ORSHIN

D3.3

Models for formal verification

Project number:	101070008
Project acronym:	**ORSHIN**
Project title:	Open-source ReSilient Hardware and software for Internet of thiNgs
Project Start Date:	1st October, 2022
Duration:	36 months
Programme:	HORIZON-CL3-2021-CS-01
Deliverable Type:	R – Document, report
Reference Number:	CL3-2021-CS-01 / D3.3 / 1.0
Workpackage:	WP 3
Due Date:	June 2025 - M33
Actual Submission Date:	30 June 2025
Responsible Organisation:	KUL
Editor:	Benedikt Gierlichs, Frank Piessens
Dissemination Level:	PU – public
Revision:	1.0
Abstract:	This deliverable reports on the results of WP3. The WP concluded successfully, and several results on countermeasures against side-channel attacks were obtained and published. To handle the problem of micro-architectural side-channel leakage, we develop hardware models that capture the security guarantees that processors offer, thus making it possible to formally verify the security of software running on these processors. Concerning physical side-channels, we examine gaps between theoretical models and practical implementations, we develop a leakage analysis tool, and we propose several new countermeasures.
Keywords:	Side-channel attacks, countermeasures, formal verification

Funded by the European Union under grant agreement no. 101070008. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

Editor

Benedikt Gierlichs, Frank Piessens(KUL)

Contributors (ordered according to beneficiary numbers)

Marton Bognar, Lesly-Ann Daniel, Job Noorman (KUL)
Josep Balasch, S. V. Dilip Kumar, Ingrid Verbauwhede (KUL),
Maria Chiara Molteni (SEC)

Reviewers

Guido Bertoni (SEC)
Aurélien Francillon (ECM)

Disclaimer

Executive Summary

This deliverable reports on the research activities carried out in Work Package 3 “Models for formal verification” of the ORSHIN project. More precisely we report on the research activities carried out in task 3.1 “Models for formal reasoning about software and micro-architectural side-channel leakage in processors” and in task 3.2 “Models for formal verification of resistance of open-source cryptographic hardware against physical side-channel and fault injection attacks”.
To handle the problem of micro-architectural side-channel leakage, we developed hardware models that capture the security guarantees that processors offer in terms of leakage, thus making it possible to formally verify the security of software running on these processors. We have defined three such models. The first model, ProSpeCT, addresses the problem of speculative execution attacks by combining hardware support for speculative taint tracking with constant time programming at the software level. The second and third model address control flow leakage attacks. The AMi model provides principled architectural support for balancing and linearizing code, important techniques that are used to make software programs constant time. The Libra model complements this with processor support that makes balancing secure on a wider range of processors. For each of the three processor models, we also designed, implemented and evaluated hardware extensions to a RISC-V open-source processor that make the processor compliant with the model, and we develop verifiable programming models that make use of the proposed processor extensions to achieve useful end-to-end security guarantees. In summary, task 3.1 has been very successful: each of the three models developed in the task has been published at a top-tier security conference, and full prototype implementations are open-source available and serve as the basis for the ORSHIN prototype deliverable D3.1.
Concerning physical side-channels, security and implementation cost are of primary importance for IoT devices, which are the focus of the ORSHIN project. Security does not come for free, and it is important to explore how much the cost of a secure implementation can be reduced.
We designed, implemented and manufactured a real silicon chip featuring three case studies of state-of-the-art countermeasures, in order to examine gaps between security guarantees provided by theoretical models and practical implementations. We also performed comparative experiments with state-of-the-art countermeasures on FPGA. In both cases our goal was to gain deeper insight into discrepancies and help bridge the gap between theory and practice, which is a primary objective of the ORSHIN project.
We have developed and implemented an open-source tool capable of analyzing hardware designs for potential side-channel leakage. The entire workflow leading up to the use of the tool is carried out using open-source electronic design automation tools, aligning with the objectives of the ORSHIN project.
We have also developed several new countermeasures. The first countermeasure challenges an assumption that is frequently made in current models for formal verification, is secure in practice, and leads to reduced implementation cost.
The second countermeasure is tailored for applications with a strict requirement for low latency. In such applications low latency is prioritized at the cost of greater chip area or higher randomness cost, but they remain secondary design goals. We also implemented and evaluated our countermeasure. It offers first-order security, is provable secure, and leads to reduced implementation cost. Our prototype circuits are formally verified and secure in practice.
The third countermeasure is an extension of the second countermeasure to higher security orders. Also here we designed the countermeasure and implemented and evaluated prototype circuits in practice. The countermeasure provides provable higher-order security, and reduced implementation cost compared to the state-of-the-art. Our prototype circuits are formally verified

and secure in practice.
In summary, task 3.2 has been very successful: each of the three countermeasures developed in the task has been published at a top-tier conference or journal, and several prototype implementations of two countermeasures are available under an open-source license and served as basis for the ORSHIN demonstrators reported in D3.2.

Table of Content

List of Figures

List of Tables

Chapter 1

Introduction

This deliverable contains some material from the interim deliverable iD3.3 provided half way through the project and a lot of novel material. More precisely, Section 2.1 in this deliverable corresponds to Chapter 2 in iD3.3 and Section 3.1 in this deliverable corresponds to Chapter 3 in iD3.3. All other chapters and sections in this deliverable were added or updated since iD3.3.

The ORSHIN project studies open source resilient hardware and software for the Internet of Things.
In this deliverable we report on the research activities carried out in Work Package 3 “Models for formal verification” up to project month 33.
More precisely we report on the research activities carried out in task 3.1 “Models for formal reasoning about software and micro-architectural side-channel leakage in processors” and task 3.2 “Models for formal verification of resistance of open-source cryptographic hardware against physical side-channel and fault injection attacks”. Following a general introduction into the topic of side-channels the structure of the document then mainly follows the two-way split.
The security of cryptographic algorithms is typically analyzed in the so called black-box model. In this model the cryptographic algorithm is an abstract object which the adversary cannot access. The adversary can only observe or choose inputs to the black-box and observe outputs. The algorithm is thus analyzed as an abstract mathematical object.
However, a deployment of the algorithm requires it to be implemented in software or hardware. And such implementations offer the adversary a broader attack surface than the black-box model. Implementations provide an adversary with additional information about the internal processing of the algorithm through what is called side-channels. In the following we distinguish software side-channels and physical side-channels.
The same observation can be made for other, non-cryptographic security building blocks, for instance a password check. In the black-box model the password check algorithm receives an input and compares it to the correct password. If they match it outputs “OK” and else it outputs “not OK”. An adversary is not supposed to learn more than this binary response. An implementation of such a password check is, however, likely to provide additional information to an adversary. For example the execution time may reveal how many characters were guessed correctly, which dramatically simplifies a password search.

Chapter 2 deals with micro-architectural side-channels. Micro-architectural side-channel attacks are mainly relevant for shared computation platforms, where several parties can be running software on the same processor. In such a scenario, an attacker program running on the platform can observe what a victim program is doing by exploiting optimization techniques such as caching,

pipelining, branch prediction, and speculative and out-of-order execution. These optimizations require the processor implementation (or *micro-architecture*) to maintain state, like the contents of the cache, or the state of the branch predictor. The victim program running on the processor has effects on this micro-architectural state, and these effects can in turn be observed by an attacker program running on the same processor.
These micro-architectural attacks, including classic attacks like cache attacks, and more recent attacks like transient execution attacks, are an important threat to the confidentiality of software running on a shared platform.
In Chapter 2, we explain how the ORSHIN project contributes countermeasures to such attacks.

Chapter 3 deals with physical side-channels. Processors executing code, and hardware circuits, require a certain amount of time, consume a certain amount of power, and emit a certain amount of electromagnetic radiation in order to perform operations. These physical observables thus carry information about what the processor or the circuit is doing. For typical IoT devices which are the focus of the ORSHIN project it is particularly true that an adversary may have physical access to them, and will be able to measure such physical quantities. It is thus naturally important to protect implementations against physical side-channel attacks. In Chapter 3, we explain how the ORSHIN project contributes to countermeasures against these attacks.

Finally, Chapter 4 provides a summary of this deliverable as well as conclusions and an outlook on future activities.

Chapter 2

Models for formal reasoning about software and micro-architectural side-channel leakage in processors

Task 3.1 of the ORSHIN project develops hardware models that capture the security guarantees that processors offer, thus making it possible to formally verify the security of software running on these processors.
Our objective is to define such models, to design hardware extensions to a RISC-V open-source processor to make the processor compliant with these models, and to develop verifiable software programming models that provably protect against well-specified classes of attacks on processors compliant with these models.
The project has developed three such models:

- ProSpeCT: a generic formal processor model providing provably secure speculation for the constant-time policy.
- AMi: a model of a processor that supports *mimic execution*, a processor mode introduced specifically to enable defending against microarchitectural side channels by software-based balancing or linearization of program code.
- Libra: a model of a processor with architectural extensions to enable secure balancing of code even on high-end processors.

For each of these three models, the project has developed the theory (with each model published in a peer-reviewed paper at one of the top tier conferences in the field [55, 171, 170]), implemented a prototype processor that is compliant with the corresponding model, and made the prototype available open-source. The corresponding prototypes are part of deliverable D3.1 of the project.
This deliverable provides an informal introduction to each of these three models, pointing to the corresponding papers for a more detailed and formal account.

2.1 ProSpeCT: Provably Secure Constant-Time Speculation

The first model, and prototype processor, was already completed in the first half of the project. The class of micro-architectural attacks that we protect against with this processor are speculative execution attacks.

More specifically, we propose ProSpeCT, a generic formal processor model providing provably secure speculation for the constant-time policy. For constant-time programs under a *non-speculative* semantics, ProSpeCT guarantees that speculative and out-of-order execution cause no microarchitectural leaks. This guarantee is achieved by tracking secrets in the processor pipeline and ensuring that they do not influence the microarchitectural state during speculative execution. Our formalization covers a broad class of speculation mechanisms, generalizing prior work. As a result, our security proof covers all known Spectre attacks, including load value injection (LVI) attacks.
In addition to the formal model, we provide a prototype hardware implementation of ProSpeCT on a RISC-V processor and show evidence of its low impact on hardware cost, performance, and required software changes. In particular, the experimental evaluation confirms our expectation that for a compliant constant-time binary, enabling ProSpeCT incurs no performance overhead.
The results reported in this Chapter of the deliverable were published in the Usenix Security 2023 conference [55], and the prototype of the processor is made available open-source at `https://github.com/proteus-core/prospect`.

2.1.1 Introduction

It is well-understood that microarchitectural optimization techniques commonly used in processors can lead to security vulnerabilities [68]. One of the most recent and challenging problems in this space is the family of Spectre attacks [95], which abuse speculative execution to leak secrets to an attacker that can observe parts of the microarchitectural state of the platform on which the victim is executing.
In response to the discovery of Spectre, a wide range of countermeasures has already been proposed [39, 14, 157, 134, 45, 64, 154, 90, 107, 17, 6, 146, 145, 167, 175, 179, 155, 91]. It is an important and difficult challenge to understand the trade-offs offered by these mitigations in terms of security, performance, and applicability to legacy hardware or software.
On the one hand, software countermeasures targeting specific transient execution attacks can still leave other attacks unmitigated [54], and they must be patched every time new speculation mechanisms are introduced (e.g., the predictive store forwarding feature newly introduced in AMD Zen3 processors [13]). On the other hand, mainstream hardware mitigations have been recently shown ineffective [18] against Spectre-v2 (BTB) attacks [95].

Hardware-based secure speculation In a recent paper, Guarnieri et al. [81] propose *hardware-software contracts* to compare hardware-based mechanisms for secure speculation and better understand how these defenses can enable software to provide end-to-end security guarantees. For instance, they show that certain types of hardware-level taint tracking [179, 167, 17] provide secure speculation for the *sandboxing* policy. On processors implementing one of these mechanisms, the software can simply enforce the sandboxing policy under a non-speculative semantics and does not need to consider the (error-prone and possibly expensive) placement of software speculation barriers.
However, none of the hardware defenses studied under the hardware-software contract framework enable secure speculation for the constant-time policy, except for completely disabling speculative execution. Hence, the classic cryptographic constant-time programming model [12] does not suffice to guarantee security on processors with these countermeasures, and significantly more complex and costly software programming models are required to recover security [116, 79, 43, 80, 54, 23, 162].

Problem statement We investigate how to provide efficient provably secure speculation for the constant-time policy under a wide range of speculation mechanisms. Specifically, we apply the

hardware-software contract framework to another class of hardware taint-tracking mechanisms explicitly tracking *secrecy* of data in the microarchitecture (e.g., systems like ConTExT [150], SpectreGuard [66], or SPT [49]). In such systems, a constant-time program informs the processor about which memory cells contain secret data. Using this additional information, hardware-based taint-tracking can provide *stronger* security guarantees than sandboxing approaches [81]. Additionally, we consider a wide variety of speculation mechanisms, whereas the model of Guarnieri et al. considers only speculation on conditional branches.

Our proposal The main contribution of this work is ProSpeCT, a generic processor model formalizing the essence of such secrecy-tracking hardware mechanisms and a proof that it provides secure speculation for the constant-time policy. Specifically, off-the-shelf constant-time cryptographic libraries can be run securely on ProSpeCT without additional protections for transient execution attacks.
ProSpeCT is modular in the implementation of predictors and covers a broad class of speculation mechanisms, including branch prediction and store-to-load forwarding. As a novel aspect, ProSpeCT additionally covers new mechanisms like predictive store forwarding [13] and even mechanisms that are not (yet) implemented in commercial processors, such as load value prediction [109] or value prediction [108]. In particular, we rigorously show that ProSpeCT protects against Spectre-v2 (BTB) attacks [95], for which mainstream hardware mitigations have recently been shown ineffective [18]. As evidence for generality, we show that our mechanism even protects against Load Value Injection (LVI) attacks [38], which are particularly challenging to mitigate. Another novel aspect of our formalization is the statement of our security condition, which allows a program to declassify a ciphertext while still requiring the processor to make sure that the attacker does not learn anything about the plaintext or the key used to compute the ciphertext.
To demonstrate the viability of our proposed mechanism, we extend a RISC-V processor to be ProSpeCT-compliant and quantify the hardware costs. Results show that the overhead of ProSpeCT in area usage and critical path is reasonable. We also demonstrate that the required software changes to cryptographic code are minimal and that the performance impact is negligible if secrets are precisely annotated. Our prototype is the first non-simulated hardware implementation of a speculative and out-of-order processor that implements secure speculation for the constant-time policy.

Contribution In summary, our contributions are:

- We present ProSpeCT, the first formal processor model providing provably secure speculation for the constant-time policy. We propose a formal model of a processor that tracks secrets during execution and temporarily blocks speculative execution if secrets could leak. The model is generic; it supports a wide range of speculation mechanisms and formalizes the guarantees provided by prior hardware-based secrecy tracking mechanisms [66, 150, 49].

- We formally prove that ProSpeCT provides secure speculation for the constant-time policy, i.e., programs that comply with the classic cryptographic constant-time discipline will not leak secrets through microarchitectural channels, including in the presence of declassification. The proof holds for a large variety of speculation mechanisms, encompassing all known Spectre and LVI attacks.

- We are the first to consider load value speculation. Interestingly, our formal analysis reveals that executions resulting from *correct* load value speculation must sometimes be rolled-back to avoid attacks based on *implicit resolution-based channels* [179].

- We provide the first non-simulated hardware implementation of a processor offering secure speculation. We implement ProSpect on a RISC-V processor supporting speculation and evaluate the costs of the proposed mechanism in terms of hardware, performance, and manual effort for precisely marking secret data.

Availability Our implementation and the experimental evaluation are open-sourced at `https://github.com/proteus-core/prospect`. A technical report containing the full formalization and proofs is available at [56].

2.1.2 Problem Statement

Transient execution attacks

Modern processors rely on heavy optimizations to improve performance. They can execute instructions out-of-order to avoid stalling the pipeline when the operands of an instruction are not available. Additionally, they employ *speculation* mechanisms to predict the instruction stream. The execution of instructions resulting from a misprediction, called *transient execution*, is reverted at the architectural level, but effects on the microarchitectural state (e.g., the cache) are persistent.
Spectre attacks [95] exploit these speculation mechanisms to force a victim to leak secrets during transient execution. An attacker can mistrain predictors to force a victim into transiently executing a sequence of instructions, called a Spectre gadget, chosen to encode secrets in the microarchitectural state. Finally, the attacker can use microarchitectural attacks to extract the secret. To this day, many variants of Spectre attacks have been discovered, exploiting a wide variety of speculation mechanisms [114, 101, 84, 13, 95, 39, 140].
Transient execution may also arise from incorrect data being forwarded by faulting instructions. For instance, on some processors, the result of unauthorized loads is transiently forwarded to subsequent instructions before the load is rolled back. This mechanism has first been exploited in Meltdown-style attacks [111, 37] to exfiltrate secret data from another security domain. It is generally accepted that Meltdown-style attacks should be mitigated in hardware by preventing such forwarding from faulting loads. We consider Meltdown-style attacks out of scope for this work.
However, these faulting loads have also been exploited to *inject* incorrect data into the victim's transient execution, and, similarly to Spectre attacks, lead the victim to leak their secrets into the microarchitectural state. In particular, these so-called load value injection (LVI) attacks [38] are still possible in the presence of Meltdown mitigations zeroing out the results of faulting loads at the silicon level (i.e., LVI-NULL). LVI attacks are related to Spectre attacks that would exploit *value speculation* during loads.
We illustrate variants of Spectre and LVI attacks in Listing 2.1, where programs in Listings 2.1c to 2.1e abuse different sources of transient execution (👻) to encode `SecretVal` in the cache using the `leak` function in Listing 2.1b. After encoding, the attacker can extract the secret from the cache using cache attacks. Note that while we illustrate these attacks using a cache side-channel, transient execution vulnerabilities are independent of the microarchitectural side-channel they exploit, such as branch predictor state [50], SIMD units [151], port contention [30, 65], micro-op cache [142], etc. Consequently, the `leak(x)` function can be replaced with any other function that reveals information on the value of `x` via a timing or microarchitectural side-channel.
The **Spectre-PHT** (Pattern History Table) or Spectre-v1 variant [95] exploits the conditional branch predictor to transiently execute the wrong side of a conditional branch. For instance, in Listing 2.1c, an attacker can first mistrain the conditional branch predictor to take the branch and

```
     0 - 15: A[16]
ptr_s (16): SecretVal
17 - 16400: B[256 * 64]
```

(a) Memory

```
1  void leak(x):
2    idx ← x * 64
3    y ← load B + idx
```

(b) Encode x into the cache.

```
4  if (idx < size_A)
5    x ← load A + idx
6    leak(x)
```

(c) Spectre-PHT (v1)

```
7  f ← trusted_func
8  x ← load ptr_s
9  jmp f(x)
```

(d) Spectre-BTB (v2)

```
10 store ptr_s 0
11 x ← load ptr_s
12 leak(x)
```

(e) Spectre-STL (v4)

```
13 idx ← load trusted_idx
14 x ← load A + idx
15 leak(x)
```

(f) LVI

Listing 2.1: Examples of code snippets vulnerable to transient execution attacks. The memory layout given in Listing 2.1a where `SecretVal` is the only secret input and `ptr_s` = 16 is common to Listings 2.1c to 2.1f. indicates instructions triggering transient executions and indicates a leakage.

then call the piece of code with `idx` = 16 to make the victim transiently execute the branch, accessing `SecretVal` at line 5 and encoding it to the microarchitectural state at line 6.
The **Spectre-BTB** (Branch Target Buffer) or Spectre-v2 variant [95] exploits indirect branch prediction to transiently redirect the control flow to an attacker-chosen location. For example, the program in Listing 2.1d calls a trusted function, which performs secure computations using `SecretVal`. An attacker can mistrain the branch predictor such that, after line 9, the victim transiently jumps to the `leak` function instead of the trusted function and leaks `SecretVal`. The **Spectre-RSB** (Return Stack Buffer) variant [101, 114] is similar to Spectre-BTB but exploits target predictions for `ret` instructions.
The **Spectre-STL** (Store-To-Load-forwarding) or Spectre-v4 variant [84] exploits the fact that load instructions can speculatively bypass preceding stores. In Listing 2.1e, the secret located at `ptr_s` is overwritten at line 10, followed by a `load` to the same address, which should return 0. With Spectre-STL, the `load` may bypass the `store` at line 10 and transiently load `SecretVal`, which would then be leaked to the microarchitectural state at line 12.
Finally, **LVI** (Load Value Injection) attacks [38] exploit a faulting `load` to directly inject incorrect data into the victim's execution. For instance, in Listing 2.1f, an attacker can prepare the microarchitectural state so that the value 16 is forwarded to `idx` by the `load` instruction at line 13, hence accessing `SecretVal` at line 14 and leaking it at line 15.

Secure speculation approaches

Since transient execution attacks were discovered, several studies have focused on adapting program semantics, security policies, and verification tools to take into account the *speculative semantics* of programs and place extra software-level protections against Spectre attacks, e.g., [47, 132, 43, 174, 23, 80, 162, 165, 82, 137, 166, 54]. However, reasoning about transient execution attacks at the software level only can be burdensome and fragile. Firstly, it necessitates knowledge of microarchitectural details that are often not publicly available. Secondly, it requires changing security policies and applying software patches every time new speculation mechanisms are introduced (e.g., the predictive store forwarding feature newly introduced in AMD Zen3 processors [13]). Finally, software countermeasures targeting specific transient execution attacks can still leave the door open to other attacks [54].
Instead, we argue that, for a given policy P, enforcement mechanisms at the software level should only consider an architectural (non-speculative) semantics, while the hardware should guarantee that transient execution does not introduce additional vulnerabilities. We call this approach *hardware-based secure speculation for* P.

Hardware-based secure speculation for sandboxing A sandboxing policy isolates a potentially malicious application by restricting the memory range it can access. A program is said to

be *sandboxed* if it never accesses memory outside its authorized address range. Sandboxed programs are vulnerable to Spectre attacks, as out-of-bounds memory locations may still be accessed transiently and have their contents leaked to the microarchitectural state. As an example, the program in Listing 2.1c is sandboxed but can still access and leak out-of-bounds data when the condition is misspeculated.
Some hardware taint-tracking mechanisms [179, 167, 17] have been shown to enable secure speculation for sandboxing [81]. For instance, Speculative Taint Tracking (STT) [179] taints speculatively accessed data and prevents tainted values from being forwarded to instructions that may form a covert channel. In Listing 2.1c, STT taints the variable `x` at line 5 until the condition at line 4 is resolved. As `x` is tainted, its value is not forwarded to the insecure `load` in the `leak` function.
Unfortunately, hardware-based secure speculation for sandboxing *only protects speculatively accessed data*, meaning that secret data loaded in registers during sequential execution may still be transiently leaked. For instance, STT does not protect the program in Listing 2.1d against Spectre-BTB. At line 5, a secret is loaded during sequential execution. As a result, `x` is not tainted by STT, and its value can still be forwarded to an insecure instruction if the `jmp` is misspeculated.
Hardware-based secure speculation for sandboxing is therefore insufficient to guarantee security for programs that compute on secrets, such as cryptographic primitives. To protect these programs, we need to enable hardware-based secure speculation for the constant-time policy.

Hardware-based secure speculation for constant-time A constant-time policy specifies that program secrets should not leak through timing or microarchitectural side channels. Before the advent of transient execution attacks, the constant-time policy was enforced with a coding discipline ensuring that the *control-flow of the program*, *addresses of memory accesses*, and *operands of variable-time instructions* do not depend on secret data. This coding discipline is the de facto standard for writing cryptographic code; it has been adopted in many cryptographic libraries [28, 180, 11] and is supported by many tools, e.g., [168, 16, 12, 61, 104, 53, 44, 36, 60, 83, 169].
A standard definition for constant-time programs (i.e., programs adhering to the constant-time policy), and the one we use in this work, is the following:

Definition 1 (Constant-time program)**.** A program is constant-time if the observation trace that it produces during *sequential execution* is independent of secret data (where the observation trace records the control flow and memory accesses).

Unfortunately, adhering to this definition is insufficient to guarantee security on modern processors vulnerable to transient execution attacks like Spectre or LVI. Indeed, all programs in Listing 2.1 are constant-time according to Definition 1, but they are vulnerable to transient execution attacks.
Hardware-based countermeasures guaranteeing secure speculation for sandboxing do not guarantee secure speculation for the constant-time policy. Therefore, to enforce the constant-time policy on speculative processors, it is still necessary to insert specific protections (typically `fence` instructions or retpolines [157]) to protect against transient execution attacks. Software developers still have to reason about speculation when they want to enforce the constant-time policy. In this work, we address the problem of *providing hardware-based provably secure speculation for the constant-time policy*.

