PhysReg,LIs}}$ tracks the set of liveness intervals associated to a given physical register. $M_{\text{LI,PhysRegs}}$ tracks the set of physical registers associated to a given liveness interval. $M_{\text{LI,W}}$ associates each liveness interval to its spill weight. At the beginning of the algorithm, the allocator implicitly computes the spill weight for each liveness interval and populates the map $M_{\text{LI,W}}$. Also, the allocator implicitly populates the queue Q_{PhysRegs} of available architectural physical registers on the target architecture by querying the target-independent code generation interface.

The algorithm resorts on four helper procedures:

`assign` : given a physical register p, a liveness interval l and the maps $M_{\text{PhysReg,LIs}}$ and $M_{\text{LI,PhysRegs}}$, it records the assignment of p to l in the two maps.

`collectInterfs` : given the set $M_{\text{PhysReg,LIs}}[p]$ of liveness intervals associated to the physical register p and a liveness interval l, it returns the set of liveness intervals assigned to p and interfering with l.

`spill` : given a liveness interval l, it inserts instructions to spill and reload the interval l. The allocator enqueues in Q_{LIs} the intervals obtained from splitting l.

`evict` : given a physical register p, a liveness interval l and the maps $M_{\text{PhysReg,LIs}}$, $M_{\text{LI,PhysRegs}}$, it removes the allocation of p to l from the two maps. Then, it spills l.

The BasicRA proceeds in three phases. At first, the BasicRA verifies whether there is a physical register p available to the currently analysed liveness interval l (Line 5–9). If it finds such p, it `assigns` p to l (Line 7) and proceeds with the visit of a new interval (Line 10–11). The BasicRA tracks, for later use, the physical registers which do not satisfy the above condition (Line 9).

In case the first phase does not end with an allocation, the BasicRA proceeds with the second phase (Line 13–23). The BasicRA looks for an interval l' assigned to a physical register p and interfering with l (Line 13–17). The algorithm chooses an l' with smaller

spill weight than l (Line 18). If it finds such l', it `evicts` l' (Line 10) and `assigns` l to c (Line 23).

If the assignment succeed, the BasicRA proceeds with the visit of a new interval (Line 24–25). Otherwise, it `spills` l (Line 26).

4.4 Enhancing Code-Generation Modules

In this section, we provide a description of the general approach to enhance code generation algorithms, of the IR annotation to support the enhancement and the enhanced version of the MS and BasicRA algorithms.

4.4.1 General Approach

In Section 4.2, we exemplified how simple strategies can effectively address micro-architecture-induced transition-based leakages, for instance a different assignment of physical registers and a different organization of the instruction order. We can take advantage of the register allocation and instruction scheduling code generation algorithms to automate the leakage detection and the application of these mitigation strategies.

As stated at the beginning of this chapter, our approach enhances existing algorithms with *leakage-awareness*. This awareness materializes as an additional constraint to the optimization problem that the code generation algorithm attempts to solve.

In essence, during the selection of an intermediate solution, the code generation algorithm *also* checks whether the candidate solution induces some leaking transitions or not. In the latter case, the algorithm accepts the candidate solution and moves on in the process. Otherwise, the algorithm looks for another non-leaking candidate intermediate solution.

The additional constraint potentially reduces the set of available intermediate solutions; in the worst case, the solution set empties. Such unfortunate case, would imply the irrevocable interruption of the whole code generation process. To solve the situation, the algorithm carries out a *flush* of the micro-architectural state. This operation acts on the state of the micro-architecture, removing the transition-based leakages that prevents the selection of an intermediate solution. Since we are presenting an ISA-based approach (Section 3.2.2), we develop the flushing procedure as an instruction sequence which acts on the micro-architectural state. In order to select solutions which satisfy the leakage constraint, the code generation algorithm needs to resort on a model of the (micro-)architecture encompassing leakage-related information. In the case of a register allocation algorithm, such model specifies which architectural register leaks according to

a transition-based model. In the case of a instruction scheduler algorithm, the model specifies the micro-architectural features leaking according to a transition-based model.

Among the possible strategies (e.g., act on the whole micro-architectural state), we opt for a *minimal flushing* policy: the code generation algorithm flushes the smallest subset of the micro-architectural state. Such choice is driven by performance reasons: minimizing the part of the state to change, we potentially minimize the number of instructions composing the flush mechanism which, in turns, reduced the performance overhead.

As already stated in Section 4.2, the correctness of the mitigation approaches strictly depends on the micro-architecture. The mechanisms we propose and implement are valid for our scope: scalar, in-order, potentially pipelined architectures.

The last ingredient our methodology needs is a piece of information that the code generation algorithms can employ to identify the occurrence of a transition-based leakage. We provide this information by annotating (or *tagging*) all the intermediate variables in the input IR with *encoding tags*, which we describe in the next section.

4.4.2 Intermediate Value Tagging

As stated at the end of the previous section, to pursue the generation of a leakage-free program (*goal*), the code generation algorithm requires to detect whether a given intermediate solution to the optimization problem (for instance, the assignment of a physical register to a liveness interval) might induce a transition-based leakage or not (*requirement #1*). Moreover, we must ensure that no *false negative* (i.e., an intermediate solution induces an information leakage, although the algorithm does not detect it) can occur (*requirement #2*).

Forbidding false positives (i.e., the algorithm detects an intermediate solution as leaking, although it does not) is not a necessary requirement for our goal: the algorithm would put in place a strategy to avoid a solution which already does not induce any security degradation. In the worst case, it simply increases the cost (e.g., the execution time) of the protected implementation.

To satisfy the first and second requirement, the code generation algorithm must:

1. uniquely identify each share in the masked program
2. uniquely identify each random variable
3. know the set of shares an intermediate variable depends on
4. know the set of random variables an intermediate variable depends on
5. put in relation intermediate variables whose recombination leaks information

In the following paragraphs, we describe the solution we employ to address the three above points.

We remind that we target the protection of software implementations masked via *first-order boolean* masking. Further, without lack of generalization, we assume to deal only with unary and binary functions.

Tracking shares and share dependencies In order to satisfy the first four points, we enrich the IR input with a piece of information that we call *encoding tag*:

Definition 4.1. ***Encoding Tag (ETag).*** *A pair* $\langle e, s \rangle \in \mathbb{N} \times (\mathbb{N} \cup \{\perp\})$*, where* e *uniquely identifies the encoding, and* s *uniquely identifies the share within the given encoding.*

Now, we can describe how we track share dependencies. We associate (*tag*) to each instruction (thus, to each computed intermediate variable V in the program) two sets $\mathbb{E}_V$ and $\mathbb{F}_V$ of ETags. The first set tracks all the input shares and random variables on which a variable depends on. The second set only tracks the input shares and random variables with uniform statistical distribution. Implicitly, a non-empty $\mathbb{F}_V$ indicates that V has a statistical distribution independent on any secret. As we will see later, we require such redundancy in order to avoid false negative cases.

Rule #1 (Input Share Tagging) : for each share $\mathbf{X}_i$:

$$\mathbf{X}_i \mapsto \mathbb{E}_{\mathbf{X}_i} = \mathbb{F}_{\mathbf{X}_i} = \{\langle e_X, i \rangle\}$$

where e_X identifies the encoding of the variable X

Rule #2 (Random Variable Tagging) : for each random variable R:

$$R \mapsto \mathbb{E}_R = \mathbb{F}_R = \{\langle e_R, \perp \rangle\}$$

where e_R represents a brand-new encoding identifier

Rule #3 (Tags Propagation) : for each intermediate variable $V = f(X, Y)$:

$$V \mapsto \mathbb{E}_V = \mathbb{E}_X \cup \mathbb{E}_Y, \mathbb{F}_V = \begin{cases} (\mathbb{F}_X \cup \mathbb{F}_Y) \setminus (\mathbb{F}_X \cap \mathbb{F}_Y) & , f \text{ is the } \texttt{xor} \text{ function,} \\ \varnothing & , \text{ otherwise.} \end{cases}$$

The idea of this rule is to propagate all the dependencies according to the DDG (union of the $\mathbb{E}$ sets), while tracking the random variables that *mask* any statistical dependency on any of the masked variables. In case f is the `xor` function, we remove only the common random variables. Otherwise, we empty the set $\mathbb{F}_V$. This

LLVM-IR CODE

```
...
%x_01 = i32 and %a_0,  %b_1
%tmp  = i32 xor %x_01, %r
%x_10 = i32 and %a_1,  %b_0
%z    = i32 xor %tmp,  %x_10
...
```

DATA-DEPENDECY GRAPH

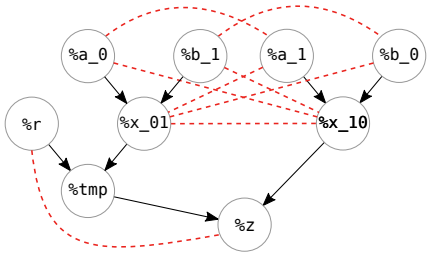


Figure 4.13: Example of Leakage Relation.

choice guarantees to prevent any false negative: indeed, in case V comes from a function f different from the `xor`, we assume its statistical distribution is not secret-independent. Admittedly, it is a conservative choice, as the statistical distribution of V might still be independent on any masked variable. As such, we potentially increase the number of false positive leakages that the code generation algorithms mitigate. Since our main concern is to avoid false negative, we leave as a future work the formalization of a less conservative propagation rule.

Identifying leaking recombination To address the fifth and last point, upon the concept of ETag we define a *leakage relation* between intermediate variables. The code generation algorithms employ this relation to determine whether an intermediate solution induces an information-leaking recombination of two intermediate variables.

Definition 4.2. ***Leakage Relation.*** *Let us consider the set $\mathbb{V}$ of intermediate values appearing in a program P. We define the relation R_{leak} such that:*

$$R_{leak} \triangleq \{(X,Y) \in \mathbb{V}\times\mathbb{V} \mid (((\mathbb{F}_X \setminus (\mathbb{F}_X \cap \mathbb{E}_Y)) \cup (\mathbb{F}_Y \setminus (\mathbb{F}_Y \cap \mathbb{E}_X))) = \varnothing) \wedge \exists\, \langle e_A, s_A\rangle \in \mathbb{E}_X,\ \langle e_B, s_B\rangle \in \mathbb{E}_Y :\ e_A = e_B \wedge s_A \neq s_B\}.$$

In other words, two intermediate variables X and Y leaks whether there is not random variable on which only X or only Y depends *and* X and Y depend on two shares of the same encoding.

Tagging: a SecMult-based example To illustrate the employment of this information and the tagging process, let us consider the example depicted in Figure 4.13. In this example, we report the DDG of a (partial) `LLVM-IR` implementation of the SecMult gadget (Algorithm 1, Section 2.3). When operating on this code snippet, the code generation

algorithm has to avoid the recombination of the following intermediate variables:

$$(\texttt{\%a\_0}, \texttt{\%a\_1}), (\texttt{\%a\_0}, \texttt{\%x\_10}), (\texttt{\%a\_1}, \texttt{\%x\_01})$$
$$(\texttt{\%b\_0}, \texttt{\%b\_1}), (\texttt{\%b\_0}, \texttt{\%x\_01}), (\texttt{\%b\_1}, \texttt{\%x\_10}),$$
$$(\texttt{\%x\_01}, \texttt{\%x\_10}), (\texttt{\%r}, \texttt{\%z})$$

Hence, we can put in *relation* the variables that leak when recombined. The DDG in Figure 4.13 reports this *leakage* relation by additional dashed red lines.

Given $i, j, k \in \mathbb{N} : \ i \neq j \neq k$, we can tag as follows `%a_0`, `%a_1`, `%b_0`, `%b_1` (rule #1) and `%r` (rule #2):

$$\texttt{\%a\_0} \mapsto \mathbb{E}_{\texttt{\%a\_0}} = \mathbb{F}_{\texttt{\%a\_0}} = \{\langle i, 0\rangle\}, \ \texttt{\%a\_1} \mapsto \mathbb{E}_{\texttt{\%a\_1}} = \mathbb{F}_{\texttt{\%a\_1}} = \{\langle i, 1\rangle\},$$
$$\texttt{\%b\_0} \mapsto \mathbb{E}_{\texttt{\%b\_0}} = \mathbb{F}_{\texttt{\%b\_0}} = \{\langle j, 0\rangle\}, \ \texttt{\%b\_1} \mapsto \mathbb{E}_{\texttt{\%b\_1}} = \mathbb{F}_{\texttt{\%b\_1}} = \{\langle j, 1\rangle\},$$
$$\texttt{\%r} \mapsto \mathbb{E}_{\texttt{\%r}} = \mathbb{F}_{\texttt{\%r}} = \{\langle k, \perp\rangle\}.$$

For the variables `%x_01`, `%x_10`, `%tmp` and `%z`, we employ rule #3 as follows:

$$\texttt{\%x\_01} \mapsto \mathbb{E}_{\texttt{\%x\_01}} = \{\langle i, 0\rangle, \langle j, 1\rangle\}, \mathbb{F}_{\texttt{\%x\_01}} = \varnothing$$
$$\texttt{\%x\_10} \mapsto \mathbb{E}_{\texttt{\%x\_01}} = \{\langle i, 1\rangle, \langle j, 0\rangle\}, \mathbb{F}_{\texttt{\%x\_10}} = \varnothing,$$
$$\texttt{\%tmp} \mapsto \mathbb{E}_{\texttt{\%tmp}} = \{\langle k, \perp\rangle, \langle i, 0\rangle, \langle j, 1\rangle\}, \mathbb{F}_{\texttt{\%tmp}} = \{\langle k, \perp\rangle\},$$
$$\texttt{\%z} \mapsto \mathbb{E}_{\texttt{\%z}} = \{\langle k, \perp\rangle, \langle i, 0\rangle, \langle j, 1\rangle, \langle i, 1\rangle, \langle j, 0\rangle\}, \mathbb{F}_{\texttt{\%z}} = \{\langle k, \perp\rangle\}.$$

Figure 4.14 reports the DDG augmented with an hypothetical correct ETag set assignment for each intermediate variable (hence, for each instruction in the IR input).

For the rest of the chapter, we assume that code generation algorithms always consume an IR input annotated with a tagging *preventing false negative detections* of transition-based leakages.

4.4.3 Leakage-aware Machine Scheduler (MS)

In this section, we describe the leakage-aware enhancement of the original MS scheduler (Section 4.3.5). Algorithm 4 reports the pseudo-code of the leakage-aware MS. With respect to the original algorithm, we extend with leakage-related information the map M_μ tracking the evolution of the micro-architectural state. Specifically, for each micro-architectural resource r leaking according to a transition-based model, we also associate the last intermediate variable V that employed this resource.

As in the original algorithm, the scheduler collects this information by querying the target-independent code generation interface. At the beginning, the scheduler marks

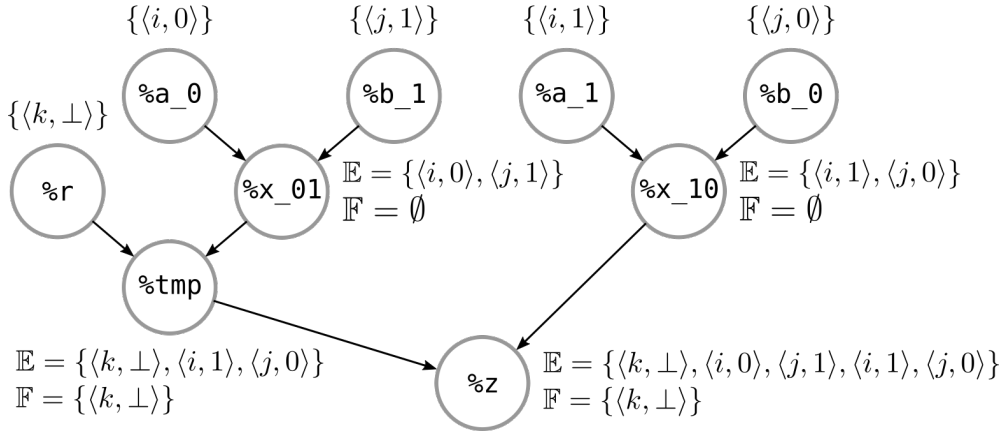


Figure 4.14: Example of ETags Assignment for the SecAnd Gadget.

each micro-architectural resource as available and with no intermediate variable associated. Furthermore, we enhance the scheduler to take into account the precise *path* each instruction's input operand employs, an important aspect to correctly address the micro-architecture-induced leakages.

We extend the set of available procedures with two new ones:

`pickNonLeakingBest` : given a queue Q and a micro-architectural state M_μ, it acts as the original `pickBest` (Section 4.3.5), but it also performs a *leakage detection* check: when choosing a potential candidate instruction i, the procedure queries for the set $\mathbb{R}_i = \mathbb{R}_i^{\text{in}} \cup \mathbb{R}_i^{\text{out}}$ of input and output resources employed by i. Then, denoted with V the output variable and with $\mathbb{I}$ the set of input variables of i, the procedure verifies the following conditions:

$$\forall\ r \in \mathbb{R}_i^{\text{in}},\ \forall\ W \in \mathbb{I} : (M_\mu[r], W) \notin R_{leak}$$
$$\forall\ r \in \mathbb{R}_i^{\text{out}} : (M_\mu[r], V) \notin R_{leak}.$$

`flushUarchState` : given an instruction i causing the leakage and a micro-architectural state M_μ, this procedure mitigates the leakage by flushing the micro-architectural resources employed by i. Specifically, the procedure enqueues in P an instruction f created for the leakage mitigation.

The enhanced version of the MS scheduler extends the original one by first looking for non-leaking candidate instructions with `pickNonLeakingBest` (Line 9). In the case

Algorithm 4: Leakage-Aware Machine Scheduler (MS)

Input: P, program to schedule
Data: M_μ, micro-architectural state
Output: P^*, scheduled program

```
1  begin
2      Q_Ready ← []; Q_Pending ← []; P* ← [];
3      for i ← P do                                   // Initialize ready queue
4          if inputsReady(i, M_μ) then
5              Q_Ready ← i
6          else
7              P ← i
8      while Q_Ready ≠ [] do
           // Get best non-leaking candidate
9          c ← pickNonLeakingBest(Q_Ready, M_μ);
10         if c = ⊥ then                          // Flush micro-architectural state
11             i ← pickBest(Q_Ready, M_μ);
12             P* ← flushUArchState(i, M_μ);
13             Q_Ready ← i;
14             c ← pickNonLeakingBest(Q_Ready, M_μ);
15         P* ← c ;                          // Schedule best non-leaking candidate
16         Q_Pending ← successors(c);
17         for i ← Q_Pending do                       // Release ready instructions
18             if inputsReady(i, M_μ) then
19                 Q_Ready ← i
20             else
21                 Q_Pending ← i
22     return P*;
```

it fails, it flushes the micro-architectural state with `flushUarchState` (Line 11 - 12) before looking again for the best non-leaking candidate (Line 14). As we will state later, the `flushUarchState` procedure guarantees to solve the detected information leakage, allowing the scheduler to find a non-leaking instruction to schedule. Then, it proceeds as in the original scheduler.

We highlight that the, right after flushing the state, we put back in Q_{Read} the instruction i we used to determine which micro-architectural resources to flush (Line 13). We opted for this choice for performance reasons: as we flush the micro-architectural state, some pending instructions might become ready for execution. Scheduling one of them than the instruction i might be more profitable. Thus, we enqueue i back and leave the scheduler choose the best instruction.

Algorithm 5: `flushUarchState` — Leakage-aware MS

Input: i, instruction originating the leakage
M_μ, micro-architectural state
Output: f, instruction flushing the micro-architectural state

```
1 begin
2     f ← cloneInstr(i);
3     replaceInputsWith(f, constOp);
4     updateUarchState(f, M_μ);
5     return f;
```

Concerning `flushUarchState`, we report its pseudo-code in Algorithm 5. It resorts on three helper functions:

`clone` : given an instruction i, it returns a copy of the instruction, encompassing inputs and output operands too.

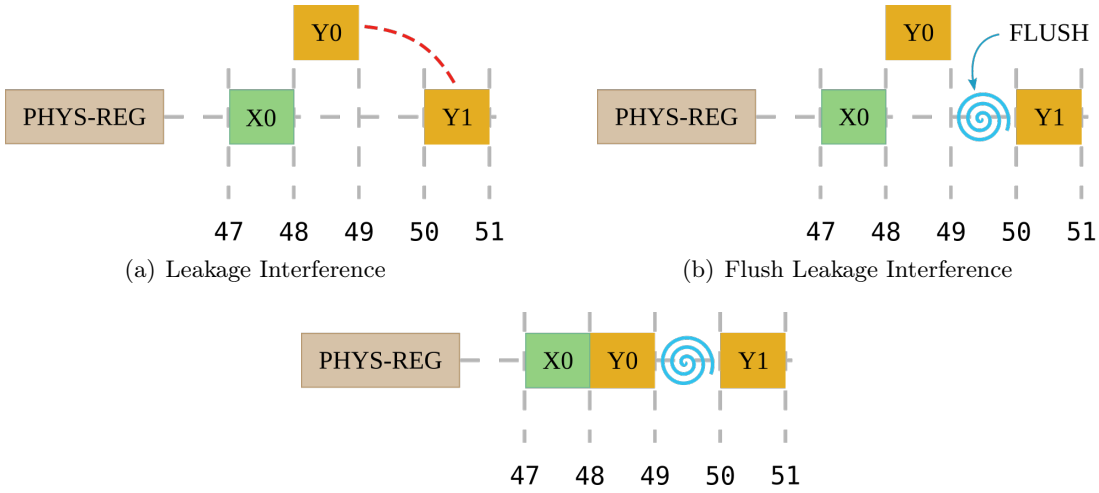
`replaceInputsWith` : given an instruction i and an operand o, it replaces i's input operands with the operand o.

`updateUarchState` : given an instruction f and a micro-architectural state M_μ, it queries the target-independent code generator interface for the set $\mathbb{R}_f = \mathbb{R}_f^{\text{in}} \cup \mathbb{R}_f^{\text{out}}$ of input and output resources employed by f. Denoting with V the output variable and with $\mathbb{I}$ the set of input variables of f, it updates the micro-architectural state as follows:

$$\forall\ r \in \mathbb{R}_f^{\text{in}},\ \forall\ W \in \mathbb{I} : M_\mu[r] = W$$
$$\forall\ r \in \mathbb{R}_f^{\text{out}} : M_\mu[r] = V.$$

Eventually, it also updates the execution state of the resources $r \in \mathbb{R}_f$ (for instance, r is busy for a given number of clock cycles).

When executed, the procedure `clone`s the leaking instruction i, generating the instruction f (Line 2). Then, resorting to `replaceInputsWith`, it replaces the inputs of f with a random operand `rndOp`, assumed available to the program P (Line 3). Eventually, it calls `updateUarchState` (Line 4) and enqueues the instruction f in the program P^* (Line 4). The procedure `flushUarchState` always guarantees to find a candidate instruction to schedule. Indeed, the instruction f, being a copy of the leaking i, will use the same data paths and micro-architectural registers of i, overwriting them with a random value. Thus, the scheduler can, at least, schedule i as next instruction.



(a) Leakage Interference (b) Flush Leakage Interference

(c) Assign Leakage Interference

Figure 4.15: Occurrences of Leakage Interferences between Live Intervals.