2.1.3 Informal Overview

In this section, we motivate our design choices, make explicit what guarantees have to be enforced by software, and sketch the requirements the hardware must enforce. Finally, we illustrate how ProSpeCT protects the programs in Listing 2.1.

Design choices

ProSpeCT relies on a hardware-software co-design where developers annotate their secret data, and the hardware guarantees that no information about these secrets can leak during transient execution. The design of ProSpeCT is motivated by two main objectives. The first objective is to support existing constant-time code with minimal software changes. To this end, we base our annotation and declassification mechanism on ConTExT [150] in which developers partition the memory into public and secret regions and can declassify secrets by writing them to public memory. The second objective is to support secure code while maintaining full performance benefits of speculative and out-of-order execution. Specifically, ProSpeCT delays speculative execution only when a secret is about to be leaked; hence *in constant-time programs* (which do not leak secrets) ProSpeCT *only blocks mispredicted instructions*.

Software contracts Software developers must comply with three contracts:

Contract 1. Secret memory locations are labeled.

For instance, in Listing 2.1, address `16` is labeled as *secret* (or *high*), denoted `H`, whereas other addresses are labeled as *public* (or *low*), denoted `L`.

Contract 2. The program is constant-time.

Contract 3. Secret values written to public memory are *intentionally declassified* by the program.

Contract 3 allows, for instance, cryptographic code to declassify ciphertexts. However, software developers must make sure to not unintentionally declassify secrets by writing them to public memory.
We prove in [56] that if programs comply with these three contracts, then execution on ProSpeCT does not leak secrets through timing and microarchitectural side channels.

Hardware requirements On the hardware side, ProSpeCT must realize the following:

Requirement 1. During the execution of a program, the processor tracks *security levels*. Concretely, it labels values loaded from memory with their corresponding security level (`L` or `H`) and soundly propagates these security levels during computations.

Requirement 2. The processor prevents values with security level `H` to be leaked during speculative execution. Hardware developers identify *insecure instructions* that may leak data through

1. changing the microarchitectural state,
2. influencing the program counter, or
3. exhibiting operand-dependent timing.

The processor prevents these instructions from being speculatively executed with secret operands.

Requirement 3. Predictions do not leak secret data, in particular:

1. predictor states are only updated using public values, and
2. speculations are rolled back (even the *correct* ones) when their outcome depends on secrets (otherwise, it would leak whether the public prediction is equal to the secret value).

ProSpeCT through illustrative examples

Consider the program in Listing 2.1c, assuming that `idx = 16` and the condition is misspeculated to $true$. When executing the `load` instruction at line 5, ProSpeCT tags the register `x` with the

security level corresponding to address 16, denoted $\mathtt{x} \mapsto (v\mathtt{SecretVal}{:}\mathtt{H})$ (by Req. 1).[1] Then, when the `leak` function is executed, the `load` instruction (line 3, Listing 2.1b) is blocked because it would leak a secret-labeled value during speculative execution (by Req. 2). Conversely, if register x contains a public-labeled value, i.e., $\mathtt{x} \mapsto (vv{:}\mathtt{L})$, the `load` instruction is not blocked. ProSpeCT only blocks speculative execution in a few restricted cases, namely when secret data is about to be leaked.
Notice that, contrary to sandboxing-based approaches, ProSpeCT also protects secrets loaded in architectural registers from being transiently leaked. For example, in Listing 2.1d, when the secret is loaded at line 8, x is labeled with H, which prevents the secret from being transiently leaked later (by Req. 2) if the `jmp` instruction at line 9 transiently jumps to the `leak` function.
So far, we have seen examples of ProSpeCT applied to Spectre-PHT and Spectre-BTB (Spectre-RSB is similar to the latter). The protection generalizes to any other source of speculation, such as load value prediction (which encompasses LVI and Spectre-STL). Take, for instance, the program in Listing 2.1f. Here, the source of speculation is the `load` instruction, which transiently forwards an incorrect value at line 13. Until the `load` is resolved, ProSpeCT considers the following instructions speculative. Consequently, (by Req. 2) it does not forward the secrets to the `load` in the `leak` function (Listing 2.1b, line 3) and prevents the LVI attack.
Finally, ProSpeCT also guarantees (by Req. 3) that predicted values do not depend on secrets. In particular, secret values cannot be speculatively forwarded to other instructions. For example, in Listing 2.1e, the `load` instruction at line 11 cannot speculatively load `SecretVal`, because the corresponding address is labeled as secret. Notice that ProSpeCT still allows forwarding public values.

2.1.4 Further information

For more information about ProSpeCT, including the full formal model, a description of the prototype implementation, and the experimental evaluation of the implementation, we refer the reader to the technical report [56].

2.2 Architectural Mimicry: Innovative Instructions to Efficiently Address Control-Flow Leakage

The control flow of a program can often be observed through side-channel attacks. Hence, when control flow depends on secrets, attackers can learn information about these secrets. Widely used software-based countermeasures ensure that attacker-observable aspects of the control flow do not depend on secrets, relying on techniques like *dummy execution* (for balancing code) or *conditional execution* (for linearizing code). In the current state-of-practice, the primitives to implement these techniques have to be found in an existing instruction set architecture (ISA) that was not designed a priori to provide them, leading to performance, security, and portability issues. To counter these issues, our second processor model proposes lightweight hardware extensions for supporting these techniques in a principled way. We propose (1) a novel hardware mechanism (*mimic execution*), that executes an instruction stream only for its attacker-observable effects, and suppresses (most) architectural effects, and (2) ISA support (called AMi, for *Architectural Mimicry*) and programming models to effectively use mimic execution to balance or linearize

[1] A more conservative design choice, adopted by ConTExT [150], would be to prevent such speculative loads from accessing secret memory locations and to prevent the execution of line 5. However, we formally show that secure speculation is possilbe with this more liberal design choice.

code. We show the feasibility and benefits of our proposal by implementing mimic execution and AMi for a 32-bit out-of-order RISC-V core that leaks control flow in multiple ways (via e.g., the branch predictor, instruction timings, and the data cache). Our experimental evaluation shows that the hardware cost is low (most importantly, no impact on the processor's critical path), and that AMi enables significant performance improvements. In particular, AMi reduces the overhead of state-of-the-art linearized code by 60% in our benchmarks.

2.2.1 Introduction

Control flow that depends on confidential information discloses (parts of) this information to an adversary that can observe the control flow via side channels. For instance, the outcome of a conditional branch might be inferred by measuring the execution time, one of many (micro)architectural timing measurements that expose control flow [69]. Countering this leakage in software typically relies on two classes of countermeasures. First, *code balancing* balances the two sides of a secret-dependent branch to equalize their observable behavior [5, 57, 135, 172, 34]. If the two different execution paths of a conditional branch exhibit the same observable behavior, an attacker can no longer distinguish them. Unfortunately, for most types of computing platforms this approach is insecure, but, when applicable, it can have performance benefits compared to other approaches [172, 34]. Second, *linearization* [35, 52, 118, 173] ensures that control flow does not depend on program secrets at all.
Balancing and linearization are important ingredients in state-of-practice software-based countermeasures (such as *constant-time programming* [10]), as well as in recent research prototypes [172, 34, 35, 153, 173]. They are based on techniques like *dummy execution* (i.e., using architectural no-ops with an appropriate side-channel footprint) and *conditional execution* (e.g., conditional moves). In the current state-of-practice, the primitives to implement these techniques have to be found in an existing instruction set architecture (ISA) that was not designed a priori to provide them, leading to performance, security, and portability issues.

Our proposal

In contrast to the above, this work investigates how to offer hardware support and a small ISA extension to support control-flow balancing and linearization in a *principled* way instead. We propose a novel hardware mechanism, called *mimic execution*. Mimic execution can be thought of as a *mode* in which the processor executes instructions only for their attacker-observable effects, and suppresses (most) architectural effects: every instruction becomes a no-op, but a side-channel attacker cannot see the difference with a normal execution of the instruction.
Mimic execution is a powerful primitive, but using it correctly in software to obtain secure and correct code is non-trivial. We design and formally specify suitable ISA support (called AMi, for *Architectural Mimicry*) to activate and deactivate mimic execution, and we show how to use it to develop efficient and portable side-channel resistant code. We provide two implementations; for a pipelined in-order, and an out-of-order 32-bit RISC-V processor. We show that AMi enables significant performance improvements for hardened code, while only incurring low hardware costs.

Contributions

In summary, we contribute the following:

- *Mimic execution*, a novel and lightweight hardware primitive for imitating computational behavior.

- *Architectural Mimicry (AMi)*, a set of innovative instructions to control mimic execution in an efficient and portable way.
- Programming models showing how to balance and linearize control flow correctly and securely with AMi.
- A simple formal ISA model of AMi and a formal characterization of AMi programming models.
- An implementation of AMi for RISC-V.
- An experimental evaluation showing that the hardware cost is low, and that AMi enables significant performance improvements for hardened code.

We evaluate the benefits of AMi when *manually* writing hardened code, in line with the state-of-practice for writing constant-time cryptographic code. But we see very interesting avenues for future work to build compilers or binary rewriters that automatically (and provably) harden code against side channels by relying on AMi. To support and enable such future work, and to improve reproducibility of our results, our RISC-V implementation of AMi, as well as the full set of benchmarks and experiments are open sourced.

2.2.2 Problem Statement

Prior work addressing the problem of control-flow leakage via software-based side channels can broadly be categorized into two classes with the common goal that the trace of observable side effects produced by a program's execution does not depend on secrets. The first approach is based on the insight that if the code is carefully balanced in such a way that all possible targets of a single control-flow transfer induce exactly the same observable behavior, then executing the code does not reveal via side-effect observations which target has been executed [5, 57, 135, 172, 34]. Unfortunately, this *balanced form* does not prevent control-flow leakage in general as it is not possible on *all* platforms to balance out *all* side effects of a control-flow transfer. For instance, to predict the most likely target of a control flow transfer, modern CPUs are equipped with a branch predictor unit, which maintains a history of recent transfers. The predictor state encodes in a direct manner which target has been selected, and consequently, balanced control flow cannot prevent this shared microarchitectural state from being exposed. For this reason, the second approach avoids secret-dependent control flow altogether and linearizes the control flow using different techniques [25, 35, 52, 118, 141, 153, 173]. Control flow in *linearized form* always executes the instructions from all possible targets in a fixed order, but makes sure that architectural state is only modified by the instructions whose associated path condition holds. The linearized form has been adopted by both the security and the architecture community as the de facto standard to prevent applications from leaking confidential data via the control flow. Avoiding secret-dependent branches is a key principle of the constant-time programming discipline [10], which is broadly adhered to for writing security-critical code.

Performance

Both the balanced and the linearized form have in common that they rely on clever software tricks to achieve some form of dummy execution. The balanced form relies on the availability of dummy instructions (i.e., no-ops) to compensate for side-effects induced by instructions in alternate execution paths. The linearized form relies on the ability to neutralize the architectural

effects of instructions that should not be executed according to program semantics. Implementing these forms of dummy execution incurs a significant performance overhead due to the use of extra instructions and additional registers.

Security

More than 25 years after Kocher introduced the concept of timing attacks [97], it is well understood how to systematically harden applications to prevent control flow from exposing secrets: *the timing behavior and the hardware resource utilization due to a control-flow transfer must not depend on confidential data*. Unfortunately, despite this fundamental understanding, vulnerabilities of this kind are being found on a regular basis, even in high-profile code [7, 8, 117]. This is partially due to the common practice of hardening applications at the level of the source code [89], which is typically written in a high-level programming language. This enables so-called cross-layer vulnerabilities [133, 152], when lower layers such as the compiler or the underlying hardware are not made aware of the security semantics of the application.

Portability

It is determined by the underlying hardware implementation what observable side effects are exposed. Since application hardening is typically done at high abstraction levels [89], a comprehensive defense is needed that is effective for all target platforms, ranging from low-cost microcontrollers to high-end servers. Current practice adopts a worst-case adversary model and assumes that the control flow leaks in all situations and on all hardware. This is a secure assumption, but overly conservative. More importantly, it tightly couples the security policy to the source code and leaves no room to adopt more relaxed policies on simpler architectures that leak less information, which could have performance benefits. Furthermore, decoupling the security policy from the source code also improves other software qualities such as readability and maintainability.

2.2.3 Assumptions and Security Objectives

System model

Our goal is to develop an extension for widely used ISAs, such as the RISC-V RV32IM ISA used by our implementation. In our formalization, however, we use a simplified ISA called AMiL. Base AMiL (i.e., without the AMi extensions) is defined in Fig. 2.1. We assume a set of registers Regs, a set of values $\mathcal{V}$ (including memory addresses), and a set of program locations $Loc \subseteq \mathbb{N}$. We let $Inst$ be the set of instructions. A program $P : Loc \to Inst$ is a mapping from locations to instructions and $P[\ell]$ denotes the instruction at location ℓ.

(Expr) e := $v \mid x$
(Inst) i := **add** x,e_1,e_2 | **mul** x,e_1,e_2 | **beqz** e,ℓ |
call ℓ | **jmp** e | **load** x,e | **store** e_1,e_2

Figure 2.1: Syntax of base instructions where x ranges over Regs, v ranges over $\mathcal{V}$ and ℓ ranges over Loc.

An *architectural configuration* is a tuple $\langle m, r, \mathrm{pc}\rangle \in \mathcal{A}$ where $m : \mathcal{V} \to \mathcal{V}$ is a memory, which maps addresses to values; r : Regs $\to \mathcal{V}$ is a register file, which maps registers to values; and pc is the program counter, a special register pointing to the next instruction to execute. The semantics of base AMiL can be defined straightforwardly as a transition system over configurations.

Attacker model

We consider software that manipulates secrets such as cryptographic keys and that aims to protect these secrets against attackers who can observe microarchitectural timing side-channels [69], revealing access to shared resources such as the instruction cache, data cache, branch predictor and TLB. Physical side-channel attacks [96], and other software-based side-channel attacks, such as fault attacks [124] and power attacks [110], are out of scope for this paper and subject of orthogonal mitigations.

Leakage model

We model the observational power of an attacker by defining a *leakage model*, which we integrate in the AMiL semantics. The semantics of base instructions is given by the relation $a \xrightarrow[inst]{o} a'$. It denotes the evaluation of a base instruction $inst$ in an architectural configuration a resulting in configuration a'. Additionally, it produces an observation $o \in \mathcal{O}$ defining the architectural information that leaks through microarchitectural side channels (which we abstract from) during the evaluation of the instruction. This is similar to existing work [24]. We parameterize the semantics by a set of leakage functions $\lambda_{inst} : \mathcal{A} \to \mathcal{O}$, which define for each instruction $inst$ what parts of the architectural configuration leak. The observation trace of an n-step execution, written $a \xrightarrow[inst]{o} {}^n a'$, is the concatenation of observations produced by individual execution steps.
In this paper, we consider two countermeasures to prevent control-flow leakage: control-flow balancing and linearization. To study these two techniques, we reduce the leakage space to two leakage models by defining two versions of the leakage functions in Fig. 2.2. For both leakage models, the λ_{add} and λ_{mul} leakage functions return a fixed (i.e., configuration-independent) observation, such as the instruction latency. The functions λ_{load} and λ_{store} model the exposure of the accessed memory address (e.g., through the data cache) when executing a `load` and a `store` instruction. Finally, λ_{call}, λ_{jmp} and λ_{beqz} model the observations produced by `call`, `jmp` and `beqz` instructions, which are instantiated differently for the two leakage models:

- In the first leakage model, it is possible to avoid exposure of the program counter. An attacker can only infer the value of the program counter when the targets of a control-flow transfer produce different observations. In this model, it is secure to balance secret-dependent branches, i.e., to make sure that the different execution paths produce the same observation trace and thus remain indistinguishable by an attacker. Hence, a developer can choose between balancing and linearizing based on a profitability analysis. This model represents the leakage of low-end microcontrollers, typically not equipped with performance-enhancing hardware.
- In the second leakage model, the program counter is inevitably exposed to an attacker. In this model, it is not secure to balance secret-dependent branches. Branch elimination (by linearizing the branch) is the only secure hardening option. This model corresponds to the constant-time leakage model [10], commonly employed in security analyses. It represents the leakage of high-end processors that typically feature performance-enhancing hardware such as a branch predictor and an instruction cache.

Security objectives

The developer identifies what parts of the program state should remain secret, and the security objective of the developer is to avoid that these secrets leak to the attacker. We model this in

Common leakage

$$\lambda_{add}(\langle m, r, \mathtt{pc}\rangle) = \mathit{add} \quad \text{if } P[\mathtt{pc}] = \mathtt{add}\ \mathtt{x}, \mathsf{e1}, \mathsf{e2}$$
$$\lambda_{mul}(\langle m, r, \mathtt{pc}\rangle) = \mathit{mul} \quad \text{if } P[\mathtt{pc}] = \mathtt{mul}\ \mathtt{x}, \mathsf{e1}, \mathsf{e2}$$
$$\lambda_{load}(\langle m, r, \mathtt{pc}\rangle) = \mathit{load}\ a \quad \text{if } P[\mathtt{pc}] = \mathtt{load}\ \mathtt{x}, \mathsf{e} \text{ and } a = [\![\mathsf{e}]\!]_r$$
$$\lambda_{store}(\langle m, r, \mathtt{pc}\rangle) = \mathit{store}\ a \quad \text{if } P[\mathtt{pc}] = \mathtt{store}\ \mathsf{e1}, \mathsf{e2} \text{ and } a = [\![\mathsf{e2}]\!]_r$$

Leakage model 1 (Control flow exposure can be avoided)

$$\lambda_{call}(\langle m, r, \mathtt{pc}\rangle) = \mathit{call} \quad \text{if } P[\mathtt{pc}] = \mathtt{call}\ \ell$$
$$\lambda_{jmp}(\langle m, r, \mathtt{pc}\rangle) = \mathit{jmp} \quad \text{if } P[\mathtt{pc}] = \mathtt{jmp}\ e$$
$$\lambda_{beqz}(\langle m, r, \mathtt{pc}\rangle) = \mathit{br} \quad \text{if } P[\mathtt{pc}] = \mathtt{beqz}\ \mathsf{e}, \ell$$

Leakage model 2 (Control flow is inevitably exposed)

$$\lambda_{call}(\langle m, r, \mathtt{pc}\rangle) = \mathit{call}\ \ell \quad \text{if } P[\mathtt{pc}] = \mathtt{call}\ \ell$$
$$\lambda_{jmp}(\langle m, r, \mathtt{pc}\rangle) = \mathit{jmp}\ \ell \quad \text{if } P[\mathtt{pc}] = \mathtt{jmp}\ e \text{ and } \ell = [\![\mathsf{e}]\!]_r$$
$$\lambda_{beqz}(\langle m, r, \mathtt{pc}\rangle) = \mathit{br}\ \ell' \quad \text{if } P[\mathtt{pc}] = \mathtt{beqz}\ \mathsf{e}, \ell \text{ and } \ell' = \begin{cases} \ell & [\![\mathsf{e}]\!]_r = 0 \\ \mathtt{pc}+1 & [\![\mathsf{e}]\!]_r \neq 0 \end{cases}$$

Figure 2.2: Leakage functions for AMiL where *add*, *mul*, *load*, *store*, *call*, *jmp*, *br* are (fixed) observations. Notice that in leakage model 1 the locations ℓ are absent (i.e., only the fixed cost leaks). The expression evaluation function $[\![e]\!]_r$ evaluates expression e using register file r.

the classic way, using a lattice with two security levels: public (low) and secret (high). A security policy $\mathcal{P}$ is a mapping from registers and memory locations to security levels, identifying them as secret or public. Two configurations σ and σ' are low-equivalent with respect to a policy $\mathcal{P}$, written $\sigma =_{\mathcal{P}} \sigma'$, if they agree on the public part of their register file and memory as defined by $\mathcal{P}$. A program is *secure* if two executions starting from low-equivalent initial configurations produce the same observation trace.

Definition 2 (Secure program)**.** A program P is secure w.r.t. a security policy $\mathcal{P}$ if for all initial configurations σ_0 and σ_0', and for all n such that $\sigma_0 =_{\mathcal{P}} \sigma_0'$, $\sigma_0 \xrightarrow{o}{}^{n} \sigma_n$, and $\sigma_0' \xrightarrow{o'}{}^{n} \sigma_n'$, then we have $o = o'$.

2.2.4 Informal overview of Architectural Mimicry

We now present Architectural Mimicry (AMi). Recall that on the hardware side, we propose a new primitive, called *mimic execution*. Mimic execution imitates instructions in terms of their timing and microarchitectural behavior, but suppresses (most of) their architectural effects. It is left to the hardware designer how to mimic an instruction since this heavily depends on the implementation.

To control mimic execution in software, AMi extends the base ISA from Section 2.2.3 with qualifiers $q \in \mathcal{Q} = \{\mathtt{s}, \mathtt{m}, \mathtt{a}, \mathtt{g}, \mathtt{p}\}$ that can be associated with base instructions, denoted $q.i$. At the

```
1  if (c != 0)
2  {
3    v = 2 * a + 7
4  }
5  else
6  {
7    v = a
8  }
9
10 .
```

(a) C code

```
1  beqz  c, 5
2    ; c != 0
3    add v, a, a
4    add v, v, 7
5    jmp 6
6    ; c == 0
7    add v, a, 0
8    ...
9
10 .
```

(b) Vulnerable code

Listing 2.2: Code with secret-dependent control flow.

assembly level, each instruction has an instruction-dependent default qualifier (discussed in detail later), which can be omitted. For instance, the `add` instruction has the *standard* qualifier (`s`) by default. In machine code, the qualifier is always present. Additionally, we propose a number of programming models that rely on AMi to balance and linearize control-flow in a correct and secure way.
We informally introduce the basics of AMi and the programming models by example and refer to the published paper for more details.

Balancing branches

We assume leakage model 1, where we can securely balance secret-dependent control flow. Consider the insecure program in Listing 2.2b. This produces an observation trace that depends on the secret condition. When the branch is not taken (lines 3-5), the observation trace is $[br \cdot add \cdot add \cdot jmp]$. When the branch is taken (line 7), the observation trace is $[br \cdot add]$. As a consequence, an attacker is able to infer the outcome of the secret-dependent branch.
A solution to harden this program is to insert instructions to *balance* the two sides of the branch so that they produce the same observation trace. AMi provides hardware support to do so using *mimic instructions*, which are prefixed with the mimic qualifier `m`. A mimic instruction `m.`$inst$ produces the same observations as the instruction $inst$ but does not update the architectural state. Therefore, mimic instructions can be used to securely balance branches (with hardware guarantees), instead of relying on ad-hoc techniques using existing instructions.
The program in Listing 2.3a illustrates how AMi can be used to build a (secure) balanced version of the code in Listing 2.2b. First, notice that instructions on lines 3 and 7 are already balanced as they produce the same observation. Second, the `add` instruction on line 4 is balanced with a mimic `add` on line 8: it produces the observation *add* but does not change the value of `v`. Finally, the `jmp` instruction on line 5 is also balanced with a jump on line 9. In this version, both sides of the branch produce the same observation trace $[br \cdot add \cdot add \cdot jmp]$, while the functional behavior of the program is equivalent to the one in Listing 2.2b.

Linearizing branches

We now assume leakage model 2. Under this leakage model, the balanced program in Listing 2.3a is insecure because the conditional branch on line 1 leaks its target, resulting in an observation trace starting with $[br\, 2]$ if `c` $\neq 0$, and $[br\, 5]$ otherwise. This way, an attacker can gain information on the secret `c`. *Linearizing* the secret-dependent region by eliminating the branch

```
1  beqz   c, 5
2    ; c != 0
3    add    v, a, a
4    add    v, v, 7
5    jmp    8
6    ; c == 0
7    add    v, a, 0
8    m.add  v, v, 7
9    jmp    8
10 ...
```

(a) Balanced form

```
1  a.beqz   c, 4
2    ; Start activating region c == 0
3    add    v, a, a
4    add    v, v, 7
5    ; End activating region c == 0
6  a.bnez    c, 6
7    ; Start activating region c != 0
8    add    v, a, 0
9    ; End activating region c != 0
10 ...
```

(b) Linearized form

Listing 2.3: Balancing and linearizing a secret-dependent region with AMi instructions.

on line 1 makes the program secure. AMi provides hardware support for linearization through *activating branches*. An activating branch is a branch instruction prefixed with the activating qualifier a. An activating branch always falls through to the next instruction, but if the branch condition evaluates to true (i.e., the branch should be taken), the processor enables *mimicry mode* for the duration of the branch (i.e., until the branch target is reached). When in mimicry mode, the CPU mimics standard instructions. It is important to understand that the activating branch is not a branch per se but an instruction to efficiently linearize secret-dependent control flow, which, just like any other linearization technique, turns (insecure) control dependencies into (secure) data dependencies. Importantly, the activating branch instruction does not introduce extra sources of overhead compared to other linearization techniques. In particular, because activating branches deterministically fall through to the next instruction, there is no uncertainty about what instructions to fetch after an activating branch and the CPU can fetch and issue subsequent instructions without any delay (the code is effectively linear). Hence, it is not necessary for security reasons to delay the fetch (and stall the pipeline) until the outcome of the activating branch condition is known.
The program in Listing 2.3b illustrates how to eliminate a branch leveraging AMi. If $c = 0$, mimicry mode is enabled on line 1, the instructions on line 3 and 4 are mimicked, and line 8 is executed normally. If $c \neq 0$, lines 3 and 4 are executed normally, but the activating branch on line 6 activates mimicry mode and line 8 is mimicked. In both cases, the observation trace produced by the execution of the program is $[br\,2 \cdot add \cdot add \cdot br\,5 \cdot add]$, thus no secret information is leaked.

2.2.5 Further information

For more information about AMi, including the advanced features, the formalization, a description of the prototype implementation, and the experimental evaluation of the implementation, we refer the reader to the published paper [171].

2.3 Libra: Architectural Support For Principled, Secure And Efficient Balanced Execution On High-End Processors

Control-flow leakage (CFL) attacks enable an attacker to expose control-flow decisions of a victim program via side-channel observations. *Linearization* (*i.e.* elimination) of secret-dependent control flow is the main countermeasure against these attacks, yet it comes at a non-negligible cost. Conversely, *balancing* secret-dependent branches often incurs a smaller overhead, but is

notoriously insecure on high-end processors. Hence, linearization has been widely believed to be *the only* effective countermeasure against CFL attacks. In this work, we challenge this belief and investigate an unexplored alternative: how to securely balance secret-dependent branches on higher-end processors?
We propose Libra, a generic and principled hardware-software co-design to efficiently address CFL on high-end processors. We perform a systematic classification of hardware primitives leaking control flow from the literature, and provide guidelines to handle them with our design. Importantly, Libra enables secure control-flow balancing without the need to disable performance-critical hardware such as the instruction cache and the prefetcher. We formalize the semantics of Libra and propose a code transformation algorithm for securing programs, which we prove correct and secure. Finally, we implement and evaluate Libra on an out-of-order RISC-V processor, showing performance overhead on par with insecure balanced code, and outperforming state-of-the-art linearized code by 19.3%.

2.3.1 Introduction

In recent years, software-based microarchitectural attacks [69, 112] have emerged as a critical security threat. When multiple stakeholders run code on the same computing device, this type of side-channel attack makes it possible for an attacker to infer program secrets just by monitoring from software how a victim uses shared hardware such as the cache, branch predictor, or prefetcher.
Of special interest to this work are so-called *control-flow leakage (CFL) attacks* [33, 48, 105, 118, 136, 160, 178] whereby an attacker tries to expose the program counter (PC) trace of a victim program via side-channel observations with the aim of revealing the outcome of conditional control-flow decisions. The program's conditional control flow exposes the outcome of the condition that determines the control flow, which poses a security threat if that condition depends on secret information. In the presence of a microarchitectural attacker, a program's control flow can, in general, be observed in the microarchitectural state of shared hardware or through contention.
A possible software countermeasure against CFL attacks is *control-flow balancing* [5, 34, 57, 100, 135, 172], a program transformation which aims to make the execution of all possible targets of a control-transfer instruction appear the same to an attacker. So far, control-flow balancing has been shown to be secure only for a class of low-power embedded processors [34, 172]. This is because modern superscalar processors feature critical performance-enhancing hardware that maintains state as a function of the PC, thus leaking the PC in an unbalanceable way when this hardware is shared between different security domains. For this reason, it is widely accepted that, to counter CFL attacks on higher-end processors, programs must be PC-secure [118], i.e., their PC should be independent from secret information. PC-secure programs are created by avoiding secret-dependent control flow and the techniques for doing so are well-documented in the literature [35, 118, 141, 161, 173].
Unfortunately, this advice has not been questioned much. Over the years, it has been evolving into a dogma and it has become an established practice to hardcode it in *constant-time* [10] source code, preventing the adoption of more relaxed policies (for simpler architectures or for weaker attacker models). Furthermore, this trend creates the fallacy that secret-dependent control flow is inherently insecure and, consequently, it discourages the search for novel mechanisms to securely execute PC-insecure programs on higher-end processors.
On the other hand, there still exists a strong desire to keep the secret-dependent control flow for performance reasons, even on high-end processors. Vendors of cryptographic libraries, for instance, sometimes take the risk and do balance secret-dependent branches [178] instead of

eliminating them. As another example, numerous offensive research papers have been published that develop new CFL attacks, accompanied by ad-hoc defenses, which are later found to be vulnerable by other offensive research, a trend that has been recently described as *the CFL arms race* [178].

Our Proposal In this work, we challenge the widely-held belief that secret-dependent control flow is inherently insecure on high-end processors and propose a well-founded hardware-software co-design for secure and efficient balanced execution. In contrast to prior works that target a single vulnerability and propose ad-hoc, incremental defenses, we propose a principled solution that addresses the CFL problem in a generic way with the goal of ending the CFL arms race. Also in contrast to prior works, we do not assume a simple processor pipeline and scheduling but support modern out-of-order processor designs.
We conduct a rigorous analysis of how hardware optimizations leak a program's control flow. A key finding is that hardware optimizations can be partitioned into two categories; those that yield *balanceable observations* and those that yield *unbalanceable observations*. Balanceable observations can be securely balanced by software-only approaches. Unbalanceable observations require hardware support. Based on the findings of our analysis, we propose Libra, a hardware-software security contract that lays the principled foundation for secure balanced execution. We introduce a novel memory layout, called *folded layout*, and an algorithm for *folding* balanced code regions, which makes it possible to keep enabled performance-critical hardware optimizations without compromising security. Additionally, we propose an ISA extension for executing folded regions.
In a nutshell, we make the following contributions:

- A novel hardware-software contract, called Libra, for secure and efficient balanced execution.
- A formalization of the ISA-level semantics of Libra and security and correctness proofs of our folding algorithm.
- A characterization of hardware optimizations regarding how they leak a program's control flow.
- Recommendations for hardware designers wishing to adopt Libra to their designs.
- An implementation of Libra on an out-of-order RISC-V core.
- An experimental evaluation showing that balanced execution is secure and efficient at a low hardware cost.

Additional material Our RISC-V implementation and evaluation are available on GitHub: `https://github.com/proteus-core/libra`.

2.3.2 Terminology and Background

We first define relevant terminology from the fields of graph theory and compiler construction and then introduce some new vocabulary (marked with $*$).

Definition 3 (Basic block)**.** A basic block is a straight-line instruction sequence always entered at the beginning and exited at the end.

```
En: br a0,t,f
 t:    br a1,tt,tf
tt:     add s1,s2,s3
        j Ex
tf:     add s2,s3,s4
        j Ex
 f:    sub s1,s2,s3
       j Ex
Ex: [...]
```

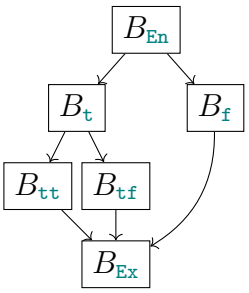


Figure 2.3: A program and its CFG.

In other words, the instructions of a basic block are always executed one after another, in a sequence. Only the first instruction can be the target of a branch, and only the last instruction can be a branch.

Definition 4 (Control-flow graph)**.** A control-flow graph (CFG) is a directed graph that represents all the paths that might be traversed through a program during its execution. The nodes of a CFG represent basic blocks, the edges represent control-flow transfers.

Without loss of generality, we assume that a CFG has a unique *entry* and a unique *exit* block. We also assume that the last instruction in a basic block is a control-transfer instruction, which designates the possible successor blocks. We refer to this instruction as the *terminating instruction* of the basic block. Figure 2.3 contains an illustration of a CFG with B_{En} the entry basic block and B_{Ex} the exit basic block.

Definition 5 (Distance)**.** The distance between two basic blocks in a CFG is the number of edges in a shortest path connecting them.

In Fig. 2.3, the distance between the basic blocks B_{En} and B_{Ex} is 2 ($B_{\text{En}} \rightarrow B_{\text{f}} \rightarrow B_{\text{Ex}}$). The distance between two instructions is defined similarly by considering individual instructions as basic blocks.

Definition 6 (Postdominance)**.** A basic block Y postdominates a basic block X (*i.e.* Y is a postdominator of X) if all paths from X to the exit block go through Y.

The closest postdominator of a basic block is called its *immediate postdominator*. In Fig. 2.3, basic block B_{Ex} postdominates basic block B_{En}. It is also the immediate postdominator of B_{En}.

Definition 7 (Level structure)**.** The level structure of a CFG is a partition of the basic blocks into subsets (levels) that have the same distance from the entry basic block.

The level structure of the CFG in Fig. 2.3 consists of three levels: $L_0 = \{B_{\text{En}}\}, L_1 = \{B_{\text{t}}, B_{\text{f}}\}, L_2 = \{B_{\text{tt}}, B_{\text{tf}}, B_{\text{Ex}}\}$.

Definition 8 ($*$Level slice)**.** The set of equidistant instructions for a distance δ with respect to basic block B forms the level slice (or simply *slice*) determined by the tuple (B, δ)

In Fig. 2.3, the slice of distance 0 is {`br a0,t,f`} and the slice of distance 1 is: {`br a1,tt,tf; sub s1,s2,s3`} (both relative to B_{En}).

Definition 9 ($*$Secret-dependent region)**.** The set of basic blocks between a secret-dependent control-transfer instruction $inst$ and its immediate postdominator form the secret-dependent region determined by $inst$.

We refer to the basic block containing the secret-dependent control-transfer instruction as the *entry block* of the region, and to its immediate postdominator as the *exit block* of the region. In Fig. 2.3, if `a1` is secret (line `t`), then $\{B_{\text{tt}}, B_{\text{tf}}\}$ is the secret-dependent region determined by the instruction on line `t`. The entry block of the region is B_{t}, the exit block B_{Ex}. Similar to the level

structure of a CFG, we define the *level structure of a secret-dependent region* as the partition of its basic blocks into subsets (levels) that have the same distance from the region's entry block.

Control-Flow Leakage Attacks

CFL attacks are a type of microarchitectural attack whereby an attacker tries to learn the outcome of a secret-dependent branch by exposing the control flow via microarchitectural side channels. Consider the program in Listing 2.4a. When the branch on line 1 evaluates to true, the instructions on lines 2-3 are executed and the program exits. When the branch evaluates to false, the instruction on line 4 is executed and the program exits. An attacker that is able to observe the program's execution time will be able to distinguish the two executions, and hence learn if `secret` evaluates to true or false.

Listing 2.4: Code vulnerable to CFL attacks (Listing 2.4a) and its balanced version (Listing 2.4b).

```
1       br secret,t,f
2  t:    add s1,s2,s3
3        j Ex
4  f:    add s2,s3,s4
5
6  Ex:  [...]
```

(a)

```
1       br secret,t,f
2  t:    add s1,s2,s3
3        j Ex
4  f:    add s2,s3,s4
5        j Ex
6  Ex:  [...]
```

(b)

Besides this start-to-end timing difference, interrupt latency [160], data cache contention [131], structural dependencies [9] or data dependencies stalling the pipeline are other examples of microarchitectural events that can be monitored by an attacker to leak the control flow. Consider Listing 2.4a again and assume that the addresses of the `add` instructions (lines 2 and 4) map to different instruction cache lines. Monitoring which cache line has been touched (for instance with the Flush+Reload attack [176]) will reveal the control flow.
Two common software countermeasures against CFL attacks are *control-flow balancing* and *control-flow linearization*. The former technique keeps the secret-dependent control flow intact while the latter eliminates it completely.

Control-Flow Balancing

Control-flow balancing is based on the idea that if the two sides of a secret-dependent branch induce exactly the same attacker-observable behavior, then executing the code does not reveal via side channels which side of the branch has been executed. Listing 2.4b gives the balanced form of Listing 2.4a. The `add` instruction on line 2 is balanced with the `add` instruction on line 4 and a jump instruction is added to the `f` path on line 5 to balance it with the jump on line 3 in the `t` path.
Recent work [34, 172] has demonstrated the security (and efficiency) of control-flow balancing for small, embedded processors with deterministic timing behavior. The authors propose a methodology consisting of three steps. First, by profiling the microarchitecture, the instruction set is classified into a number of *leakage classes* such that executing instructions from the same leakage class induces the same side-channel observations. Second, a dummy (no-op) instruction is composed for every leakage class. Lastly, the secret-dependent branches are algorithmically balanced [172] with respect to the leakage classification, and by inserting dummy instructions when

necessary. This approach ensures that the dynamic instruction trace of balanced code always produces the same sequence of leakage classes.
Although control-flow balancing counters attacks exploiting microarchitectural optimizations on low-end devices [117, 160], higher-end devices (the target of our work) typically feature optimizations yielding observations that are unbalanceable in software alone. Yet, for performance reasons, balanced control flow is sometimes found in security-critical libraries targeting these devices [117, 178]. Thus, how to make balanced execution secure on these higher-end devices remains an important research question.

Control-Flow Linearization

Control-flow linearization is a key principle of the widely-established constant-time programming discipline [10]. By eliminating secret-dependent branches, control-flow linearization ensures that the PC does not get tainted (i.e., that the PC trace is independent of secrets). Several linearization techniques have been proposed in the literature [35, 118, 141, 153, 161, 173].
Listing 2.3 contains the linearized form of the running example from Listing 2.4a, based on a state-of-the-art method that was first proposed by Molnar et al. [118]. Compared with the balanced form from Listing 2.4b, the linearized form comes with a higher cost due to the use of additional instructions and registers.

```
1  seqz  t1,secret
2
3  addi  t1,t1,-1   # t1 = true mask  (in {0xffff, 0x0000})
4  not   t2,t1      # t2 = false mask (in {0xffff, 0x0000})
5  and   t3,s1,t1   # start of else
6  add   s1,s2,s3
7  and   s1,s1,t2
8  or    s1,s1,t3   # start of then
9  and   t3,s2,t2
10 add   s2,s3,s4
11 and   s2,s3,t1
12 or    s2,s2,t3
```

Listing 2.3: Linearized form of the vulnerable code in Listing 2.4a.

This work

The goal of this work is to make sure that executing balanced code (which contains secret-dependent control flow) on high-end processors does not leak more information than executing the equivalent linearized code (which does *not* contain secret-dependent control flow). We demonstrate that, with minimal hardware support, it is possible to securely balance secret-dependent control flow on higher-end platforms, without disabling performance-critical hardware resources that are shared between different stakeholders.

2.3.3 Threat Model

We consider an adversary with the goal to infer secrets (e.g., cryptographic keys) by learning the secret-dependent control flow of a victim application. We consider an adversary with the same capabilities as an adversary under the *classic* constant-time threat model, and thus assume that applications are hardened against transient execution attacks [40]. More specifically, an adversary with the capabilities of this threat model is able to run arbitrary code alongside an

architecturally isolated victim (e.g., via process isolation) on the same machine and it shares hardware resources, such as the branch predictor, cache hierarchy and execution units with the victim. This setting enables the adversary to precisely observe the execution time of the victim, and how it uses the shared resources. If these observations depend on the secret control flow, the adversary is able to learn something about the secret.
We consider software-based timing channels, i.e., the adversary monitors the microarchitectural resource usage via timers from software [69, 112]. Side channels that require physical access and physical equipment to measure quantities such as power consumption [96] or EM emissions [138] are out of scope for this paper. Similarly, other types of software-based side-channel attacks, such as software-based fault attacks [124] and software-based power attacks [110] are out of scope and subject of orthogonal mitigations.
We make no further assumptions on the type of (software-based) microarchitectural side-channels attacks that can be mounted by the adversary, ranging from classic cache attacks [131] to more recent contention-based attacks [9].

2.3.4 Overview of Libra

A program's control flow can leak through observations induced by various microarchitectural optimizations. Some of these observations, such as instruction latency, are independent of the value of the PC. We refer to optimizations yielding this type of observation as *sources of balanceable leakage* as their observations can be balanced by software. However, some performance-critical optimizations commonly found in modern hardware (e.g., the instruction cache and the instruction prefetcher) yield observations that are dependent on the value of the PC. They inevitably leak the control flow. We refer to these optimizations as *sources of unbalanceable* leakage as they cannot be dealt with by software alone. In the published paper [170], we study this distinction further and provide a comprehensive characterization of hardware optimizations regarding how they leak the control flow.
Existing control-flow balancing solutions are ineffective against unbalanceable leakage. It is the goal of Libra to address this gap via a novel hardware-software security contract for secure and efficient balanced execution. On the one hand, the software is responsible for balancing secret-dependent control flow under a *weak observer mode* (accounting for the balanceable leakage) in which the PC does not leak. On the other hand, the hardware provides support to deal with the sources of unbalanceable leakage to ensure that the program remains secure in a *strong observer mode*, representative of our threat model (Section 2.3.3) for high-end processors.

Leakage Contract

Libra requires the hardware to augment the ISA with a *leakage contract* that provides sufficient information on how to balance the control flow. Software, such as a compiler, can then rely on this contract 1) to securely balance secret-dependent control flow (making control-flow balancing a *principled* code transformation) or 2) to verify that secret-dependent control flow is securely balanced. This stands in contrast to prior works [5, 100, 34, 57, 172, 135], where it is the responsibility of the software to empirically figure out *how* to balance corresponding instructions.
The Libra leakage contract classifies an instruction set into two dimensions. First, it partitions instructions into *leakage classes* [34, 172] such that instructions from the same leakage class yield identical side-channel observations. Importantly, any instruction can be used to balance any other instruction from the same leakage class. For every leakage class, the contract additionally designates a canonical *dummy instruction*, which does not produce architectural effects (*e.g.*

`mv x1, x1`). Finally, the hardware provides a blocklist of instructions that are not supported in balanced regions. Blocklisted instructions have to be rewritten in terms of non-blocklisted instructions before performing control-flow balancing.
Second, the leakage contract partitions the instruction set into safe and unsafe instructions [177]. *Safe instructions* are instructions whose timing and shared microarchitectural resource usage are independent of the values of their operands. For instance, an `add` instruction is typically implemented in a safe way, while a `load` typically exposes the value of the address operand on systems with a data cache (making it an *unsafe instruction*). It is insecure to pass secrets to unsafe instructions but it is secure to use unsafe instructions in balanced regions if it can be proven that the operands of any two equidistant unsafe instructions are the same for all possible executions. For instance, the code `if (secret) load x0 a else load x1 a` is secure as the resulting observation is independent of `secret` (under the assumption that the `load` is only unsafe in its address operand).

ISA Extension

The goal of Libra is to securely execute balanced code regions on high-end CPUs without disabling performance-critical optimizations. In particular, Libra aims at keeping *all* modern hardware optimizations fully enabled when executing security-insensitive code (*i.e.*the common case), and keeping *as many optimizations as possible* in secret-dependent regions.
To this end, Libra proposes an ISA extension introducing two main novel features:

- A novel memory layout for balanced code, termed *folded layout*, which interleaves the instructions from balanced regions by placing the level slices sequentially in memory.
- A new instruction, the *level-offset branch* (`lo.br`), which informs the CPU how to navigate a folded region. Additionally, it signals to the CPU that it is about to execute a secret-dependent region such that it can adapt the behavior of some optimizations.

Importantly, even though folding sequentially lays out instructions of balanced regions in memory (reminiscent of linearization), the *original control flow of the program is preserved*, *i.e.*only one side of a folded conditional branch is executed, as prescribed by the original CFG (just like with standard code balancing).
The level-offset branch `lo.br` $c, \texttt{off}_\texttt{t} : \texttt{off}_\texttt{f} : \texttt{bbc}$ specifies how to navigate a folded region:

1. The level offsets $\texttt{off}_\texttt{t}$ and $\texttt{off}_\texttt{f}$ indicate what instructions of the next level to execute, depending on whether the condition c is true or false;
2. The basic block count `bbc` indicates the number of basic blocks of the next level (the slice size of the next level) and is used to increment the PC by the correct value.

Listing 2.4b illustrates how to fold the balanced code from Listing 2.4a. First, the two add and the two j instructions are sequentially placed in memory. Second, the conditional branch is rewritten using a `lo.br` with $\texttt{off}_\texttt{t} = 0$, $\texttt{off}_\texttt{f} = 1$ and $\texttt{bbc} = 2$. After the `lo.br`, the CPU will execute the folded region slice by slice, incrementing the PC by 2. If the condition is true, the first (offset $\texttt{off}_\texttt{t}$) instruction of each slice is executed, otherwise the second (offset $\texttt{off}_\texttt{f}$) instruction is executed. Finally, the terminating j instructions are replaced by `lo.br` instructions to reset the level offset and `bbc` and resume “normal” execution at the `Ex` label.

Listing 2.4: Balanced code (Listing 2.4a) and its folded version (Listing 2.4b).

```
1       br secret,t,f
2  t:    add s1,s2,s3
3        j Ex
4  f:    add s2,s3,s4
5        j Ex
6  Ex: [...]
```

(a)

```
1       lo.br secret,0:1:2 # offT:offF:bbc
2  L1:    add s1,s2,s3
3         add s2,s3,s4
4         lo.br zero,0:0:1 # offT:offF:bbc
5         lo.br zero,0:0:1 # offT:offF:bbc
6  Ex: [...]
```

(b)

How does Libra address unbalanceable leakage? The design of Libra is tailored to address unbalanceable leakage in hardware efficiently, *i.e.*by keeping essential hardware optimizations enabled. Yet, to establish the security guarantees, Libra requires that the PC does not leak at a finer granularity than a slice, possibly requiring adaptations to the behavior of some optimizations. Importantly, the folded memory layout is crucial to keep enabled performance-critical optimizations of modern hardware (e.g., the instruction cache) without, or with only minimal, adaptations. By virtue of folding (which creates a linear memory layout), the hardware can efficiently implement a data-oblivious instruction memory access pattern by always prefetching all the slices in the same order, effectively making it independent of the outcomes of conditional branch(es).
While some sources of unbalanceable leakage do not require hardware modifications, some will, possibly degrading performance. However, because the hardware is informed when it is executing a folded region, these modifications can be limited to folded regions only. For instance, some hardware structures, such as the branch predictor, must be disabled for the `lo.br` instruction to prevent control-flow exposure to an attacker sharing the branch predictor. However, the linear layout of a folded region makes the branch predictor unnecessary for `lo.br` instructions, because there is no uncertainty (at slice granularity) what address the sequential prefetcher should fetch from, so it can fill the cache with the instructions that are about to be fetched by the CPU.
In the published paper [170], we present, based on a rigorous study of the attack literature, a characterization of the sources of unbalanceable leakage (with folding in mind), and we provide guidelines about how to handle them.

2.3.5 Advanced Features

Nested branches

When folding a region with a nested branch (as in Listing 2.5a), the software must fold the level structure of the entire outer region, as shown in Listing 2.5b. The slice size grows with the level of nesting. In the example from Listing 2.5b, each slice of the second level consists of four instructions. Recall that the hardware has to make sure to fetch instructions without exposing their offset within the current level. For instance, if a slice occupies multiple cache lines, the hardware must ensure to always touch all the cache lines in the same order, irrespective of the current instruction's offset.
Note that when a nested branch does not depend on secret information (*e.g.* a loop with a constant trip count), it can be more efficient to keep the branch instead of folding it. In that case, for correctness, the software must ensure that the level offsets of the target instructions are consistent regarding the offsets of the branch instructions. Moreover, for security, the software must ensure that the branch targets of the branches in the source slice all point to targets in the same target slice.