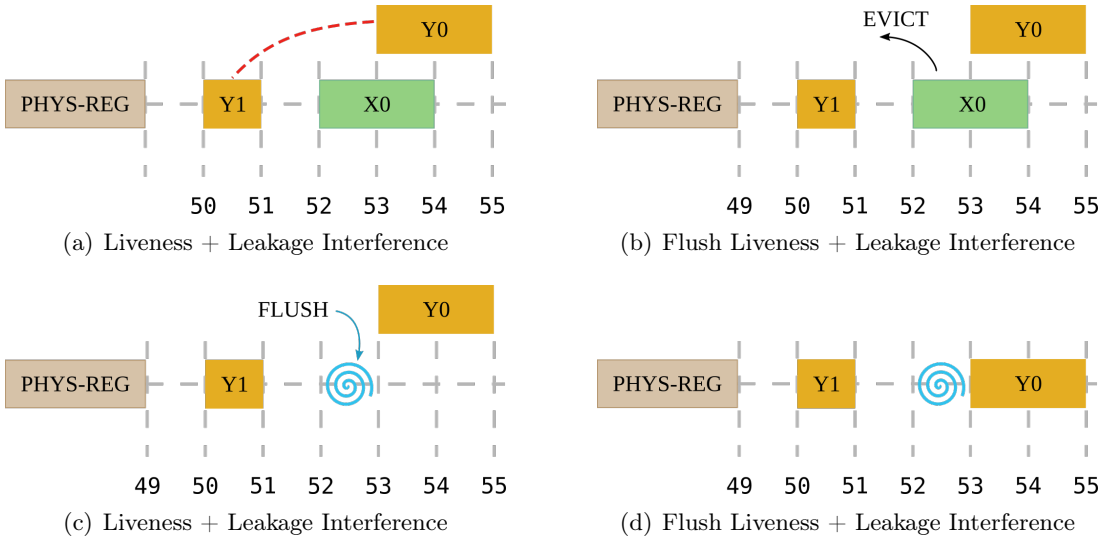
4.4.4 Leakage-aware Register Allocator

In this section, we describe the leakage-aware enhancement of the original BasicRA algorithm (Section 4.3.5). We recall that, since the BasicRA works on an IR in SSA form, we have a one-to-one correspondence between a variable's liveness interval and the instruction defining that variable. Thus, a liveness interval inherits the ETag set associated to the defining instruction. By checking the leakage relation R_{Leak} (Section 4.4.2) on the liveness intervals, the allocator verifies whether a physical register assignment leaks.

According to the approach we described in Section 4.4.1, the allocator assigns physical registers to liveness intervals per the original algorithm specification, but it also checks whether the assignment does not imply a transition-based leakage.

To this end, we add a new type of interferences: the *leakage* interference. Given two liveness intervals l, l' and their associated variable V, W, a leakage interference puts in relation l and l' if and only if $l < l' \wedge (V, W) \in R_{leak}$. We stress the importance of the $l < l'$ condition, which allows the two intervals to be consecutively assigned to the same physical register (Section 4.3.3). As such, their consecutive assignment induces a transition-based leakage. Figure 4.15(a) depicts an example of leakage interference: the allocator attempts to assign the physical register to the liveness interval associated to the share $\mathbf{Y}_0$. Although no liveness interferences occur, the allocator can't perform the assignment, as there is a leakage interference with the $\mathbf{Y}_1$'s interval (red dashed line).

We remark that the two types of interference, liveness and leakage, can occur together. Figure 4.16(a) reports an example of this case: the allocator tries to allocate



(a) Liveness + Leakage Interference (b) Flush Liveness + Leakage Interference

(c) Liveness + Leakage Interference (d) Flush Liveness + Leakage Interference

Figure 4.16: Occurrences of Leakage Interferences between Live Intervals.

$\mathbf{Y}_0$'s interval to the physical register, but both a liveness interference (with $\mathbf{X}_0$'s interval) and a leakage interference (with $\mathbf{Y}_1$'s interval) occur. To provide a leakage-free allocation, it is important that the allocator checks also for leakage interferences when a liveness interference occurs. Otherwise, considering our example, it might evict $\mathbf{X}_0$ and replace it with $\mathbf{Y}_0$, which would induce a transition-based leakage.

When the allocator detects a leakage interference (alone or in conjunction with a liveness interference), it can either *post-pone* or *solve* the leakage. With *post-poning*, we mean that the allocator delays the allocation of l, *optimistically* hoping that the allocation of another interval implicitly solves the detected leakage interference. With *solving*, we mean that the allocator applies a procedure to remove the detected interference leakage, the `flushArchState` procedure (Algorithm 9).

This procedure simply creates a *flush* instruction (by means of the helper function `createInstr`) to overwrite the content of the target physical register p with a constant value `constValue`. We place this instruction before the interval l which would generate the leakage. Figure 4.15(c) shows the application of the `flushArchState`: the flush instruction, represented by a spiral, is placed before $\mathbf{Y}_1$'s interval. From such example, it is easy to see that this procedure always allow the allocation of an interval for which we detected a transition-based leakage.

Actually, when a liveness interference also occurs, the allocator performs first an `eviction`, then it flushes the architectural state. Figure 4.16 reports an example: the allocator `evict` $\mathbf{X}_0$'s interval, assuming $\mathbf{X}_0$'s interval is cheaper (Figure 4.16(b)); then,

Table 4.2: Decisions the Register Allocator takes (top-most row) according to occuring interference (left-most column).

	Evict	Spill	Post-pone	Flush	Evict + Flush
Liveness	✓	✓	-	-	-
Leakage	-	-	✓	✓	-
Liveness + Leakage	-	✓	✓	-	✓

flushes the target physical register (Figure 4.16(c)); eventually, the allocator assigns $\mathbf{Y}_0$'s interval to the physical register (Figure 4.16(b)). In case the eviction is not possible ($\mathbf{X}_0$'s interval more expensive), the allocator spills $\mathbf{Y}_0$'s interval. By spilling $\mathbf{Y}_0$'s interval, we implicitly remove the leakage interference, as this interval won't be assigned to the given physical register. Thus, the allocator does not flush the architectural state.

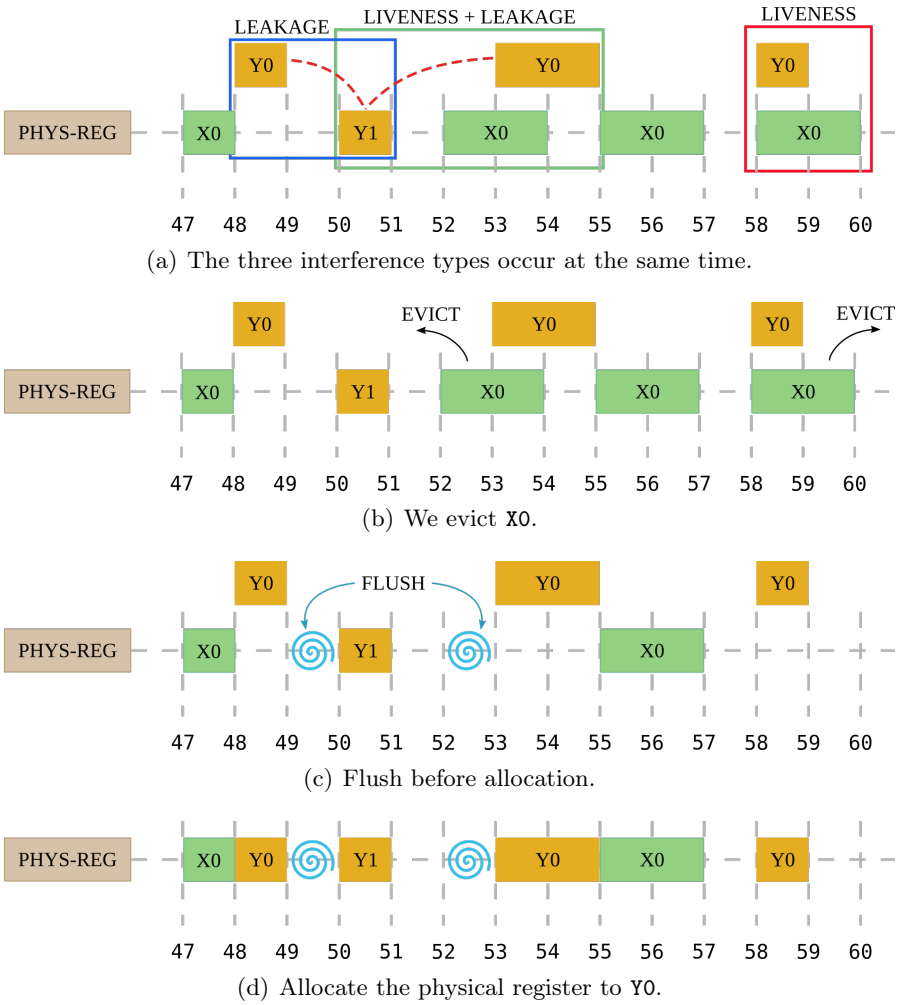
Table 4.2 summarizes the decisions for the leakage-aware register allocation, according to the different interferences.

Before continuing with the description of the enhanced BasicRA algorithm and its helper procedures, we remark an important fact: liveness, leakage and liveness + leakage interferences can occur *at the same time* on a physical register. Figure 4.17(a) reports an example of this simultaneous occurrence. We observe a leakage interference in the interval $[48, 51)$, a leakage + liveness interference in the interval $[50, 55)$ and a liveness interference in the interval $[58, 59)$. To take care of all them, the allocator proceeds as follows, assuming $\mathbf{X}_0$'s intervals are cheaper than $\mathbf{Y}_0$'s intervals: the allocator removes all the liveness interferences by eviction (Figure 4.17(b)); then, it removes all the leakage interferences by flushing the physical register (Figure 4.17(c)); finally, it assigns $\mathbf{Y}_0$'s intervals to the physical register (Figure 4.17(d)). In case at least one among $\mathbf{X}_0$'s intervals has a higher spill weight than one $\mathbf{Y}_0$'s interval, the allocator cannot evict and opts for spilling $\mathbf{Y}_0$'s intervals.

Now, we proceed with the description of the enhanced BasicRA algorithm. The enhanced register allocation composes of a *driver* component (Algorithm 6) and of the `assignPostponeOrFlush` helper procedure (Algorithm 10).

The driver component proceeds in two phases. In the first phase (Line 3–12), the allocator operates as the original algorithm, but postponing the allocation to a liveness interval in case a leakage interference or a liveness + leakage interference occurs. In this phase, if a physical register gets assigned to an interval, the driver prioritises the allocation of postponed intervals (Line 4–7)

In the second phase (Line 13–14), the allocator operates as the original algorithm, but it solves the (liveness +)leakage interferences by (evicting +)flushing the architectural state.



(a) The three interference types occur at the same time.

(b) We evict `X0`.

(c) Flush before allocation.

(d) Allocate the physical register to `Y0`.

Figure 4.17: Handling of liveness interval assignment when the three types of interferences occur at the same time. We assume `Y0` has a higher spill weight than `X0`.

The helper procedure `assignPostponeOrFlush` extends the original allocation strategy (Algorithm 3, Section 4.3.5). Before detailing it, we describe the five new helper functions on which it relies:

`collectLiveInterfs`: given the set $M_{\text{PhysReg,LIs}}[p]$ of liveness intervals associated to the physical register p and a liveness interval l, it returns the set of liveness intervals assigned to p and interfering with l (liveness interference case).

`collectLeakInterfs`: given the set $M_{\text{PhysReg,LIs}}[p]$ of liveness intervals associated to the physical register p and a liveness interval l, it returns the set of liveness intervals

Algorithm 6: Leakage-Aware Basic Register Allocator (Driver)

Input: Q_{LIs}, queue of liveness intervals
Data: $M_{\text{PhysReg,LIs}}$, map physical registers to liveness intervals
$M_{\text{LI,PhysRegs}}$, map liveness intervals to physical registers
$M_{\text{LI,W}}$, map liveness intervals to spill weights
Q_{PhysRegs}, list of physical registers
Output: $M_{\text{LI,PhysRegs}}$, map liveness intervals to physical registers

```
1  begin
2      M_PhysRegs, LIs ← []; M_LI, PhysRegs ← []; Q_Post ← [];
3      while Q_LIs ≠ [] do
4          for l ← Q_Post and M_LI,PhysRegs[l] = ⊥ do          // Handle postponed first
5              assignPostponeOrFlush(l, /*canPostpone=*/true);
6              if M_LI,PhysRegs[l] = ⊥ then
7                  Q_Post ← l;
8          l ← Q_LIs;                                          // Allocate new interval
9          assignPostponeOrFlush(l, /*canPostpone=*/true);
10         if M_LI,PhysRegs[l] = ⊥ then
11             Q_Post ← l;
12     Q_LIs ← Q_Post ;                                        // Copy Q_Post's content
13     while l ← Q_LIs do                                      // Handle lasts postponed
14         assignPostponeOrFlush(l, /*canPostpone=*/false);
15     return M_LI,PhysRegs;
```

assigned to p and interfering with l (leakage interference case).

`collectLiveLeakInterfs`: given the set $M_{\text{PhysReg,LIs}}[p]$ of liveness intervals associated to the physical register p and a liveness interval l, it returns the set of liveness intervals assigned to p and interfering with l (liveness + leakage interference case).

`handleLiveInterfs`: this procedure takes in input a physical register p and a liveness interval l and attempts to evict the liveness intervals l' (assigned to p) for which it detects a liveness interference with l. It extrapolates the very same eviction procedure reported in Algorithm 3 (Line 13–Line 23, Section 4.3.5), for which we already provided a description.

`handleLeakInterfs`: this procedure takes in input a physical register p and a liveness interval l and flushes p. It first collects all the liveness interval l' assigned to p and in leakage interference with l. Then, it determines whether the flushing mechanism should flush p before or after the interval l.

Now, we detail the `assignPostponeOrFlush` procedure (Algorithm 10). For reference, we find the original register allocation logic in the snippets between Line 4–Line 6 (collect

Algorithm 7: `handleLiveInterfs` — Leakage-aware BasicRA.

Input: p, candidate physical register
l, liveness interval to assign
Data: $M_{\text{PhysReg,LIs}}$, map physical registers to liveness intervals
$M_{\text{LI,PhysRegs}}$, map liveness intervals to physical registers
$M_{\text{LI,W}}$, map liveness intervals to spill weights
Q_{LIs}, queue of liveness intervals
Output: **true**, if p available for assignment; **false** otherwise

```
1  begin
2      Q_Evict ← []; canEvict ← true;
3      for l' ← collectLiveInterfs(M_PhysReg,LIs[p],l) do
4          if M_LI,W[l'] > M_LI,W[l] then
5              canEvict ← false;
6              break;
7          Q_Evict ← l';
8      if canEvict then
9          for e ← Q_Evict do                    // Free candidate physical register p
10             Q_LIs ← evict(M_PhysReg,LIs, M_LI,PhysRegs, p, e);
11     return;
```

Algorithm 8: `handleLeakInterfs` — Leakage-aware BasicRA.

Input: p, candidate physical register
l, liveness interval to assign
Data: $M_{\text{PhysReg,LIs}}$, map physical registers to liveness intervals
$M_{\text{LI,PhysRegs}}$, map liveness intervals to physical registers
Output: **true**, if p available for assignment; **false** otherwise

```
1  begin
2      for l' ← collectLeakInterfs(M_PhysReg,LIs[p],l) do
           // Make available candidate physical register p
3          if l < l' then
4              flushArchState(p, l');
5          else
6              flushArchState(p, l);
7      return ;                                  // Register assigned
```

liveness interferences), Line 11–Line 13 (assign free physical register), Line 14–Line 21 (evict cheaper liveness intervals) and Line 37 (spill analysed liveness interval).

The procedure now considers the presence of leakage interferences and the possibility to post-pone liveness intervals. The new allocation strategy develops as follows:

1. Interferences collection (Line 4–Line 13): for each available physical register p in

Algorithm 9: `flushArchState` — Leakage-aware BasicRA

Input: p, a physical register
$l = [i, j)$, a live interval
Data: P, the program under analysis
Output: None

```
1 begin
2     f ← createInstr(move, p, constValue);
3     addAt(P, i, f);
4     return;
```

Q_{PhysRegs}, the allocator collects the liveness intervals assigned to p and interfering with the currently analysed interval l. Specifically, the allocator checks for all the interferences that might happen on p. When a given interference is detected, the allocator stores the register p in a specific queue to track the occurring interference.

2. Assign free physical register (Line 11–Line 13): in case the allocator does not detect any interference, it assigns the analysed interval l to the physical register p.

3. Evict cheaper intervals (Line 14–Line 21): the allocator scans the physical registers on which we have *only* liveness interferences ($Q_{\text{PhysLives}}$), and attempts to evict all the already assigned intervals from one of them. If it succeeds, it assigns the analysed interval l to the freed physical register.

4. Post-pone (Line 22–Line 23): in case the allocator calls this procedure during the driver's first phase, it post-pones the assignment of a physical register if, on all the available physical register, it detects a leakage interference (either alone or in conjunction with a liveness interference).

5. Flush and assign (Line 24–Line 30): if the allocator detected *only* leakage interferences on a physical register (excluding also the ones in conjunction with liveness interferences) it flushes the physical register and assigns it to the analysed interval.

6. Evict, flush and assign (Line 31–Line 36): the allocator detected multiple interferences at the same time on all the available physical registers. We remark that, at this point, the queue $Q_{\text{PhysLeaks}}$ is empty. Thus, we omit it from this snippet. It first attempts the eviction of all the intervals with which there is a liveness interference. If the eviction succeeds, the allocator flushes the leakage interferences and assigns the freed physical register to the analysed interval.

7. Spill (Line 37): if the allocator could not find a free physical register (or it could not free one), it spills the analysed interval l and do not allocate any physical register.

Algorithm 10: `assignPostponeOrFlush` — Leakage-aware BasicRA.

Input: l, liveness interval to assign
canPostpone, boolean variable
Data: $M_{\text{PhysReg,LIs}}$, map physical registers to liveness intervals
$M_{\text{LI,PhysRegs}}$, map liveness intervals to physical registers
$M_{\text{LI,W}}$, map liveness intervals to spill weights
Q_{PhysRegs}, list of physical registers
Q_{LIs}, queue of liveness intervals
Output: None

```
1  begin
       // Queues of liveness intervals
2      Q_Lives ← []; Q_Leaks ← []; Q_LiveLeaks ← [];
       // Queues of physical registers
3      Q_PhysLives ← []; Q_PhysLeaks ← []; Q_PhysLivesLeaks ← [];
4      for p ← Q_PhysRegs do                          // Collect PhysRegs with Interfs
5          if collectLiveInterfs(M_PhysReg,LIs[p],l) ≠ ∅ then
6              Q_PhysLives ← p
7          if collectLeakInterfs(M_PhysReg,LIs[p],l) ≠ ∅ then
8              Q_PhysLeaks ← p
9          if collectLivesLeakInterfs(M_PhysReg,LIs[p],l) ≠ ∅ then
10             Q_PhysLivesLeaks ← p
           // If no interfs, assign
11         if p ∉ (Q_PhysLives ⋃ Q_PhysLeaks ⋃ Q_PhysLivesLeaks) then
12             assign(M_PhysReg,LIs, M_LI,PhysRegs, p, l);
13             return ;                                          // Register assigned
       // Only liveness interfs:  evict cheaper intervals
14     for p ← Q_PhysLives do
15         if p ∈ (Q_PhysLeaks ⋃ Q_PhysLivesLeaks) then
16             Q_PhysLives ← p;
17             continue;
18         isAvailable ← handleLiveInterfs(p, l);
19         if isAvailable then
20             assign(M_PhysReg,LIs, M_LI,PhysRegs, p, l);
21             return ;                                          // Register assigned
22     if (Q_LeakIntfs ⋃ Q_LiveLeakIntfs) ≠ ∅ ∧ canPostpone then
23         return ;                                  // Postpone analyzed interval
       // Only leakage interfs:  flush and assign
24     for p ← Q_PhysLeaks do
25         if p ∈ (Q_PhysLives ⋃ Q_PhysLivesLeaks) then
26             Q_PhysLeaks ← p;
27             continue;
28         handleLeaksInterfs(p, l);
29         assign(M_PhysReg,LIs, M_LI,PhysRegs, p, l);
30         return ;                                          // Register assigned
       // Multiple interfs types:  evict, flush and assign
31     for p ← (Q_PhysLives ⋃ Q_PhysLivesLeaks) do
32         isAvailable ← handleLiveInterfs(p, l);
33         if isAvailable then
34             handleLeaksInterfs(p, l);
35             assign(M_PhysReg,LIs, M_LI,PhysRegs, p, l);
36             return ;                                          // Register assigned
37     Q_LIs ← spill(l);                                 // Spill analyzed interval
38     return;
```

4.5 Implementation Aspects

In this section, we describe the relevant elements characterizing the implementation of our methodology in the `LLVM` libraries.

4.5.1 Intermediate Value Tagging

In Section 4.4.2 we described the principles and the rules for tagging the intermediate variables of a masked program. The tags associated to a variable must flow through the compiler's back-end untouched, in order to reach the enhanced code generation algorithms and enable the leakage-free generation of masked programs.

In the `LLVM-IR`, there is no data-structure explicitly representing the concept of *variable*. On the other hand, as the `LLVM-IR` is in SSA form, there is one, and only one, instruction defining a given variable. Thus, we can equivalently tag the instruction defining a certain variable.

Although `LLVM` supports the association of custom metadata, this can be modified or removed when flowing through the compiler's back-end. As an alternative, we rely on ad hoc *intrinsic functions*. An intrinsic function marks a particular code pattern for which the target architecture might have special hardware support. The `LLVM` libraries do not support out-of-the-box processing of new intrinsics, leaving them untouched throughout the whole compilation process.

For each `LLVM-IR` instruction we target, we introduce an ad hoc intrinsic function to which, in addition to the same input operands of the original instruction, we assign as additional input operands the ETags. Since we work on Boolean-masked implementations, we target the following instructions: bitwise and, eXclusive or, logical shift left and right, bitwise negation. In addition, we consider load and store instructions, which we require in case of memory accesses.

4.5.2 Micro-architectural Model

In the following experimental evaluations (Section 4.6), we use as a target CPU the ARM Cortex-M4. To support our leakage-aware instruction scheduler, we provide it with a leakage-enhanced model of the Cortex-M4 micro-architecture. We encode these leakage-related information within the `.td` files describing the Cortex-M4 micro-architecture (Section 4.3.5), such that we can take advantage of the target-independent code-generator interface provided by `LLVM`.

Specifically, we take as a base the micro-architectural description provided with the `LLVM` version 9.0.1. This description encodes a very basic model of the Cortex-M4 micro-architecture, modelling the whole execution pipeline as a pipelined functional unit to

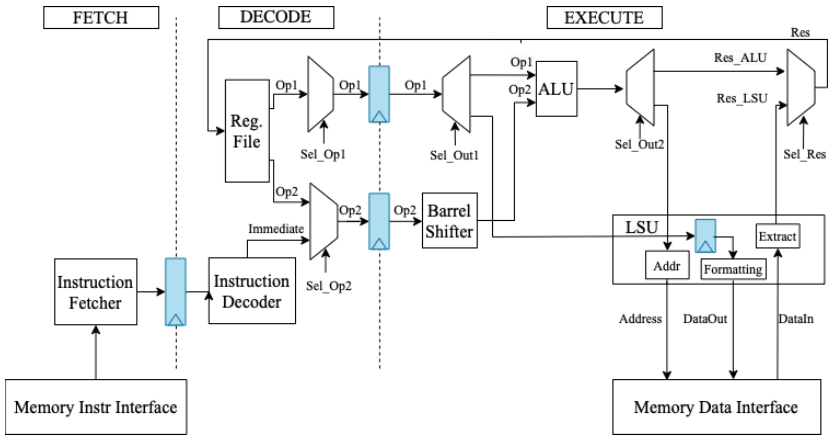


Figure 4.18: Leakage-enhanced Micro-architectural Model for the Cortex-M4.

which every instruction has access. No other function unit finds description. Instruction latencies match the reported public information. Also, the model correctly describe the CPU as single-issue.

This models lack of several details paramount for our methodology. We resort on public information [Bar+21; dHM22; MPW22] to extend it as follows:

- We add the description of the different functional units hosted on the micro-architecture (ALU, barrel-shifter, Load-Store Unit).
- We extend the description of the execution pipeline with information on the micro-architectural registers between each pipeline stage.
- We extend the description of each functional unit with information on the micro-architectural register it hosts (the Load-Store Unit register).
- We map each instruction to the correct set of micro-architectural resources it employs.

Figure 4.18 reports the described leakage-enhanced model, highlighting in blue the micro-architectural registers that induce transition-based leakages.