Listing 2.5: Region with nested branches (Listing 2.5a) and its folded version (Listing 2.5b).

```
      br secret,t,f
  t:   br c,tt,tf
 tt:     add r,r,4
         j Ex
 tf:     add r,r,8
         j Ex
  f:   br c,ft,ff
 ft:     sub r,r,4
         j Ex
 ff:     sub r,r,8
         j Ex
 Ex: [...]
```

(a)

```
      lo.br secret,0:1:2
 L1:    lo.br c,0:1:4
        lo.br c,2:3:4
 L2:      add r,r,4
          add r,r,8
          sub r,r,4
          sub r,r,8
          lo.br zero,0:0:1
          lo.br zero,0:0:1
          lo.br zero,0:0:1
          lo.br zero,0:0:1
 Ex: [...]
```

(b)

Function calls

To support function calls in balanced code, prior work on control-flow balancing [34, 172] proposed to create a dummy function for each function called from a secret-dependent region. A dummy function is mostly made up of dummy (no-op) instructions designed to mirror the behavior of the real function. These dummy instructions ensure that both the dummy and real functions cause identical changes in the microarchitectural state. As a result, an attacker cannot distinguish between the execution of the dummy function and that of the real function. A call to a function in a secret-dependent region can then be balanced with a call to its dummy version. Libra supports this scheme, yet in order not to expose the control flow on higher-end CPUs (e.g., via the instruction cache), functions must be folded with their dummy counterpart. Libra provides hardware support to efficiently invoke a folded function and extends the ISA with a new instruction, the *level-offset call*: `lo.call` $b\ \ell$. The instruction jumps to the folded function and, according to the boolean immediate b, either executes the real part or the dummy part of the folded function. Additionally, the CPU must save/restore the Libra state (*i.e.* current offset and `bbc`) of the caller upon calls/returns. Libra proposes a two-level hardware stack, used for storing and restoring the Libra state of the caller. For non-leaf functions (i.e., to support more than one level of nesting, including recursion), the software is responsible to save and restore the Libra state on a software-based stack.

Exceptions

Instructions that may throw exceptions are inherently unsafe because whether an exception is thrown depends on the value of their operands and handling an exception impacts both the timing and resource usage of an application. Therefore, such instructions should be treated similarly to other unsafe, balanceable instructions, by balancing the unsafe operands and their dependencies.

2.3.6 Hardware-Software Security Contract

In summary, with Libra we propose a hardware-software security contract for balanced execution. If both parties fulfill their part of the contract, then executing a balanced code region will not leak more information than the equivalent linearized region.

On the hardware side, Libra imposes the following requirements:

HR1 A leakage contract for control-flow balancing is provided.

HR2 The PC does not leak at a finer granularity than a slice.

HR2a The instruction memory access pattern does not depend on the outcome of the level-offset branch (implied by **HR2**).

HR3 The level-offset branch and the level-offset call are safe instructions.

On the software-side, Libra relies on:

SR1 A correct identification of secret-dependent regions and functions called from secret-dependent regions.

SR2 A secure balancing according to a weak observer mode as prescribed by the leakage contract. In practice, this entails making sure that secrets do not directly flow to unsafe instructions, applying a balancing algorithm (such as the one from [172]), and providing dummy versions for functions called from secret-dependent regions.

SR3 A correct folding of the balanced regions and functions. In the full paper [170], we give a folding algorithm.

2.3.7 Further information

For more information about Libra, including the formalization, a description of the prototype implementation, and the experimental evaluation of the implementation, we refer the reader to the paper [170].

Chapter 3

Models for formal verification of resistance of open-source cryptographic hardware against physical side-channel and fault injection attacks

Task 3.2 of the ORSHIN project is concerned with models that capture information leakage through physical side-channels, or the sensitivity to active fault injection attacks, and that allow formal verification of security properties of open-source hardware. Current models will be investigated. Their weaknesses will be identified, and new models will be proposed. This task also comprises the development of demonstrators and practical experiments in the state-of-the-art electronics security evaluation lab of KUL.
This chapter of the deliverable reports our development of three new countermeasures against side-channel attacks and their prototype implementations.
The first countermeasure challenges an assumption that is frequently made in state-of-the-art models, satisfying which leads to an increased implementation cost. We demonstrate that our prototype implementation of a cryptographic algorithm protected by our countermeasure achieves both practical security and low implementation cost. We therefore show that it is not necessary to satisfy the assumption in order to achieve a secure and low-cost implementation. These results were published at the DATE 2023 conference [102]. An extended version of the DATE paper was published in the journal IEEE Transactions on Information Forensics and Security [103].
The second countermeasure against side-channel attacks is tailored for low-latency applications. Countermeasures against side-channel attacks come with implementation overheads, specifically secure hardware masking requires to add register stages which ultimately increases the processing time from input to output. This is hardly acceptable in some applications with a low-latency requirement, for instance memory encryption, where such an increased latency would slow down the entire system. Hardware masking for low-latency applications prioritizes low latency at the cost of greater chip area or higher randomness cost. Our countermeasure achieves the same low latency as the state of the art, or better, but with lower overheads in terms of chip area and randomness. We prove our countermeasure secure and formally verify the security of our implementations with state of the art tools. Overall we demonstrate that provable secure and formally verified implementations can have less overheads. These results were published in the journal IACR Transactions on Cryptographic Hardware and Embedded Systems [158].
The third countermeasure is an extension of the second countermeasure to higher security orders. We designed the countermeasure and implemented and evaluated prototype circuits in

practice. The countermeasure provides provable higher-order security, and reduced implementation cost compared to the state-of-the-art. Our prototype circuits are formally verified and secure in practice. These results were published in the journal IACR Transactions on Cryptographic Hardware and Embedded Systems [159].
The prototype implementations of the second and third countermeasure served as basis for the demonstrator described in deliverable D3.2, which is publicly available as open-source hardware. This chapter of the deliverbale also reports our efforts to gain deeper insight into discrepancies and help bridge the gap between theory and practice, which is a primary objective of the ORSHIN project. We designed, implemented and manufactured a real silicon chip featuring three case studies of state-of-the-art countermeasures, in order to examine gaps between security guarantees provided by theoretical models and practical implementations. We also performed comparative experiments with state-of-the-art countermeasures on FPGA.
This chapter of the deliverable also reports our work on pre-silicon open-source evaluation tools. We have developed and implemented an open-source tool capable of analyzing hardware designs for potential side-channel leakage. The entire workflow leading up to the use of the tool is carried out using open-source electronic design automation tools, aligning with the objectives of the ORSHIN project.

3.1 Low-cost first-order secure boolean masking in glitchy hardware

We describe how to securely implement the masked logical AND of two bits in hardware in the presence of glitches without the need for fresh randomness, and we provide guidelines for the composition of circuits. As a case study, we design, implement, and evaluate masked DES cores. We focus on first-order secure Boolean masking and do not aim for provable security. Our goal is a practically relevant trade-off between area, latency, randomness cost, and security. We provide two low-cost solutions. Our first solution focuses on strong security while simultaneously aiming for low implementation costs. The resulting DES engine shows no evidence of first-order leakage in a non-specific leakage assessment with 50M traces. Our second solution follows the opposite approach: we focus on lowering implementation costs, latency to be specific, while not sacrificing much on security. Our low-latency DES engine exhibits signs of first-order leakage only after approximately 15M traces.

3.1.1 Introduction

Over the last few decades, much attention has been dedicated to researching and developing fast and efficient cryptographic implementations that are secure against power analysis attacks [99]. Masking [46, 73] is a well-known technique that can be used to protect both hardware and software implementations. Its core idea is to split the data being processed by an implementation into random shares, effectively eliminating its correlation with the device's power consumption.
In this work, we focus on first-order Boolean masking, where each sensitive (intermediate) value x is randomly split into two shares x_0 and x_1 such that $x = x_0 \oplus x_1$. First-order masked implementations can, in theory, be broken with higher-order attacks, which combine leakage of multiple (or all) shares to derive sensitive values. We nevertheless focus on first-order masking because performing a successful higher-order attack can be made very difficult by adding noise (the number of traces needed increases exponentially in the attack order, with the noise factor in the basis) [46].

In hardware, masking is commonly applied at the gate level. As logic gates are used as a fundamental building block in gate-level masking, any cost reduction in building a masked logic gate significantly benefits the overall cost of a masked circuit. A significant hurdle in hardware masking is to overcome the effect of glitches, i.e. undesired signal transitions in the circuit, as they are known to temporarily reveal unmasked sensitive values [115].
A methodology for implementing a masked circuit in hardware requires at least a masked AND gadget, a masked XOR gadget and rules for the composition of gadgets to build a masked circuit. A masked XOR is easy because one can simply apply the XOR to each share separately. A masked AND is more difficult as the computation needs to involve all shares of all variables at some point. One needs to be very careful not to reveal any unmasked sensitive intermediate values, as demonstrated in many previous works. Composition is also difficult because circuit effects, uniformity of inputs and outputs, and their dependencies need to be tracked and corrected as necessary.
Modern hardware masking techniques, such as Threshold Implementation (TI) [127] and Domain-oriented Masking (DOM) [75], have been designed to address the problem caused by glitches. In contrast to classical Boolean masking, they control the propagation of glitches through register layers and maintain the uniformity of the intermediate values by injecting fresh randomness. As a result, they achieve *provable* security against first-order attacks. Threshold Implementation is shown to be provable secure [58] under the glitch-robust probing model [63]. Although DOM does not enjoy a security proof, it has shown many times to be secure in practice. Some of them further generalize to higher-orders. However provable security comes with higher costs. Protected implementations using modern masking schemes require a lot more resources in terms of area, latency, and randomness than classical Boolean masking [156].
In this work, we develop a low-cost Boolean masked AND gate to build masked circuits that provide *practical* security in the presence of glitches. Our contributions are as follows:

- Starting from the software-oriented masked AND construction by Biryukov et al. [32], we derive a low-cost AND gadget suitable for hardware implementations which requires no fresh randomness. We propose two solutions to prevent glitches by controlling the arrival time of input operands.
- We provide guidelines for composition and exemplary circuits for securely computing the logical AND of more than two terms and circuits with AND and XOR gadgets. We pay particular attention to the need to remask and explain when and how to do it.
- We design and implement two masked DES encryption engines building on the proposed low-cost AND gadget and guidelines for composition. We add security measures only where needed for practical security.
- We evaluate the performance of our designs both in terms of cost (area, latency, randomness) and first-order side-channel leakage on an FPGA platform.

3.1.2 Low-Cost Masked AND2 Gadget

If a regular AND2 computes $z = x \cdot y$, a straightforward masked AND2 could for instance compute $z_0 = x_0 \cdot y_0 \oplus x_0 \cdot y_1$ and $z_1 = x_1 \cdot y_0 \oplus x_1 \cdot y_1$ such that $z = z_0 \oplus z_1$. This would, however, not be secure because $z_0 = x_0 \cdot (y_0 \oplus y1) = x_0 \cdot y$ depends on unmasked y, and similar for z_1. A simple solution to this problem, as first proposed by Trichina [156] consists in the introduction of a fresh

random bit r in the equations:

$$\begin{aligned} z_0 &= r \oplus (x_0 \cdot y_0) \oplus (x_0 \cdot y_1) \oplus (x_1 \cdot y_1) \oplus (x_1 \cdot y_0) \\ z_1 &= r \end{aligned} \tag{3.1}$$

This construction is secure only if the order of evaluation is from left to right. A well-known problem arises when implementing such a gadget in hardware, because the order of evaluation is unknown and glitches in the combinational circuit can happen. Previous work, such as TI and DOM, provide solutions for this problem. They require fresh random bits, too. The cost in terms of the number of random bits is an important criterion when comparing masked implementations. Gross et al. [78] propose an AND2 gadget and rules for composition which allow implementing, e.g. an entire masked AES-128 using only two bits of randomness. The security of their approach was proven in the t-probing model [88]. However, in hardware, this approach leads to a significant penalty in latency. The software implementation provided by Gross et al. was found to be insecure [26]. Like Gross et al. [78] we start our work from the masked AND2 gadget proposed by Biryukov et al. [32] for software implementations:

$$\begin{aligned} z_0 &= (x_0 \cdot y_0) \oplus (x_0 + \overline{y_1}) \\ z_1 &= (x_1 \cdot y_0) \oplus (x_1 + \overline{y_1}) \end{aligned} \tag{3.2}$$

$\cdot, \oplus, +$ denote AND, XOR and OR, respectively. We refer to this gadget as `secAND2` from now on. A remarkable property of this gadget is that it does not require fresh randomness to be secure. Yet, due to the lack of a fresh mask, the output is not independent of the input, which needs to be considered during composition. Another advantage of the `secAND2` gadget over the one by Trichina is that it requires fewer elementary logic operations (AND, XOR, etc.) and will thus lead to a faster implementation in software or a smaller implementation in hardware. While Gross et al. aimed for provable security in the presence of glitches with minimal randomness requirements, we strive for an overall practically relevant tradeoff between area, latency, randomness cost and security.

Secure Hardware Implementation of `secAND2`

A straightforward ASIC implementation of `secAND2` using a common standard cell library, i.e. using AND2, XOR2, OR2 and INV gates, will be insecure due to glitches in the circuit. A similar problem occurs when implementing the logic equations on FPGAs using Look Up Tables (LUT). We have verified this by performing leakage assessment tests on a Spartan6 FPGA. Our results clearly show that programming the equations for the outputs of `secAND2` (z_0 and z_1) directly into LUTs leaks, which we attribute to glitches.
Glitches on the output of a logic gate are created by different arrival times of its input signals. Predicting the order in which the inputs arrive in a large circuit is impossible. But if we have control over the order of and the delay between the arrival of input signals, it might be possible to send the inputs in a safe sequence such that there are no glitches and thus no leakage of any information about the sensitive inputs or intermediate values. In the next subsection, we investigate the existence of such safe sequences for `secAND2`.

Identifying Safe Input Sequences

We experiment with the issue of glitches on the same Spartan6 FPGA by forcibly sending the inputs of `secAND2` (x_0, x_1, y_0, y_1) at different time instances. Sending one input after another can

be executed in a controlled manner with the help of registers by connecting them directly to the inputs of secAND2. First, we reset all four registers to 0. Then, we update one register at a time over four consecutive clock cycles with the desired input sequence. Finally, we observe if there is any leakage for each of the $4! = 24$ possible sequences. Inputs x and y are independently shared with uniformly distributed random bits.

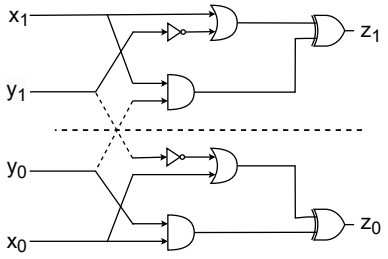


Figure 3.1: secAND2 gate schematic.

Clock Cycle	1	2	3	4	
Input	*	*	*	x_1 or x_0	$\rightarrow$ Sequence leaks
Input	*	*	*	y_1 or y_0	$\rightarrow$ Sequence does not leak

Table 3.1: Leakage behaviour of secAND2 for different input sequences. '*' denotes any of the remaining input shares.

The results of this experiment are summarized in Table 3.1. In short, we observe leakage in sequences where either x_0 or x_1 arrive in the last clock cycle, but not in sequences where either y_0 or y_1 arrive last. These results can be explained from the secAND2 equations in (3.2). Our secAND2 gadget is not non-complete with respect to y (refer to the non-completeness property of TI), as the equations involve both shares y_0 and y_1. The equation for z_0 depends on x_0, y_0 and y_1 whereas z_1 depends on x_1, y_0 and y_1. Therefore, if a glitch occurs, the late arrival of x_0 can reveal information about the unshared input $y(= y_0 \oplus y_1)$ and similar for x_1. By forcing y_0 or y_1 to arrive last, we essentially make x_0 and x_1 arrive early. We observe no leakage in sequences where y_0 or y_1 arrive last because x_0 and x_1 do not evaluate on the combined value of y_0 and y_1, which would leak the unshared input $y(= y_0 \oplus y_1)$. Looking at the secAND2 circuit in Figure 3.1, in the first three clock cycles, no signal or gate has enough information to be able to leak anything about either sensitive unshared inputs, *x* and *y*. And as a result, we can achieve a *temporary* non-completeness property for both output bits during the evaluation in the first three clock cycles. In the fourth clock cycle, only a single input bit arrives straight from a register. Any signal in the secAND2 circuit, see Figure 3.1, will toggle at most once. In other words, glitches cannot occur in the last clock cycle. Thus, no sensitive information can be leaked even though secAND2 is no longer non-complete. The Hamming Distance of the outputs, z_0 and z_1, before and after the fourth clock cycle does not depend on either sensitive inputs, *x* or *y*. Therefore, the final cycle does not leak either sensitive input, and any sequence that ends with y_0 or y_1 can securely compute a product of two shared variables.

Although we have identified safe sequences, we have to address a few issues related to our initial assumption and proposed solution. We began with the assumption that the four registers connected to secAND2 to provide inputs are reset to 0. But this is hardly the case in practice. Our secAND2 gadgets are typically expected to be reused for computation as part of a cryptographic circuit. It might not always be feasible to reset the input registers between computations, for example, if the circuit is pipelined. Additionally, this approach would significantly increase latency as each secAND2 evaluation would take four clock cycles to compute instead of one. Lastly, a

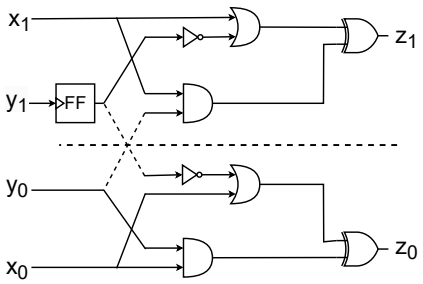


Figure 3.2: secAND2 gate with internal FF or secAND2-FF.

masked cryptographic circuit typically contains several AND gates connected to one another; sending the inputs in four clock cycles for every multiplication would require extra registers to temporarily buffer the intermediate values, which also increases the area cost. In what follows, we propose two solutions to tackle these problems.

Solution 1: secAND2 with a flip-flop (secAND2-FF)

To create a secure low-cost masked AND gate, any circuit in which one of the two shares of y, i.e. y_0 or y_1, arrives last will guarantee no leakage. This can be achieved by delaying the processing of either of the two inputs, for example, y_1. Our secAND2 could hence be constructed as illustrated in Figure 3.2, using an internal delay flip flop (FF). The flip-flop delays the input y_1 and ensures secure computation. This optimization reduces the number of clock cycles to calculate a multiplication from four to two. We shall refer to this faster two-cycle secAND2 as secAND2-FF from now on. We also verified its security with leakage assessment experiments.
In subsection 3.1.2, we explained that the order in which inputs arrive could determine whether the computation is secure or not and that late arrival of x_0 (or x_1) has the potential to reveal information about the unshared input $y(= y_0 \oplus y_1)$. Suppose we compute two multiplications consecutively on the same secAND2-FF gadget: let the inputs for the first multiplication be (m_0, m_1, n_0, n_1) and the inputs for the second multiplication (a_0, a_1, b_0, b_1). If we did not reset the inputs between the multiplications and a_0 arrives before b_0 and b_1, the existing inputs of the secAND2-FF, n_0 and n_1 would remain unchanged when a_0 arrives. Hence, a_0 would leak information about the previous computation, $n(= n_0 \oplus n_1)$. Our first solution, secAND2-FF, reduces latency, but it must be reset between successive computations.

Solution 2: secAND2 with path delay (secAND2-PD)

We propose our second solution to address the issues of secAND2. We eliminate the need for resetting between consecutive multiplications while also reducing the latency. Using a flip-flop as a delay element in our previous solution guarantees that one of the inputs arrives late. Instead of using a flip-flop, we now propose using path delay to achieve the same result, for instance, by making one of the input signals travel through a longer path so it arrives late. This solution follows a more practical approach and comes with certain constraints, such as placement and routing, which might not be as straightforward to implement. We explain in the extended version of the paper that this solution can indeed be achieved in practice. Using path delay as a delay element, instead of a flip-flop, eliminates the critical need to reduce the number of cycles required to send the inputs. Unlike our previous solution, we could send our inputs one after another, each input with a different amount of delay, while not increasing the cycle count of our implementation. In

fact, we could compute secAND2 in a single clock cycle. This may of course increase the critical path delay and thus reduce the circuit's maximum clock frequency.

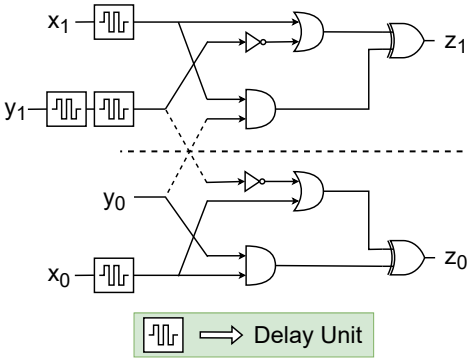


Figure 3.3: secAND2 gate with path delay or secAND2-PD.

Consequently, this approach of sending each input with a different amount of delay would help us compute consecutive multiplications without the need to reset the inputs between computations. We propose such a cost-efficient delayed sequence in Figure 3.3, which we shall call secAND2-PD. Each input is either delayed by zero, one, or two DelayUnits. We refer to a replicable amount of delay as a DelayUnit, and we explain in the extended version of the paper how this can be realized in practice. Input y_0 is not delayed and arrives first in order to protect against information leakage about the previous computation. It is followed by the delayed x_0 and x_1. And finally, y_1 arrives as the last input as explained above.
Referring to our previous example, in Section 3.1.2, b_0 arrives before a_0 (or a_1) does. Therefore a_0 (or a_1) cannot leak information about $n(= n_0 \oplus n_1)$ from the previous computation, as n_0 is replaced by b_0. And the final input b_1 arrives after a_0 and a_1, thereby protecting information leakage about the current computation, i.e. a_0 (or a_1) cannot leak information about $b(= b_0 \oplus b_1)$. In conclusion, secAND2-PD does not require an input reset and also decreases the latency of our secAND2 gadget to a single cycle.

3.1.3 Composing Secure Masked Circuits

secAND2-FF and secAND2-PD gadgets can be used as a building block to securely implement more complex masked circuits. In this section, we provide guidelines on two important steps which are generally required to build circuits. We first show how to compute products of more than two variables securely. And then, we explain how to add *dependent* variables securely.

Computing Product Terms using secAND2-FF

As we start building a circuit, it is common to have situations where we need to compute a product of more than two variables. To illustrate, we compute a product of four variables, $z = a \cdot b \cdot c \cdot d$, which we assume here to be independently shared. Implementing this expression securely can be done with the circuit in Figure 3.4, which evaluates

$$z = \texttt{secAND2-FF}(\texttt{secAND2-FF}(a, b), \texttt{secAND2-FF}(c, d))$$

using three secAND2-FF gadgets and has a latency of three clock cycles.
By carefully controlling when we sample the internal FFs, we can achieve a secure construction with no additional (i.e. external) FFs, which also helps us keep the area footprint low. All the inputs

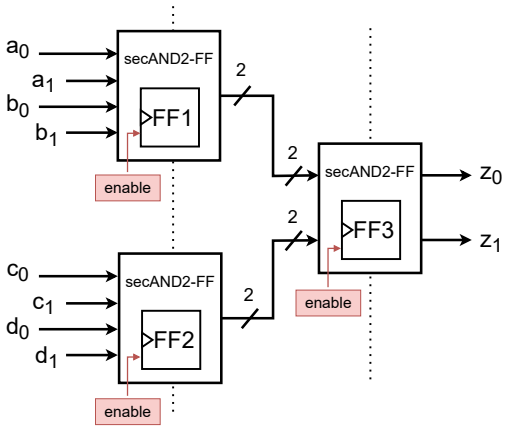


Figure 3.4: Product of four masked variables using `secAND2-FF`.

arrive in the first clock cycle, and in the second clock cycle, the enable signal corresponding to flip-flops FF1 and FF2 is set to high. The enable signal controls when the FF samples the input. FF1 and FF2 sample b_1 and d_1 respectively, therefore $\texttt{secAND2-FF}(a, b)$ and $\texttt{secAND2-FF}(c, d)$ are computed securely. The enable signal of FF3 remains disabled during the second clock cycle, so the `secAND2-FF` computing $z = \texttt{secAND2-FF}(\texttt{secAND2-FF}(a, b), \texttt{secAND2-FF}(c, d))$ is inactive. And in the third and final clock cycle, the enable signal of FF3 is toggled to high, thereby securely completing the computation of output (z_0, z_1) in three clock cycles.
In the general case, implementing a product of n independent variables requires $n-1$ `secAND2-FF` gadgets arranged into $log_2(n)$ layers, such that all different sub-products are cascaded. The latency of the circuit becomes $log_2(n) + 1$ cycles.

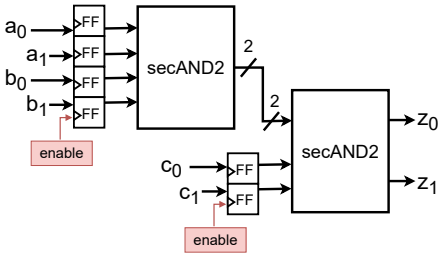


Figure 3.5: `secAND2` with input registers.

In specific cases, it might be advantageous to take the internal FFs out of the `secAND2-FF` gadgets and instead place them at the beginning. For instance, when we compute low-degree products. The flip-flop inside `secAND2-FF` serves the purpose of delaying one of the input signals. Equivalently, we can replace `secAND2-FF` with a `secAND2` and place registers before the gate to buffer the input shares, as shown in Figure 3.5. We can then use a Finite State Machine (FSM) to control when the FFs sample, thus guaranteeing a safe arrival sequence of the input operands to the `secAND2` gadgets. Unlike internal FFs inside `secAND2-FF` gadgets, which solely belong to that gadget, the input registers we now use can be commonly shared by multiple `secAND2` gadgets. For instance, consider the two multiplications $a * b * c$ and $a * b * d$. The input registers used to store $a_0, a_1, b_0, b_1,$ can be shared for the two multiplications. This is usually the case when we are computing polynomials. We would have to compute several different products with common inputs. The resulting circuit can have a slightly larger area due to extra input FFs, but it can be beneficial for evaluation purposes. It allows us, for instance, to test and compare different input sequences or to reset the FFs at any given time.

Computing Product Terms using `secAND2-PD`

Our construction for computing a product of more than two variables using `secAND2-PD` resembles a chain-like structure in contrast to the tree structure we illustrated in the previous subsection. This decision was made for practical reasons, as it helped implement hardware delays easier. In our experience, it is relatively easy to enforce delays on inputs of `secAND2-PD` that arrive directly from registers. But in a typical tree structure which is organized in layers, the outputs of `secAND2-PD` gadgets of one layer are fed as inputs to the next layer of `secAND2-PD` gadgets. Based on our experience, it was not as easy to enforce delays on outputs of `secAND2-PD` gadgets. A construction for a product of three variables, $z = a \cdot b \cdot c$, which we assume here to be independently shared, is shown in Figure 3.6.

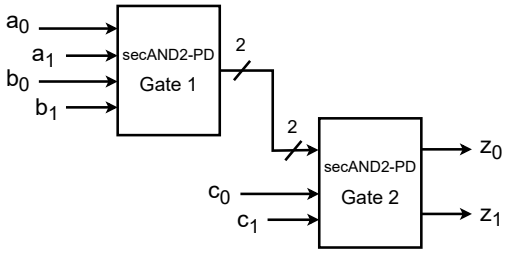


Figure 3.6: Product of three masked variables using `secAND2-PD`.

An appropriately delayed input sequence, as shown in Table 3.2, can be used to compute the product of three variables in a single clock cycle.

Product of 3 variables $z = a \cdot b \cdot c$	$c_0 \rightarrow b_0 \rightarrow a_0, a_1 \rightarrow b_1 \rightarrow c_1$
Product of 4 variables $z = a \cdot b \cdot c \cdot d$	$d_0 \rightarrow c_0 \rightarrow b_0 \rightarrow a_0, a_1 \rightarrow b_1 \rightarrow c_1 \rightarrow d_1$

Table 3.2: Delay sequence for a product 3 or 4 variables

To create the delayed sequence, we use a `DelayUnit`, like the one we used in Section 3.1.2. c_0 is not delayed, b_0 is delayed by one `DelayUnit`, a_0 and a_1 are delayed by two `DelayUnits`, b_1 is delayed by three `DelayUnits` and c_1 is delayed by four `DelayUnits`. The reasoning behind the input sequence remains the same as before, a_0 (and a_1) not only has the potential to leak information about values $b(= b_0 \oplus b_1)$ in Gate 1, see Figure 3.6, but also $c(= c_0 \oplus c_1)$ in Gate 2. Hence, c_0 is sent first to protect against information leakage about the value of $c(= c_0 \oplus c_1)$ from any previous computation and c_1 arrives last to protect against leakage about the current computation. The rest of the sequence, between c_0 and c_1, is identical to what was discussed in subsection 3.1.2. Similarly, we can construct an input sequence for a product of four variables, see Table 3.2.

To generalize, implementing a product of n independent variables requires $n - 1$ `secAND2-PD` gadgets arranged into $n - 1$ layers. In theory, with a proper input sequence, a product of any number of variables can be computed in a single clock cycle. It is up to the designer to realise such a sequence in practice. In this work, we successfully and securely computed a product of three variables in a single clock cycle. We have not explored computing the product of more than three variables in a single cycle, as it was not needed for our secure DES implementation.

Addition of Product Terms

Masked AND and XOR gates are fundamental to building a masked circuit. So far, we concentrated on masking AND gates. But it is also essential to ensure no loss of security during XOR. Both our `secAND2-FF` and `secAND2-PD` gadgets do not consume fresh randomness. Instead, the uniformity of the output is achieved by reusing the randomness of the inputs. This characteristic becomes critical when implementing circuits that combine several terms, for instance, through addition. It can lead to decreased security if the added terms are not independent.

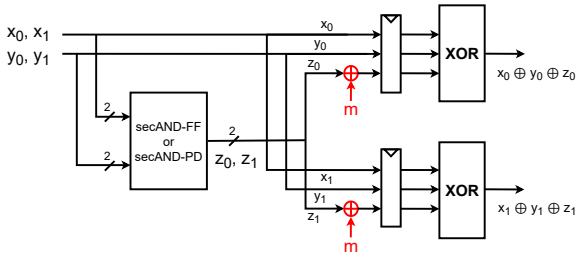


Figure 3.7: $f = x \oplus y \oplus x \cdot y$ (secure).

Consider the function $f = x \oplus y \oplus x \cdot y$, where the product term $z = x \cdot y$ is computed with either a `secAND2-FF` or `secAND2-PD` gadget. In this situation, the masked output z is not independent of x and y, leading to a data-dependent distribution of the masked inputs of the XOR plane. Securing this function, as well as any other which combines dependent shares, requires selectively *refreshing* the (intermediate) dependent variables. Figure 3.7 depicts a circuit to compute f securely. It requires 1 bit of randomness m to refresh the shares of z and guarantee a uniform output distribution.

3.1.4 Further information

For more information about our low-cost Boolean masking scheme, an architecture of a protected implementation of the Data Encryption Standard, concrete performance results as well as results of a practical leakage assessment on FPGA, we refer the reader to our publications at the DATE 2023 conference [102] and in the journal IEEE Transactions on Information Forensics and Security [103].

3.2 Time sharing - A novel approach to low-latency masking

We present a novel approach to small area and low-latency first-order masking in hardware. The core idea is to separate the processing of shares in time in order to achieve non-completeness. Resulting circuits are proven first-order glitch-extended PINI secure. This means the method can be straightforwardly applied to mask arbitrary functions without constraints which the designer must take care of. Furthermore we show that an implementation can benefit from optimization through EDA tools without sacrificing security. We provide concrete results of several case studies. Our low-latency implementation of a complete PRINCE core shows a 32% area improvement (44% with optimization) over the state-of-the-art. Our PRINCE S-Box passes formal verification with a tool and the complete core on FPGA shows no first-order leakage in TVLA with 100 million traces. Our low-latency implementation of the AES S-Box costs roughly one third (one quarter with optimization) of the area of state-of-the-art implementations. It shows no first-order leakage in TVLA with 250 million traces.

3.2.1 Introduction

Implementing secure cryptographic algorithms in a computer system without compromising their promised security has always been challenging. Early research demonstrating the vulnerabilities of cryptographic implementations by Kocher *et al.* [98] showed that it is possible to find the secrets that a computer processes by monitoring its execution time and thereby highlighted the need to build secure implementations. This led to the consolidation of side-channel analysis as a field of study that attempts to gain information from the implementation of a chip or computer system by monitoring its physical effects rather than exploiting a weakness of the implemented algorithm. Along with timing analysis [98], power analysis [99] and electromagnetic analysis [67, 139] represent some of the best-known side-channel attacks. Power analysis, in particular, is perhaps the most popular due to its low setup cost, non-invasive nature, and devastating effectiveness.
In the past few decades, there has been a great deal of research on securing cryptographic implementations against side-channel attacks. Chari *et al.* [46] as well as Goubin and Patarin [73] independently proposed a generic countermeasure called masking that splits the data being processed into random shares to thwart power analysis attacks. The idea behind the countermeasure is to eliminate the correlation between the secret data and the data being processed, since the device's power consumption depends on the latter, which is now random. But processing multiple shares also comes with overheads in terms of implementation area, execution time, online randomness, etc. Along with securing implementations, it has also been important to minimize these overheads. Masking proved to be quite successful for securing software implementations, but it was later found that masked hardware implementations still leak information about the secrets due to glitches [115]. Several modern masking techniques such as Threshold Implementations (TI) [127], Consolidating Masking Schemes (CMS) [143], and Domain Oriented Masking (DOM) [75] were proposed to securely mask hardware in the presence of glitches. They were quite successful in creating secure and efficient hardware implementations with low area and randomness usage. Overhead reductions are typically achieved by decomposing complex non-linear functions, such as an S-Box, into smaller sub-circuits with low algebraic degrees that can be masked efficiently. The composition of the sub-circuits requires careful use of register stages to prevent glitch propagation and re-masking intermediate values to maintain uniformity.
Due to the recent advent of IoT devices, which are very accessible to an attacker, there is a need for embedded real-time applications to have fast data processing, such as memory encryption, to ensure security. As a consequence, there is a new motivation to design masking schemes suitable for low-latency implementations. One of the first generic approaches called GLM was proposed by Groß *et al.* and was used to design low-latency S-Boxes in [74]. GLM, built upon DOM, reduces latency by eliminating register stages required for share compression after non-linear operations. Skipping share compression exponentially increases the share count after every non-linear operation, drastically increasing the overall area and randomness utilization. This especially makes the approach impractical for masking large functions such as higher-degree S-Boxes. Other research into low-latency masking includes LLTI [15] based on TI and other methods involving asynchronous circuits [122, 125].
Although masking techniques are typically proven secure in the t-probing model [88], most are not generic and are not trivial to compose with other design elements. In other words, converting any unprotected circuit to a protected one is not straightforward and is usually laborious. Recently in [42], Cassiers *et al.* introduced a new security notion called Probe Isolating Non-Interference (PINI), which allows for trivial composition. Any PINI gadget is directly composable with other (linear and non-linear) PINI gadgets, without significant overheads. In [41], the authors propose two small multiplication gadgets called HPC1 and HPC2 that can be composed in the glitch-extended probing model, introduced by Faust *et al.* [63], to create more complex circuits. Later

in [92], Knichel *et al.* proposed the HPC3 gadget specifically intended for low-latency applications. Although one can build any circuit with these gadgets, the latency of the circuit grows with the algebraic degree of the function. To the best of our knowledge, there exists only one algorithm-level approach (which does not simply compose elementary gadgets) to generate first-order PINI secure circuits, namely GHPC [94]. Despite being PINI secure, their low latency version GHPC_{LL} also suffers from the high area and randomness overheads for larger functions, like GLM. A single-cycle AES S-Box using GHPC_{LL} costs 64.1 kGEs and 2048 bits of randomness, similar to the cost of GLM. But GHPC_{LL} has the advantage that it is proven to be composable secure while GLM is not.

Contributions. We present a new masking method for low-latency applications that is first-order PINI composable secure, and - more importantly - brings substantially less overhead than other composable low-latency masking schemes. Our contributions are the following:

- We present a masking method that secures any function against first-order attacks and uses only a single register stage, thus executes in a single clock cycle.
- We provide a formal description and follow up with a proof that shows any circuit secured by our approach is first-order glitch-extended PINI secure.
- Compared to previously published algorithm-level approaches such as GLM [74] and GHPC [94] that implement single-cycle Boolean functions, our method shows a substantial improvement both in terms of area as well as online randomness required, for realistically complex circuits.
- We apply our proposed method to produce a masked first-order secure PRINCE implementation that executes in one cycle per round and show the improvements in the utilization results. We demonstrate the security of our PRINCE S-Box with a formal verification tool and show that the complete PRINCE core on FPGA exhibits no first-order leakage in TVLA with 100 million traces.
- We apply our proposed method to mask a more complex function, *i.e.* the AES S-Box, in order to demonstrate its potential for efficient implementations. We show significant improvements in utilization costs and demonstrate that our method scales well especially when masking larger functions. Our AES S-Box on FPGA shows no first-order leakage in TVLA with 250 million traces.

3.2.2 Preliminaries

In this section we briefly introduce the notation and recall relevant background.

Notation

Boolean masking splits each bit $x \in \mathbb{F}_2$ into n uniform random shares x_i such that $x = x_0 \oplus \ldots \oplus x_{n-1}$. The storage of a variable in a register is denoted by curly brackets $\{\cdot\}$.

Probing Model

In the probing model, introduced by Ishai, Sahai, and Wagner [88], an adversary $\mathcal{A}$ is allowed to observe a set of at most t (predefined) wires of a circuit at each execution of the masking. The security of a given implementation is proven by showing that a simulator $\mathcal{S}$ can perfectly simulate any set of at most t probes without any knowledge of the input shares $(x_0, \ldots, x_{n-1})$. A circuit ensuring this condition for any set of size t is said to be t-probing secure.
The probing model provides a way to prove the security of a circuit. However, this algorithmic circuit does not trivially map to practice where, due to leakage effects, an adversary can gain more information than what a probe typically captures. The main leakage effect to be considered is that of glitches. In this work, we model glitches by bundling groups of wires over which a glitch could carry information from one wire to another. Whereas one of the adversary's probes normally results in the value of a single wire, a glitch-extended probe allows obtaining the values of all wires in a bundle. This extension of the probing model has been discussed in the work of Reparaz et al. [143] and formalized by Faust et al. [63]. The formulation of the latter work is as follows: "For any ϵ-input circuit gadget G, combinatorial recombinations (aka glitches) can be modeled with specifically ϵ-extended probes so that probing any output of the function allows the adversary to observe all its ϵ inputs."

Related Work

Due to the growing interest in low-latency masking, several masking techniques have been developed in recent years which are specifically focused on reducing latency. Some techniques relevant to our work are GLM [74], GHPC [94] and LMDPL [147]. Among these, the constructions for GHPC and GLM are most comparable to our technique, while LMDPL employs a distinct dual-rail precharge logic.
GLM is a low-latency masking approach proposed by Groß *et al.* that can be applied to protect any security-sensitive circuit [74]. GLM is based on the Domain Oriented-Masking (DOM) scheme [75], which was introduced to create low-area and low randomness designs. DOM is a gate-level masking technique that uses masked AND gates (DOM multipliers) to build and secure more complex circuits. A masked circuit built with DOM is split into independent circuits called "domains" based on the share index of variables. Non-linear operations compute on all shares of a variable requiring communication between domains and are called "cross-domain terms". For a secure computation, the cross-domain terms are refreshed and stored in registers before they are compressed and merged with inner-domain terms to limit the number of shares and to reduce area.
While DOM optimizes for area and randomness, GLM trades area and randomness for reduced latency. The register stages in DOM multipliers increase latency in a design. GLM reduces latency by eliminating these stages. However, constructing a low-latency circuit by eliminating these register stages introduces complications, leading to increased area and randomness requirements for the circuit. First, the number of output shares increases after every non-linear operation as there is no share compression due to the lack of register stages. The cross-domain terms cannot be merged with the inner-domain terms, increasing the number of shares. Furthermore, as the non-linear logic depth increases, the number of shares of the intermediate values in the circuit also increases exponentially, resulting in a significant increase in area. Second, removing the registers causes the circuit to be susceptible to variable collisions. GLM requires the inputs to non-linear gates to be independently shared. If the circuit violates this condition, the colliding variables must be duplicated with multiple shared instances of the same variable with independent sharings. To resolve collisions, it might also be necessary to duplicate the entire fan-in

circuitry causing the collision. All of these fixes increase the area and randomness overhead of the circuit. As a final step, secure share compression is performed by refreshing the shares with randomness and storing them in registers before the compression, which also increases the cost of area and randomness since many shares need to be refreshed and stored.
GHPC, introduced by Knichel *et al.* [94], is a low-latency masking technique that uses Shannon Decomposition to transform arbitrary Boolean functions into secure PINI composable gadgets. For simplicity, we will illustrate the technique by applying it to a 4 $\times$ 4 function, $F(x, y, z, w) : \mathbb{F}_2^4 \rightarrow \mathbb{F}_2^4$. The technique is applied independently to each coordinate function, decomposing it into cofactors by fixing the input shares within a single share domain. Consider $f(x_0 + x_1, y_0 + y_1, z_0 + z_1, w_0 + w_1) : \mathbb{F}_2^4 \rightarrow \mathbb{F}_2$ as the shared representation of one of the four coordinate functions of F, where the subscript denotes the share domain of the inputs. The function f is decomposed into 16 cofactors by considering all combinations of inputs from the second share domain, *i.e.* $\{x_1, y_1, z_1, w_1\} \in \{0, 1\}^4$. For example, if $\{x_1, y_1, z_1, w_1\} = \{0, 1, 1, 0\}$, then the corresponding cofactor would be $f(x_0, \overline{y_0}, \overline{z_0}, w_0)$. The resulting Shannon cofactors only depend on the inputs from the first share domain, *i.e.* $\{x_0, y_0, z_0, w_0\}$. A secure implementation of F with GHPC necessitates two register layers. A secure low-latency implementation of F with GHPC$_{\text{LL}}$ reduces this to one register layer at the cost of more randomness. For each coordinate function, in the first phase, the 16 cofactors are calculated, refreshed, and registered using inputs from the first share domain. In the second phase, the inputs from the second share domain serve as selection bits to choose the correct cofactor out of the sixteen for output. In GHPC, the number of cofactors, registers, and randomness required is determined by the number of inputs and not the algebraic degree of the function.

3.2.3 Time Sharing Masking

We introduce our novel approach to securely first-order mask any (vectorial) Boolean function in hardware with a single register layer. We will refer to it using the acronym TSM, short for Time Sharing Masking, in the remainder of the paper. We begin the explanation with a toy example, applying TSM to a single AND gate, in Section 3.2.3. In Section 3.2.3, we write out TSM formally so it applies to any Boolean function, in particular also vectorial Boolean functions. Finally, in Section 3.2.3, we prove that TSM is first-order glitch-extended PINI composable secure.

Preliminary Example

Let x and y be the two inputs of the AND gate that computes $z = x \cdot y$. And let $(x_0, x_1), (y_0, y_1)$ be their sharings such that $x = x_0 + x_1$ and $y = y_0 + y_1$. A key aspect of TSM is to separate *in time* the processing of $share_0$ inputs from the processing of $share_1$ inputs with the help of a register layer, see Figure 3.8. Before the computation begins, we refresh the inputs with two random bits r_3, r_4 for the masked AND gate to be composable secure (see Section 3.2.3):

$$x'_0 = x_0 + r_3 \qquad x'_1 = x_1 + r_3$$
$$y'_0 = y_0 + r_4 \qquad y'_1 = y_1 + r_4$$

Then, in the first phase, all cross product combinations of $share_0$, *i.e.*, $(x'_0, y'_0, x'_0 y'_0)$, are computed, refreshed, and stored in registers. In the second phase, all cross product combinations of $share_1$, *i.e.*, $(x'_1, y'_1, x'_1 y'_1)$, are computed. Finally, in order to produce the output of the AND gate, products of masked combinations of $share_0$ and combinations of $share_1$ are summed up, see Eq. (3.3).

$$
\begin{aligned}
z_0 &= \Big\{x'_0y'_0 + r_0\Big\} + \Big\{x'_0 + r_1\Big\} \cdot \Big\{y'_1\Big\} + \Big\{y'_0 + r_2\Big\} \cdot \Big\{x'_1\Big\} \\
z_1 &= \Big\{r_0\Big\} + \Big\{r_1\Big\} \cdot \Big\{y'_1\Big\} + \Big\{r_2\Big\} \cdot \Big\{x'_1\Big\} + \Big\{x'_1\Big\} \cdot \Big\{y'_1\Big\}
\end{aligned}
\tag{3.3}
$$

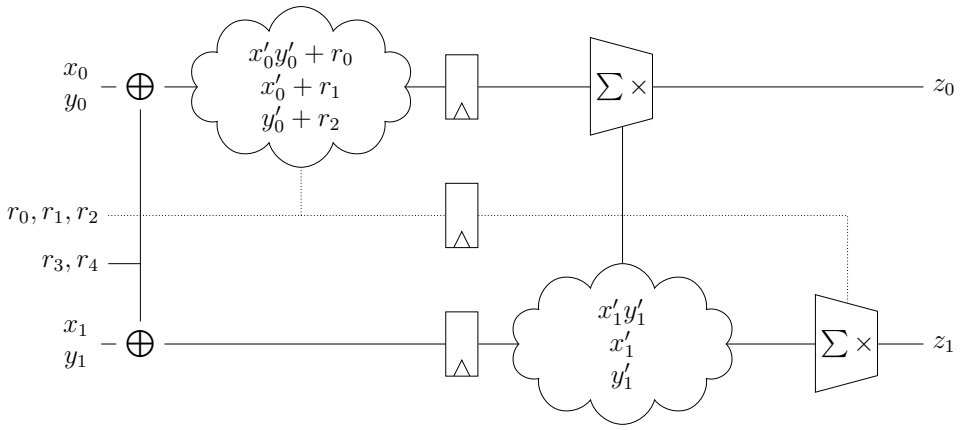


Figure 3.8: Application of TSM to a single AND gate.

The computation requires five fresh random bits and eight registers to store the intermediate shares. The area and randomness utilization for computing a single AND gate is high, but we use this toy example only for illustration. TSM should be mainly applied to mask more complex non-linear functions as a whole, and not individual AND gates.

Formal Description

We provide a description of TSM working on an arbitrary (vectorial) Boolean function. The outline is presented in Figure 3.9.
Specifically in this section, we change the notation to denote bits in a word by square brackets $(x[0], ..., x[k-1])$ instead of using different letters (*e.g.*, x, y in the previous section). We denote $x \in \mathbb{F}_2^k$ a k-bit word where its two-share Boolean masking is denoted by $\bar{x} = (x_0, x_1) \in \mathbb{F}_2^{2k}$ such that $x_0 + x_1 = x$ with $x_0 = (x_0[0], ..., x_0[k-1])$ and $x_1 = (x_1[0], ..., x_1[k-1])$ the notation of the share-words in separate bits. This change allows a simpler, more compact presentation of what follows next.

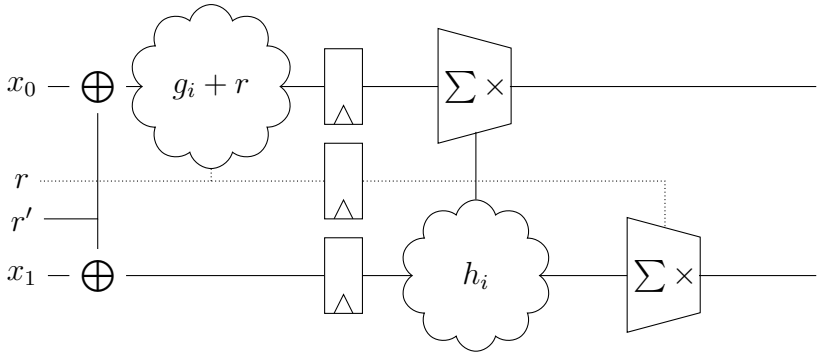


Figure 3.9: Application of TSM to an arbitrary (vectorial) Boolean function described by the functions g_i and h_i.