We recall that the code generation algorithms work on a low-level IR specification of the original program, derived from the original IR in input to the compiler's back-end (Section 4.3). We also remind that the `LLVM` libraries do not support the addition of new intrinsics out-of-the-box (Section 4.5.1), in particular their conversion to equivalent low-level versions. Thus, to enable the actual application of our approach, we need (1) to extend the low-level IR with ad hoc instructions representing the intrinsic functions carrying ETags (Section 4.5.1) and (2) to instruct the compiler's back-end to convert each

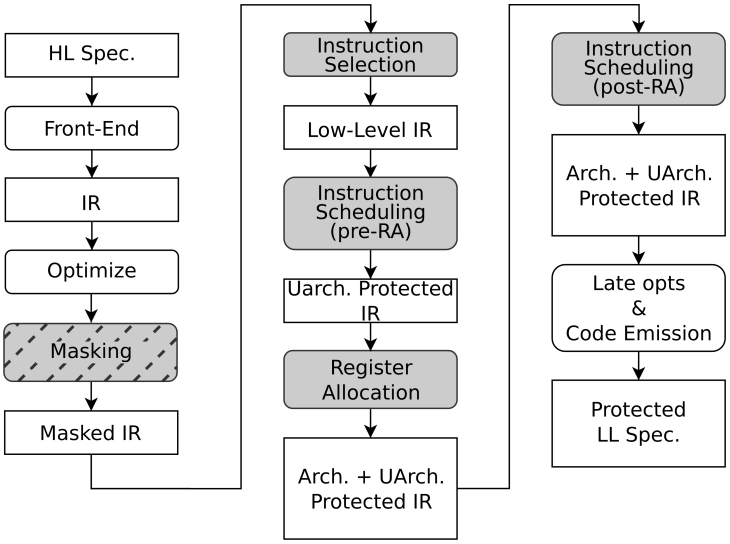


Figure 4.19: Tool-Chain Integration. In grey, we report the modified passes.

ad hoc intrinsic function into their low-level equivalent *and* properly copy the ETags from the former to the latter.

4.5.3 Tool-Chain Integration

We integrate our code generation-based approach to an internally-developed `LLVM`-based compilation tool-chain (based on our same commit), which automate the generation of Boolean-masked software implementations from an unmasked `C` specification. Figure 4.19 reports a schematic of the full compilation tool-chain and the whole compilation and transition-based leakage hardening process.

The tool-chain masks the whole program after application of almost all the optimization passes specified for the `-O3` level of `CLANG`. In particular, the tool-chain applies an aggressive loop unrolling, which provides a fully unrolled IR version of the input program. From these, we exclude optimizations promoting vectorization and conversion of memory operations (for instance, memory copy) to instructions with specific hardware support. Such cases require additional handling at middle-end and back-end level, which we leave as future work.

The application of the Boolean masking consists in replacing each instruction with semantically equivalent, `LLVM-IR`-based, gadgets (`Masking` block, Figure 4.19). Specifically, we implement each gadget by means of the ad hoc `LLVM` intrinsic functions. The middle-end directly inline each gadget implementation. By inlining, the implementation does not pay the cost of several function calls to the gadgets and allows our code generation-based

methodology to fully benefits of the higher degree of instruction parallelism and.

Since `LLVM` does not support the addition of new intrinsics function out-of-the-box, we modified the back-end's instruction-selection phase to lower the ad hoc intrinsics to low-level IR equivalent instructions.

Notice that, right after the post-RA instruction scheduling, the compiler back-end can run further optimizations. Such optimizations might introduce code modifications jeopardizing the work done by the leakage-aware code generation algorithms.

This is the case when targeting the ARM targets: we remarked that the `LLVM`-based tool-chain runs several passes to optimize for code-size. Such optimization attempts to convert the encoding of a `Thumb-2` 32-bit into a semantically equivalent 16-bit variant. In particular, one of them exploits the commutativity of register-tied instructions (instructions which read and write the same architectural register) before attempting the above instruction encoding conversion. Since 32-bit and 16-bit encodings potentially exhibit different leakage patterns [dHM22], and since commuting the input register operands of an instruction potentially induce leaking transitions, we force the code generation phase to emit only `Thumb-2` 32-bit instructions via specific compiler options.

4.5.4 Instruction Scheduling

Here we describe two implementation aspects concerning the flushing mechanism and the instruction scheduling algorithm.

Flushing

As reported in Algorithm 5 (Section 4.4.3), the flushing mechanism resorts on a constant value. We store this value to a fixed stack location at the beginning of the program. When the instruction scheduler needs to flush the micro-architectural state, it first reads the constant value from its stack location.

Post-RA Instruction Scheduling and Flushing

The post-RA instruction scheduling works on an IR version of the implementation which handle *physical* registers instead of virtual ones. By working with physical registers, instructions crafted to flush the micro-architectural state potentially generate an output. Since we must preserve the semantics of the program, we instruct the instruction scheduler to save the result of a flushing instruction in a *dead* physical register (i.e., a register that will be overwritten before being read again).

In case no physical register can be safely used to store the flush result, the instruction scheduler (1) spills an intermediate variable V from its physical register p (taking care

it does not induce leaking transitions with a previous memory instruction), (2) flushes the micro-architecture, (3) saves the output in the register p and (4) reload the spilled variable V in its original local (that is, the register p).

4.6 Experimental Evaluations

In this section, we put forward the experimental procedures carried out to evaluate our automated methodology.

Our analyses develop along two axes: a security analysis and an overhead analysis. For these evaluations, we selected as a use case the SIMON128/128 block cipher, a lightweight cryptosystem developed by the National Security Agency (NSA) [Bea+13]. Starting from a common specification in `C` language, we compile several implementation of the block cipher: unmasked, first-order masked (with and without micro-architectural protection) and second-order masked. In the security evaluation, we assess the information leaked from each of these implementation with the TVLA methodology. In the overhead evaluation, we assess the impact of our approach in terms of execution time, code size and required randomness measuring these metrics from the same masked implementations.

We first overview the employed experimental setup: we describe the side-channel acquisition setup, the analysed SIMON128/128 software implementations, the security and overhead analysis procedures and the employed experimental parameters. We follow with the presentation of the security analysis results first, of the overhead analysis then. A discussion section elaborates on the observed results and we provide a critical comparison with the state of the art. We conclude the chapter with considerations on potential future works.

4.6.1 Experimental Setup

Here, we present the experimental setup employed in our analyses. We first provide an overview of the side-channel acquisition setup and of the target device on which we execute the SIMON128/128 software implementations. Then, we describe the SIMON128/128 software implementations we execute and analyse. We finish with a description of the experimental procedure for both the security and the overhead analysis.

Acquisition Equipment and Target Device

We execute the SIMON128/128 implementations on the STM32F303 micro-controller, hosting an ARM Cortex-M4 CPU.

To reduce the execution time variability across runs of the same code, we fetch code from the Flash, disable the instruction and data cache and set the Flash access latency to 0 clock cycles. We collect power-based side-channel traces via the ChipWhisperer setup, with an acquisition board CW-308 UFO and the CW-1200 oscilloscope [NEWa]. We set the micro-controller clock frequency to 7.384 MHz, and the oscilloscope samples the power consumption at a rate of 29.538 MHz. Hence, 4 samples per clock cycle are measured.

SIMON128/128 Software Implementations

Each SIMON128/128 implementation comes from the compilation of the same *round-reduced* unmasked `C` specification, which follows the official description [Bea+19]. We resort on a round-reduced version, due to limited amount of Flash memory available on our target device. We contextually specify the number of rounds composing the implementation. We carry out the compilation thanks to the tool-chain we described in Section 4.5.3, generating machine-code implementation tuned for the ARM Cortex-M4 micro-architecture. We report and discuss the results concerning 4 distinct implementations: an unmasked implementation; a first-order masked implementation (without micro-architecutral protection); a first-order masked implementation (with micro-architectural protection); a second-order masked implementation (without micro-architectural protection). To distinguish between the two first-order variants, we employ the adjectives *unprotected* and *protected*. Concerning this last implementation, we generate it enabling the leakage-aware variant of both the pre-RA and post-Ra instruction scheduling, as well as of the register allocation.

In order to provide fair comparisons, we enable pre-RA instruction scheduling, register allocation and post-RA instruction scheduling for all the implementations. We control the execution of each leakage-aware variant through a compiler option.

Concerning the masked implementations, we provide them the required randomness via a pointer to a stack location containing a vector filled with fresh random values. We fill this pool of randomness before each invocation of the implementation by means of the PRNG XOROSHIRO** 1.0 [BV21].

Security Evaluation

To evaluate the protection applied by our approach, we resort on the TVLA methodology (Section 2.4.1). For each of the 10-round-reduced implementations, we run a fixed-vs-random plaintext on a set of 75k traces (10k for the unmasked implementation). We measure each trace considering both the key-scheduling and the body of the cipher. Each t-test employs the official SIMON128/128 test-vectors [Bea+13].

Overhead Evaluation

To evaluate the overheads implied by our approach, we measure the following three metrics: *execution time* (in clock cycles), *code size* (in bytes) and *required randomness* (in bytes). In particular, this last metric counts for the randomness required to mask the cipher *and* to mitigate the transition-based leakages.

We provide these metrics for different number of rounds, measuring them for the 10, 15, 20 and 25-round-reduced versions.

We collect the execution time, considering both the key-scheduling and the body of the cipher, by measuring the number of samples for which the oscilloscope's trigger is set high (`trig_count` attribute provided by the class `scope.adc` of the ChipWhisperer software). Dividing by the number of samples per clock cycle, we get the corresponding number of clock cycles. We remark that the execution time metric does not encompass the time spent for the generation of randomness, as it is externally generated and provided as an input to the masked program.

For code size, we measure the size of the executable's `.text` segment. We do not consider also the `.data` and `.bss` segments as they remain unchanged.

Finally, for the number of randomness bytes, we get the measure from an option specific to the employed compilation toolchain (Section 4.5.3).

4.6.2 Security Evaluation Results and Discussion

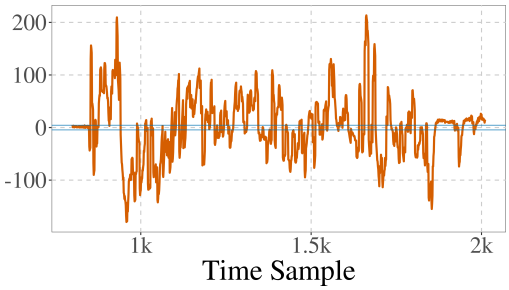
In the following, we present and discuss the results of the security analysis carried out on the 10-round-reduced software implementations of SIMON128/128. The analysis complies with the description provided in Section 4.6.1.

Results

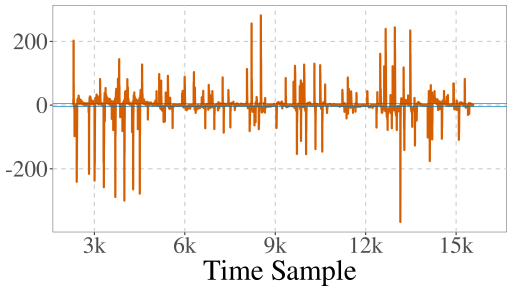
Figure 4.20 reports the results of the leakage assessments on the unmasked (Figure 4.20(a)), unprotected first-order masked (Figure 4.20(b)), protected first-order masked (Figure 4.20(c)) and second-order masked (Figure 4.20(d)).

Since we performed the side-channel acquisition on the whole implementation, also encompassing the unprotected key-scheduling, we remove the first 2.5k samples. We also remove the last 300 samples from the masked implementations, to leave out the epilogue of the program, which handles the unmasked variables.

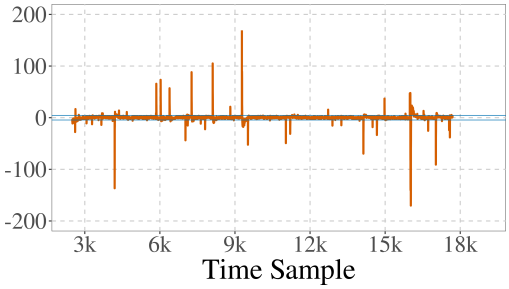
We observe that both the unmasked and unprotected first-order masked implementations leaks consistently along the whole execution. The t-test plot for the protected first-order implementation reports an overall reduction of the information leakage along the side-channel traces. Quantitatively, we witness a ×6 reduction factor (from $6,026$ to



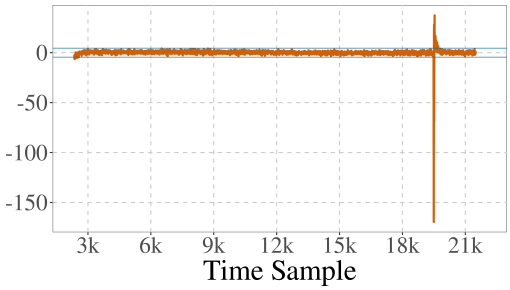
(a) Unmasked



(b) 1-st Order Masked



(c) Pre-RA + Post-RA Instr. Sched. + Reg. Alloc.



(d) 2-nd Order Masked

Figure 4.20: Non-Specific T-test results carried out on the following SIMON128/128 software implementations (10-round-reduced variants): unmasked (Figure 4.20(a)), unprotected 1-st order masked (Figure 4.20(b)), protected 1-st order masked (Figure 4.20(c)) and 2-nd order masked (Figure 4.20(d)).

984) of the occurring leaking samples between the unprotected and protected first-order implementations. The second-order version does not exhibit any leaking samples, except in the window 18k-21k: the leakage originates from a decoding gadget (and subsequent manipulation of the unmasked variables) which the compiler emits before the end of the implementation.

Discussion

Listing 4.1: UB-ST-LD-LD workload

```
<workload_st_ld_ld>:
  str.w  R_src, addr[Z] ; Z -> addr[Z]
  ldr.w  R_dst0, addr[X] ; R_dst0 <- addr[X]
  nop
  nop
  nop
  nop
```

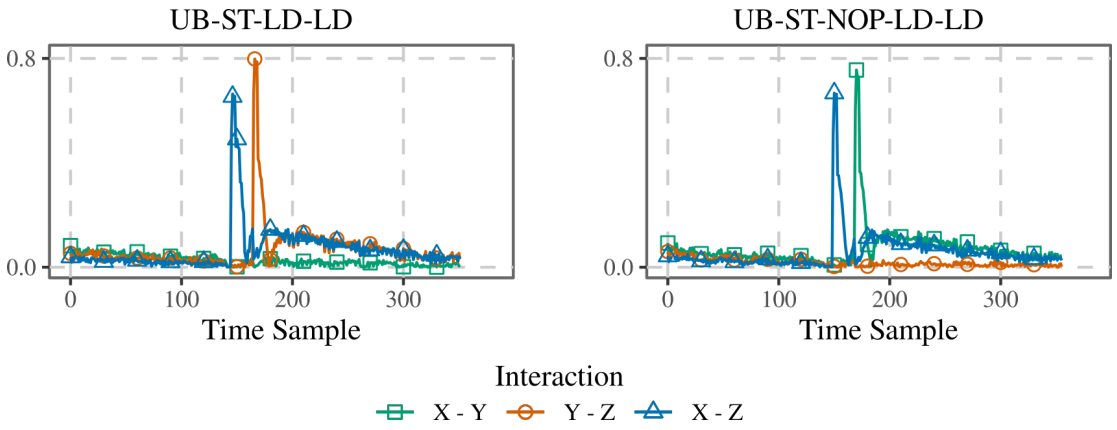


Figure 4.21: Correlation plots for the UB-ST-LD-LD (Listing 4.1, left plot) and UB-ST-NOP-LD-LD (Listing 4.2, right plot) workloads. We ran the correlation analysis on a set of 20k power-based traces, collected from our STM32F303 target.

```
  ldr.w  R_dst1, addr[Y]    ; R_dst1 <- addr[Y]
```

Listing 4.2: UB-ST-NOP-LD-LD workload

```
<workload_st_nop_ld_ld>:
  str.w  R_src, addr[Z] ; Z -> addr[Z]
  nop
  ldr.w  R_dst0, addr[X] ; R_dst0 <- addr[X]
  nop
  nop
  nop
  nop
  ldr.w  R_dst1, addr[Y]    ; R_dst1 <- addr[Y]
```

The observed information leakage affecting the unmasked and unprotected first-order implementations meets the predictions. The results regarding the second-order masked implementation agree with the theoretic expectations, as they predict that a second-order masking provides first-order security in the presence of transition-based leakages, Section 2.3.3). As already pinpointed, the protected first-order masked version addresses a large amount of the leaking points in the unprotected counterpart, although not covering all of them.

This outcome is somewhat counter-intuitive: the leakage-aware post-RA instruction scheduling should address the remaining micro-architecture-induced leakages; for instance, the ones originated by spill and reload code. We find a potential explanation in the *incompleteness* of the leakage-enhanced micro-architectural model the scheduler

employs: while investigating the root causes of the observed leakages, we discovered a novel leakage interaction between memory-related instructions.

In Listing 4.1 we report a minimal code-snippet for which we observed the effect: the variable Y loaded by the second `ldr.w` interacts with the variable Z stored by the `str.w` instruction, although one would expect such iteration to not take place due to the first `ldr.w` loading the variable X. We found out that spacing the memory write and read by 1 clock cycles is enough to get rid of the effect (Listing 4.2). Figure 4.21 reports the correlation plot for these two micro-benchmarks (left plot for Listing 4.1, right plot for Listing 4.2).

A potential explanation stands in hardware-oriented optimization characterizing the memory subsystem: when scheduled back-to-back, the memory subsystem logic might serve first the load, putting the store on hold. Thus, the variable requested by the first load transits on the memory bus, followed by the one handled by the store; eventually, this latter interacts with the variable demanded by the second load, explaining the witnessed transition-based leakage.

We now move to presentation of the overheads results, which will help us to discuss about the overhead implications of our approach.

Table 4.3: Execution time (in clock cycles) of each SIMON128/128 implementation. The figures do not encompass the time spent to generate randomness. We sort table entries by increasing execution time. With "unprotected" and "protected" we refer to the first order masked implementation without and with micro-architectural protection, respectively.

Implem.	10 Rounds	15 Rounds	20 Rounds	25 Rounds
Unmasked	550	815	1,088	1,366
Unprotected	2,474	3,712	5,884	6,255
Protected	4,499	6,902	9,366	13,647
2-nd	5,448	7,908	11,799	16,004

4.6.3 Overheads Evaluation Results and Discussion

With the previous section, we analysed the contributions carried by our automated approach from a security point of view. Now, we present and discuss the results of the overhead analyses of each of the SIMON128/128 software implementation. The analysis complies with the description we provided in Section 4.6.1. To ease the reading, we sort the entries of each table by increasing value of the metric.

Results

We start by presenting the results concerning the execution time. Table 4.3 reports the execution time measurements (in clock cycles) for each implementation. We observe a positive monotone trend along the "rounds" axis, for all the considered implementations. Along the "implementation" axis, the unmasked implementation reports the lowest execution time. The unprotected first-order implementation consistently lower bounds the masked implementations, whereas the second-order one upper-bounds the figures of the other implementations. The protected first-order implementation reports intermediate execution time overhead with respect to the other masked variants.

Table 4.4 reports the size (in bytes) of the `.text` segment for each implementation. We leave out the figures concerning the `.data` and `.bss` segments, as they do not vary across the analysed implementations. We report the very same trend witnessed for the execution time metric: a positive monotone trend along the "rounds" axis; the lowest execution time reported by the unmasked implementation, with the unprotected first-order and the second-order version lower and upper bounds the masked implementations, respectively. Again, the protected first-order one locates in an intermediate position among the other masked variants.

Table 4.5 reports the randomness requirement measurement (in bytes) for each implementation. When applying our approach, we witness an unvaried requirement of

Table 4.4: Segment `.text` size (in bytes) of each SIMON128/128 implementations. We sort table entries by increasing values of segment size. With "unprotected" and "protected" we refer to the first order masked implementation without and with micro-architectural protection, respectively.

Implem.	10 Rounds	15 Rounds	20 Rounds	25 Rounds
Unmasked	9,524	10,360	11,228	12,080
Unprotected	15,676	19,560	23,760	27,644
Protected	22,632	30,544	38,628	46,380
2-nd	25,108	33,388	45,040	51,536

Table 4.5: Number of random bytes of each SIMON128/128 implementations. We sort table entries by increasing values of randomness. With "unprotected" and "protected" we refer to the first order masked implementation without and with micro-architectural protection, respectively.

Implem.	10 Rounds	15 Rounds	20 Rounds	25 Rounds
Unmasked	-	-	-	-
Unprotected Protected	76	116	156	196
2-nd	376	576	776	976

randomness with respect to the unprotected one, whereas the second-order shows a ×5 increase. Again, we report a positive monotone trend along the "round" axis, for all the masked implementations. For the unmasked version, we report no randomness overhead.

Discussion

We first comment the positive monotone trend along the "rounds" axis as expected: as the number of rounds composing the cipher's body increases, so does the number of instructions, which potentially handle randomness bytes. By consequence, execution time, randomness requirements and code size increase as well.

Discussing the execution time and code size overhead along the "implementation" axis, we ascribe the highest impact on the second-order implementation to the intrinsic cost of higher-order masking. Concerning the first-order masked implementations, we first recall that the execution time and code size increase (also) relates to (micro-)architectural state flushing: by flushing, we introduced new instructions to mitigate the transition-based leakages. Thus, we impute the lower overhead affecting the unprotected first-order implementations to the lack of a leakage constraint: the instruction scheduling

and register allocation algorithms have the possibility to provide more performant—but insecure—solutions. Being an implementation encompassing no side-channel countermeasure, as expected, the unmasked implementation reports the lowest execution time and code size overhead.

Concerning the randomness overhead, we observe a null overhead concerning the implementations generated by our methodology. We impute this to the fact that the leakage-aware instruction scheduling and register allocation do not use randomness to mitigate transition-based leakages. The higher randomness requirement of the second-order implementation complies with the employment of a higher-order masking scheme. Clearly, for the unmasked version, no overhead can affect it as it does not rely on any random value.

Eventually, we observe that a fair comparison between the protected first-order masked implementation and the second-order masked one is not possible. Indeed, the former relies on an incomplete model of the Cortex-M4 micro-architecture. Since the execution time and code size increase (also) relates to micro-architectural state flushing, a more complete model might imply a heavier implementation in terms of execution time and code size.

Yet, regarding the randomness overhead, our approach does not rely on fresh randomness to mitigate transition-based leakages. Thus, even considering more precise models of the micro-architecture, we would not pay any randomness cost. This feature advantages our approach with respect to relying on higher-order masking. We recall that, although in our analyses we considered the required randomness to be already ready—thus, nullifying its impact on the execution time—a real masked implementation would generate the required randomness on the fly by means of a PRNG. Either provided as software or hardware modules, an PRNG can provides a limited throughput (bytes of randomness per clock cycle), which can represent the main bottleneck for the masked implementation.

We exemplify the PRNG throughput impact on the execution time considering the overhead figures we collected for our protected first-order and second-order masked implementations.

We consider two potential cases: an ideal one and a real one. In the ideal one, a byte of randomness is ready and accessible at *each* clock cycle (1 cycles/byte). The real one considers an hardware PRNG generating randomness at two different frequency: PRNG and CPU runs at the same frequency (10 cycles/byte) and the PRNG runs slower than the CPU (40 cycles/byte) (reference figures for the LFSR-based PRNG on the STM32F405 [Mic]). The bar chart in Figure 4.22 reports the ratio of the whole execution time of the protected masked implementation and the second order implementation with respect to the unprotected one (25-round-reduced implementations). On top of each bar, we also report the overhead factor.

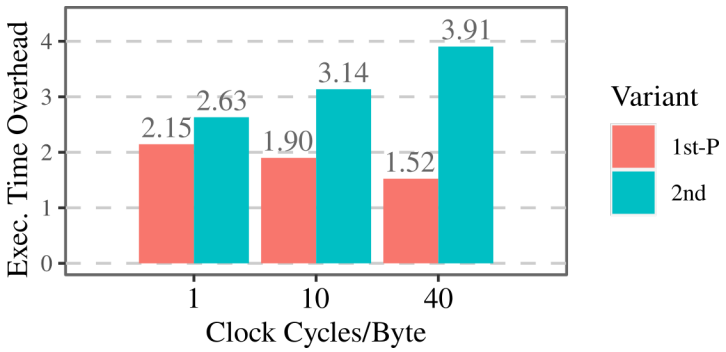


Figure 4.22: Execution time overhead induced by on-the-fly randomness generation. We consider three possible PRNG throughput (1, 10, 40 cycles/byte). On top of each bar, we report the overhead factor with respect to unprotected first-order implementation. With "1st-P" we refer to the protected first-order masked implementation. We consider the 25-round-reduced implementations.