We explain the TSM method in a constructive manner where we first rewrite in Eq. (3.4) the algebraic normal form of a shared function as the sum of non-complete terms where each term

is the multiplication between share domain 0 and share domain 1. We then rewrite this equation to Eq. (3.5) by adding fresh randomness allowing us to safely form the two output shares outlined in Eq. (3.6). Finally, the inputs of the gadget are first re-masked to ensure composable security.
We start informally, where we first rewrite the equations of a shared monomial. Namely, note that for the product of the bits $x[j]$ for some set of indices $j \in J \subset \{0, ..., k-1\}$

$$\prod_{j\in J} x[j] = \prod_{j\in J}(x_0[j] + x_1[j]) = \sum_{i\in\mathbb{F}_2^k}\prod_{j\in J} x_{i[j]}[j] = \sum_{i\in\mathbb{F}_2^k} \prod_{j\in J \text{ s.t. } i[j]=0} x_0[j] \prod_{j\in J \text{ s.t. } i[j]=1} x_1[j]\,.$$

In words, each monomial can be split as the sum of non-complete terms and each of these terms can be split as the multiplication of shares from domain 0 and shares from domain 1.
Consider an arbitrary Boolean function

$$f : \mathbb{F}_2^k \to \mathbb{F}_2 : x = (x[0], ..., x[k-1]) \mapsto f(x[0], .., x[k-1])\,.$$

We denote its two-share masking by $\bar{F} : \mathbb{F}_2^{2k} \to \mathbb{F}_2^2 : \bar{x} \mapsto (F_0(\bar{x}), F_1(\bar{x}))$ such that $F_0(\bar{x}) + F_1(\bar{x}) = f(x_0 + x_1)$. The above insight can be applied to each monomial in the algebraic normal form of f. We thus say that there exist functions g_i and h_i such that

$$f(x_0 + x_1) = \sum_{I\in\mathcal{P}_k} g_{\pi(I)}((x_0[i])_{i\in I}) h_{\pi(I)}((x_1[i])_{i\in\Omega/I})\,, \tag{3.4}$$

where we denote $(x_0[i])_{i\in I}$ as the set of all bits $x_0[i]$ for i in I. We also denote by $\mathcal{P}_k$ the power set of the indices $\Omega = \{0, ..., k-1\}$, namely all possible sets of indices in Ω. It is clear that $|\mathcal{P}_k| = 2^k$. The sets in $\mathcal{P}_k$ are numbered and indicated by the function π.
The functions g_i, h_i in Eq. (3.4) work only on share domain 0 and 1, respectively. To go back to the masked AND gate example from Section 3.2.3, the functions g_i, h_i are the following

$$\begin{array}{cccc} g_0(x_0, y_0) = x_0y_0 & g_1(x_0) = x_0 & g_2(y_0) = y_0 & g_3(\emptyset) = 1 \\ h_0(\emptyset) = 1 & h_1(y_1) = y_1 & h_2(x_1) = x_1 & h_3(x_1, y_1) = x_1y_1\,. \end{array}$$

Multiplying and adding the above terms, for $\mathcal{P}_2 = ((0,1), (0), (1), \emptyset)$, we get

$$\begin{aligned} xy &= \sum_{I\in\mathcal{P}_2} g_{\pi(I)}((x_0[i])_{i\in I}) h_{\pi(I)}((x_1[i])_{i\in\Omega/I}) \\ &= g_0(x_0, y_0)h_0(\emptyset) + g_1(x_0)h_1(y_1) + g_2(y_0)h_2(x_1) + g_3(\emptyset)h_3(x_1, y_1) \\ &= x_0y_0 + x_0y_1 + x_1y_0 + x_1y_1\,. \end{aligned}$$

The above sharing is already correct, however, it misses randomness for its security.
We thus further adapt Eq. (3.4) by adding randomness. Namely, by adding 2^k random bits $r = (r_0, ..., r_{2^k-1})$, we get

$$\begin{aligned} f(x_0 + x_1) = &\sum_{I\in\mathcal{P}_k} (g_{\pi(I)}((x_0[i])_{i\in I}) + r_{\pi(I)}) h_{\pi(I)}((x_1[i])_{i\in\Omega/I}) \\ &+ \sum_{I\in\mathcal{P}_k} r_{\pi(I)} h_{\pi(I)}((x_1[i])_{i\in\Omega/I})\,. \end{aligned} \tag{3.5}$$

By re-masking, we can split the computation in two parts (read two phases), the computation and refreshing of $g_i(\cdot)$ on the first shares, and the computation and recombination of $h_i(\cdot)$ on the second shares. This is also depicted in Figure 3.9. In this figure, we also observe that the

shares x_0, x_1 are first refreshed with the randomness $r' \in \mathbb{F}_2^k$. This is done in order to make TSM composable secure as proven in Section 3.2.3.
Finally, the two shares $F_0(\bar{x}), F_1(\bar{x})$ are composed as follows

$$F_0(\bar{x}) = \sum_{I \in \mathcal{P}_k} (g_{\pi(I)}((x_0[i])_{i \in I}) + r_{\pi(I)}) h_{\pi(I)}((x_1[i])_{i \in \Omega/I}) \tag{3.6}$$
$$F_1(\bar{x}) = \sum_{I \in \mathcal{P}_k} r_{\pi(I)} h_{\pi(I)}((x_1[i])_{i \in \Omega/I}) \,.$$

Since the functions g_i and h_i (or their product) consist of all terms up to degree k, any Boolean function can be made from these g_i and h_i. This is extended for vectorial functions ($\mathbb{F}_2^k \to \mathbb{F}_2^\ell$) by re-using the g_i and h_i functions for each coordinate function. While this can make the output shares non-uniform (in the extreme case, two coordinate functions are equal), the security comes from the register layer being filled with uniquely re-masked values and from each function working only on one share domain at a time. This is made formal in the next section where we show that TSM is PINI composable secure.

Security

We prove that TSM is first-order probing secure and that it is, moreover, composable first-order secure in the Probe-Isolating Non-Interference (PINI) framework by Cassiers *et al.* [42]. Namely, we show that any circuit secured with TSM allows for trivial composition. Since TSM is designed to work over hardware, we use the glitch-extended probing model by Faust *et al.* [63] to extend the PINI framework into the glitch-extended PINI framework. This PINI security is particularly important since it allows for the composition between gadgets without the need to place additional registers between them. Since all maskings of linear layers (where the linear function is applied share-wise) are PINI secure, we can trivially secure linear functions without adding additional registers or additional randomness.
Before starting the proof that the approach delivers PINI secure solutions, we need to introduce the necessary concepts to introduce PINI security. We start by providing the notion of simulation.

Definition 10 (Simulatability [42])**.** Let $P = \{p_1, ..., p_\ell\}$ be a set of ℓ probes of a gadget C and C_P the tuple of values of the probes for an execution of C. Let $I = \{(i_1, j_1), ..., (i_k, j_k)\} \subset \{0, ..., d-1\} \times \{0, ..., m-1\}$ be a set of input wires of C. A simulator is a randomized function $\mathcal{S} : \mathbb{F}_q^k \to \mathbb{F}_q^\ell$. The set of probes P can be simulated with the set of input wires I if there exists a simulator $\mathcal{S}$ such that for any inputs $x_{*,*}$, the distributions $C_P(x_{*,*})$ and $\mathcal{S}(x_{i_1,j_1}, ..., x_{i_k,j_k})$ are equal, where the probability is over the random coins in C and $\mathcal{S}$.

The above definition defines the security game in terms of a simulation game. This framework is extended to PINI security where we define which information is given to the simulator.

Definition 11 (PINI [42])**.** Let G be a gadget over d shares and P a set of t_0 (glitch-extended) probes on wires of G (called internal probes). Let A be a set of t_1 share indices. G is t-PINI if for all P and A such that $t_0 + t_1 \leq t$, there exists a set B of at most t_0 share indices such that probes on the set of wires $P \cup y_{A,*}$ can be simulated with the wires $x_{A \cup B}$, with $x_{i,*}$ denoting all inputs of share i and $y_{i,*}$ denoting all outputs of share i.

Given the above definition of PINI, we show that any circuit secured by the TSM method from Section 3.2.3 is composable secure. Intuitively, the reason TSM is composable secure is due to each registered value (in the single register stage of the method) being re-masked by unique randomness.

Theorem 1. *Any circuit secured by TSM (Section 3.2.3) is first-order glitch-extended PINI.*

Proof. Denoting the k-bit input shares x_i and the output shares y_i. Looking at Definition 14 for $t = 1$ (considering glitch-extended probes), we find that we need to prove two cases. Namely,

- for $t_0 = 0$ and $t_1 = 1$, in which case we need to show that the output shares y_i can be simulated using the input shares x_i for $i \in \{0, 1\}$.
- for $t_0 = 1$ and $t_1 = 0$, in which case we need to show that a single intermediate probe can be simulated using either x_0 or x_1.

We begin with the first case, we have to prove that y_i can be simulated using x_i. We split up the proof depending on i.

- For y_0, the output is calculated from the values g_i re-masked by r and by values h_i which operate on the second shares x_1 re-masked by r'. Since g_i is re-masked by r and x_1 is re-masked by r', a simulator can sample r and r' and perfectly simulate the values y_0 as uniform randomness (in particular, the simulator does not need the values x_0).
- For y_1, since it is created using only x_1 and randomness r', a simulator can perfectly simulate y_1 given x_1 and by uniformly random sampling r' (in fact, due to r' the simulation would also work from scratch in which case the simulator can simulate the probed values as uniform randomness).

For the proof of the second case where we simulate an intermediate probe, we consider only probes in the first phase of the circuit, since probes on the second phase were already considered in the previous case. However, for probes on the first phase, it is clear that these can be perfectly simulated since the computation is done share-wise (a probe either only sees values from x_0 or from x_1) in which case the simulator simply gets either the zero or the one shares and performs the computation following the algorithm.
Since both cases are proven, the masking is first-order glitch-extended PINI. □

As a result, since the TSM circuit is first-order glitch-extended PINI secure, it can be composed with any other PINI gadget (which includes all linear operations too) without adding extra register stages or randomness following the proofs of composable security from the original work [42].

3.2.4 Advantages of TSM

In this section we mainly outline the general efficiency of TSM and contrast it with the first-order case of GLM and GHPC$_{\text{LL}}$. In general, we emphasize the advantages of TSM through a comparison of area, considering both the number of registers and logic, as well as the randomness required for masking a $\mathbb{F}_2^k \to \mathbb{F}_2^k$ function with an algebraic degree of $k - 1$ (which represents the highest algebraic degree for a k-bit permutation).

Registers and Randomness Cost

In Section 3.2.3, we described our approach by applying it to an arbitrary Boolean function $f : \mathbb{F}_2^k \to \mathbb{F}_2$. The Boolean function is computed as a combination of the functions g_i and h_i. Importantly, the functions g_i and h_i solely depend on the k shared inputs. We emphasize that TSM can be extended to vectorial Boolean functions with multiple outputs ($\mathbb{F}_2^k \to \mathbb{F}_2^k$) because

all coordinate functions share the same k inputs. The intermediate registers which store the refreshed results of the g_i functions, the random bits, and the second share inputs can be commonly used to calculate the shared outputs of all coordinate functions without increasing the register and randomness cost. In other words, the number of intermediate registers and randomness remains constant irrespective of the number of outputs.
TSM requires at most $2^k - 2$ registers to store the results of the g_i functions since $|\mathcal{P}_k| = 2^k$ and there is no degree k term (removing one register) and we do not store a constant term (removing the second register). TSM then requires at most another $2^k - 2$ registers to store the random bits r. Finally, TSM requires k registers to store the second share inputs. This gives a total of at most $2^{k+1} + k - 4$ registers. Similarly, for the randomness, TSM requires at most $2^k - 2$ bits to refresh the g_i functions and another k bits for r' in Figure 3.9.
We compare these numbers with GLM and GHPC$_{\text{LL}}$ in Table 3.4. We note that both TSM and GLM can be more efficient than what is reported in the table, depending on the function to which the method is applied. Namely, we report the *worst case metrics* such that any function of degree $k - 1$ can be implemented with the given register and randomness costs.

Name	# Registers	# Register Layers	# Random Bits
TSM	$2^{k+1} + k - 4$	1	$2^k + k - 2$
GLM [74]	$2^k k$	1	$2^k k$
GHPC$_{\text{LL}}$ [94]	$2^k k + k$	1	$2^k k$

Table 3.3: Comparison for a $\mathbb{F}_2^k \to \mathbb{F}_2^k$ function of algebraic degree $k - 1$.

We observe roughly a factor $k/2$ improvement in the number of registers and a factor k in random bits over both GLM and GHPC$_{\text{LL}}$. As previously mentioned, this improvement is a direct result of re-using the registered values for each coordinate function (of the k outputs).
If we compare TSM with GLM for masking an AES S-Box, we see significant, concrete savings in registers and random bits when implementing a higher algebraic degree function with many outputs. Groß *et al.* [74] report the cost of masking an AES S-Box with a single register layer to be $16 \cdot 2^7 (= 2048)$ registers and $16 \cdot 2^7 (= 2048)$ random bits. To compare these numbers with TSM, we fill in the value $k = 8$ in Table 3.4. TSM requires only 516 registers and only 262 random bits per AES S-Box.
In the published paper we discuss the implementation of the AES S-Box with TSM in more detail and provide concrete numbers for the area cost, including combinational logic.

Combinational logic

Without loss of generality, let us consider the PRINCE S-box, a 4×4 function $S : \mathbb{F}_2^4 \to \mathbb{F}_2^4$ for illustration. We show the equations for the function S in Algebraic Normal Form (ANF) in Eq. (3.7). We denote (a, b, c, d) as the four input bits and f^0, f^1, f^2, f^3 as the coordinate functions which produce the four output bits.

$$\begin{aligned} f^0 &= 1 + dc + b + cb + dcb + a + da + ba \\ f^1 &= 1 + db + cb + dcb + ca + cba \\ f^2 &= d + dc + a + da + ca + dca + cba \\ f^3 &= 1 + c + cb + dcb + a + dca + ba + dba \end{aligned} \tag{3.7}$$

Before delving into the benefit of TSM, let us briefly discuss how the function S would be masked using GLM. In the first stage, every coordinate f^0, f^1, f^2, f^3 is split into eight share domains. Cubic terms, such as *bcd*, are split into eight shared multiplication terms, $b_0c_0d_0, b_0c_0d_1, ..., b_1c_1d_1$. One multiplication term is assigned to each of the eight shared domains of the coordinate functions. Quadratic terms, such as *bc*, are split into four shared multiplication terms $b_0c_0, b_0c_1, b_1c_0, b_1c_1$ and are distributed among four of the eight shared domains of the coordinate functions. The share domains are then refreshed with fresh randomness and are registered. In the second stage, share compression is performed to reduce the number of shares from eight to two. In summary, the first stage involves expanding the number of shares, followed by the second stage, where the shares are compressed. On the other hand, to mask the function *S*, GHPC applies "Shannon decomposition", an identity that splits any Boolean function into parts called cofactors, to each coordinate function f^0, f^1, f^2, f^3. The coordinate functions are independently expanded into 16 cofactors by setting one share of the inputs a,b,c, and d to either 0 or 1. A common characteristic between GLM and GHPC, which may be regarded as a potential drawback, is that every coordinate function is treated as a separate entity even though they commonly share the same inputs.

Applying TSM to the function S, in the first stage, all inputs are remasked, then all cross-products of the $share_0$ inputs are computed, *i.e.*, $(a_0, b_0, a_0b_0, ..., b_0c_0d_0)$, and finally those are refreshed and stored in the register layer. In the second stage, the cross-products of the $share_1$ inputs are computed, and they are then multiplied and summed with the masked cross-products of the $share_0$ inputs to produce the outputs of the coordinate functions.

TSM allows to reduce the cost of combinatorial logic by efficient reuse in several ways. First, we can deduplicate identical terms across coordinate functions, i.e. compute them only once and then reuse them. For example, dc is needed to compute f_0 and f_2 but there is no need to compute dc twice. Overall this allows to reduce the number of distinct terms to compute from 20 to 14. The decrease in logic becomes more prominent with an increase of the number of coordinate functions. This also reduces the number of random bits needed for refreshing in phase 1, and the number of registers.

Second, we can reuse already computed lower degree terms to compute higher degree terms. For example, we can compute dc and reuse it for computing dcb. Eq. (3.8) shows the sharing of the coordinate function f^0 and Eq. 3.9 shows in the first three lines the straightforward computation for $(b)_0$, $(cb)_0$ and $(dcb)_0$. In the fourth line it shows a more efficient computation of $(dcb)_0$ by reusing the already computed $(cb)_0$ which results in a reduction of the number of logic gates, thereby lowering the area. The decrease in logic gates becomes more prominent with an increase in the algebraic degree of the terms, particularly when masking higher algebraic degree functions such as the AES S-Box.

$$\begin{aligned} f_0^0 &= 1 + (dc)_0 + (b)_0 + (cb)_0 + (dcb)_0 + (a)_0 + (da)_0 + (ba)_0 \\ f_1^0 &= (dc)_1 + (b)_1 + (cb)_1 + (dcb)_1 + (a)_1 + (da)_1 + (ba)_1 \end{aligned} \tag{3.8}$$

$$\begin{aligned} (b)_0 &= \left\{b'_0 + r_1\right\} \\ (bc)_0 &= \left\{b'_0c'_0 + r_7\right\} + \left\{c'_0 + r_2\right\} \cdot \left\{b'_1\right\} + \left\{b'_0 + r_1\right\} \cdot \left\{c'_1\right\} \\ (bcd)_0 &= \left\{b'_0c'_0d'_0 + r_{13}\right\} + \left\{b'_0c'_0 + r_7\right\} \cdot \left\{d'_1\right\} + \left\{b'_0d'_0 + r_8\right\} \cdot \left\{c'_1\right\} + \left\{c'_0d'_0 + r_9\right\} \cdot \left\{b'_1\right\} \\ &\quad + \left\{b'_0 + r_1\right\} \cdot \left\{c'_1\right\} \cdot \left\{d'_1\right\} + \left\{d'_0 + r_3\right\} \cdot \left\{b'_1\right\} \cdot \left\{c'_1\right\} + \left\{c'_0 + r_2\right\} \cdot \left\{d'_1\right\} \cdot \left\{b'_1\right\} \\ (bcd)_0 &= (bc)_0 \cdot \left\{d'_1\right\} + \left\{b'_0c'_0d'_0 + r_{13}\right\} + \left\{b'_0d'_0 + r_8\right\} \cdot \left\{c'_1\right\} + \left\{c'_0d'_0 + r_9\right\} \cdot \left\{b'_1\right\} \\ &\quad + \left\{d'_0 + r_3\right\} \cdot \left\{b'_1\right\} \cdot \left\{c'_1\right\} \end{aligned} \tag{3.9}$$

Optimization during Logic Synthesis

Importantly, since the combinational logic in phase 1 (before the register layer) and the combinational logic in phase 2 (after the register layer) is non-complete, it is safe to allow (or even, one should enforce) the logic optimization through Electronic Design Automation (EDA) tools, without the need to carefully place logic in distinct modules. The only kind of optimization which must not be allowed is register re-timing, as that may move combinational logic across the register stage which may lead to a violation of non-completeness. Our case studies in the published paper include the impact of logic optimization on area, maximum frequency and security.

3.2.5 Further information

For more information about our low-latency Boolean masking scheme, case studies with application to PRINCE and the AES S-Box, concrete performance results, formal security evaluation, as well as results of a practical leakage assessment on FPGA, we refer the reader to our publication at TCHES 2024 [158].

3.3 Higher-Order Time Sharing Masking

Time Sharing Masking (TSM) was introduced as a novel low-latency masking technique for hardware circuits. TSM offers area and randomness efficiency, as well as glitch-extended PINI security, but it is limited to first-order security. We address this limitation and generalize TSM to higher-order security while maintaining all of TSM's advantages. Additionally, we propose an area-latency tradeoff. We prove HO-TSM glitch-extended PINI security and successfully evaluate our circuits using formal verification tools. Furthermore, we demonstrate area- and latency-efficient implementations of the AES S-box, which do not exhibit leakage in TVLA on FPGA. Our proposed tradeoff enables a first-order secure implementation of a complete AES-128 encryption core with 92 kGE, 920 random bits per round, and 20 cycles of latency, which does not exhibit leakage in TVLA on FPGA.

3.3.1 Introduction

Cryptographic algorithms implemented on computer systems are susceptible to physical attacks that can extract sensitive information being processed by the system. An adversary could monitor side-channel information, such as power consumption [99], execution time [98], or electromagnetic emanations [67, 139], to uncover sensitive data like cryptographic secret keys. Masking [46, 73] is a popular countermeasure that splits data into multiple shares, thereby removing the correlation between the data and side-channel information to protect implementations against such attacks.

Over the past years, masking techniques for hardware have, most importantly, aimed to guarantee security in the presence of glitches [115]. To securely mask hardware implementations, several techniques have been developed, including popular methods such as Threshold Implementations (TI) [127], Consolidating Masking Schemes (CMS) [143], and Domain Oriented Masking (DOM) [75]. These methods primarily focus on reducing the area overhead and minimizing the fresh randomness required to ensure security. Another line of research in hardware masking that has gained traction in recent years is focused solely on reducing latency. An initial effort, LMDPL [106], introduced by Leiserson *et al.*, achieved the construction of a first-order secure

AES S-Box in just 2 cycles. Subsequent improvements by Sasdrich et al. [147] further refined this approach to create a first-order secure round-based AES-128 implementation operating in ten cycles. Later, Gross et al. introduced an alternative low-latency approach, GLM [74], derived from DOM, capable of securely masking vectorial Boolean functions of any algebraic degree with a latency of just one cycle, regardless of the security order. Recent works include GHPC [94] and TSM [158], both of which are low-latency approaches capable of producing first-order single-cycle masked gadgets. They offer the additional benefit of compositional security under the Probe Isolating Non-Interference (PINI) security notion [42]. A prevailing theme across all low-latency approaches is the trade-off between area and randomness to minimize latency. The single-cycle first-order GLM AES S-Box is 60.7 kGE in size and requires 2048 bits of randomness, while the GHPC_{LL} AES S-Box has an area of 64.1 kGE and also requires 2048 bits of randomness. In comparison, the single-cycle TSM AES S-Box shows improvements in both area and randomness, with a size of 14.3 kGE and a randomness requirement of 262 bits. However, the area and randomness requirements remain substantial compared to other S-Boxes with higher latency that employ techniques like DOM, TI. Notably, many low-latency approaches are limited to provide only first-order security.

Contributions. We extend TSM to achieve higher-order security (HO-TSM) and introduce an area-latency trade-off construction that leverages TSM and HO-TSM, leading to substantial reductions in area and randomness costs. Our contributions are as follows:

- We introduce a higher-order extension, HO-TSM, that builds upon the foundation of first-order TSM [158]. The fundamental principle of HO-TSM is to process one share of each input in every clock cycle while maintaining isolation of these computations through the use of registers. Notably, the latency of HO-TSM is dictated by the security order, rather than the algebraic degree of the function. For a d^{th}-order secure implementation, the latency is precisely *d* cycles.
- We formally outline our construction and demonstrate that any function secured using our method achieves d^{th}-order glitch-extended PINI and SNI (Strong Non-Interference) [21] security.
- We develop a two-cycle, second-order HO-TSM AES S-Box that occupies less area and requires less randomness than the only other low-latency, second-order AES S-Box by Gross *et al.* (GLM) [75]. Our FPGA implementation of this S-Box exhibits no signs of first- or second-order leakage in TVLA with 100 million traces.
- We present an area-latency trade-off for HO-TSM that significantly reduces both area and randomness costs, with only a single cycle increase in latency. We construct a new two-cycle, first-order PINI and SNI secure AES S-Box that shows substantial improvements in utilization costs and develop a complete round-based AES-128 implementation that executes in twenty cycles. The FPGA implementation of our full AES-128 demonstrates no first-order leakage in TVLA with 250 million traces.
- Using our trade-off construction, we develop another three-cycle, second-order PINI and SNI secure AES S-Box. Our S-Box exhibits no first- or second-order leakage in TVLA with 100 million traces.
- We use SILVER [93] to validate small examples of both HO-TSM and our trade-off construction, providing evidence for PINI and SNI security. In addition, we also verify our S-Boxes using the formal verification tools `maskVerif` [20] and PROLEAD [123].

3.3.2 Preliminaries

We introduce essential concepts and notations which form the basis for the methodologies and security proofs we present in later sections.

Notation

In Boolean masking, each value $x \in \mathbb{F}_2^k$ is split into $d+1$ uniform random shares x_i such that $x = x_0 + \ldots + x_d$. For $x \in \mathbb{F}_2^k$ a k-bit word, we denote the bits of the word by square brackets $(x[0], ..., x[k-1])$. We represent the sharing of x as $\bar{x} = (x_0, x_1, \cdots, x_d) \in \left(\mathbb{F}_2^k\right)^{d+1}$ such that $x_0 + x_1 \cdots + x_d = x$ with $x_i = (x_i[0], ..., x_i[k-1])$ the notation of the share-words in separate bits. For a set of share indices A $\subset \{0, \ldots, d\}$, we denote $x_A = \{x_i : x \in A\}$. We denote $x \xleftarrow{\$} X$ to represent selecting a value uniformly and randomly from the set X.