From the chart, we observe how the more limited is the PRNG throughput, the higher it is the cost we pay, in terms of execution time, for each byte of randomness needed. In the ideal case, we already observe that the higher randomness requirement for the second order implementation induces a higher execution time than the protected first order one (×2.15 vs ×2.63) the execution time. In the real cases, already for the 10 cycles/byte case, we observe a pronounced difference between the protected first order and the second order implementations (×1.90 vs ×3.14); in the 40 cycles/byte case, we observe an almost ×4 factor for the second order implementation, where as the protected first order one shows a mild ×1.52 overhead factor. These figures consider a simple, non-cryptographically secure LFSR PRNG, which already provides a good throughput. A recent work of Cassiers et al. shows the potential insecurity on masked implementations of such PRNGs, suggesting to rely on cryptographically secure, but slower constructions [Cas+23]. Thus, the actual impact on a second-order masked implementation potentially worsen.

To conclude, even in the case the employment of a more complete micro-architectural model induces a higher impact on the execution time, the higher randomness requirement of second-order masking plays a major drawback. A drawback that our methodology does not suffer. These considerations suggest that our approach potentially benefits all those applications where software is executed on resource-limited platforms, in particular in terms of PRNG throughput.

4.7 Related Work

In this chapter, we presented a code generation-based methodology to automate the mitigation of micro-architecture-induced transition-based leakages. With respect to most of the micro-architecture-aware ISA-based solutions (Section 3.2.2), our approach operates on the IR specification of the masked program. A first evident advantage is the higher portability of the solution: given a single implementation, we can generate micro-architecture-resilient implementations for any platform whose micro-architectural model is known. A second advantage is the exploitation of the code generation phase: by integrating a leakage constraint in the optimization problems, we potentially allow to generate more performant code while being intrinsically leakage-resilient. With respect to the gadget-based approach proposed by Abromeit et al. [Abr+21], which replace each instruction with a function call to an equivalent, micro-architecture-resilient gadget, our methodology works at a finer granularity. Indeed, given an annotated IR implementation, where also the gadgets implementations are fully-inlined, we can take advantage of the instruction scheduling to provide *interleaved gadget execution*. With respect to the former solution, which serializes the evaluation of the gadgets, the interleaved execution potentially improves the execution time of the whole program. Yet, this comes to the disadvantage of more potential leaking transitions, as the two gadgets, sharing the same hardware resources (that is, the micro-architecture) have more chances to interact. Wang et al. [WSW19] and Tsoupidi et al. [Tso+23] proposed two approaches relying on SMT solvers to find a leakage-free register allocation and instruction scheduling. In contrast, our approach is more *greedy*—we do not look for the absolute optimal—and does not rely on external components, although the resulting solution might be more expensive. Furthermore, these two methods assume to always find a leakage-free solution. On the other hand, we do not rely on this strong assumption, and we instruct code generation algorithms to pro-actively solve the detected leaking transitions.

The work of Seuschek et al. [SSG17] shares a similar concept to our approach. Their approach provides probabilistic convergence to a leakage-free solution. Specifically, when their instruction scheduling algorithm cannot identify a leakage-free re-organization of the code, the whole process aborts. Such probabilistic behaviour potentially requires iterated attempts until a secure solution is found. Furthermore, the approach does not provide any performance guarantees, as it relies on a simplified micro-architectural model which does not take into consideration, for instance, the execution latencies of the instructions.

On the contrary, given a certain input program, our solution has the potential to *deterministically* converges towards a leakage-free solution. Also, relying a more complete model of the micro-architecture, we can better address certain micro-architectural effects while containing the performance impacts.

The existing leakage-simulator-based solutions [She+21b; She+21a] rely on a model of the leakage behaviour of target platform, which comes from a preliminary profiling step. Their mitigation capabilities depends on the completeness of this model which, in turn, strictly depends on the measure side channel and the measurement setup employed during this step. In the general case, our approach suffers of the same model dependency, as certain information concerning the micro-architecture might be classified or simply undocumented. Indeed, the mitigation capabilities strictly depend on the accuracy of the model, as we experienced (Section 4.6.2). Still, the methodology we proposed mitigates the transition-based leakage (1) knowing what shares are manipulated and (2) knowing the micro-architectural state. Thus, the code generation algorithms can consciously generate code that does not leak, while containing the performance overhead. Indeed, during the generation of the code, we avoid as much as possible the introduction of leakages, attempting also to minimize the employment of potentially expensive mechanisms (i.e., flushing) to remove the latter. On the other hand, leakage-simulator-based ones do not possess such information, and have to go through an iterative patching process until a no more leakage is detected. We remark that a local modification of the code might induce further leakages in the surrounding code. As such, this iterative process potentially imply a higher impact on the performances.

4.8 Conclusion

With this chapter, we presented a methodology for the automated generation of first-order masked software resilient against transition-based leakages induced by a CPU micro-architecture. The approach, an ISA-based proactive one according to the classification developed in Chapter 3, operates on an annotated intermediate representation of the masked software. We enhance instruction scheduling and register allocation algorithms to interpret the annotated program in input and automate the generation of masked software implementations resilient against transition-based leakages induced by the executing micro-architecture.

We integrate this methodology as part of an `LLVM`-based compilation tool-chain, which generates first-order boolean-masked implementations from an unmasked `C` input specification. We evaluate our methodology both from a security and a performance perspective. To this end, thanks to the enhanced `LLVM`-based compilation tool-chain, we generate several micro-architecture-protected first-order implementations of the SIMON128/128 cryptosystem, which we compare against an unmasked implementation, a first-order masked implementation and a second-order masked implementation.

From the security analyses, we remark that our implementation already addresses a large part of the leakage impacting the unprotected first-order implementation. Nonethe-

less, at the time of writing, the current implementation of the methodology does not mitigate all the transition-based leakages. Bugs in the implementation and, in particular, the lack of a precise and complete model of the micro-architecture justify this drawback.

Thus, a fair comparison between the protected first-order implementation and the second-order one is not possible. Yet, from the overhead analysis, we highlight an interesting fact: not relying on additional randomness to mitigate the transition-based leakages, we do not suffer of the exponential execution time penalties experienced by higher-order masking.

Concerning the potential future works, we remark that we target primary transition-based leakages. To some extent, our approach could be employed to automate the mitigation of glitch-based leakages. For instance, Gigerl et al. [GPM21] suggest to mitigate glitch-based leakages, generated from the forwarding logic of execution pipelines, by separating the issue of instructions handling shares belonging to the same encoding by a factor dependent on the micro-architecture. The micro-architectural description can encode such information, and the instruction scheduler uses it to check whether an instruction might be issued or not.

The solution proposed in this chapter focuses on first-order masking and scalar, in-order CPUs. Two clear follow ups would be the extension of our approach to higher-order masked implementations and the protection against leakages induced by more complex micro-architectures. e.g., super-scalar ones.

Although unlikely our approach could address all the possible leakage sources generated by the micro-architecture (for instance, glitch-based), we remark that it could be paired with other approaches to reach better security guarantees in practice. For instance, our approach could be easily adapted to take advantage of the hardware-based flushing mechanism of the ISE-based approach proposed by Gao et al. [Gao+20b]. This combination would combine the automated generation of code and minimization of the micro-architectural flushing with a precise flushing mechanism. We leave as a future work the investigation of such combination of approaches.

We remark that our approach natively takes advantage of instruction-level parallelism, thanks to the employment of the instruction scheduling module. As such, implementations relying on programming or masking techniques promoting the concept of parallelism would gain particular advantage; for instance, bit-slicing or threshold implementations. Connected to this last observation, we remark that our approach works on an annotated intermediate representation of the masked program. It is completely independent from any optimization or transformation done before the compiler's back-end. As we did for the `LLVM`-based tool-chain we employed, we could transparently integrate our methodology to any tool for which we support the IR (the `LLVM-IR` in our particular case), as long as the tool generates and annotates the IR as described in Section 4.4.2. Thus, we

could enhance existing tools (for instance, the compiler-based threshold implementation tool CATI [Luo+17]) to investigate the potential security enhancement stemming from the combination. We leave this exploration as an interesting line of work.

Chapter 5

On the Practical Resilience of Masked Software Implementations

In Chapter 3, we put forward the difficulties to provide practically secure masked software implementations and how the currect state of the art faces such challenge. In brief, recombination effects allow an attacker to acquire information on multiple shares through one single observation, degrading the proven security of masking. To bridge the gap between the theoretical and the practical security of masking in software, the recent literature relies on the *suppression* of this information recombination.

An intrinsic limitation of this approach stands in some degree of dependency from the target micro-architecture running the implementation, with several downsides. By proposing an automated methodology to suppress transition-based leakages (Chapter 4), we directly witnessed such limitations; in particular the need of an accurate model of the leaking features of the target micro-architecture.

Yet, in addition to the literature gaps identified in Chapter 3, we highlight two further gaps: (1) the limitation to Boolean masking and (2) the limitation to recombination effects. Concerning the first point, few works explored the theoretical impact of recombination effects on other types of masking schemes, such as the Arithmetic-Sum Masking and the Inner-Product Masking (Chapter 2); to the best of our knowledge, none attempted a comprehensive analysis of the practical impact of micro-architecture-induced leakages on such schemes. Concerning the second point, current works do not take into account the *data-parallelism* intrinsic to modern micro-architectures. Such data-parallelism potentially induces Parallel Processing of Shares (PPS), which an attacker can fruitfully take advantage of.

The literature urges for a more comprehensive analyses of the security degradation of masking schemes when impacted by micro-architecture-induced recombination effects

and data-parallelism.

With this chapter, we take on this challenge. We evaluate the practical security offered by first-order *Boolean*, *arithmetic-sum* and *inner-product* masking against transition-based leakages and PPS-based leakages in software. We develop our study in three main steps:

1. We characterise micro-architectural leakage effects: we carefully handcraft micro-benchmarks to assess the presence of transition-based and PPS-based leakages in software (Section 5.2).

2. We characterise the impact of the observed leakage effects on masking encodings: we quantify the leaked information and investigate its exploitability (Section 5.3, Section 5.4).

3. We characterise the impact of the observed leakage effects on masked implementations: once evaluated leakage impact on the encodings, we assess the practical security of fully masked software implementations (Section 5.5). Specifically, we target as a use-case the AES-128 block-cipher [NIS01].

To provide a comprehensive analysis, we split the security assessment in a first *information leakage assessment*, to analyse the information leaked by the encoding or the fully masked implementation, and in a *information leakage exploitation*, to evaluate the exploitability of such information. In addition, as the design and implementation of the execution platform potentially impacts the observed leakage [MMT20; Aro+21; MPW22], we lead our investigation on two different micro-controllers, an STM32F215 and an STM32F303.

5.1 Preliminaries

In this section, we first introduce a leakage model to describe the leakage behaviour of PPS. Second, we introduce a preprocessing technique to exploit the information intrinsic to the PPS phenomenon. We employ these two elements to analyse PPS phenomenon in the remainder of the chapter.

5.1.1 A Leakage Model for Parallel Processing of Shares

The CMOS technology is still mainstream in digital design, and the overall power consumption of a CMOS-based circuit is the superposition of the power consumptions of its sub-elements [MOP07]. We can describe the induced leakage via the *Sum-of-Hamming-Weights* leakage function:

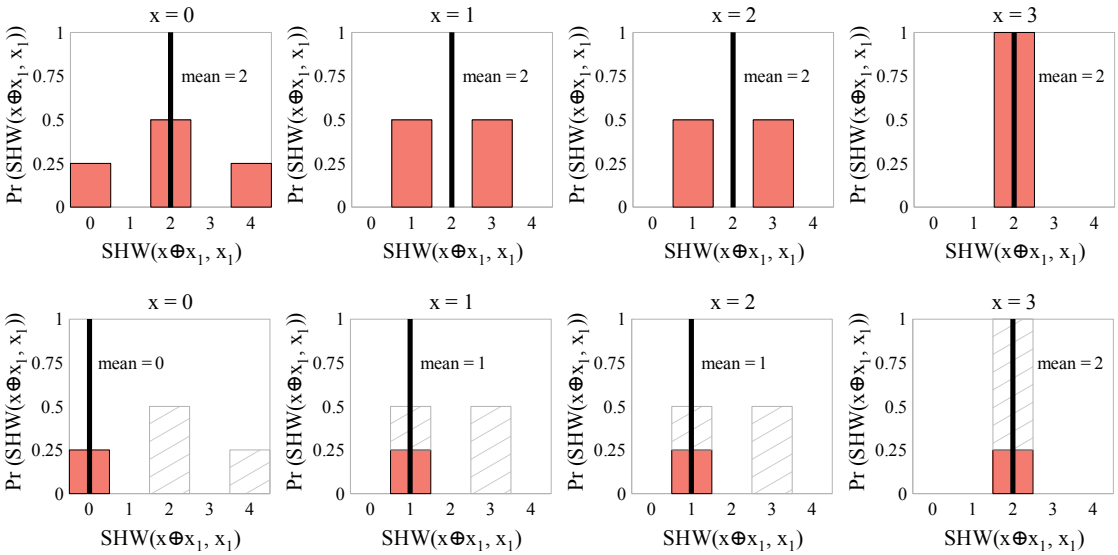


Figure 5.1: SHW distributions obtained for various secret values masked with first-order Boolean masking. x is the secret value, and x_1 a random value used for Boolean masking. Top row: distributions of SHW without preprocessing. Bottom row: distribution obtained when keeping only the lowest k% values ($k = 25\%$ here). While the mean is independent of the secret without preprocessing, it becomes dependent on the secret when only the lowest k% samples are kept.

$$\mathrm{L}(X, Y) = \mathrm{SHW}(X, Y) + \mathcal{N}(0, \sigma). \tag{5.1}$$

Such AGN model assumes as its deterministic component the SHW function:

$$\mathrm{SHW}(X, Y) = \mathrm{HW}(X) + \mathrm{HW}(Y). \tag{5.2}$$

In this paper we use the binary form of the SHW function, which can be readily extended to accept an arbitrary number of arguments.

5.1.2 Biasing Leakage Distributions to Attack Masked Parallel Implementations

The strength behind masking stands in the need, for an attacker, to compute higher-order statistical moments and/or to perform multivariate statistical analyses. When considering hardware masked implementations, security evaluators assume a parallel computation model. Under this computation model, the implementation can treat related shares at the same time sample. Considering a nth-order masking scheme, the attacker, which observes all the $n + 1$ shares of a key-dependent encoded value, needs, at least, the

statistical moment of order $n+1$ to detect any key-dependent information. Moos and Moradi proposed a preprocessing technique to reduce such minimal key-dependent order moment [MM17]. Informally, the technique consists in selecting, for each trace sample, a subset of the measured traces, preserving only certain leakage values. Such Biasing Leakage Distribution (BLD) preprocessing biases the leakage distribution of each trace sample, converting higher-order leakages to lower-order leakages.

To exemplify this technique, let us consider a first-order BM encoding of $X \in \mathcal{X}_{2^2}$. Further, let us assume that the two shares $\mathbf{X}_0, \mathbf{X}_1$ are processed in parallel, and that the implementation leaks according to a noise-free SHW model (Eq. 5.1). Figure 5.1, top row, reports the marginal distributions of each realisation of X. Each marginal distribution exhibits the same first-order moment (e.g., mean). That is, the first-order moment is independent on the encoded value X, as it is expected for a first-order masking scheme. Figure 5.1, bottom row, reports the marginal distributions of the realisations of X after a preprocessing keeping the $k = 25\%$ of samples with the lowest values of the leakage distributions [MM17]. The preprocessed first-order moments of the marginal distributions depend on the secret value, making possible to mount first-order attacks. In practice, the resulting order reduction varies depending on the value of *threshold* k, and on the heuristic used for traces pruning (e.g., keeping the ones with the lowest leakage values) [MM17].

5.2 Parallel Processing of Shares in Software

As our goal is to evaluate the practical security of masked software implementations (Section 5.3, Section 5.4, Section 5.5), we need first to assess the *potential* sources of leakage. To this end, we proceed as follows: we firstly provide a rationale explaining how the complexity of a CPU micro-architecture potentially induces PPS (Section 5.2.1). Then, we describe the three carefully hand-crafted assembler code (called micro-benchmarks, or *UBenches*) that we designed to investigate the presented rationale (Section 5.2.2). To confirm or reject the presence of PPS, we run side-channel analyses on each UBench (Section 5.2.4).

As presented in Section 3.1.2, the micro-architecture of modern CPUs constitutes a rich source of recombination effects; in particular, of transition-based leakages. Hence, we also include a UBench exercising a transition-based leakage originating within the micro-architecture. We have released these micro-benchmarks (C and binary code) as publication artefacts (`https://zenodo.org/record/8094516`).

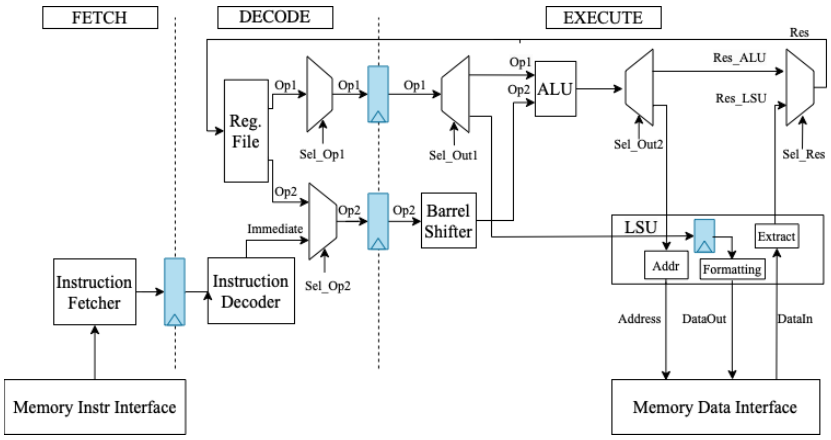


Figure 5.2: Simplified model of a 3-stage, in-order micro-architecture.

5.2.1 Rationale

As explained in Section 3.1.1, the micro-architecture of modern CPUs extensively relies on hardware-oriented techniques to increase the instruction throughput [HP12]. Figure 5.2 reports a simplified 3-stage, in-order, micro-architecture. We recall that the Instruction Fetch (`IF`) stage fetches the next instruction to be executed, the Instruction Decode (`DE`) interprets the instruction (e.g., selecting operands from the Register File), whereas the Instruction Execute (`EXE`) executes the instruction. In such example, we assume that the execution of memory-related instructions (e.g., load and store) requires 2 clock cycles, whereas arithmetic-logic instructions require 1. For memory accesses, the target address is sent to the memory in the first cycle of the `EXE` stage. During the second cycle, the data to be stored is sent to the memory, or the data to be read is received from the memory. The address computation phase employs the `ALU`. To avoid any resource conflict, during the address computation phase, the fetch and decode stages are stalled. Although being quite simple, such model captures the micro-architecture organisation of real micro-controller-graded CPUs (e.g., ARM Cortex-M3 and Cortex-M4 [dHM22; Bar+21]). With such model in mind, it gets easy to understand how PPS can happen in software. Indeed, as mentioned above, each stage takes care of one part of the instruction life cycle: the execution of the `DE` stage happens *in parallel* with the execution of the `EXE` stage. As a consequence, whenever the two stages of the simplified micro-architecture manipulate related shares, the micro-architecture processes shares *in parallel*.

Listing 5.1: Common Structure of Leakage Micro-Benchmarks

```
<ubench_template>:
  <preamble_ubench> ; Prepare UArch state
  <padding>         ; eor with random inputs x4
  bl <trigger_high> ; begin traces collection
  <padding>         ; eor with random inputs x8
  <workload_ubench> ; See Listing 2, 3, 4, 5
  <padding>         ; eor with random inputs x8
  bl <trigger_low>  ; end traces collection
```

5.2.2 Micro-Benchmarks

We design three distinct micro-benchmarks, one for each potential PPS case we identified. Each UBench shares the same structure: a preamble followed by a workload (Listing 5.1). We implement the UBenches in `Thumb-2` assembler, targeting ARM-based target platforms (Section 5.2.3).

The UBench preamble consists in a sequence of machine instructions preparing the architectural and micro-architectural states and the inputs for the workload. The preparation of the micro-architectural state consists in the randomization of the state of specific elements (e.g., micro-architectural registers, memory data-path), which may otherwise induce unintended leakage. The workload consists in a sequence of machine instructions, which attempts to exercise a desired leakage effect. The `trigger_high()` and `trigger_low()` functions, which surround the workload, respectively start and stop the collection of power-based side-channel traces. To clearly identify the workload-induced leakage effect, we pad the workload's beginning and ending with `eor.w` instructions provided with random inputs. To make clear the handling of these values, we comment each UBench instruction with its effect.

Notations We denote the UBench target words as `X0` and `X1`, whereas `rndN` refers to one of the UBench random input values. We denote with `R_val` a generic 32-bit register containing the value `val`. As a special case, we denote with `R_destN` a 32-bit register containing the result of the N-th UBench instruction. We refer to the immediate address of a value `val` with `addr[val]`. We denote a constant value `const` with `#const`.

PPS-related UBench #1 The first PPS-related UBench stimulates the parallel manipulation of bytes when loading a specific one from a given memory address. The preamble crafts a 32-bit word and stores it on the memory stack. Such word contains the least-significant byte (LSB) of both `X0` and `X1`. The workload reads the `X0`'s LSB by accessing to its address. The manipulation of the LSB of each share allows different word's

layouts. Listing 5.2 reports the workload employed in our evaluations, hereafter referred to as UB-SHW-LDRB. We comment the workload with a byte-oriented representation of the word's layout (LSB on the right).

Listing 5.2: UB-SHW-LDRB workload

```
<workload_ubench_shw>:
  ; [R_addr] = [ LSB(X1) | 0 | LSB(X0) |  0       ]
  ; R_dst0  <- [ 0       | 0 | 0       | LSB(X0) ]
  ldrb.w R_dst0, [R_addr, #1]
```

PPS-related UBench #2 The second PPS-related UBench stimulates the parallel manipulation of values during the readings of `X0` and `X1` from the memory and the register file, respectively. Listing 5.3 reports the corresponding workload, hereafter referred to as UB-SHW-LDR-EOR. The `ldr.w` instruction enters the `EXE` stage at clock cycle `#k`. `X0` enters the micro-architecture at clock cycle `#k+1`. Due to the pipeline stall inserted during the address generation, the `eor.w` instruction passes the `DE` stage at clock cycle `#k+1`. During the `DE` stage, the `X1` is read from the register file. As a consequence, at clock cycle `#k+1`, the values `X0` and `X1` are simultaneously alive in the micro-architecture.

Listing 5.3: UB-SHW-LDR-EOR workload

```
<workload_ubench_shw_ldr_eor>:
  ldr.w R_dst0, addr[X0]      ; R_dst0 <- X0
  eor.w R_dst1, R_rnd0, R_X1  ; R_dst1 <- rnd0 ^ X1
```

PPS-related UBench #3 The third PPS-related UBench stimulates the parallel manipulation of values by processing `X0` and `X1`, each handled by a distinct `ALU` instruction. Listing 5.4 reports the corresponding workload, hereafter referred to as UB-SHW-MOV-EOR. The `mov.w` instruction (and thus, its input operand `X0`) enters in the `EXE` stage at clock cycle `#k`. At the same clock cycle, the `eor.w` instruction enters the `DE` stage, where the target value `X1` is read from the register file. As a consequence, the values `X0` and `X1` will be both in the micro-architecture within the same clock cycle `#k`.

Listing 5.4: UB-SHW-MOV-EOR workload

```
<workload_ubench_shw_mov_eor>:
  mov.w R_dst0, R_X0      ; R_dst0 <- X0
  eor.w R_dst1, R_X1, #0  ; R_dst1 <- X1
```

Transition-related UBench This UBench tests the transition-based leakage stemming from the update of the inter-stage pipeline registers. Listing 5.5 reports the corresponding workload, hereafter referred to as UB-HD. At clock cycle `#k`, the first `eor.w` instruction enters the `DE` stage. `X0` is read from the register file and stored in the `DE/EXE` inter-stage register. At clock cycle `#k+1` the second `eor.w` enters the `DE` stage. `X1` is read from the register file, and it is stored in the `DE/EXE` register. The update of the `DE/EXE` potentially causes a transition-based leakage.