Circuit Model

For the purpose of security proofs and to explain masking methods, we represent algorithms in the shape of a directed-acyclic graph called a circuit. In this circuit, an edge represents a bit ($\mathbb{F}_2$) or a word ($\mathbb{F}_2^k$) and a node represents an operation of its fan-in such as an XOR or an AND gate. We also consider nodes without input which can output a uniformly distributed random bit or word.

Probing Model

We use the probing model, as originally proposed by Ishai *et al.* [88], to model the side-channel adversary. More specifically, the d^{th}-order probing adversary is one who has access to the layout of the circuit he is attacking and who is able to request (before the execution of the circuit) the digital value of up to d wires (probes) in the circuit.
The above adversary is expanded following the glitch-extended robust probing model by Faust *et al.* [63] where each probe is expanded such that it returns not only the wire value but all registered values leading up to that wire. This adversarial expansion is a way to capture the effect of glitches and propagation delays in the circuit's physical implementation.
We call a circuit d^{th}-order glitch-extended probing secure if there exists a simulator which can simulate probed values in such a way that an adversary cannot distinguish the actual circuit from the simulator. The adversary has the power to choose the circuit's input secrets and the simulator is not given this information. This simulation-based security is a clever way of saying that the probed values' distribution (made by the circuit's internal randomness) is independent of its input secrets.

Composable Probing Security

We consider a simulation security game for the probing adversary, because apart from easing certain proofs, it allows for composable security. Namely, using the Strong Non-Interference (SNI) framework by Barthe *et al.* [21] and its extension to Probe-Isolated Non-Interference (PINI) by Cassiers and Standaert [42], one can show that if a small part of the circuit (gadget) is composable secure then its composition with other composable secure gadgets remains secure.
We first provide the definition of simulatability as given by in [42].

Definition 12 (Simulatability [42])**.** Let $P = \{p_1, ..., p_\ell\}$ be a set of ℓ probes of a gadget C and C_P the tuple of values of the probes for an execution of C. Let $I = \{(i_1, j_1), ..., (i_m, j_m)\} \subset \{0, ..., d-1\} \times \{0, ..., k-1\}$ be a set of input wires of C. A simulator is a randomized function $\mathcal{S} : \mathbb{F}_q^m \to \mathbb{F}_q^\ell$. The set of probes P can be simulated with the set of input wires I if there exists a simulator $\mathcal{S}$ such that for any inputs $\bar{x}$, the distributions $C_P(\bar{x})$ and $\mathcal{S}(x_{i_1}[j_1], ..., x_{i_m}[j_m])$ are equal, where the probability is over the random bits in C and $\mathcal{S}$.

In this work, we consider composability in the form of both the SNI [21] and the PINI [42] frameworks. The reason for opting for a more restrictive simulation-based model will become clear in the published paper, where we provide a low-cost area-latency trade-off. In the more restrictive NI model, we consider simulation where the simulator is not given any input shares when probing an output. We introduce both PINI and SNI.

Definition 13 (SNI)**.** A gadget is called t–SNI if for any set of t_0 (glitch-extended) probes on intermediate variables and every set of t_1 (glitch-extended) probes on output shares such that $t_0 + t_1 \leq t$, the totality of the probes can be simulated by only t_0 shares of each input.

Definition 14 (PINI)**.** Let G be a gadget over d shares and P a set of t_0 (glitch-extended) probes on wires of G (called internal probes). Let A be a set of t_1 share indices. G is t-PINI if for all P and A such that $t_0 + t_1 \leq t$, there exists a set B of at most t_0 share indices such that probes on the set of wires $P \cup y_A$ can be simulated with the wires $x_{A \cup B}$, with x_i denoting all inputs of share i and y_i denoting all outputs of share i.

In the regular PINI notion, the simulator is given the input share (the share with the same index) for every probed output share and in the regular SNI notion, the simulator is given arbitrary shares of each input instead of the same index for each input. Instead, we demand that the simulation of the gadget can be made using the same index for each input share for each placed internal probe and that the simulator is not given any shares when probing the gadget's output.

Time Sharing Masking

Time Sharing Masking (TSM) is a low-latency masking scheme introduced by Kumar S. V. *et al.* [158] designed to perform first-order masking of any (vectorial) Boolean function in hardware using a single register layer. Moreover, the technique has been proven to be secure under the PINI composable security notion. In this subsection, we provide a comprehensive description of TSM as applied to an arbitrary (vectorial) Boolean function.
Consider an arbitrary Boolean function

$$f : \mathbb{F}_2^k \ \to \ \mathbb{F}_2^\ell \quad : \quad x = \big(x[0], \ldots, x[k-1]\big) \ \mapsto \ f\big(x[0], \ldots, x[k-1]\big).$$

We denote its two-share masking by $\bar{F} : \mathbb{F}_2^{2k} \to \mathbb{F}_2^{2\ell}$, such that $\bar{x} \mapsto (F_0(\bar{x}), F_1(\bar{x}))$ and $F_0(\bar{x}) + F_1(\bar{x}) = f(x_0 + x_1)$. The sharing of the ℓ coordinate functions is given by $F_0 = (F_0[0], \ldots, F_0[\ell-1])$ and $F_1 = (F_1[0], \ldots, F_1[\ell-1])$.
We explain the TSM method constructively, beginning with a reformulation of the algebraic normal form (ANF) for one coordinate function $f[m]$ of f, where $m \in \{0, \ldots, \ell-1\}$. The ANF decomposes $f[m]$ in a way that facilitates its masking. Any single-output Boolean function $f[m]$ can be expressed in its ANF as a sum of products of input bits:

$$f[m]\big(x[0], \ldots, x[k-1]\big) \ = \ \sum_{S \in \mathcal{P}_k} \alpha_S \prod_{j \in S} x[j] \quad \text{with} \quad \alpha_S \in \mathbb{F}_2. \tag{3.10}$$

Here, $\mathcal{P}_k$ denotes the power set of the indices $\{0, \ldots, k-1\}$, so that $|\mathcal{P}_k| = 2^k$. The sum is over all elements $S \in \mathcal{P}_k$. For any S, α_S determines whether the product of input bits $x[j]$ for $j \in S$ is present in the ANF of $f[m]$.

We define:

$$\mathcal{M}_m \;=\; \big\{S \in \mathcal{P}_k \;\mid\; \alpha_S = 1\big\} \tag{3.11}$$

as the set of all indices of products present in $f[m]$.

Decomposing Product Terms in the ANF

Each product indexed by $S \in \mathcal{M}_m$ is masked using Boolean masking and decomposed into a sum of products of shares from domains 0 and 1:

$$\prod_{j\in S} x[j] \;=\; \prod_{j\in S}\big(x_0[j] + x_1[j]\big) \;=\; \sum_{I\subseteq S}\Big(\prod_{j\in I} x_0[j]\Big)\,\Big(\prod_{j\in S\setminus I} x_1[j]\Big). \tag{3.12}$$

Here, the sum is over all $I \in \mathcal{P}(S)$, the power set of S. Recall that S represents one specific combination of input bits $x[j]$. The sum is thus over all possible combinations of bits in S. The first product involves shares from domain 0 for bits in I, while the second involves shares from domain 1 for bits which are not in I, but in S.
We define two sets of functions g^0 and g^1, each operating on a single share domain, 0 and 1, respectively:

$$g^0_{\pi(I)}(x_0) \;=\; \prod_{j\in I} x_0[j], \qquad g^1_{\pi(I)}(x_1) \;=\; \prod_{j\in I} x_1[j] \tag{3.13}$$

where $I \subseteq \{0,\ldots,k-1\}$. For brevity, we write $g^0_{\pi(I)}(\cdot)$ to denote $g^0_{\pi(I)}$ evaluated on the share x_0. Similarly, $g^1_{\pi(I)}(\cdot)$ for the function $g^1_{\pi(I)}$ evaluated on x_1. The indexing function π maps each $I \in \mathcal{P}_k$ to a unique integer in $\{0,\ldots,2^k-1\}$. For the empty set, $g^0_{\pi(\emptyset)}(\cdot) = g^1_{\pi(\emptyset)}(\cdot) = 1$.
Using these definitions in Eq.(3.13), we rewrite Eq.(3.12) as:

$$\prod_{j\in S} x[j] \;=\; \prod_{j\in S}\big(x_0[j] + x_1[j]\big) \;=\; \sum_{I\subseteq S}\Big(g^0_{\pi(I)}(x_0)\Big)\,\Big(g^1_{\pi(S\setminus I)}(x_1)\Big). \tag{3.14}$$

A concrete example of these notations is a 2-input masked AND gate with inputs $x[0]$ and $x[1]$. For $k = 2$, the power set of $\{0,1\}$ consists of four subsets: $\emptyset, \{0\}, \{1\}, \{0,1\}$, indexed as: $\pi(\emptyset) = 0$, $\pi(\{0\}) = 1$, $\pi(\{1\}) = 2$, $\pi(\{0,1\}) = 3$. The corresponding functions g^0 and g^1 are summarized below:

Subset I	$\emptyset$	$\{0\}$	$\{1\}$	$\{0,1\}$
$g^0_{\pi(I)}(x_0)$	1	$x_0[0]$	$x_0[1]$	$x_0[0]\cdot x_0[1]$
$g^1_{\pi(I)}(x_1)$	1	$x_1[0]$	$x_1[1]$	$x_1[0]\cdot x_1[1]$

Applying Eq. (3.14) to the monomial $\prod_{j\in\{0,1\}} x[j]$, we obtain:

$$\begin{aligned}
\prod_{j\in\{0,1\}} x[j] &= \sum_{I\subseteq\{0,1\}} g^0_{\pi(I)}(x_0)\cdot g^1_{\pi(\{0,1\}\setminus I)}(x_1)\\
&= \underbrace{g^0_0(x_0)\,g^1_3(x_1)}_{I=\emptyset} \;+\; \underbrace{g^0_1(x_0)\,g^1_2(x_1)}_{I=\{0\}} \;+\; \underbrace{g^0_2(x_0)\,g^1_1(x_1)}_{I=\{1\}} \;+\; \underbrace{g^0_3(x_0)\,g^1_0(x_1)}_{I=\{0,1\}}\\
&= x_1[0]\cdot x_1[1] \;+\; x_0[0]\cdot x_1[1] \;+\; x_0[1]\cdot x_1[0] \;+\; x_0[0]\cdot x_0[1]\\
&= \big(x_0[0] + x_1[0]\big)\big(x_0[1] + x_1[1]\big).
\end{aligned}$$

Applying TSM to a Single Coordinate

This decomposition is applied to each term in the ANF of $f[m]$. Summing over all terms $S \in \mathcal{M}_m$, we have:

$$f[m](x_0 + x_1) \;=\; \sum_{S\in\mathcal{P}_k} \alpha_S \sum_{I\subseteq S} g^0_{\pi(I)}(x_0)\, g^1_{\pi(S\setminus I)}(x_1) \;=\; \sum_{S\in\mathcal{M}_m} \sum_{I\subseteq S} g^0_{\pi(I)}(x_0)\, g^1_{\pi(S\setminus I)}(x_1) \tag{3.15}$$

The expression in Eq.(3.15) can be reorganized by subsets $I \in \mathcal{P}_k$ associated with $g^0_{\pi(I)}$. This reorganization will be essential later when we extend the approach to vectorial Boolean functions. In this reorganization, for each $I \in \mathcal{P}_k$, we capture the set of g^1 functions that need to be summed and multiplied with the corresponding $g^0_{\pi(I)}$ function.

$$\sum_{S\in\mathcal{M}_m} \sum_{I\subseteq S} g^0_{\pi(I)}(\cdot)\, g^1_{\pi(S\setminus I)}(\cdot) \;=\; \sum_{I\in\mathcal{P}_k} \Big(g^0_{\pi(I)}(\cdot) \;\cdot \sum_{\substack{S\in\mathcal{M}_m \\ I\subseteq S}} g^1_{\pi(S\setminus I)}(\cdot) \Big) \tag{3.16}$$

Specifically, for each $I \in \mathcal{P}_k$, the set of g^1 functions is constructed by selecting all $S \in \mathcal{M}_m$ where $I \subseteq S$, and including the functions $g^1_{\pi(S\setminus I)}$ in this set.
Formally, this set of g^1 functions can be defined using the indices:

$$J_{m,\pi(I)} \;=\; \big\{ \pi\big(S \setminus I\big) \;\big|\; S \in \mathcal{M}_m,\; I \subseteq S \big\}. \tag{3.17}$$

The set $J_{m,\pi(I)}$ represents those terms in the ANF of $f[m]$ (denoted by $\mathcal{M}_m$) that contribute to the product decomposition involving $g^0_{\pi(I)}$. The final expression of $f[m]$ is given by:

$$f[m](x_0 + x_1) \;=\; \sum_{I\in\mathcal{P}_k} \Big(g^0_{\pi(I)}(\cdot) \;\cdot \sum_{\substack{S\in\mathcal{M}_m \\ I\subseteq S}} g^1_{\pi(S\setminus I)}(\cdot) \Big) \;=\; \sum_{I\in\mathcal{P}_k} \Big(g^0_{\pi(I)}(\cdot) \;\cdot \sum_{j\in J_{m,\pi(I)}} g^1_j(\cdot) \Big). \tag{3.18}$$

If $J_{m,\pi(I)}$ is empty, it implies that the subset I does not contribute to the ANF of $f[m]$ under the specified conditions. Consequently, the sum $\sum_{j\in J_{m,\pi(I)}} g^1_j(\cdot)$ evaluates to zero.

First-Order Security

Next, to ensure first-order security, we refresh the outputs of g^0 with fresh random bits r, where $r = \{ r_0, \ldots, r_{2^k-1} \}$. Specifically, the output of each $g^0_{\pi(I)}$ is refreshed by adding the corresponding random bit $r_{\pi(I)}$, leading to

$$f[m](x_0 + x_1) \;=\; \sum_{I\in\mathcal{P}_k} \Big(\big(g^0_{\pi(I)}(\cdot) + r_{\pi(I)} \big) \cdot \sum_{j\in J_{m,\pi(I)}} g^1_j(\cdot) \Big) \;+\; \sum_{I\in\mathcal{P}_k} \big(r_{\pi(I)} \cdot \sum_{j\in J_{m,\pi(I)}} g^1_j(\cdot) \big).$$

We can split this expression into two phases, as depicted in Figure 3.10:

(i) computing and refreshing g^0 on the first share (with addition of r),

(ii) computing g^1 on the second share and recombining.

These steps allow us to safely construct the two output shares:

$$\begin{aligned} F_0[m] &= \sum_{I\in\mathcal{P}_k} \Big(\big\{ g^0_{\pi(I)}(\cdot) + r_{\pi(I)} \big\} \;\cdot \sum_{j\in J_{m,\pi(I)}} g^1_j(\cdot) \Big), \\ F_1[m] &= \sum_{I\in\mathcal{P}_k} \Big(\big\{ r_{\pi(I)} \big\} \;\cdot \sum_{j\in J_{m,\pi(I)}} g^1_j(\cdot) \Big). \end{aligned} \tag{3.19}$$

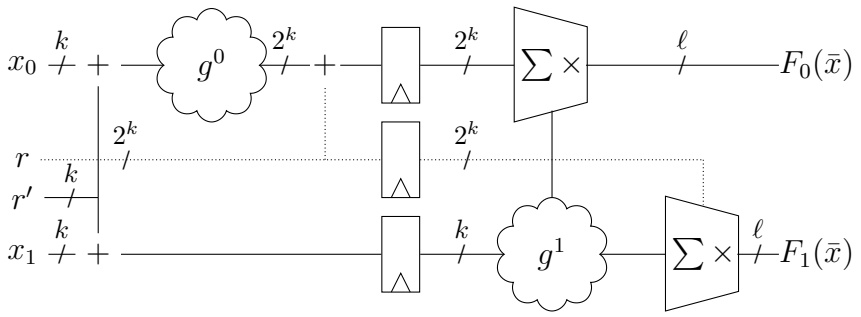


Figure 3.10: Application of TSM to an arbitrary (vectorial) Boolean function described by the set of functions g^0 and g^1.

where the curly brackets indicate storing a value in a register, which prohibits glitch propagation. Lastly, before processing the gadget, the inputs x_0, x_1 themselves are refreshed with randomness $r' \in \mathbb{F}_2^k$ (i.e., $x_0 \leftarrow x_0 + r'$ and $x_1 \leftarrow x_1 + r'$) to ensure composable security.

Extending TSM to Vectorial Functions

We now generalize this approach to vectorial functions by ensuring that the masking process is applied efficiently across all coordinate functions $f[m]$, $m \in \{0, \ldots, \ell - 1\}$. From Eq.(3.12), it is evident that all monomials up to degree k can be constructed using the functions g^0 and g^1. For each coordinate function $f[m]$, $m \in \{0, \ldots, \ell-1\}$, there is a corresponding set $J_{m,\pi(I)}$ determined by the terms that actually appear in $f[m]$, denoted by $\mathcal{M}_m$, see Eq.(3.17). Consequently, the same g^0 and g^1 building blocks can be efficiently re-used to extend TSM to a vectorial Boolean function ($f : \mathbb{F}_2^k \rightarrow \mathbb{F}_2^\ell$) by applying the above construction to each output coordinate $f[m]$.
To establish correctness, we demonstrate that the sum of the output shares reconstructs the original Boolean function f. The randomness $r_{\pi(I)}$, introduced during the refreshing of $g^0_{\pi(I)}$, cancels out between the two shares, leaving only the correctly reconstructed terms from the ANF of $f[m]$:

$$\begin{aligned} F_0[m] + F_1[m] &= \sum_{I \in \mathcal{P}_k} \Big(\big\{ g^0_{\pi(I)}(\cdot) + r_{\pi(I)} \big\} + \big\{ r_{\pi(I)} \big\} \Big) \cdot \sum_{j \in J_{m,\pi(I)}} g^1_j(\cdot) \\ &= \sum_{I \in \mathcal{P}_k} \Big(g^0_{\pi(I)}(\cdot) \cdot \sum_{j \in J_{m,\pi(I)}} g^1_j(\cdot) \Big) = f[m](x_0 + x_1). \end{aligned} \tag{3.20}$$

For vectorial functions $f : \mathbb{F}_2^k \rightarrow \mathbb{F}_2^\ell$, this reasoning extends independently to each coordinate $f[m]$, $m \in \{0, \ldots, \ell - 1\}$. Consequently, the sum of the shares satisfies $F_0(\bar{x}) + F_1(\bar{x}) = f(x_0 + x_1)$.

3.3.3 Higher-Order Time Sharing Masking (HO-TSM)

We now extend TSM to achieve higher-order security while retaining the benefits of the first-order construction. We begin with a second-order secure AND gate as an illustrative example in Section 3.3.3. Next, we provide the construction of our higher-order secure method in Section 3.3.3. In Section 3.3.3, we prove that our proposed solution is glitch-extended PINI and SNI secure. Finally, we apply our method to develop a two-cycle second-order masked AES S-Box in Section 3.3.3.

Preliminary Example

Before formally introducing HO-TSM, we illustrate its core concept with a simple example of a second-order secure AND gate. Consider two inputs, x and y, for the AND gate, producing an output $z = x \cdot y$. Assume x and y are shared as $\bar{x} = (x_0, x_1, x_2)$ and $\bar{y} = (y_0, y_1, y_2)$, respectively. The HO-TSM approach processes one share of each input per clock cycle while ensuring the isolation of computations using registers, as shown in Figure 3.11.

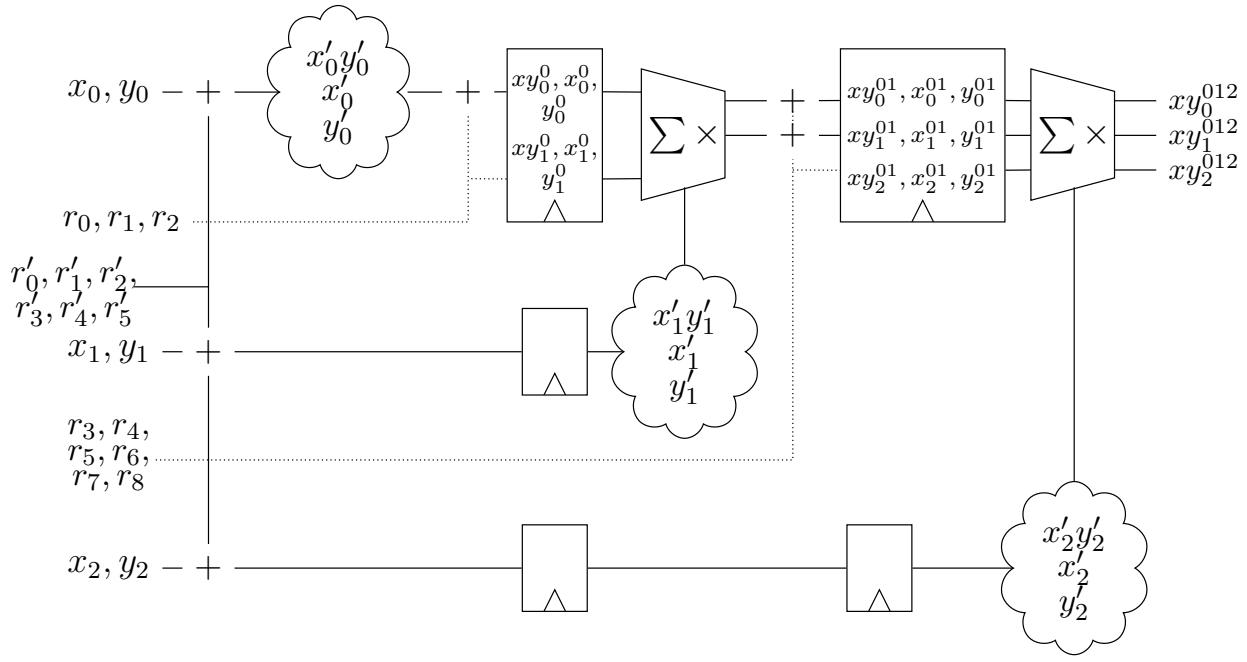


Figure 3.11: Application of HO-TSM$_2$: a second-order secure AND gate.

To ensure composable security, the input shares (x_0, x_1, x_2) and (y_0, y_1, y_2) are initially refreshed with six random bits $\{r'_0, r'_1, r'_2, r'_3, r'_4, r'_5\}$, as follows:

$\boldsymbol{x' = x'_0 + x'_1 + x'_2}$	$\boldsymbol{y' = y'_0 + y'_1 + y'_2}$
$x'_0 = x_0 + r'_0 + r'_1$	$y'_0 = y_0 + r'_3 + r'_4$
$x'_1 = x_1 + r'_0 + r'_2$	$y'_1 = y_1 + r'_3 + r'_5$
$x'_2 = x_2 + r'_1 + r'_2$	$y'_2 = y_2 + r'_4 + r'_5$

In the first phase, all cross-product combinations of $(x'_0, y'_0, x'_0 y'_0)$ are calculated, refreshed using random bits $\{r_0, r_1, r_2\}$, and stored in registers:

$\boldsymbol{xy^0 = x'_0 \cdot y'_0}$	$\boldsymbol{x^0 = x'_0}$	$\boldsymbol{y^0 = y'_0}$
$xy_0^0 = x'_0 y'_0 + r_0$	$x_0^0 = x'_0 + r_1$	$y_0^0 = y'_0 + r_2$
$xy_1^0 = r_0$	$x_1^0 = r_1$	$y_1^0 = r_2$

In the second phase, all cross-product combinations of $(x'_1, y'_1, x'_1 y'_1)$ are computed and combined with the register contents from Phase 1. This phase also increases the number of intermediate shares from two to three:

$\boldsymbol{xy^{01} = (x'_0 + x'_1) \cdot (y'_0 + y'_1)}$	$\boldsymbol{x^{01} = (x'_0 + x'_1)}$	$\boldsymbol{y^{01} = (y'_0 + y'_1)}$
$xy_0^{01} = xy_0^0 + x_0^0 \cdot y_1' + y_0^0 \cdot x_1' + r_3$	$x_0^{01} = x_0^0 + r_5$	$y_0^{01} = y_0^0 + r_7$
$xy_1^{01} = xy_1^0 + x_1^0 \cdot y_1' + y_1^0 \cdot x_1' + x_1'y_1' + r_4$	$x_1^{01} = x_1^0 + x_1' + r_6$	$y_1^{01} = y_1^0 + y_1' + r_8$
$xy_2^{01} = r_3 + r_4$	$x_2^{01} = r_5 + r_6$	$y_2^{01} = r_7 + r_8$

In the third phase, cross-product combinations of $(x'_2, y'_2, x'_2y'_2)$ are computed and combined with the intermediate shares from Phase 2 to produce the final outputs:

$\boldsymbol{xy^{012} = (x'_0 + x'_1 + x'_2) \cdot (y'_0 + y'_1 + y'_2)}$
$xy_0^{012} = xy_0^{01} + x_0^{01} \cdot y_2' + y_0^{01} \cdot x_2'$
$xy_1^{012} = xy_1^{01} + x_1^{01} \cdot y_2' + y_1^{01} \cdot x_2'$
$xy_2^{012} = xy_2^{01} + x_2^{01} \cdot y_2' + y_2^{01} \cdot x_2' + x_2'y_2'$

This computation requires 15 fresh random bits and 21 registers to store the intermediate shares, with a latency of two clock cycles. While the area and randomness costs of HO-TSM may be significant for a single AND gate, which serves merely as an illustrative example, the method is intended for use with complex non-linear functions, where its efficiency and advantages are most pronounced.