Listing 5.5: UB-HD workload

```
<workload_ubench_hd>:
  eor.w R_dst0, R_X0, R_rnd0 ; R_dst0 <- X0 ^ rnd0
  eor.w R_dst1, R_X1, R_rnd1 ; R_dst1 <- X1 ^ rnd1
```

5.2.3 Experimental Setup

In this subsection, we briefly present the experimental setup (target devices, compilation toolchain and side-channel acquisition setup).

We execute the UBenches on the STM32F215 and STM32F303 micro-controllers. The former hosts a ARM Cortex-M3 CPU, whereas the latter a ARM Cortex-M4 CPU.

We compile each UBench with `arm-none-eabi-gcc` version 9.2.1. We tune the compilation with `-Os`, `-mthumb`, and `-mcpu=cortex-m3` and `-mcpu=cortex-m4` for the STM32F215 and the STM32F303, respectively. To minimise execution time variability across runs of the same code, we fetch code from the Flash, disable the instruction and data cache and set the Flash access latency to 0 clock cycles. We collect power-based side-channel traces via the ChipWhisperer setup, with an acquisition board CW-308 UFO and the CW-1200 oscilloscope [NEWa]. We set the micro-controllers clock frequency to 7.384 MHz, and the oscilloscope samples the power consumption at a rate of 29.538 MHz. Hence, 4 samples per clock cycle are measured. The STM32F215 comes with an internal voltage regulator, which we left turned-on and set to 1.2V [NEWb].

5.2.4 Evaluation

For each UBench, we generate two datasets of randomly chosen input values: the *test* dataset and the *control* dataset. Then, for each input dataset, we collect a trace set of $30,000$ power-consumption traces, each of 90 samples. Finally, for *both* the collected traces sets, we compute $\rho(\mathrm{L}(\texttt{X0},\texttt{X1})_{\mathrm{d}}, \mathbf{T}^{i}_{30\mathrm{k}\times 90})$, where $i \in [0, 90)$ and `X0`, `X1` belong to the *test* dataset (i.e., the *control* input dataset is unused). With this procedure, we verify that any correlation stems from `X0` and `X1` manipulation, not from other experimental factors.

The two leftmost columns of Figure 5.3 report the results under the SHW leakage model (Eq. 5.1), whereas the two rightmost columns report the results under the HD leakage model (Eq. 2.4). Except for the UB-SHW-LDRB on the STM32F215, we observe that, when using the proper leakage model (i.e., SHW and HD for PPS-oriented and transition-oriented UBenches, respectively) we observe a higher correlation in the *test* traces, confirming the presence of the targeted leakage effect. When looking for other effects (i.e., transitions in the PPS-oriented UBenches or PPS in transition-oriented UBenches), we do not observe any significant correlation, indicating that the searched effect is negligible. Concerning the UB-SHW-LDRB, as explained in Section 5.2.2, we test all the different word layouts. For the sake of brevity, we only report the results of UB-SHW-LDRB for the layout illustrated in Listing 5.2, but all the other word layouts give similar results.

Finally, we observe lower correlation values for the STM32F215 as compared to the STM32F303. Such difference, potentially stemming from micro-architectural differences and/or the noise generated by the STM32F215's internal regulator (Section 5.2.3), provides us two distinct noise settings for the same leakage model. We will take advantage of this difference to explore the practical resilience of BM, ASM and IPM in different noise settings.

In this section, we have experimentally shown that both transition-based and PPS-based leakages potentially occur in software. In the following section, we employ the developed UBenches to assess the security of masking encodings against transition-based and PPS-based leakages.

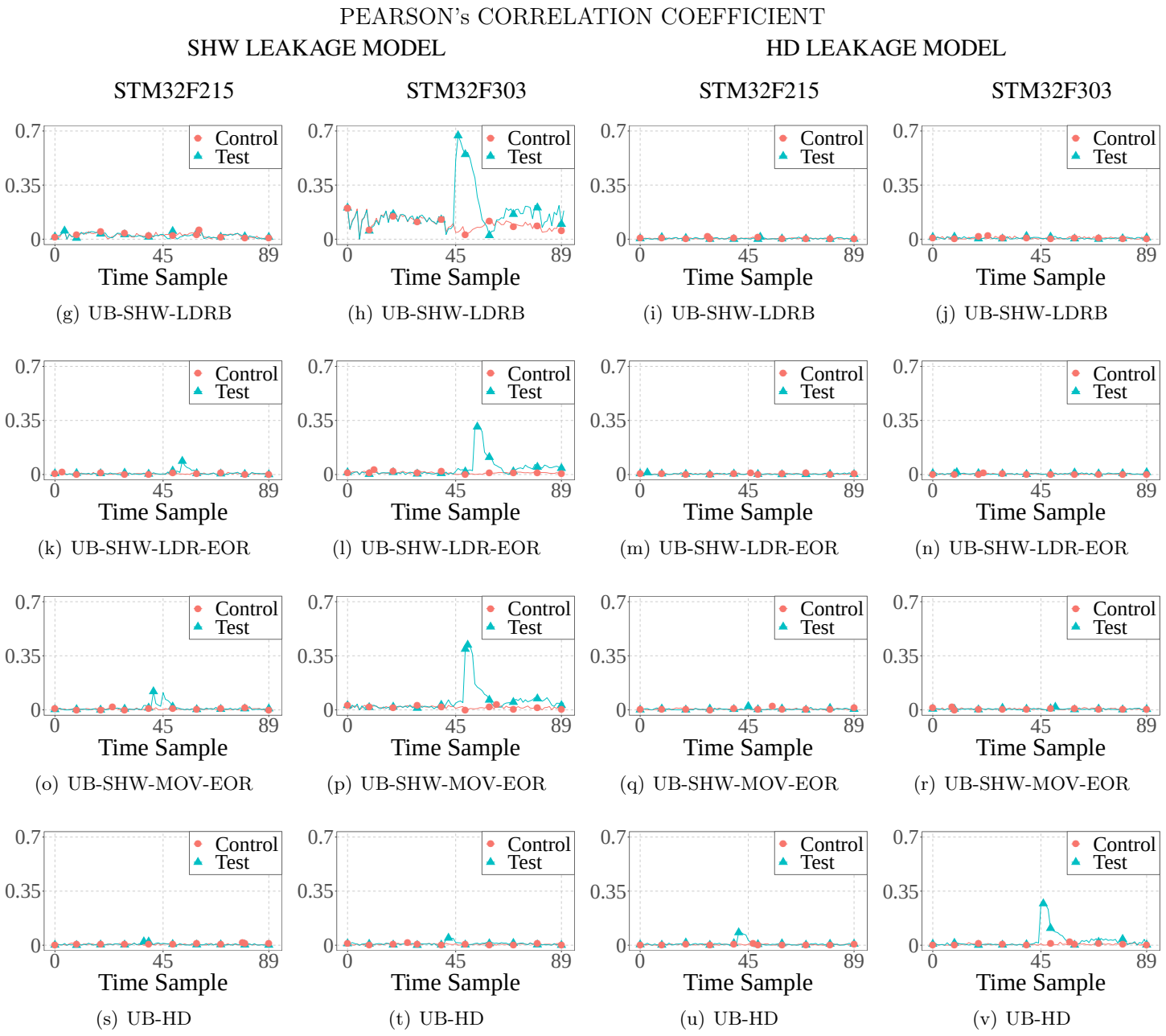


Figure 5.3: PCorrl -based evaluation of PPS-based and transition-based leakages. Each row reports the PCorrl from a different UBench: first row for UB-SHW-LDRB (Listing 5.2), second row for UB-SHW-LDR-EOR (Listing 5.3), third row for UB-SHW-MOV-EOR (Listing 5.4), fourth row for UB-HD (Listing 5.5). The two first columns report the results under the SHW leakage model, and the two last columns under the HD leakage model. The first and third column report the results for the STM32F215 board, whereas the second and fourth ones for the STM32F303 board. Each UBench is evaluated on two sets (*test* and *control*) of 30,000 power-consumption traces.

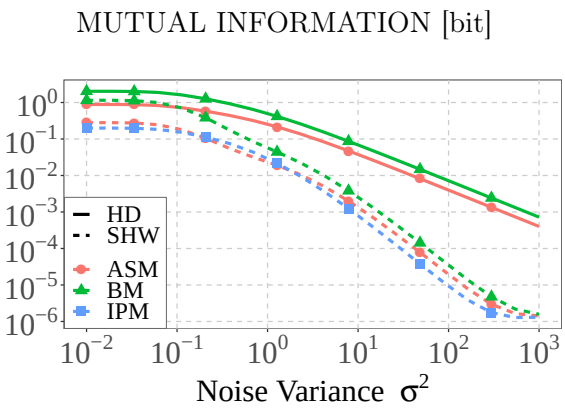


Figure 5.4: Information-Theoretic leakage resilience analyses results. The plot reports the numerically estimated $\mathrm{MI}(X, \mathrm{L}(\mathbf{X}_0, \mathbf{X}_1))$ evolution according to an increasing noise variance σ^2 (both in Log10 scale). We describe the leakage L as an AGN leakage model (Eq. 2.2), where $\mathrm{L}(\cdot)_\mathrm{d}$ = SHW or $\mathrm{L}(\cdot)_\mathrm{d}$ = HD, for PPS-based and transition-based leakages, respectively. Due to estimation errors, for $\sigma^2 \geq 10^2$, the SHW curve diverges from the expected straight line. As IPM reaches perfect independence from X in the HD case, we omit the related curve.

5.3 Evaluation of the Practical Resilience of Masking Encodings

In the previous section, we verified the presence of both transition-based and PPS-based leakages on our two target micro-controllers. This section evaluates the practical resilience of first-order masking encodings against such leakage sources. We develop the evaluation in two settings: an *ideal* one (leakage model and leakage effect match); a *real* one (leakage model and leakage effect potentially differ). For the latter case, we rely on the UBenches designed to assess the presence of PPS and transition-based leakages (Section 5.2.2). We analyse the encodings' resilience in two steps: (a) quantification and comparison of the leaked information (Section 5.3.1); (b) exploitation of the leaked information through first-order analyses (Section 5.3.2).

5.3.1 Theoretical Evaluation

As remarked in Section 5.2.4, the SHW and HD leakage models might not perfectly describe the actual behaviour of our target boards. In order to evaluate the leakage resilience in the case such models capture the leakage behaviour, we firstly conduct an information-theoretic analysis. For such purpose, we numerically estimate $\mathrm{MI}(X, \mathrm{L}(\mathbf{X}_0, \mathbf{X}_1))$, where $X \in \mathcal{X}_{2^4}$ and the shares $\mathbf{X}_0, \mathbf{X}_1 \in \mathcal{X}_{2^4}$ encode X according to BM, ASM or IPM. For IPM, we arbitrarily select $\mathbf{L} = (1, 6) \in \mathbb{F}_{2^4}^2$. We describe the leakage L via an AGN leakage

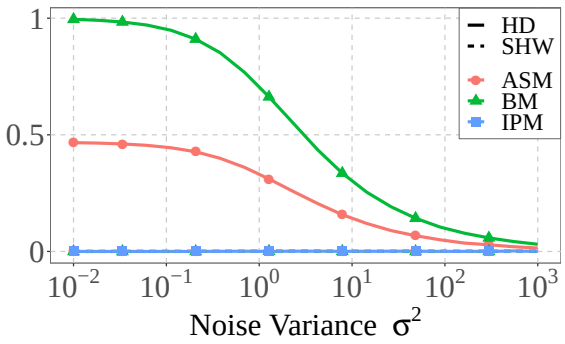


Figure 5.5: PCorrl -based leakage resilience analyses results on simulated traces. The plot reports $\rho(\mathrm{HW}(X), \mathrm{L}(\mathbf{X}_0, \mathbf{X}_1))$ according to an increasing noise variance σ^2 (Log_{10} scale), for the HD and SHW models. We generate $1,000,000$ power-consumption traces, each of 1 sample. We simulate the traces according to an AGN leakage model (Eq. 2.2), where $\mathrm{L}(\cdot)_\mathrm{d}$ = SHW or $\mathrm{L}(\cdot)_\mathrm{d}$ = HD, for PPS-based and transition-based leakages, respectively. The metric does not detect correlation with X under the SHW for BM, ASM and IPM.

model (Eq. 2.2). According to the targeted leakage effect, we employ either $\mathrm{L}(\cdot)_\mathrm{d}$ = SHW or $\mathrm{L}(\cdot)_\mathrm{d}$ = HD.

Figure 5.4 reports the results of the information-theoretic leakage evaluation. We observe that the BM encoding leaks the most, while the IPM one leaks the least. Comparing the information leakage between the two leakage models, the SHW model not only provides the least information quantity, but it decreases faster. This is witnessed by the slope of the curves, as the SHW curve reports a slope of -2, whereas the HD one reports a slope of -1. As reported by Duc et al., such slope reports the minimal statistical moment to break the encoding [DFS19].

We verify this observation by mounting a first-order correlation analysis on simulated power-consumption traces. Specifically, we generate $1,000,000$ traces, each of 1 sample, via an AGN leakage model, and we compute $\rho(\mathrm{HW}(X), \mathrm{L}(X_0, X_1))$, where $X, \mathbf{X}_0, \mathbf{X}_1 \in \mathcal{X}_{2^4}$. Figure 5.5 reports the results of the first-order analyses. As expected, under the HD model, we detect correlation for both the BM and ASM encodings. Consistently with the information-theoretic analysis, we do not detect correlation for the IPM encoding. Concerning the SHW case, the first-order analysis does not identify correlation with the encoded value X. Such evidence illustrates the need of, at least, a second-order statistical moment to correlate with X.

From the information-theoretic analyses, we observed that ASM and IPM encodings tend to better mitigate transition-based and PPS-based leakages. We corroborated such analyses by first-order moment analyses, evaluating the correlation between the encoded

value and the simulated power-consumption. We observed results consistent with the information-theoretic ones. Furthermore, we highlighted how first-order moments cannot detect any information in the presence of PPS-based leakage.

Having obtained an overview of the resilience of first-order BM, ASM and IPM in an ideal setting (i.e., the leakage models perfectly describes the target leakage effect), in the next section we evaluate the resilience of such encodings in a more realistic context.

5.3.2 Experimental Evaluation

In the previous sub-section, we analysed the information-theoretic resilience of first-order BM, ASM and IPM. We completed the analyses with a PCorrl -based evaluation on simulated traces. Such evaluation remarked the better leakage resilience of ASM and IPM encodings. Although the interest provided by an ideal setting (i.e., simulated traces), masked software implementations are executed in an imperfect one, where the leakage behaviour potentially deviates from the hypothetical one. As such, in this section we evaluate the leakage resilience of the three masking schemes when the first-order encodings are manipulated on our two boards, the STM32F215 and STM32F303. For this purpose, we re-use the UBenches of Section 5.2.2, which stimulate PPS-based and transition-based leakages. Differently from the information-theoretic analyses, for IPM we arbitrarily select $\mathbf{L} = (1, 170) \in \mathbb{F}_{2^8}^2$.

For each UBench, we capture $4,000,000$ traces, each of 90 samples. We first quantify the leaked information by computing $\mathrm{HI}(X, \mathbf{T}^i_{4\mathrm{M}\times 90})$. We set the random target inputs `X0`, `X1`, manipulated by each UBench, to the realisation of the shares $\mathbf{X}_0$, $\mathbf{X}_1 \in \mathcal{X}_{2^8}$, in each of the studied masking encodings BM, ASM, and IPM. As stated in Section 2.2.3, the HI provides an upper bound of MI. This property is of particular interest in our case as we want to assess conservatively the amount of leakage. HI also converges towards the true MI as the number of traces gets higher [Bro+19].

The first two columns of Figure 5.6 present the results of the HI analysis for the considered masking encodings and UBench. We compute the HI via the `ENNEMI` Python library [Laa22] which implements a k-nearest-neighbour algorithm. Although the high number of traces and the univariate setting which favours HI convergence, we observe weak information leakage on the STM32F215 for both UB-SHW-LDR-EOR and UB-SHW-LDRB. As shown in Figure 5.3, PPS leakage seems very low on this board, which may explain this result. On the STM32F303, UB-SHW-LDR-EOR and UB-SHW-LDRB show a tiny peak of information, whose significance is uncertain. By contrast, peaks of information are clearly visible for UB-SHW-MOV-EOR and the UB-HD on both boards. As expected, the BM encoding leaks the most information, while leakage is hardly visible for the IPM for the given number of traces.

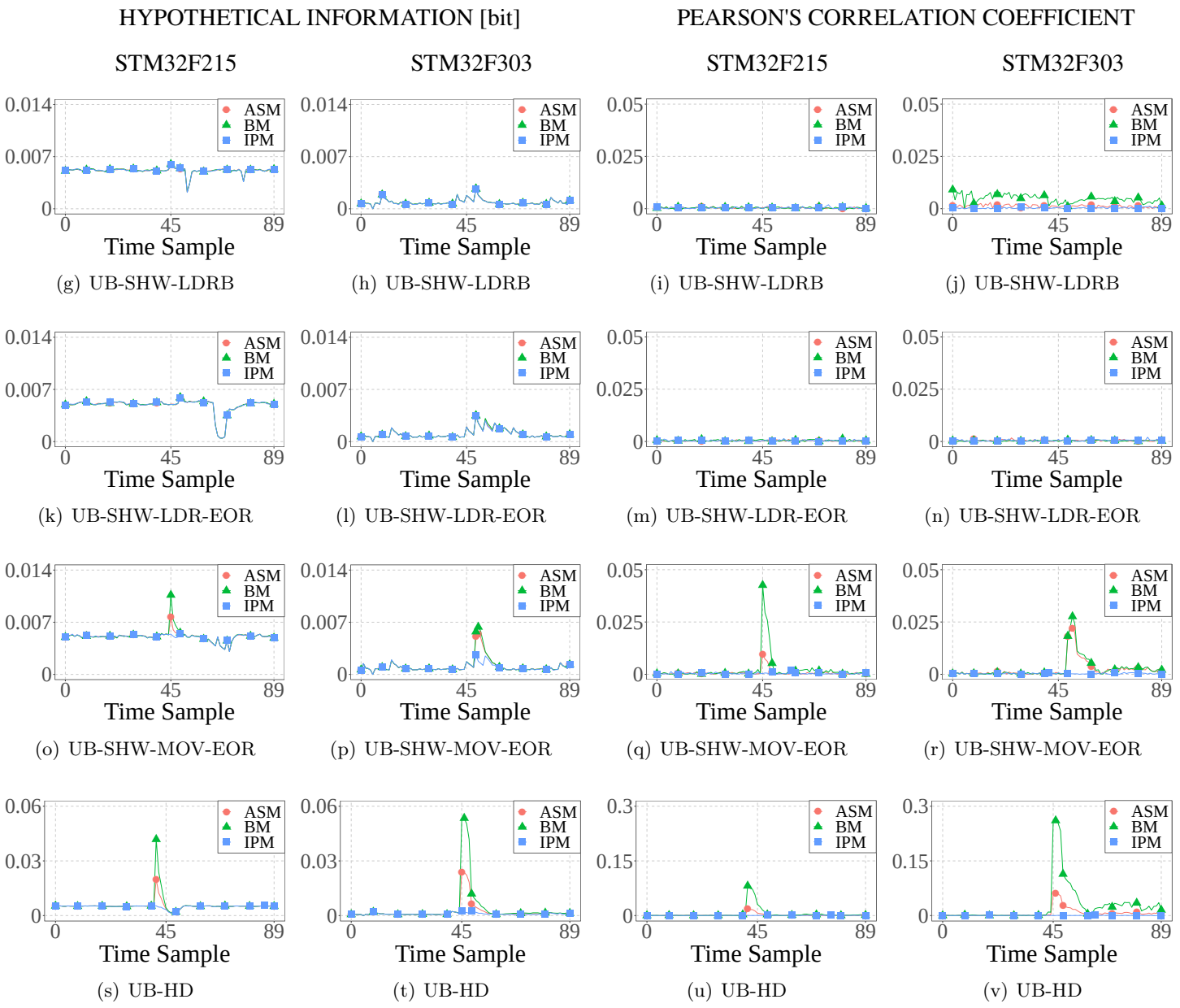


Figure 5.6: Experiment-based quantification of the transition-based and PPS-based leakages. Each row reports the PCorrl from a different UBench: first row for UB-SHW-LDRB (Listing 5.2), second row for UB-SHW-LDR-EOR (Listing 5.3), third row for UB-SHW-MOV-EOR (Listing 5.4), fourth row for UB-HD (Listing 5.5). The first two columns report the HI metric, whereas the last two report the PCorrl metric. The first and third column reports the results for the STM32F215 board, whereas the second and fourth one for the STM32F303 board. For each UBench and board, we compute the PCorrl on a $4,000,000$ power-consumption trace set.

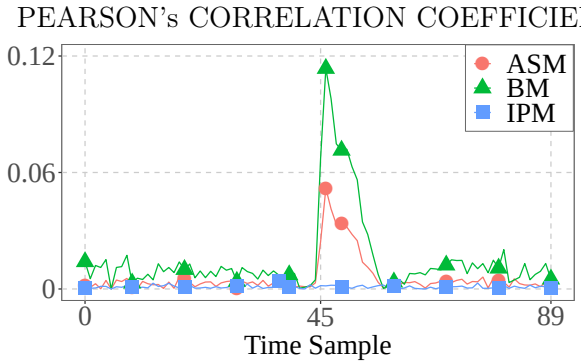


Figure 5.7: Evaluation of the BLD approach (Section 5.1.2). We collect $4,000,000$ power-consumption traces and apply the BLD approach for $k = 10$. We compute the PCorrl by means of the HW model. We collect the traces during the execution of the UB-SHW-LDRB (Listing 5.2) on the STM32F303 board.

For completeness, we also run first-order moment analyses on the same traces sets. Specifically, we compute $\rho(\mathrm{HW}(X), \mathbf{T}^i_{4\mathrm{M}\times 90})$ where $i \in [0, 90)$. The last two columns of Figure 5.6 report the results.

Unexpectedly, we observe a correlation peak for the UB-SHW-MOV-EOR. As explained in Section 5.3.1, a first-order moment cannot detect correlation with an encoded value via PPS-based leakage. Still, the peak takes place at the same time sample where we verified the presence of PPS-based leakage (Section 5.3). Hence, we ascribe the observed correlation to a recombination effect that occurs *simultaneously* with the PPS event.

Up to now, we evaluated the leakage resilience of different masking encodings against transition-based and PPS-based leakages. Concerning transition-based leakages, the results highlight the better leakage resilience of ASM and IPM encodings. Concerning the PPS-based ones, although the use of $4,000,000$ traces, the HI-based analyses hardly identify any PPS-based information leakage. Nonetheless, a different approach e.g., use of the BLD preprocessing [MM17], could better take advantage of the existing information leakage.

With this last remark, we employ the BLD preprocessing proposed by Moos and Moradi [MM17]. Their approach takes advantage of the PPS, converting higher-order leakages into lower-order ones, reducing the security order of the encoding (Section 5.1.2). We directly focus on experimental analyses, as simulation-based ones are extensively provided in the original work [MM17]. Due to its high correlation with the PPS-based leakage (Figure 5.3), we limit our analysis to the trace set collected with the UB-SHW-LDRB execution on the STM32F303. From experimental attempts, we identified $k = 10\%$

(i.e., 400,000 traces per sample) as a good threshold. Figure 5.7 provides the correlation curves from the BLD-based analyses. This time, we detect correlation peaks for both BM and ASM encodings, confirming the potential exploitability of PPS-based leakage.

This section has shown that transition-based and PPS-based leakages represent a concrete vulnerability in software masking implementations, leaking exploitable information through simple first-order analyses. Among the selected candidates, the IPM was found to be the least vulnerable, preventing even the exploitation of higher-order leakages by means of the BLD approach. Yet, such approach relies on the HW model's distribution of the encoded value X, independently of the targeted masking scheme and leakage source. In reality, the distribution of HD and SHW model changes with the masking encoding. In the following section, we take advantage of this observation to break all the evaluated software masked implementations of AES with first-order analysis.

5.4 Exploitation of Leakage Model Distribution in Improved Correlation Attacks

In the previous section, we have evaluated the resilience of BM, ASM and IPM first-order encodings, remarking the better leakage resilience of ASM and IPM ones. This result stems from the consistent employment of the HW to model the leakage of the encoded variable X. In general, such model provides low discrimination capabilities when targeting recombination effects as transitions. For instance, given a first-order IPM encoding of an arbitrary X, $\mathrm{HD}(\mathbf{X}_0, \mathbf{X}_1) \neq \mathrm{HW}(X)$. The same observation holds for PPS-based leakages. In this section, we take advantage of the above remark to enhance the practical security investigation of masking encodings. We proceed as follow: we first elaborate on the unsuitability of the HW model when targeting transition-based and PPS-based leakages; we discuss how to exploit the leakage model's distribution to build more efficient ones. Then, we put in practice the developed models, mounting first-order analyses and compare the new security results with the previous ones.

5.4.1 Rationale

When targeting leakages involving multiple shares, generally the HW model provides low discrimination capabilities. Considering the case of transitions and PPS-based leakages, the HD and SHW distributions are different from the HW's (Eq. 5.3, Eq. 5.4).

$$\mathcal{D}_{(\mathrm{HD}(\mathbf{X}_0,\mathbf{X}_1),X)} \neq \mathcal{D}_{(\mathrm{HW}(X),X)} \tag{5.3}$$

$$\mathcal{D}_{(\mathrm{SHW}(\mathbf{X}_0,\mathbf{X}_1),X)} \neq \mathcal{D}_{(\mathrm{HW}(X),X)} \tag{5.4}$$

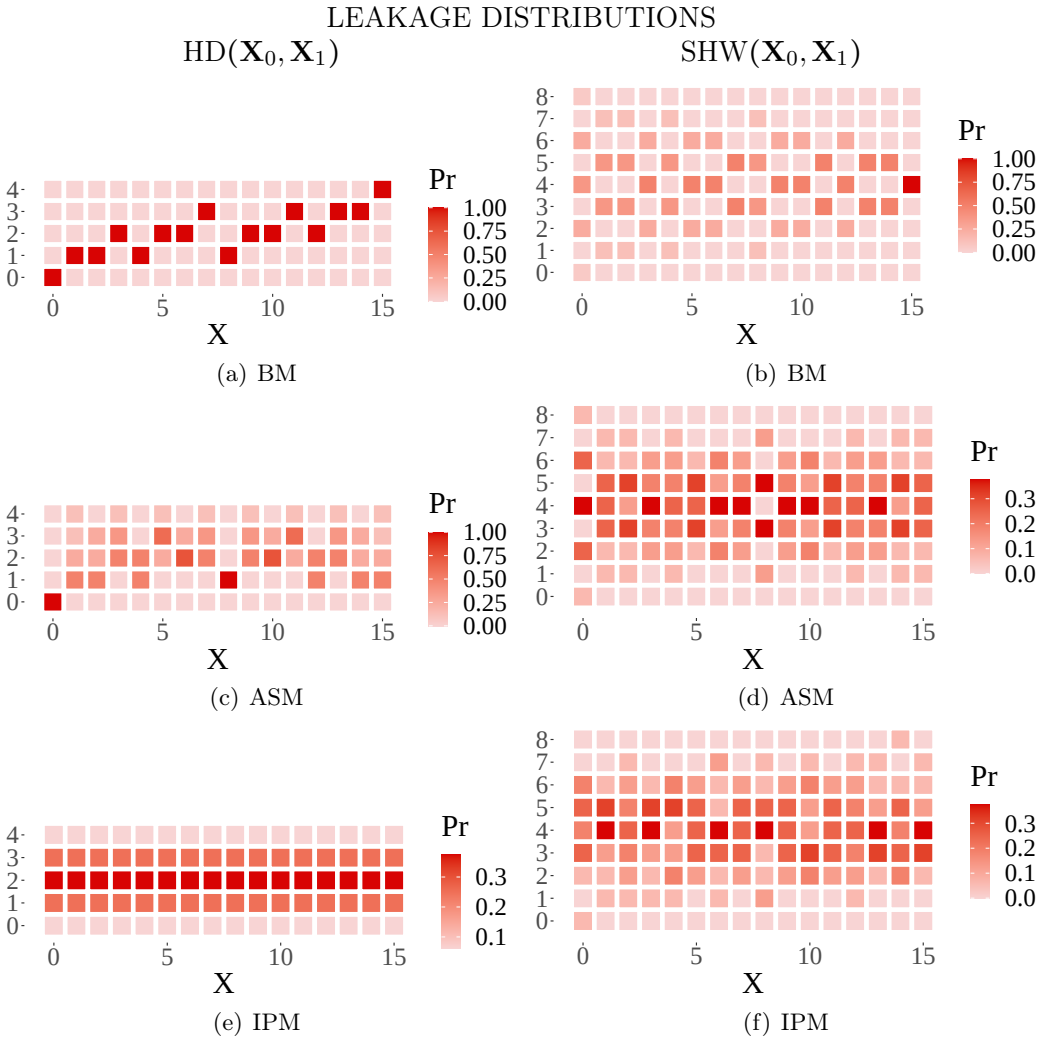
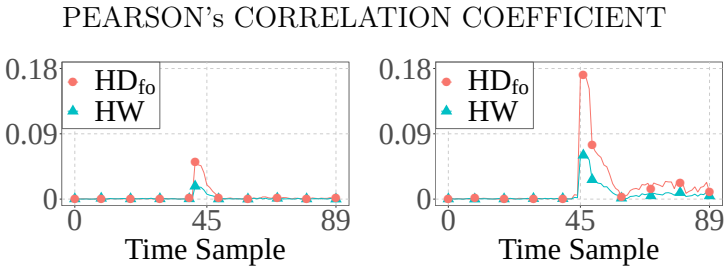


Figure 5.8: Distribution of the HD and SHW leakage models. Given $X \in \mathcal{X}_{2^4}$, the first row reports $\mathcal{D}_{(\mathrm{HD}(\mathbf{X}_0,\mathbf{X}_1),X)}$, whereas the second one reports $\mathcal{D}_{(\mathrm{SHW}(\mathbf{X}_0,\mathbf{X}_1),X)}$, where $\mathbf{X}_0, \mathbf{X}_1 \in \mathcal{X}_{2^4}$ represent the shares obtained from the application of BM (first column), ASM (second column) or IPM (third column) to X.

Figure 5.8 reports $\mathcal{D}_{(\mathrm{HD}(\mathbf{X}_0,\mathbf{X}_1),X)}$ and $\mathcal{D}_{(\mathrm{SHW}(\mathbf{X}_0,\mathbf{X}_1),X)}$ for BM, ASM and IPM. As the distributions differ, so the marginal distributions do. It is possible to exploit such difference to define (statistical-)moment-based leakage models.

For instance, we can associate to each X's realisation the first-order moment of the marginal $\mathcal{D}_{(\mathrm{HD}(\mathbf{X}_0,\mathbf{X}_1),X=x)}$:

$$\mathrm{HD}_{\mathrm{fo}}(x) = \frac{1}{|\mathbb{F}_{2^8}|^2} \sum_{\mathbf{x}_i \in \mathbb{F}_{2^8}, x = \bigodot_i x_i} \mathrm{HD}(\mathbf{x}_0, \mathbf{x}_1) \tag{5.5}$$



(a) STM32F215 (b) STM32F303

Figure 5.9: Experiment-based comparison of the $\mathrm{HD}_{\mathrm{fo}}$ and the HW leakage models. We consider the ASM case. We compute PCorrl on $4,000,000$ power-consumption traces. We collect traces during the execution of the UB-HD (Listing 5.5) on the STM32F215 and the STM32F303 boards.

Nevertheless, such moment-based approach cannot improve the PCorrl results in the IPM case, as the $\mathcal{D}_{(\mathrm{HD}(\mathbf{x}_0,\mathbf{x}_1),X)}$ is independent of X (and thus, any statistical moment is independent of X).

Concerning the SHW model, which we use to model the PPS-based leakages, the $\mathcal{D}_{(\mathrm{SHW}(\mathbf{x}_0,\mathbf{x}_1),X)}$'s first-order moment is independent of X, for all the three masking schemes. Thus, we can not straightforwardly employ $\mathrm{SHW}_{\mathrm{fo}}$ (Eq. 5.6) model.

$$\mathrm{SHW}_{\mathrm{fo}}(x) = \frac{1}{|\mathbb{F}_{2^8}|^2} \sum_{\mathbf{x}_i \in \mathbb{F}_{2^8}, x = \bigodot_i x_i} \mathrm{SHW}(\mathbf{x}_0, \mathbf{x}_1) \tag{5.6}$$

Yet, we can resort on the BLD preprocessing to make $\mathcal{D}_{(\mathrm{HD}(\mathbf{x}_0,\mathbf{x}_1),X)}$'s mean secret-dependent. We define the biased version of the $\mathrm{SHW}_{\mathrm{fo}}$ model:

$$\mathrm{SHW}_{\mathrm{fo},k\%}(x) = \frac{1}{|\mathbb{F}_{2^8}|^2} \sum_{\mathbf{x}_i \in \mathbb{F}_{2^8}, x = \bigodot_i x_i} \mathrm{SHW}_{k\%}(\mathbf{x}_0, \mathbf{x}_1) \tag{5.7}$$

where

$$\mathrm{SHW}_{k\%}(\mathbf{x}_0, \mathbf{x}_1) = \begin{cases} \mathrm{SHW}(\mathbf{x}_0, \mathbf{x}_1), \text{if } \mathrm{SHW}(\mathbf{x}_0, \mathbf{x}_1) \in \mathbb{O}_{k\%}(x) \\ 0, otherwise \end{cases}$$

and $\mathbb{O}_{k\%}(x)$ contains the $k\%$ lowest (or highest) realisation of $\mathrm{SHW}(\mathbf{X}_0, \mathbf{X}_1)$ when $X = x$.

5.4.2 Evaluation

We start with a first evaluation of the $\mathrm{HD}_{\mathrm{fo}}$ leakage model for transition-based leakages. We target the ASM scheme, as for BM we cannot improve the results, and the IPM is intrinsically immune to this leakage type. We compute $\rho(\mathrm{HD}_{\mathrm{fo}}(X), \mathbf{T}^i_{4\mathrm{M}\times 90})$, with $\mathbf{T}_{4\mathrm{M}\times 90}$ the trace set collected from the STM32F303 board executing UB-HD. Figure 5.9 confirms

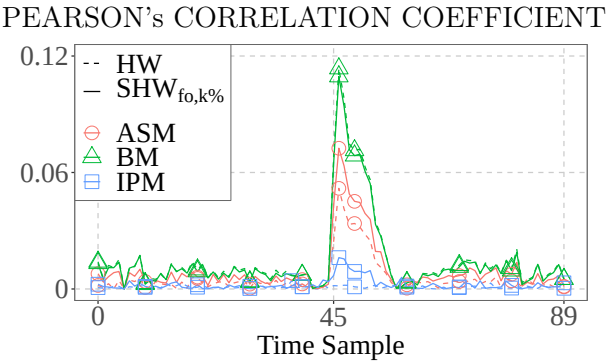


Figure 5.10: Experiment-based comparison of the HW and the $\text{SHW}_{\text{fo},k\%}$ model. We consider the case of the BLD-based PCorrl analyses, for $k = 10$ (Section 5.1.2). For the $\text{SHW}_{\text{fo},k\%}$ model, we set $k = 10$. We compute PCorrl over $4,000,000$ power-consumption traces. We collect the traces during the execution of the UB-SHW-LDRB (Listing 5.2) on the STM32F303 board.

the better suitability of the first-order moment leakage model, as we get a higher PCorrl value with respect to the HW model. Then, we test the improvements concerning the exploitation of PPS-based leakages. We employ the $\text{SHW}_{\text{fo},k\%}$ model against each masking scheme, computing $\rho(\text{SHW}_{\text{fo},k\%}(X), \mathbf{T}^i_{4\text{M}\times 90})$, with $\mathbf{T}_{4\text{M}\times 90}$ the trace set collected from the STM32F303 board executing UB-SHW-LDRB. We experimentally select $k = 10\%$ (i.e., $400,000$ traces per each $0 \leq i < 90$ sample) as it works well for BM, ASM and IPM. Figure 5.10 compares the PCorrl when employing the HW model and our moment-based leakage model. The HW allows the detection of correlation peaks in the case of BM and ASM schemes, but none in the IPM case. In contrast, our moment-based model not only improves the correlation results for the ASM, but it detects a correlation peak in the IPM case.

Such results corroborate the observations made in Section 5.3, remarking the better leakage resilience of ASM and IPM encodings against transition-based and PPS-based leakages.

5.5 Side-Channel Resilience of Software Masked AES-128

With Section 5.3 and Section 5.4 we assessed the practical security of different first-order masking encodings. Such analyses are fundamental get insights on the achievable security of masked implementations. Inner-product encoding showed perfect resistance against transition-based leakage, while Boolean and arithmetic encodings were more vulnerable. All masking encodings showed vulnerability to PPS-based leakage. We question how these findings translate on a full implementation.

This section aims at evaluating the impact of transition-based and PPS-based leakages on 4 software implementations of the AES-128 block-cipher: an unprotected version (*vanilla* from now on) and three masked ones, one for each masking scheme investigated. We have released all our investigated implementations (both C and binary codes) as publication artefacts (`https://zenodo.org/record/8094516`).

Our security assessment splits in two phases: at first, we evaluate whether the masked implementations leak information; the second one assess the resistance of such implementation against the exploitation of the (potential) leakage. The first phase relies on the TVLA methodology (Section 2.4.1) to provide an assessment independent on the class of attacker. The second phase relies on the same techniques employed to analyse the masking encodings (Section 5.3, Section 5.4). Specifically, we evaluate the security with and without the BLD technique (Section 5.1.2). We start by exploiting univariate first-order moment leakages, then we exploit univariate higher-order moment leakages with filtering. This last phase is particularly important to assess the practical security against PPS, since its first-order moment leakages can't be directly exploited (Section 5.3.1).

5.5.1 Experimental Setup

The vanilla implementation follows the FIPS-PUB-197 specification [NIS01], except for the key-scheduling: the implementation generates the next round key between the SubByte and the MixColumns steps.

Each first-order masked implementation follows by the manual application of the related masking scheme to the vanilla implementation. In particular, the BM and IPM ones follow the specification of Rivain et al. [RP10] and Balasch et al. [BFG15], respectively. For the IPM version, we resort to $\mathbf{L} = (1, 170) \in \mathbb{F}_{2^8}^2$, the same we employed for the experiment-based analyses (Section 5.3, Section 5.4). We implement the finite field multiplication using log/exp tables [GR17].

Concerning the ASM implementation, an inherent difficulty is the masking of the field addition (i.e., the eXclusive-OR, XOR). Indeed, the XOR is *non-linear* with respect to the arithmetic-sum operation. We mask the XOR operation by means of a masked look-up table. A straightforward tabulation of the operation would require 2^{16} byte of memory. To reduce the memory consumption, we tabulate the XOR on 4 bits, where the concatenation of the least (and most) significant inputs' nibbles indexes the table. We compute the XOR between two 8-bit inputs as a double access to such table: one to process the least significant nibbles of the inputs, and one to process the most significant ones. We remark that, the output carry of the arithmetic-sum potentially leaks information on the processed values. To prevent such leakage, we pre-charge the landing bit of the output-carry with a fresh random value.

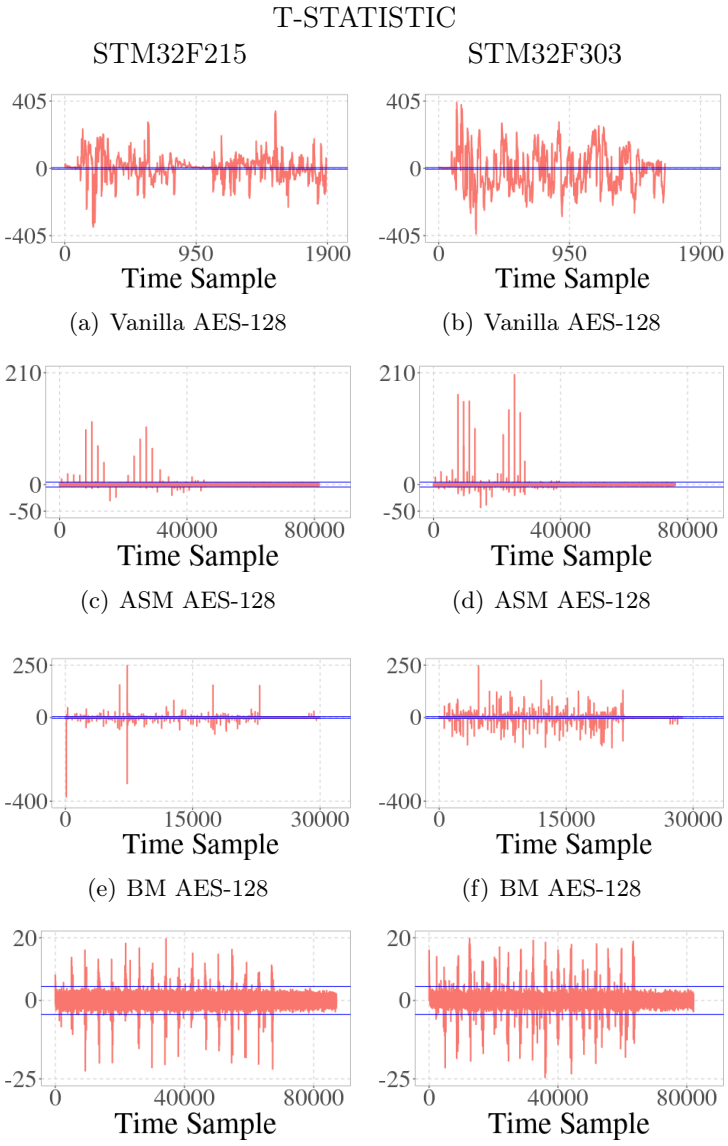
Table 5.1: Mean execution time (in clock cycles), number of calls to the PRNG, and segment size (in bytes) of each AES-128 implementation.

Version	Execution time		PRNG	Segment size		
	STM32F303	STM32F215	calls	`.text`	`.data`	`.bss`
Vanilla	3,524	4,180	0	1,016	525	0
BM	70,310	73,478	35	4,384	276	324
ASM	469,615	498,805	13,463	14,936	280	840
IPM	202,318	213,580	3,234	12,836	8,756	152

In the vanilla implementation, for performance reasons, we tabulate the SBOX and the XTIME functions. In the ASM implementation, we implement the same functions by means of masked look-up tables. Concerning the BM and IPM implementations, we compute those functions on the fly.

We resort to the experimental setup introduced in Section 5.2.3 (software toolchain and side-channel measurement setup). We develop each implementation in C language, and compile them with the compiler toolchain and compilation options reported in Section 5.2.3. Table 5.1 reports the mean execution time, number of PRNG calls, and memory impact of each AES-128 implementation. We report such parameters for both STM32F215 and STM32F303. Each masked implementation draws fresh randomness from the XOROSHIRO64** 1.0 PRNG [BV21]. The execution time from Table 5.1 includes time spent in the PRNG. We remark the long execution time ($500,000$ clock cycles on the STM32F215) for the ASM implementation. We ascribe it to the MixColumns step, which performs several accesses to the table-based XOR implementation. We remark that our experimental setup provides us with correctly-aligned side-channel traces. Hence, we do not require any re-alignment of the side-channel traces.

For the purpose of our analyses (e.g., leakage resilience against physical effects), we have to guarantee the correct application of the masking scheme. Each of the selected scheme considers a *value-based* leakage model. Thus, we verify that no value-based leakage can be detected from each implementation. To this end, we run TVLA analyses on simulated power-traces collected during the execution of each implementation on a ISA-level simulator of the ARMv7 profile. Specifically, we simulate the power consumption stemming from the usage of the register file and memory requests via load and store instructions. For all the implementations, we accept the null hypothesis (i.e., the implementation does not leak in the value-based model), proving the correct application of the three considered masking schemes.



(a) Vanilla AES-128 (b) Vanilla AES-128

(c) ASM AES-128 (d) ASM AES-128

(e) BM AES-128 (f) BM AES-128

(g) IPM AES-128 (h) IPM AES-128

Figure 5.11: TVLA results on the 4 AES-128 implementations. In red, we report the *maximum* t-statistic between two t-tests. In blue, the t-statistic threshold (±4.5) for the null hypothesis rejection. We execute each t-test by using a distinct fixed key. The first and third columns refer to the STM32F215 board, whereas the second and fourth ones to the STM32F303 board. Each plot refers to a $15,000$-vs-$15,000$ t-test, except for the IPM AES-128, which refers to a $90,000$-vs-$90,000$ t-test.

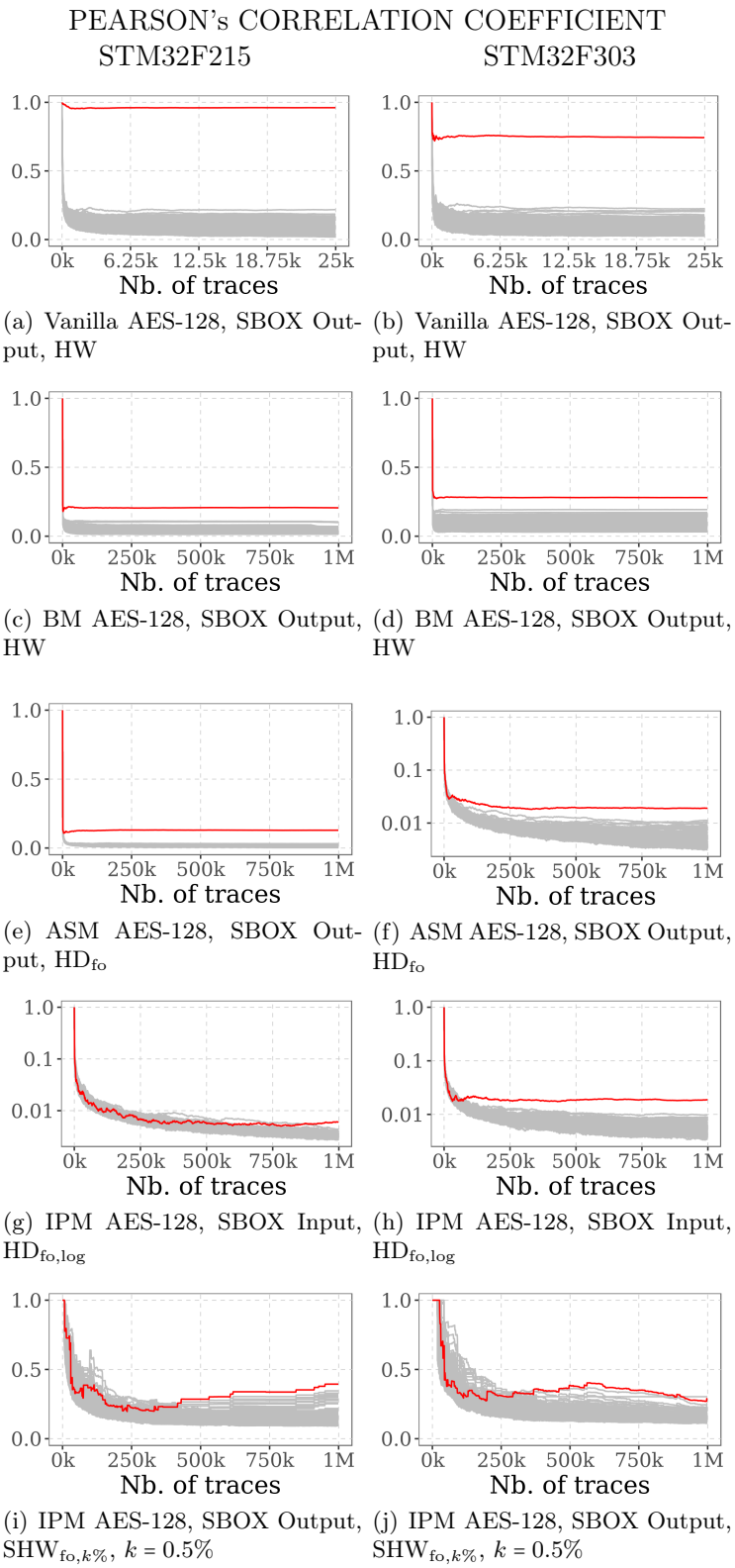
5.5.2 Information Leakage Evaluation

As a first step in the leakage resilience assessment of our AES-128 implementations, we proceed with the TVLA methodology. Precisely, we analyse the full first round of each implementation, except for the ASM implementation: as pointed out in Section 5.5.1, the MixColumns step counts for the largest part of the execution time. To reduce the trace collection time without compromising the validity of our results, we exclude the ASM's MixColumns from the leakage evaluation. As introduced in Section 2.4.1, the TVLA allows an evaluator to determine whether an implementation leaks or not, independently on the particular attack or leakage model. For the vanilla, BM and ASM implementations, we collect $15,000$ power-consumption traces for both fixed and random sets, respectively. Concerning the IPM implementation, we observed that it is characterised by a higher leakage resilience (Section 5.3, Section 5.4). To be more confident in its evaluation, perform the same assessment with $90,000$ power-based traces for both the fixed and random trace set. As explained in Section 2.4.1, the TVLA methodology is prone to errors of type I and II, where the latter represents the most problematic ones. To cope with them, for each implementation, we repeat the TVLA assessment two times, each with a distinct fixed key, and we measure the maximum absolute t-statistic for each sample point of the traces. Figure 5.11 reports the TVLA results for each AES-128 implementation and each target board.

The vanilla, BM and ASM implementations leak information along the whole first round. As we verified that the masking countermeasure is correctly applied at binary level, and as first-order statistical moments cannot detect leakage from PPS, we ascribe such leakage to recombination effects (e.g., transitions).

We remark that the ASM implementation presents fewer leaking samples than the BM. The algebraic structure of the ASM encoding potentially contribute to such observation.

Unexpectedly, the leakage assessment on the IPM implementations reveal several leakage points along the full first round. We found out that the source of such leakages stem from recombination effects that impact the log/exp-based field multiplication. Specifically, we verified the statistical dependence between $\mathrm{HD}(\log_3(\mathbf{X}_0), \log_3(\mathbf{X}_1))$ and the encoded value x. We conjecture that the non-linear nature of the logarithm function introduces some *bit-interaction* effect between the share's bits. Such effect counteracts the *randomness diffusion* of the IPM, making transition-based leakage again exploitable. Yet, we remark that, despite the higher number of employed traces, we observe a way lower magnitude of the t-statistic with respect to the one of the other implementations.



(a) Vanilla AES-128, SBOX Output, HW

(b) Vanilla AES-128, SBOX Output, HW

(c) BM AES-128, SBOX Output, HW

(d) BM AES-128, SBOX Output, HW

(e) ASM AES-128, SBOX Output, HD_{fo}

(f) ASM AES-128, SBOX Output, HD_{fo}

(g) IPM AES-128, SBOX Input, $HD_{fo,log}$

(h) IPM AES-128, SBOX Input, $HD_{fo,log}$

(i) IPM AES-128, SBOX Output, $SHW_{fo,k\%}$, $k = 0.5\%$

(j) IPM AES-128, SBOX Output, $SHW_{fo,k\%}$, $k = 0.5\%$

Figure 5.12: CPA results for the four AES-128 implementations. In grey, the wrong key hypotheses, whereas in red the correct one. Figure 5.12(f), 5.12(g) and 5.12(h) report the PCorrl in Log10 scale. For each implementation, we employ a different leakage model (Table 5.2). For the $SHW_{fo,k\%}$ model, the X-axis reports the number of collected traces (i.e., before trace filtering). Each row refers to a different implementation/leakage model combination. First and second columns refer, respectively, to the STM32F215 and STM32F303 board.

Table 5.2: Summary of the leakage models used for the side-channel analysis of each AES-128 implementation.

Implementation(s)	Leakage Model	
	Transitions	PPS
Vanilla, BM	HW (Eq. 2.5)	-
ASM	HW, $\mathrm{HD}_{\mathrm{fo}}$ (Eq. 5.5)	-
IPM	HW, $\mathrm{HD}_{\mathrm{fo,log}}$ (Eq. 5.8)	$\mathrm{SHW}_{\mathrm{fo},k\%}$ (Eq. 5.7)

5.5.3 Information Leakage Exploitation

In the previous section, we assessed the leakage resilience of our AES-128 implementations. We observed results consistent to the encoding analyses (Section 5.3, Section 5.4), except for the IPM. In fact, we observed unexpected leakage stemming from the finite field multiplication. Despite the presence of leakage, the TVLA methodology does not provide any clue concerning the *exploitability* of the leaked information.

With this section, we explore the resilience of our software masked implementations against information leakage exploitation; specifically, against univariate side-channel attacks.

To this end, we rely on standard, BLD-based (Section 5.1.2) and moment-based-model (Section 5.4) CPA attacks. For each implementation and target board, we measure $1,000,000$ power traces.

The side-channel analysis proceeds as follows. We analyse the usage of the first secret key byte during the SubByte step of the first round, and we compute $\rho(\mathrm{L}(X)_{\mathrm{d}}, \mathbf{T}^{j}_{1\mathrm{M}\times m})$, where m varies according to the target implementation. Table 5.2 summarises the leakage models $\mathrm{L}(\cdot)_{\mathrm{d}}$ employed to attack each implementation.

For the IPM implementations, we also target the SubByte's input, which comes as result of the field implementation. We employ the first-order-moment leakage model $\mathrm{HD}_{\mathrm{fo,log}}$:

$$\mathrm{HD}_{\mathrm{fo,log}}(x) = \frac{1}{|\mathbb{F}_{2^8}|^2} \sum_{\mathbf{x}_i \in \mathbb{F}_{2^8}} \mathrm{HD}(\log_3(\mathbf{x}_0), \log_3(\mathbf{x}_1)) \tag{5.8}$$

Figure 5.12 reports the results of the different CPA attacks, and Table 5.3 reports the minimum number of traces required to mount a successful CPA attack. Despite the correct application of the masking scheme on the binaries, we exploit only 140 and $241,000$ traces to break the BM and ASM implementations, respectively. Consistently with the result from Section 5.4, the $\mathrm{HD}_{\mathrm{fo}}$ model improves the attack efficiency against the ASM implementation, reducing up to $\times 8.6$ times the minimum number of traces to mount a successful CPA attack, with respect to a plain use of the HW model.

Table 5.3: Minimum number of traces to mount a successful CPA attack against the AES-128 implementations. We report `failed` in case of attack failure with $1,000,000$ traces.

Device	STM32F215					STM32F303				
Version	Vanilla	BM	ASM		IPM	Vanilla	BM	ASM		IPM
Number of traces	5	140	2970 1710	(HW) (HD_{fo})	`failed` (HW) 867k ($HD_{fo,log}$)	16	49	241k 28k	(HW) (HD_{fo})	`failed` (HW) 82k ($HD_{fo,log}$)

By targeting the SBOX input, we successfully retrieve the target key byte on IPM implementations. This suggests that the design of masking schemes should also consider the implementation of the employed algorithms (e.g., finite field multiplication). We remark that the attack on the STM32F215 takes longer to succeed. This may be due to the lower accuracy of the HD model for this device and/or the higher noise affecting the platform (Section 5.2.4).

We conclude the leakage exploitation analyses with the BLD-based CPA attacks (Section 5.1.2). We evaluate the resilience of each implementation according to several k values. Table 5.4 reports the rank of the correct key hypothesis with $1,000,000$ traces, and the minimum traces number to reach that rank. On the STM32F303, the correct key hypothesis frequently appears among the best correlated key candidates. Table 5.4 reports the number of traces necessary to observe the correct key byte hypothesis among the 4 best correlated key candidates. Then, an attacker can brute-force the 4^{16} possible 128-bit keys.

We remark that (1) the choice of the threshold value k is relevant to mount a successful CPA attack, (2) that low k values increase the probabilities of a successful side-channel attack. We ascribe this to the higher noise setting compared to more controlled context of the encoding analyses (Section 5.3, Section 5.4).

Our results emphasise the threat that PPS and recombination effects represent. Also, we highlight the practical security impact of different representations of data in a masked software implementation (e.g., logarithm of a share). As a first guideline to mitigate PPS-based leakages exploitation, developers should avoid packing shares within the same word (Listing 5.2). However, such condition is necessary, but not sufficient, as PPS potentially stems from other sources (Section 5.2.2).

5.6 Discussion

In this section, we warn about unanticipated sources of weaknesses in masked implementations, then we discuss how parallel-oriented architectures and programming models

Table 5.4: Key ranking of correct key guess when employing the $\mathrm{SHW}_{\mathrm{fo},k\%}$ against IPM implementations. We report the correct key-guess rank and related number of traces for $k \in \{0.1\%, 0.2\%, \ldots, 1\%, 2\%\}$. We omit the entries for $k > 2\%$, as we did not succeed in the attack. The number of traces corresponds to the number of *collected* traces (i.e., not the number of traces actually analysed).

Threshold (%)		0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1	2
STM32F215	Key rank at 1M traces	1	1	1	1	1	1	1	1	6	7	1
	Number of traces to get to this rank	430k	608k	417k	333k	430k	512k	430k	426k	417k	417k	814k
	Number of traces to get to top 4 rank	416k	417k	235k	333k	412k	401k	417k	417k	—	—	531k
STM32F303	Key rank at 1M traces	3	6	10	7	2	1	4	4	7	7	4
	Number of traces to get to this rank	460k	998k	997k	368k	368k	997k	997k	994k	974k	987k	784k
	Number of traces to get to top 4 rank	460k	—	—	—	324k	494k	997k	994k	—	—	784k

can introduce PPS in software, and we give some principles to prevent the vulnerabilities created by PPS.

5.6.1 On the Resilience of IPM to Transition-based Leakage

In Section 5.3, we have shown that IPM encodings are immune to transition-based leakages, which is consistent with literature knowledge. Yet, in Section 5.5 we were able to successfully attack IPM masked implementations through a leakage model targeting such leakages. We found the root cause in the use of logarithms in the finite field multiplication implementation. Transition-based leakages on logarithm representation of the encodings induced exploitable leakage. Such gap underlines the importance of studying the masking resistance both theoretically and practically. It suggests that the different representations of masked encodings used in an implementation should all be considered for security assessment.

5.6.2 PPS and Parallel-Oriented Architectures

The PPS threat emerges whenever *data processing parallelism* can be achieved. From a hardware point-of-view, PPS readily extends to any architecture encompassing any kind of feature implying data parallelism. In our work, we focused on simple micro-architectures encompassing instruction pipelining, which implies a sort of data paral-

lelism. Gigerl et al. show that super-scalar micro-architectures exhibit more sources of transition-based leakage [GPM21] due to pipeline depth and multiple issuing of instructions. In such micro-architectures, data parallelism is exacerbated, and so the possible occurrence of PPS.

Instruction Set Extensions (ISE) play an important role in the introduction of PPS. Miayjan et al. suggest the employment of SIMD (Single Instruction Multiple Data) ISE to provide efficient and secure software masked implementations [Miy+15]. The SIMD ISE enables *data-level* parallel processing, handling multiple data via a single instruction [HP12]. The explicit data parallelism naturally implies PPS. Such remark extends also to GPU architectures, designed to intrinsically support data-level parallelism. Still, we are not aware of any work concerning their usage to accelerate software masked implementations. Finally, FPGAs represent an interesting case: they can be employed for either the implementation of hardware implementations, or for the implementation of full CPUs [GMV20]. In both cases, the designs might rely on some parallel features, e.g., [VRM17], potentially introducing the PPS vulnerability.

5.6.3 Preventing PPS in Software

PPS emerges whenever the micro-architecture handles related shares in parallel. As discussed, architectures encompassing parallel features and certain programming models potentially introduce the PPS threat. As a naïve solution, the programmer should rely on programming techniques which do not promote data parallelism, and execute the implementation on architecture not endowed with parallel features. Yet, such approach would increase the already high cost of a masked implementation, in particular for masked instances of order $n > 1$.

Instead, we advocate for a more principled approach, based on the concept of *Non-Completeness*. Non-Completeness is a security property defined in the context of *Threshold Implementations* [Bil+14]. Informally, by seeing an n-th order masked algorithm as a composition of sub-functions, each sub-function has to handle no more than n shares. Gaspoz and Dhooghe extend this property to provide *necessary* security properties against micro-architecture-induced recombination effects [GD23]. In particular, we remark their *Horizontal Register Non-Completeness* as a necessary condition to avoid PPS. Such property contrasts certain programming techniques, e.g., *share-slicing* [Gao+20a], which aim at the efficient implementation of masked software implementations.

Yet, their notion of non-completeness does not take into consideration the PPS stemming from the pipeline's depth (i.e., number of pipeline stages). Indeed, PPS originates also from related shares manipulated in different pipeline stages. It is possible to extend the non-completeness property at pipeline level, requiring that the pipeline does not pro-

cess more than n shares at a time. Gigerl et al. suggest a stricter version of this *Pipeline Non-Completeness* property, separating the processing of related shares according to the pipeline's depth and number of instructions that can be executed in parallel to prevent glitch-based leakage [GPM21].

Admittedly, register and pipeline non-completeness might not be sufficient to prevent PPS. Indeed, the register file, caches and memory, potentially store all the shares of an encoding. Static power leakage potentially allows an attacker to observe these shares, enabling successful attacks [Mor14]. The risk implied by static power leakage is still unexplored in the software context.

We conclude this discussion by remarking that the IPM scheme (more generally, the family of *code-based* masking) can *amplify* the security order naturally expected [Wan+16; Che+21; Wu+22]. That is, given a masking of order n, according to the particular public vector $\mathbf{L}$, the security order can be higher than n. Although we analysed IPM instantiated with *non-optimal* codes (i.e., which do not amplify the security order), the use of optimal codes can be a sound way to better mitigate PPS-based leakage. We leave as an interesting future work the investigation of the practical security guarantees of optimal code-based software masked implementations when register and pipeline non-completeness are satisfied.

5.7 Conclusion

Recent literature has highlighted the CPU micro-architecture as a rich source of recombination effects (e.g., transitions), which severely decrease the security of masking. Although the pervasiveness of such effects, our work shows that they do not represent the only threat to the practical security of masking in software: the parallel processing of share (PPS), exercised by a CPU micro-architecture, represents a potential threat too. Relying on an adaptation of the preprocessing technique proposed by Moos and Moradi [MM17], we show how to exploit PPS-based leakage against first-order instances of Boolean, arithmetic-sum and inner-product masking. Furthermore, despite the fact that some schemes, such as the inner-product masking, provide immunity to transition-based leakage, particular operations can remove such immunity. Specifically, we show how the employment of the log operation in the field multiplication algorithm allows the successful exploitation of transition-based leakage against the inner-product masking.

Chapter 6

Conclusion and Perspectives

With this sixth and conclusive chapter, we provide the conclusion we can draw up from each thesis' contribution. We close the thesis by elaborating on the potential research perspectives and potential future works.

6.1 Conclusion

Masking schemes provide a sound and proven approach to counteract the exploitation of side-channel information. However, the physical effects characterizing the micro-architecture of CPUs severely reduces the proven security guarantees. Long-term solutions ask for a CPU hardware taking into account such problematic. On the other hand, the lack of mitigation mechanism in current CPU designs calls for short-term solutions to develop leakage-resilient masked implementations in practice. With this thesis, we addressed the challenge of developing practically-secure masked software. As a first contribution, we presented an automated methodology for the development of first-order masked software resilient to micro-architecture-induced transition-based leakages. This methodology relies on code-generation algorithms to automate the generation of masked implementations while mitigating transition-based leakages stemming from both the architectural and micro-architectural details of the target CPU. Operating on an annotated intermediate representation of a masked software, and supported by a description of the (micro-)architectural features involved with transition-based leakages, we enhance the register allocation and instruction scheduling algorithms with *leakage-awareness*. By implementing our approach as part of the `LLVM` Core Libraries, and integrating it with an `LLVM`-based tool-chain for the automated generation of first-order Boolean-masked software implementations, we show how our approach mitigates portions of the leaking samples occurring on a plain first-order implementation, while containing the overheads of the protected implementations in terms of execution time, randomness requirements

and code size with respect to a second-order masked solution. Yet, this approach, as the rest of the current existing works in the state of the art, rely on a description of the leaking micro-architectural features. This kind of dependence impacts the quality of the final solution—in terms of practical security and overheads—as well as the portability of the security guarantees across different micro-architecture designs.

A natural question is whether one can envision a more target-independent approach to achieve security in practice, despite the security degradations implied by the micro-architecture. In this regard, our second contribution consists in the evaluation of the impact of micro-architecture-induced leakages on different masking schemes. Specifically, we investigated the practical security guarantees on first-order arithmetic-sum and inner-product masking, comparing them to first-order Boolean masking. At the same time, we explored the potential impact that the data parallelism, potentially induced by the CPU's micro-architecture, might have on the practical security of a masked software implementation. The investigation first elaborated and practically showed how data parallelism can manifest and induce information leakage even on simple scalar micro-controllers. As a second step, we evaluated the side-channel resilience of the masked variables when exposed to transition-based leakages and the leakage induced by data parallelism. Eventually, we evaluated the security guarantees of several software implementations of the AES-128 cryptosystem, each masked at first-order with one of the studied masking schemes. We remark that, within this investigation, we did not attempt the mitigation or suppression of the informative side-channels but, rather, we evaluated the intrinsic mitigation of the information leakage provided by each considered masking schemes. From this study, we show that, although the different side-channel resilience of the considered masking schemes, their as-it application is not viable as, just relying on slightly elaborated attacks, they do not withstand the exploitation of both types of information leakage.

6.2 Perspectives

In the scope of each contribution, we can elaborate several future works. Concerning our approach to automate the mitigation of micro-architecture-induced leakages, its integration with approaches that expose more instruction-level parallelism could provide highly efficient, side-channel resilient implementations. For instance, we find in Threshold Implementation masking and the bit-slicing paradigm potential candidates: the former splits the masking of an algorithm in the masking of several sub-functions, which can be potentially computed in parallel; the latter naturally exhibit parallel computation of data, which has been already studied in the masking context [Bel+20]. Natural extensions of the approach encompass higher-order masking, other types of masking schemes (for in-

stance Inner-Product masking) and description of more complex micro-architectures (for instance, super-scalar ones) and explore the potential benefits our code-generation-based approach in such contexts. Along the idea of promoting collaboration between hardware and software, the combination of our approach with ISE-based ones has the potential to generate efficient and practically secure implementations. Indeed, a leakage-aware generation of the code potentially minimize the need to flush the micro-architecture, which an underlying hardware mechanism can handle to precisely target the leakage sources.

Concerning the study we conducted on the leakage resilience of different masking schemes, we remark that for the Inner-Product masking we employed an arbitrary code for the construction of the masked variables. Cheng et al. investigated codes optimizing the leakage resilience of such type of masking scheme against value-based leakage models [Che+21]. As such, we leave as an interesting future work the analysis of the contribution of such optimal codes might provide in practice against micro-architectural leakages. Our study limits to first-order masked implementations. Thus, a natural follow-up is the evaluation of the side-channel resilience of higher-order masking schemes. In particular, in light of the shown threat that data parallelism can represent, we find interesting challenging the idea of resorting to higher-order masking to provide first-order security. Another interesting research direction we consider is the evaluation of the practical side-channel resilience in the context of more complex CPU designs. One might wonder what are the consequences, in terms of side-channel resilience, when more performant architectures get in the way of the masking designer. Barenghi et al. and Gigerl et al. already started this investigation with respect to the potential recombination effects characterizing super-scalar CPUs [BP18] [GPM21]. As we overviewed in Chapter 3, a highly performant processor potentially exhibits increased parallelism capabilities. Such higher degree of parallelism becomes a potential detriment not only for first-order masked implementations, but also for higher-order ones, challenging the idea of using higher-order masking to provide practical side-channel security.

Bibliography

[ABP21] Francesco Antognazza, Alessandro Barenghi, and Gerardo Pelosi. "Metis: An Integrated Morphing Engine CPU to Protect Against Side Channel Attacks". In: *IEEE Access* 9 (2021), pp. 69210–69225. DOI: 10.1109/ACCESS.2021.3077977 (cit. on p. 26).

[Abr+21] Arnold Abromeit, Florian Bache, Leon A. Becker, Marc Gourjon, Tim Güneysu, Sabrina Jorn, Amir Moradi, Maximilian Orlt, and Falk Schellenberg. "Automated Masking of Software Implementations on Industrial Microcontrollers". In: *DATE*. 2021 (cit. on pp. 25, 27, 29, 30, 37, 89).

[Aro+21] Vipul Arora, Ileana Buhan, Guilherme Perin, and Stjepan Picek. "A Tale of Two Boards: On the Influence of Microarchitecture on Side-Channel Leakage". In: *CARDIS*. 2021 (cit. on pp. 25, 30, 94).

[Ath+20] Konstantinos Athanasiou, Thomas Wahl, A. Adam Ding, and Yunsi Fei. "Automatic Detection and Repair of Transition- Based Leakage in Software Binaries". In: *Software Verification - 12th International Conference, VSTTE 2020, and 13th International Workshop, NSV 2020, Los Angeles, CA, USA, July 20-21, 2020, Revised Selected Papers*. Ed. by Maria Christakis, Nadia Polikarpova, Parasara Sridhar Duggirala, and Peter Schrammel. Vol. 12549. Lecture Notes in Computer Science. Springer, 2020, pp. 50–67. DOI: 10.1007/978-3-030-63618-0_4 (cit. on pp. 25, 27, 29).

[Bal+14] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. "On the Cost of Lazy Engineering for Masked Software Implementations". In: *CARDIS*. 2014 (cit. on pp. 10, 17, 18, 23).

[Bar+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. "Strong Non-Interference and Type-Directed Higher-Order Masking". In: *ACM SIGSAC*. 2016 (cit. on p. 29).

[Bar+21] Alessandro Barenghi, Luca Breveglieri, Niccolò Izzo, and Gerardo Pelosi. "Exploring Cortex-M Microarchitectural Side Channel Information Leakage". In: *IEEE Access* (2021) (cit. on pp. 3, 76, 97).

[Bea+13] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. "The SIMON and SPECK Families of Lightweight Block Ciphers". In: *IACR Cryptol. ePrint Arch.* (2013) (cit. on pp. 79, 80).

[Bea+19] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. *SIMON and SPECK Implementation Guide*. English. National Security Agency. Jan. 19, 2019. 37 pp. (cit. on p. 80).

[Bec+22] Arthur Beckers, Lennert Wouters, Benedikt Gierlichs, Bart Preneel, and Ingrid Verbauwhede. "Provable Secure Software Masking in the Real-World". In: *COSADE*. 2022 (cit. on p. 38).

[Bel+18] Nicolas Belleville, Karine Heydemann, Damien Couroussé, Thierno Barry, Bruno Robisson, Abderrahmane Seriai, and Henri-Pierre Charles. "Automatic Application of Software Countermeasures Against Physical Attacks". In: *Cyber-Physical Systems Security*. Ed. by Çetin Kaya Koç. Springer, 2018, pp. 135–155. DOI: 10.1007/978-3-319-98935-8_7 (cit. on p. 26).

[Bel+20] Sonia Belaïd, Pierre-Évariste Dagand, Darius Mercadier, Matthieu Rivain, and Raphaël Wintersdorff. "Tornado: Automatic Generation of Probing-Secure Masked Bitsliced Implementations". In: *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part III*. Ed. by Anne Canteaut and Yuval Ishai. Vol. 12107. Lecture Notes in Computer Science. Springer, 2020, pp. 311–341. DOI: 10.1007/978-3-030-45727-3_11. URL: https://doi.org/10.1007/978-3-030-45727-3%5C_11 (cit. on p. 124).

[Bel+23] Sonia Belaïd, Gaëtan Cassiers, Camille Mutschler, Matthieu Rivain, Thomas Roche, François-Xavier Standaert, and Abdul Rahman Taleb. "Towards Achieving Provable Side-Channel Security in Practice". In: *IACR Cryptol. ePrint Arch.* (2023), p. 1198 (cit. on p. 25).

[BFG15] Josep Balasch, Sebastian Faust, and Benedikt Gierlichs. "Inner Product Masking Revisited". In: *EUROCRYPT*. 2015 (cit. on p. 112).

[Bil+14] Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. "Higher-Order Threshold Implementations". In: *IACR Cryptol. ePrint Arch.* (2014), p. 751 (cit. on pp. 18, 34, 120).

[BJ22] Vahhab Samadi Bokharaie and Ali Jahanian. "Power side-channel leakage assessment and locating the exact sources of leakage at the early stages of ASIC design process". In: *J. Supercomput.* 78.2 (2022), pp. 2219–2244. DOI: 10.1007/s11227-021-03927-w (cit. on p. 26).

[Boa73] David G. Boak. *A History of U.S. Communication Security.* Tech. rep. National Security Agency, July 1973. URL: https://www.governmentattic.org/2docs/Hist_US_COMSEC_Boak_NSA_1973.pdf (cit. on p. 1).

[BP18] Alessandro Barenghi and Gerardo Pelosi. "Side-channel security of superscalar CPUs: evaluating the impact of micro-architectural features". In: *DAC*. 2018 (cit. on pp. 3, 23, 24, 125).

[Bro+19] Olivier Bronchain, Julien M. Hendrickx, Clément Massart, Alex Olshevsky, and François-Xavier Standaert. "Leakage Certification Revisited: Bounding Model Errors in Side-Channel Security Evaluations". In: *CRYPTO*. 2019 (cit. on pp. 12, 105).

[BV21] David Blackman and Sebastiano Vigna. "Scrambled Linear Pseudorandom Number Generators". In: *ACM Trans. Math. Softw.* (2021) (cit. on pp. 80, 113).

[Cas+23] Gaëtan Cassiers, Loïc Masure, Charles Momin, Thorben Moos, Amir Moradi, and François-Xavier Standaert. "Randomness Generation for Secure Hardware Masking - Unrolled Trivium to the Rescue". In: *IACR Cryptol. ePrint Arch.* (2023), p. 1134. URL: https://eprint.iacr.org/2023/1134 (cit. on p. 88).

[CGD18] Yann Le Corre, Johann Großschädl, and Daniel Dinu. "Micro-architectural Power Simulator for Leakage Assessment of Cryptographic Software on ARM Cortex-M3 Processors". In: *Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings*. Ed. by Junfeng Fan and Benedikt Gierlichs. Vol. 10815. Lecture Notes in Computer Science. Springer, 2018, pp. 82–98. DOI: 10.1007/978-3-319-89641-0_5. URL: https://doi.org/10.1007/978-3-319-89641-0%5C_5 (cit. on p. 4).

[Cha+81] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. "Register Allocation Via Coloring". In: *Comput. Lang.* 6.1 (1981), pp. 47–57. DOI: 10.1016/0096-0551(81)90048-5. URL: https://doi.org/10.1016/0096-0551(81)90048-5 (cit. on p. 52).

[Cha+99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. "Towards Sound Approaches to Counteract Power-Analysis Attacks". In: *CRYPTO*. 1999 (cit. on pp. 15, 17).

[Che+21] Wei Cheng, Sylvain Guilley, Claude Carlet, Jean-Luc Danger, and Sihem Mesnager. "Information Leakages in Code-based Masking: A Unified Quantification Approach". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst* (2021) (cit. on pp. 38, 121, 125).

[Cor14] Jean-Sébastien Coron. "Higher Order Masking of Look-Up Tables". In: *EUROCRYPT*. Ed. by Phong Q. Nguyen and Elisabeth Oswald. 2014 (cit. on p. 15).

[CP23] Hao Cheng and Daniel Page. "eLIMInate: a Leakage-focused ISE for Masked Implementation". In: *IACR Cryptol. ePrint Arch.* (2023), p. 966. URL: `https://eprint.iacr.org/2023/966` (cit. on pp. 4, 25, 31, 33, 35).

[CRR02] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. "Template Attacks". In: *Cryptographic Hardware and Embedded Systems - CHES 2002, 4th International Workshop, Redwood Shores, CA, USA, August 13-15, 2002, Revised Papers*. Ed. by Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar. Vol. 2523. Lecture Notes in Computer Science. Springer, 2002, pp. 13–28. DOI: `10.1007/3-540-36400-5\_3` (cit. on p. 13).

[DDF14] Alexandre Duc, Stefan Dziembowski, and Sebastian Faust. "Unifying Leakage Models: From Probing Attacks to Noisy Leakage". In: *EUROCRYPT*. 2014 (cit. on p. 17).

[De +17] Thomas De Cnudde, Begül Bilgin, Benedikt Gierlichs, Ventzislav Nikov, Svetla Nikova, and Vincent Rijmen. "Does Coupling Affect the Security of Masked Implementations?" In: *COSADE*. 2017 (cit. on pp. 17, 23).

[DFS19] Alexandre Duc, Sebastian Faust, and François-Xavier Standaert. "Making Masking Security Proofs Concrete (Or How to Evaluate the Security of Any Leaking Device), Extended Version". In: *J. Cryptol.* (2019) (cit. on pp. 10, 104).

[dHM22] Arnaud de Grandmaison, Karine Heydemann, and Quentin L. Meunier. "ARMISTICE: Microarchitectural Leakage Modeling for Masked Software Formal Verification". In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* (2022) (cit. on pp. 3, 23, 24, 42, 76, 78, 97).

[Fau+18] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. "Composable Masking Schemes in the Presence of Physical Defaults & the Robust Probing Model". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.3 (Aug. 2018), pp. 89–120. DOI: 10.13154/tches.v2018.i3.89-120. URL: https://tches.iacr.org/index.php/TCHES/article/view/7270 (cit. on p. 18).

[Gao+20a] Si Gao, Ben Marshall, Dan Page, and Elisabeth Oswald. "Share-slicing: Friend or Foe?" In: *IACR Trans. Cryptogr. Hardw. Embed. Syst* (2020) (cit. on pp. 3, 23, 24, 32, 37, 120).

[Gao+20b] Si Gao, Ben Marshall, Dan Page, and Thinh Hung Pham. "FENL: an ISE to mitigate analogue micro-architectural leakage". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* (2020) (cit. on pp. 3, 4, 21, 25, 31–34, 91).

[Gao+21] Si Gao, Johann Großschädl, Ben Marshall, Dan Page, Thinh Hung Pham, and Francesco Regazzoni. "An Instruction Set Extension to Support Software-Based Masking". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021.4 (2021), pp. 283–325. DOI: 10.46586/tches.v2021.i4.283-325 (cit. on pp. 25, 31–34).

[GD23] John Gaspoz and Siemen Dhooghe. "Threshold Implementations in Software: Micro-architectural Leakages in Algorithms". In: *CHES* (2023) (cit. on pp. 4, 23, 25, 34, 37, 120).

[GHR15] Sylvain Guilley, Annelie Heuser, and Olivier Rioul. "A Key to Success - Success Exponents for Side-Channel Distinguishers". In: *Progress in Cryptology - INDOCRYPT 2015 - 16th International Conference on Cryptology in India, Bangalore, India, December 6-9, 2015, Proceedings*. Ed. by Alex Biryukov and Vipul Goyal. Vol. 9462. Lecture Notes in Computer Science. Springer, 2015, pp. 270–290. DOI: 10.1007/978-3-319-26617-6_15 (cit. on p. 11).

[Gig+21] Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. "Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs". In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. Ed. by Michael D. Bailey and Rachel Greenstadt. USENIX Association, 2021, pp. 1469–1468. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/gigerl (cit. on p. 4).

[GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. "Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order". In: *Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016 Vienna, Austria, October, 2016*. Ed. by Begül Bilgin, Svetla Nikova, and Vincent Rijmen. ACM, 2016, p. 3. DOI: 10.1145/2996366.2996426. URL: https://doi.org/10.1145/2996366.2996426 (cit. on p. 18).

[GMV20] T. Gokulan, Akshay Muraleedharan, and Kuruvilla Varghese. "Design of a 32-bit, dual pipeline superscalar RISC-V processor on FPGA". In: *23rd Euromicro Conference on Digital System Design*. 2020 (cit. on p. 120).

[GPM21] Barbara Gigerl, Robert Primas, and Stefan Mangard. "Secure and Efficient Software Masking on Superscalar Pipelined Processors". In: *ASIACRYPT*. 2021 (cit. on pp. 3, 17, 18, 23–25, 27, 34, 37, 91, 120, 121, 125).

[GR17] Dahmun Goudarzi and Matthieu Rivain. "How Fast Can Higher-Order Masking Be in Software?" In: *EUROCRYPT*. 2017 (cit. on p. 112).

[Gro+16] Hannes Groß, Manuel Jelinek, Stefan Mangard, Thomas Unterluggauer, and Mario Werner. "Concealing Secrets in Embedded Processors Designs". In: *Smart Card Research and Advanced Applications - 15th International Conference, CARDIS 2016, Cannes, France, November 7-9, 2016, Revised Selected Papers*. Ed. by Kerstin Lemke-Rust and Michael Tunstall. Vol. 10146. Lecture Notes in Computer Science. Springer, 2016, pp. 89–104. DOI: 10.1007/978-3-319-54669-8_6 (cit. on p. 26).

[Gro15] Hannes Groß. "Sharing is Caring - On the Protection of Arithmetic Logic Units against Passive Physical Attacks". In: *Radio Frequency Identification. Security and Privacy Issues - 11th International Workshop, RFIDsec 2015, New York, NY, USA, June 23-24, 2015, Revised Selected Papers*. Ed. by Stefan Mangard and Patrick Schaumont. Vol. 9440. Lecture Notes in Computer Science. Springer, 2015, pp. 68–84. DOI: 10.1007/978-3-319-24837-0_5 (cit. on p. 26).

[HP12] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach, 5th Edition*. 2012 (cit. on pp. 3, 21, 22, 97, 120).

[HRG14] Annelie Heuser, Olivier Rioul, and Sylvain Guilley. "Good Is Not Good Enough - Deriving Optimal Distinguishers from Communication Theory". In: *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*. Ed. by Lejla Batina and Matthew Robshaw. Vol. 8731. Lecture

Notes in Computer Science. Springer, 2014, pp. 55–74. DOI: 10.1007/978-3-662-44709-3_4 (cit. on p. 11).

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. "Private Circuits: Securing Hardware against Probing Attacks". In: *CRYPTO*. 2003 (cit. on p. 17).

[Kia+19] Pantea Kiaei, Darius Mercadier, Pierre-Évariste Dagand, Karine Heydemann, and Patrick Schaumont. "SKIVA: Flexible and Modular Side-channel and Fault Countermeasures". In: *IACR Cryptol. ePrint Arch.* (2019), p. 756. URL: https://eprint.iacr.org/2019/756 (cit. on pp. 4, 25, 31–33).

[KS20] Pantea Kiaei and Patrick Schaumont. "Domain-Oriented Masked Instruction Set Architecture for RISC-V". In: *IACR Cryptol. ePrint Arch.* (2020), p. 465. URL: https://eprint.iacr.org/2020/465 (cit. on pp. 25, 31, 33, 34).

[KS22] Pantea Kiaei and Patrick Schaumont. "SoC Root Canal! Root Cause Analysis of Power Side-Channel Leakage in System-on-Chip Designs". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.4 (2022), pp. 751–773. DOI: 10.46586/TCHES.V2022.I4.751-773. URL: https://doi.org/10.46586/tches.v2022.i4.751-773 (cit. on p. 4).

[Laa22] Petri Laarne. *polsys/ennemi: 1.1.1*. Version v1.1.1. 2022. DOI: 10.5281/zenodo.5848134. URL: https://doi.org/10.5281/zenodo.5848134 (cit. on p. 105).

[LBS19] Itamar Levi, Davide Bellizia, and François-Xavier Standaert. "Reducing a Masked Implementation's Effective Security Order with Setup Manipulations And an Explanation Based on Externally-Amplified Couplings". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.2 (2019), pp. 293–317. DOI: 10.13154/tches.v2019.i2.293-317 (cit. on pp. 23, 24).

[Luo+17] Pei Luo, Konstantinos Athanasiou, Liwei Zhang, Zhen Hang Jiang, Yunsi Fei, A. Adam Ding, and Thomas Wahl. "Compiler-Assisted Threshold Implementation against Power Analysis Attacks". In: *2017 IEEE International Conference on Computer Design, ICCD 2017, Boston, MA, USA, November 5-8, 2017*. IEEE Computer Society, 2017, pp. 541–544. DOI: 10.1109/ICCD.2017.94 (cit. on p. 92).

[MGH19] Elke De Mulder, Samatha Gummalla, and Michael Hutter. "Protecting RISC-V against Side-Channel Attacks". In: *Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019*. ACM, 2019, p. 45. DOI: 10.1145/3316781.3323485 (cit. on p. 26).

[Mic] ST Microelectronics. *STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced Arm ® -based 32-bit MCUs*. English. ARM (cit. on p. 87).

[Miy+15] Abdulaziz Miyajan, Zhijie Shi, Chun-Hsi Huang, and Turki F. Al-Somani. "Accelerating higher-order masking of AES using composite field and SIMD". In: *IEEE ISSPIT*. 2015 (cit. on p. 120).

[MM17] Thorben Moos and Amir Moradi. "On the Easiness of Turning Higher-Order Leakages into First-Order". In: *COSADE*. 2017 (cit. on pp. 96, 107, 121).

[MMT20] Lauren De Meyer, Elke De Mulder, and Michael Tunstall. "On the Effect of the (Micro)Architecture on the Development of Side-Channel Resistant Software". In: *IACR Cryptol. ePrint Arch.* (2020), p. 1297 (cit. on pp. 3, 25, 34, 37, 38, 94).

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007. ISBN: 978-0-387-30857-9 (cit. on pp. 8–10, 12, 94).

[Mor14] Amir Moradi. "Side-Channel Leakage through Static Power - Should We Care about in Practice?" In: *CHES*. 2014 (cit. on p. 121).

[MP02] Hanspeter Mössenböck and Michael Pfeiffer. "Linear Scan Register Allocation in the Context of SSA Form and Register Constraints". In: *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*. Ed. by R. Nigel Horspool. Vol. 2304. Lecture Notes in Computer Science. Springer, 2002, pp. 229–246. DOI: 10.1007/3-540-45937-5_17 (cit. on pp. 51, 56).

[MP21] Ben Marshall and Dan Page. "SME: Scalable Masking Extensions". In: *IACR Cryptol. ePrint Arch.* (2021), p. 1416. URL: https://eprint.iacr.org/2021/1416 (cit. on pp. 25, 31–34).

[MPW22] Ben Marshall, Dan Page, and James Webb. "MIRACLE: MIcRo-ArChitectural Leakage Evaluation A study of micro-architectural power leakage across many devices". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst* (2022) (cit. on pp. 3, 17, 22–24, 42, 76, 94).

[NEWa] NEWAE. *CW1200 ChipWhisperer-Pro - NewAE Hardware Product Documentation*. URL: https://rtfm.newae.com/Capture/ChipWhisperer-Pro/ (visited on 04/25/2022) (cit. on pp. 80, 100).

[NEWb] NEWAE. *CW308-STM32F2 - VCC Internal Regulator*. URL: https://rtfm.newae.com/Targets/UFO%20Targets/CW308T-STM32F/#vcc-int-supply (visited on 03/22/2022) (cit. on p. 100).

[NIS01] NIST. *Advanced Encryption Standard (AES)*. FIPS 197. NIST. 2001. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197-upd1.pdf (visited on 07/18/2023) (cit. on pp. 94, 112).

[Nyq28] H. Nyquist. "Thermal Agitation of Electric Charge in Conductors". In: *Phys. Rev.* 32 (1 July 1928), pp. 110–113. DOI: 10.1103/PhysRev.32.110 (cit. on p. 9).

[Pap+22] Kostas Papagiannopoulos, Ognjen Glamocanin, Melissa Azouaoui, Dorian Ros, Francesco Regazzoni, and Mirjana Stojilovic. "The Side-Channel Metric Cheat Sheet". In: *IACR Cryptol. ePrint Arch.* (2022), p. 253. URL: https://eprint.iacr.org/2022/253 (cit. on pp. 9, 18).

[Pap+23] Kostas Papagiannopoulos, Ognjen Glamocanin, Melissa Azouaoui, Dorian Ros, Francesco Regazzoni, and Mirjana Stojilovic. "The Side-channel Metrics Cheat Sheet". In: *ACM Comput. Surv.* 55.10 (2023), 216:1–216:38. DOI: 10.1145/3565571 (cit. on p. 20).

[PV17] Kostas Papagiannopoulos and Nikita Veshchikov. "Mind the Gap: Towards Secure 1st-Order Masking in Software". In: *COSADE*. 2017 (cit. on pp. 3, 23, 24, 27).

[Ren+11] Mathieu Renauld, François-Xavier Standaert, Nicolas Veyrat-Charvillon, Dina Kamel, and Denis Flandre. "A Formal Study of Power Variability Issues and Side-Channel Attacks for Nanoscale Devices". In: *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*. Ed. by Kenneth G. Paterson. Vol. 6632. Lecture Notes in Computer Science. Springer, 2011, pp. 109–128. DOI: 10.1007/978-3-642-20465-4_8 (cit. on p. 12).

[Rep+15] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. "Consolidating Masking Schemes". In: *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*. Ed. by Rosario Gennaro and Matthew Robshaw. Vol. 9215. Lecture Notes in Computer Science. Springer, 2015, pp. 764–783. DOI: 10.1007/978-3-662-47989-6_37. URL: https://doi.org/10.1007/978-3-662-47989-6%5C_37 (cit. on p. 16).

[Ros10] Sheldon M. Ross. *Introductory Statistics*. 3rd. 2010 (cit. on p. 19).

[RP10] Matthieu Rivain and Emmanuel Prouff. "Provably Secure Higher-Order Masking of AES". In: *CHES*. 2010 (cit. on pp. 15, 112).

[RS09] Mathieu Renauld and François-Xavier Standaert. "Algebraic Side-Channel Attacks". In: *Information Security and Cryptology - 5th International Conference, Inscrypt 2009, Beijing, China, December 12-15, 2009. Revised Selected Papers*. Ed. by Feng Bao, Moti Yung, Dongdai Lin, and Jiwu Jing. Vol. 6151. Lecture Notes in Computer Science. Springer, 2009, pp. 393–410. DOI: 10.1007/978-3-642-16342-5_29 (cit. on p. 13).

[Sha79] Adi Shamir. "How to Share a Secret". In: *Commun. ACM* 22.11 (1979), pp. 612–613. DOI: 10.1145/359168.359176. URL: https://doi.org/10.1145/359168.359176 (cit. on p. 14).

[She+21a] Madura A. Shelton, Lukasz Chmielewski, Niels Samwel, Markus Wagner, Lejla Batina, and Yuval Yarom. "Rosita++: Automatic Higher-Order Leakage Elimination from Cryptographic Code". In: *CCS '21: ACM SIGSAC*. 2021 (cit. on pp. 25, 27, 29, 30, 37, 39, 90).

[She+21b] Madura A. Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. "Rosita: Towards Automatic Elimination of Power-Analysis Leakage in Ciphers". In: *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021. URL: https://www.ndss-symposium.org/ndss-paper/rosita-towards-automatic-elimination-of-power-analysis-leakage-in-ciphers/ (cit. on pp. 4, 23–25, 27, 29, 30, 37, 39, 90).

[SLP+19] Patanjali SLPSK, Prasanna Karthik Vairam, Chester Rebeiro, and V. Kamakoti. "Karna: A Gate-Sizing based Security Aware EDA Flow for Improved Power Side-Channel Attack Protection". In: *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2019, Westminster, CO, USA, November 4-7, 2019*. Ed. by David Z. Pan. ACM, 2019, pp. 1–8. DOI: 10.1109/ICCAD45719.2019.8942173 (cit. on p. 26).

[SLP05] Werner Schindler, Kerstin Lemke, and Christof Paar. "A Stochastic Model for Differential Side Channel Cryptanalysis". In: *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*. Ed. by Josyula R. Rao and Berk Sunar. Vol. 3659. Lecture Notes in Computer Science. Springer, 2005, pp. 30–46. DOI: 10.1007/11545262_3. URL: https://doi.org/10.1007/11545262%5C_3 (cit. on p. 13).

[SM15] Tobias Schneider and Amir Moradi. "Leakage Assessment Methodology - A Clear Roadmap for Side-Channel Evaluations". In: *CHES*. 2015 (cit. on pp. 19, 20).

[SMY09] François-Xavier Standaert, Tal Malkin, and Moti Yung. "A Unified Framework for the Analysis of Side-Channel Key Recovery Attacks". In: *Advances in Cryptology - EUROCRYPT 2009, 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009. Proceedings*. Ed. by Antoine Joux. Vol. 5479. Lecture Notes in Computer Science. Springer, 2009, pp. 443–461. DOI: 10.1007/978-3-642-01001-9_26 (cit. on p. 18).

[SSG17] Hermann Seuschek, Fabrizio De Santis, and Oscar M. Guillen. "Side-channel leakage aware instruction scheduling". In: *Proceedings of the Fourth Workshop on Cryptography and Security in Computing Systems, CS2@HiPEAC 2017, Stockholm, Sweden, January 24, 2017*. Ed. by Mats Brorsson, Zhonghai Lu, Giovanni Agosta, Alessandro Barenghi, and Gerardo Pelosi. ACM, 2017, pp. 7–12. DOI: 10.1145/3031836.3031838 (cit. on pp. 25, 27, 28, 30, 37, 39, 89).

[Tso+23] Rodothea-Myrsini Tsoupidi, Roberto Castañeda Lozano, Elena Troubitsyna, and Panagiotis Papadimitratos. "Securing Optimized Code Against Power Side Channels". In: *36th IEEE Computer Security Foundations Symposium, CSF 2023, Dubrovnik, Croatia, July 10-14, 2023*. IEEE, 2023, pp. 340–355. DOI: 10.1109/CSF57540.2023.00016 (cit. on pp. 25, 27, 28, 30, 37, 39, 89).

[VGS14] Nicolas Veyrat-Charvillon, Benoît Gérard, and François-Xavier Standaert. "Soft Analytical Side-Channel Attacks". In: *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I*. Ed. by Palash Sarkar and Tetsu Iwata. Vol. 8873. Lecture Notes in Computer Science. Springer, 2014, pp. 282–296. DOI: 10.1007/978-3-662-45611-8_15 (cit. on p. 13).

[VRM17] Jo Vliegen, Oscar Reparaz, and Nele Mentens. "Maximizing the throughput of threshold-protected AES-GCM implementations on FPGA". In: *IEEE IVSW*. 2017 (cit. on p. 120).

[VS09] Nicolas Veyrat-Charvillon and François-Xavier Standaert. "Mutual Information Analysis: How, When and Why?" In: *CHES*. 2009 (cit. on p. 12).

[Wan+16] Weijia Wang, François-Xavier Standaert, Yu Yu, Sihang Pu, Junrong Liu, Zheng Guo, and Dawu Gu. "Inner Product Masking for Bitslice Ciphers and Security Order Amplification for Linear Leakages". In: *CARDIS*. 2016 (cit. on p. 121).

[Wat+20] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture*. Tech. rep. University of Cambridge, Oct. 2020. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-951.pdf (cit. on p. 30).

[WSW19] Jingbo Wang, Chungha Sung, and Chao Wang. "Mitigating power side channels during compilation". In: *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. Ed. by Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo. ACM, 2019, pp. 590–601. DOI: 10.1145/3338906.3338913 (cit. on pp. 25, 27, 28, 30, 89).

[Wu+22] Qianmei Wu, Wei Cheng, Sylvain Guilley, Fan Zhang, and Wei Fu. "On Efficient and Secure Code-based Masking: A Pragmatic Evaluation". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.3 (2022), pp. 192–222 (cit. on pp. 38, 121).

[ZMM23] Jannik Zeitschner, Nicolai Müller, and Amir Moradi. "PROLEAD_SW Probing-Based Software Leakage Detection for ARM Binaries". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2023.3 (2023), pp. 391–421. DOI: 10.46586/TCHES.V2023.I3.391-421. URL: https://doi.org/10.46586/tches.v2023.i3.391-421 (cit. on p. 4).

[Zon+18] Davide Zoni, Alessandro Barenghi, Gerardo Pelosi, and William Fornaciari. "A Comprehensive Side-Channel Information Leakage Analysis of an In-Order RISC CPU Microarchitecture". In: *ACM Trans. Design Autom. Electr. Syst.* 23.5 (2018), 57:1–57:30. DOI: 10.1145/3212719. URL: https://doi.org/10.1145/3212719 (cit. on p. 3).