Construction

In this section, we provide a formal description of HO-TSM. The d-th order HO-TSM, denoted by HO-TSM$_d$, extends the principles of TSM to higher-order security. To provide a comprehensive understanding, we first present a general overview of HO-TSM. We then explain how to construct HO-TSM$_d$ step by step using a recursive approach.
HO-TSM$_d$ operates on $(d + 1)$ shares and requires d register stages (or equivalently, $(d + 1)$ phases). In each phase, the algorithm processes one share, computes all its monomial combinations denoted by g^i(functions operating on individual shares), and combines the result with the output from the previous phase stored in the register. This combination is then refreshed to generate an additional output share, except in the final phase, where the combination output serves as the final output of the gadget . The d-th register stage maintains a $(d + 1)$-sharing of cross-products using shares $\{0, \ldots, d\}$.
For example, Figure 3.12 illustrates HO-TSM$_2$. In the first phase, share x_0 is processed to compute its monomials, which are refreshed to produce two output shares. In the subsequent phase, x_1 is processed and combined with the refreshed outputs from the first phase, resulting in three shares. Finally, in the third phase, share x_2 is processed and combined with the outputs of the previous phase to produce the final output shares of HO-TSM$_2$.
An intuitive way to understand the HO-TSM method is to recognize that it is constructed recursively from lower-order HO-TSM instances. As illustrated in Figure 3.13, HO-TSM$_d$ is built from HO-TSM$_{d-1}$, which securely operates on the first d shares $(x_0, \ldots, x_{d-1})$. A new set of functions, g^d, is then defined to operate exclusively on the additional share x_d. Finally, the outputs of HO-TSM$_{d-1}$ are combined with the outputs of g^d, merging the partial results to produce a secure $(d + 1)$-share masking of the target function.
Consider an arbitrary Boolean function

$$f : \mathbb{F}_2^k \rightarrow \mathbb{F}_2^\ell, \quad x = \big(x[0], \ldots, x[k-1]\big) \mapsto f\big(x[0], \ldots, x[k-1]\big). \tag{3.21}$$

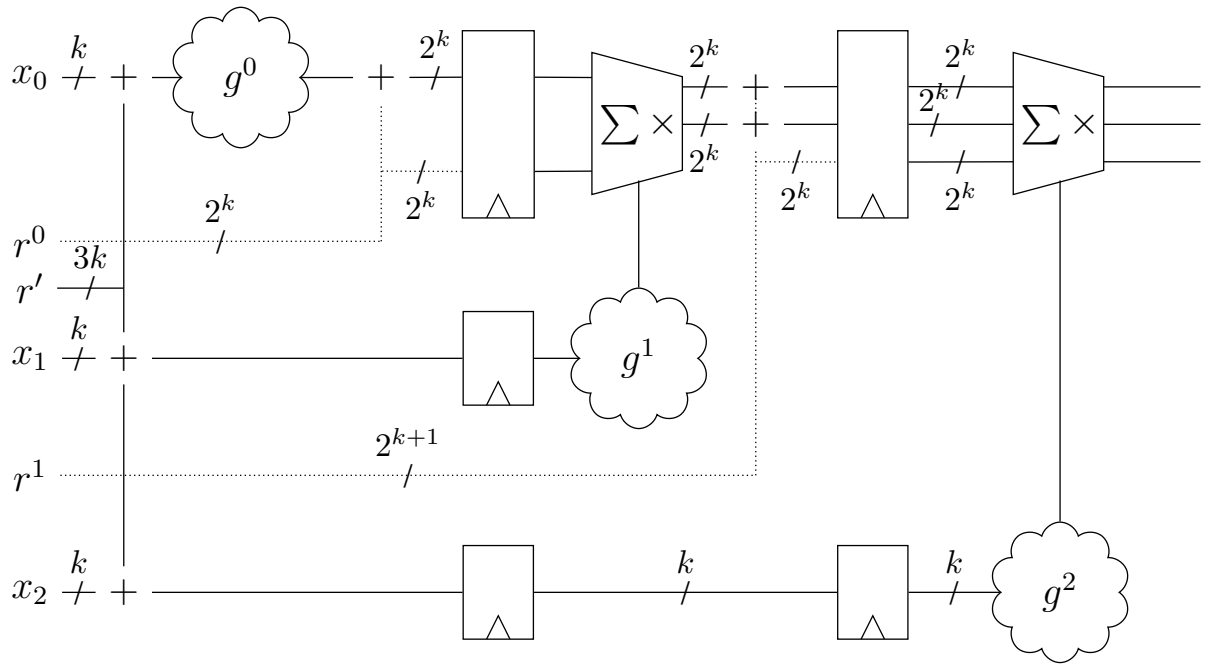


Figure 3.12: Application of HO-TSM$_2$ to an arbitrary (vectorial) Boolean function described by the functions g^0, g^1, and g^2.

We denote its $(d+1)$-share masking by $\bar{F} : \mathbb{F}_2^{k(d+1)} \to \mathbb{F}_2^{\ell(d+1)}, \bar{x} \mapsto \big(F_0(\bar{x}), \ldots, F_d(\bar{x})\big)$, such that $F_0(\bar{x}) + F_1(\bar{x}) + \cdots + F_d(\bar{x}) = f\big(x_0 + x_1 + \cdots + x_d\big)$.
The description of HO-TSM builds on the concepts introduced in TSM, including the decomposition of terms, refreshing of shares, and the reformulation of the algebraic normal form (ANF) for masked computation. It is recommended to review Section 3.3.2 beforehand, as many of these concepts are directly reused and extended in this section.
Following the same strategy as in Section 3.3.2, we first focus on a single coordinate function $f[m]$, $m \in \{0, \ldots, \ell-1\}$, and subsequently generalize to the full vectorial case. Extending to all ℓ outputs involves applying the same procedure efficiently to each coordinate function.

Decomposing Product Terms

In analogy to how TSM uses g^0 and g^1, we define two sets of functions:

(i) A set h operating on the first d share domains (0 to $d-1$),

(ii) A set g^d operating on the last (i.e., d-th) share domain.

The functions $h_{\pi(I)}$ compute cross-products across the first d shares, while $g^d_{\pi(I)}$ focuses on the monomials involving the d-th share exclusively. Concretely:

$$h_{\pi(I)}(x_0, \ldots, x_{d-1}) \;=\; \prod_{i \in I} \Big(x_0[i] + x_1[i] + \cdots + x_{d-1}[i]\Big), \quad g^d_{\pi(I)}(x_d) \;=\; \prod_{i \in I} x_d[i]. \tag{3.22}$$

Here, $I \subseteq \{0, \ldots, k-1\}$, and π is an indexing function over the power set of $\{0, \ldots, k-1\}$. To simplify notation, we use $h_{\pi(I)}(\cdot)$ to represent the function $h_{\pi(I)}$ applied to the first d shares. Similarly, $g^d_{\pi(I)}(\cdot)$ refers to $g^d_{\pi(I)}$ evaluated using the final share x_d. More details on the notations used can be found in Section 3.3.2.

As in the two-share case for first-order TSM, each shared term in $f[m]$ (of the form $\prod_{j\in J} x[j]$) can be decomposed into a sum of products that splits between the d-th share and the sum of the previous d shares. Formally, for $J \subseteq \{0,\dots,k-1\}$,

$$\prod_{j\in J} x[j] = \prod_{j\in J} ((x_0[j] + \cdots + x_{d-1}[j]) + x_d[j]) = \sum_{I\subseteq J} h_{\pi(I)}(x_0,\dots,x_{d-1}) \cdot g^d_{\pi(J\setminus I)}(x_d). \tag{3.23}$$

To illustrate, consider the second-order masked AND gate introduced in Section 3.3.3, representing $x[0]\cdot x[1]$ with $k=2$. The corresponding h-functions and g^2-functions for all subsets $I \in \mathcal{P}_2$ are summarized below:

Subset I	$\emptyset$ $(\pi(I)=0)$	$\{0\}$ $(\pi(I)=1)$	$\{1\}$ $(\pi(I)=2)$	$\{0,1\}$ $(\pi(I)=3)$
$h_{\pi(I)}(x_0,x_1)$	1	$(x_0[0]+x_1[0])$	$(x_0[1]+x_1[1])$	$(x_0[0]+x_1[0])(x_0[1]+x_1[1])$
$g^2_{\pi(I)}(x_2)$	1	$x_2[0]$	$x_2[1]$	$x_2[0]\cdot x_2[1]$

By applying Eq. (3.23), we decompose the term $\prod_{j\in\{0,1\}} x[j]$, where each $x[j]$ is $\big(x_0[j]+x_1[j]+x_2[j]\big)$:

$$\begin{aligned}\prod_{j\in\{0,1\}} \big(x[j]\big) &= \sum_{I\subseteq\{0,1\}} h_{\pi(I)}(x_0,x_1)\ \cdot\ g^2_{\pi(\{0,1\}\setminus I)}(x_2)\\ &= \underbrace{h_0(\cdot)\,g^2_3(\cdot)}_{I=\emptyset} + \underbrace{h_1(\cdot)\,g^2_2(\cdot)}_{I=\{0\}} + \underbrace{h_2(\cdot)\,g^2_1(\cdot)}_{I=\{1\}} + \underbrace{h_3(\cdot)\,g^2_0(\cdot)}_{I=\{0,1\}}\\ &= \big(x_0[0]+x_1[0]+x_2[0]\big)\big(x_0[1]+x_1[1]+x_2[1]\big).\end{aligned}$$

Decomposing a Single Coordinate

Analogous to Eq. (3.18) in the first-order TSM case, the decomposition in Eq. (3.23) is applied to each term in the algebraic normal form of $f[m]$. By utilizing the same sets $J_{m,\pi(I)}$ as defined in Eq. (3.17), we express:

$$f[m]\Big((x_0+\cdots+x_{d-1})+x_d\Big) \;=\; \sum_{I\in\mathcal{P}_k}\Big(h_{\pi(I)}(x_0,\dots,x_{d-1})\ \cdot \sum_{j\in J_{m,\pi(I)}} g^d_j(x_d)\Big). \tag{3.24}$$

The sets $J_{m,\pi(I)}$ from TSM can be directly reused in HO-TSM because they are constructed solely based on the structure of the ANF of $f[m]$, which remains unchanged regardless of the number of shares.

Recursive Construction of HO-TSM$_d$

Having decomposed $f[m]$ into the functions h and g^d, we now outline the recursive steps required to construct HO-TSM$_d$. These steps detail how the outputs of h and g^d are securely computed and combined to achieve the $(d+1)$-share masking of $f[m]$. As illustrated in Figure 3.13, a HO-TSM$_{d-1}$ block securely implements each of the $h_{\pi(I)}$ functions, shared across d shares, while the g^d cloud computes all the functions $g^d_{\pi(I)}$. The recursive process proceeds as follows:

1. The input shares $\big(x_0,\dots,x_d\big)$ are first refreshed with randomness $r' = \big(r'_0,\dots,r'_d\big)$ to ensure composable security. Each $r'_i \in \mathbb{F}_2^k$ for $i \in \{0,\dots,d-1\}$ is an independent random

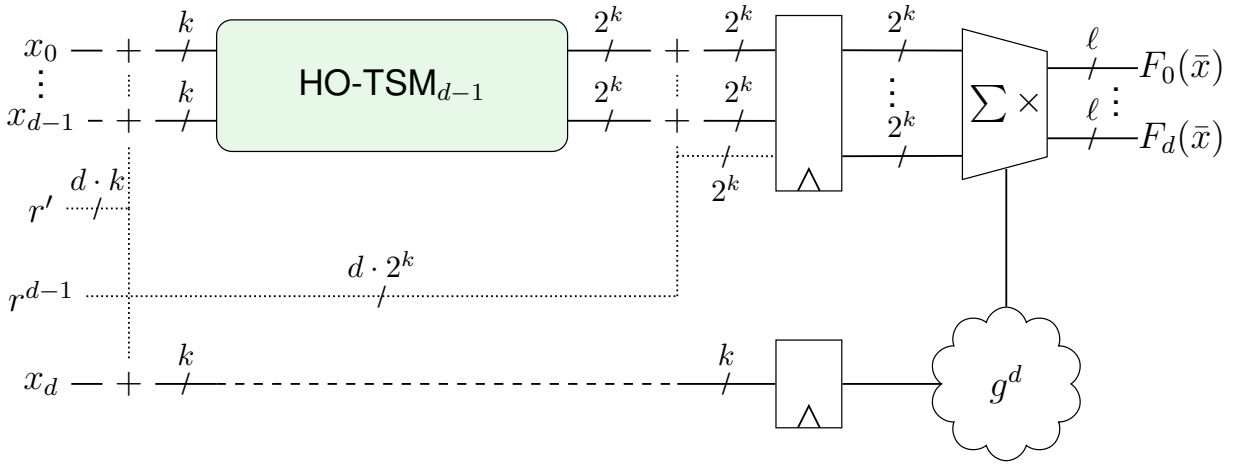


Figure 3.13: Recursive method of HO-TSM$_d$.

value, while $r'_d = \sum_{i=0}^{d-1} r'_i$. The refreshed shares are $\left(x'_0, \ldots, x'_d\right)$, with $x'_i = x_i + r'_i$ for $i \in \{0, \ldots, d\}$.[1]

2. The first d refreshed input shares, $(x'_0, \ldots, x'_{d-1})$, are securely processed by the HO-TSM$_{d-1}$ block. This block computes the outputs of all $h_{\pi(I)}$ functions (as defined in Eq. (3.22)), producing intermediate d-shared outputs:

$$\overline{h_{\pi(I)}}(x'_0, \ldots, x'_{d-1}) = \left(h_{\pi(I)}(\cdot)|_0, h_{\pi(I)}(\cdot)|_1, \ldots, h_{\pi(I)}(\cdot)|_{d-1}\right),$$

where $h_{\pi(I)}(\cdot)\big|_i$ represents the i-th share of the function $h_{\pi(I)}$, evaluated over the first d shares $(x'_0, \ldots, x'_{d-1})$.

3. Each of the d shares $h_{\pi(I)}(\cdot)\big|_i$ is refreshed using a dedicated random word r_i^{d-1}, where each r_i^{d-1} consists of 2^k bits (one random bit per subset $\pi(I)$). Altogether, $d \cdot 2^k$ fresh random bits are required in this step. Specifically, $h_{\pi(I)}(\cdot)\big|_i$ is refreshed as $\left(h_{\pi(I)}(\cdot)\big|_i + r_i^{d-1}[\pi(I)]\right)$. A corresponding new share $r_d^{d-1} = \sum_{i=0}^{d-1} r_i^{d-1}$ is also generated in this step.

4. The refreshed outputs

$$\left\{ h_{\pi(I)}(\cdot)\big|_i + r_i^{d-1}[\pi(I)] \right\} \quad \text{for } i = 0, \ldots, d-1$$

along with the newly generated share r_d^{d-1}, are stored in a register. Consequently, the register now contains $(d+1)$ shares for each $h_{\pi(I)}$ function. This process ensures that $h_{\pi(I)}(\cdot)$, which was previously shared across d shares, is now securely represented in a $(d+1)$-share domain.

5. In the last phase, the remaining refreshed input share x'_d is processed to compute the outputs of the $g^d_{\pi(I)}$ functions. These results are then combined with the refreshed outputs of $h_{\pi(I)}$ functions, which were previously computed by the HO-TSM$_{d-1}$ block, to produce

[1] Due to the recursive structure, the input shares may be re-shared multiple times. Although one could attempt to reduce this overhead, it is not the main source of randomness usage and the current approach allows for a clean recursion and straightforward security arguments.

the final $(d+1)$-shared outputs $\big(F_0[m], \ldots, F_d[m]\big)$. The output equations are as follows:

$$\begin{aligned} F_i[m] &= \sum_{I \in \mathcal{P}_k} \left(\left\{ h_{\pi(I)}(\cdot)\big|_i + r_i^{d-1}[\pi(I)] \right\} \cdot \sum_{j \in J_{m,\pi(I)}} g_j^d(x'_d) \right) && \text{for } i < d, \\ F_d[m] &= \sum_{I \in \mathcal{P}_k} \left(\left\{ r_d^{d-1}[\pi(I)] \right\} \cdot \sum_{j \in J_{m,\pi(I)}} g_j^d(x'_d) \right) && \text{for } i = d. \end{aligned} \tag{3.25}$$

The recursion proceeds iteratively, reducing the order at each step until it reaches the base case of first-order masking, HO-TSM$_1$ (i.e., TSM), which is securely implemented using the original first-order method of [158].

Extending HO-TSM to Vectorial Functions

Extending HO-TSM to vectorial Boolean functions $(f : \mathbb{F}_2^k \to \mathbb{F}_2^\ell)$ involves applying the decomposition described for a single coordinate function $f[m]$, $m \in \{0, \ldots, \ell-1\}$, to each coordinate. The key observation is that the sets $J_{m,\pi(I)}$ (as defined in Eq. (3.17)), which determine which g^d-functions are summed together for each coordinate, need to be updated to reflect the ANF of each $f[m]$. Importantly, the sets $J_{m,\pi(I)}$ constructed for TSM can be directly reused in HO-TSM, as they depend solely on the structure of the ANF of $f[m]$. Furthermore, the refreshed and registered shares of $h_{\pi(I)}$, along with the functions $g^d_{\pi(I)}$, which decompose the terms across the $(d+1)$-share domain, are universal and can be reused across all coordinates of f. This ensures an efficient extension of HO-TSM to vectorial functions by leveraging the same decomposition principles, Eq. (3.24), and building blocks, Eq. (3.22), for all coordinates.

Method	# Registers	# Register Layers	# Random Bits
HO-TSM$_d$	$2^k \frac{d(d+3)}{2} + k\frac{d(d+1)}{2}$	d	$2^k \frac{d(d+1)}{2} + k\frac{d(d+1)}{2}$
GLM [74]	$(d+1)^k k$	1	$(d+1)^k k$

Table 3.4: Comparison of algorithmic costs of HO-TSM and GLM for order d when masking a function $\mathbb{F}_2^k \to \mathbb{F}_2^k$ function of algebraic degree $k-1$.

We provide the algorithmic costs of number of registers, register stages, and random bits in Table 3.4. Comparing HO-TSM with other methods is difficult as it easily becomes dependent on the actual function that is implemented. Nevertheless, we compare to the single cycle case of the GLM method where we clearly see that we trade-off the number of registers and randomness for more register layers. A better view of the cost of HO-TSM is given in Section 3.3.3 where it is applied to create a second-order masked AES S-box.

Correctness and Security

We begin with the correctness and security proof for HO-TSM$_d$ and then provide SILVER results for some small examples.

Theorem 2. *Any circuit secured by HO-TSM$_d$ is correct.*

Proof. We establish the correctness of HO-TSM$_d$ by induction on d, the order of the masking.

- **Base Case** ($d = 1$): When $d = 1$, HO-TSM$_1$ is simply the first-order two-share masking (TSM). As shown in Section 3.3.2 and established in [158], we have, for each output coordinate m: $F_0[m] + F_1[m] = f[m](x_0 + x_1)$, and therefore $F_0(\bar{x}) + F_1(\bar{x}) = f(x_0 + x_1)$. Hence, correctness holds for $d = 1$.
- **Inductive Hypothesis:** Assume that HO-TSM$_{d-1}$, which produces d-shared outputs, satisfies correctness. That is, for some $(d-1) \geq 1$, the outputs $F_0[m], \ldots, F_{d-1}[m]$ satisfy $\sum_{i=0}^{d-1} F_i[m] = f[m](x_0 + x_1 + \cdots + x_{d-1})$, meaning each coordinate function $f[m]$ is correctly reconstructed from the sum of all d-output shares in the HO-TSM$_{d-1}$ design.
- **Inductive Step** (d): To complete the proof, we verify that HO-TSM$_d$ correctly produces $(d+1)$-shared outputs satisfying the required property. By construction:

  1. The input shares $(x_0, \ldots, x_d)$, refreshed with randomness $r' = (r'_0, \ldots, r'_d)$ for composable security, do not affect correctness as the refreshing process preserves $\sum_{i=0}^{d} x_i$. This is because $\sum_{i=0}^{d} r'_i = \sum_{i=0}^{d-1} r'_i + r'_d = 0$.
  2. HO-TSM$_{d-1}$ is applied to the first d refreshed shares $(x'_0, \ldots, x'_{d-1})$, yielding intermediate outputs:
  $$\overline{h_{\pi(I)}(\cdot)} = \left(h_{\pi(I)}(\cdot)\big|_0, h_{\pi(I)}(\cdot)\big|_1, \ldots, h_{\pi(I)}(\cdot)\big|_{d-1}\right),$$
  whose sum of shares equals $\sum_{i=0}^{d-1} h_{\pi(I)}(\cdot)\big|_i = h_{\pi(I)}(x'_0, \ldots, x'_{d-1})$. By the inductive hypothesis, HO-TSM$_{d-1}$ is correct.
  3. Each share $h_{\pi(I)}(\cdot)\big|_i$ is refreshed with randomness $r_i^{d-1}[\pi(I)]$, producing $\left\{h_{\pi(I)}(\cdot)\big|_i + r_i^{d-1}[\pi(I)]\right\}$. Adding the refreshed d-shares and the extra random share $r_d^{d-1} = \sum_{i=0}^{d-1} r_i^{d-1}$, the randomness cancels out, recovering: $\sum_{i=0}^{d-1}\left(h_{\pi(I)}(\cdot)\big|_i + r_i^{d-1}[\pi(I)]\right) + r_d^{d-1}[\pi(I)] = h_{\pi(I)}(x'_0, \ldots, x'_{d-1})$.
  4. The functions $g^d_{\pi(I)}$, which operate exclusively on the last refreshed share x'_d, are then computed. These results are combined with the refreshed shares of $h_{\pi(I)}(\cdot)$, producing the final $(d+1)$-shared outputs $(F_0[m], \ldots, F_d[m])$. Summing the outputs gives:

$$\begin{aligned}\sum_{i=0}^{d} F_i[m] &= \sum_{I \in \mathcal{P}_k} \left(h_{\pi(I)}(x'_0, \ldots, x'_{d-1}) \cdot \sum_{j \in J_{m,\pi(I)}} g_j^d(x'_d)\right) \\ &= f[m](x'_0 + \cdots + x'_d) = f[m](x_0 + \cdots + x_d).\end{aligned}$$

This correctness argument applies independently to all coordinates $f[m]$, $m \in \{0, \ldots, \ell - 1\}$, as the sets $J_{m,\pi(I)}$ ensure proper reconstruction for each $f[m]$ based on its ANF. The final $(d+1)$-shared outputs for the vectorial function f are obtained by concatenating the outputs for all coordinates, ensuring that $F_0(\bar{x}) + F_1(\bar{x}) + \cdots + F_d(\bar{x}) = f(x_0 + x_1 + \cdots + x_d)$, where $F_i(\bar{x}) = \left(F_i[0], \ldots, F_i[\ell - 1]\right)$ for $i \in \{0, \ldots, d\}$.

□

Theorem 3. *Any circuit secured by HO-TSM$_d$ is d^{th}-order glitch-extended PINI and SNI.*

Proof. We prove the theorem by induction on d.

- **Base Case** ($d = 1$): From [158], it is established that the method is secure for $d = 1$. We observe in the proof that the simulation can be performed from scratch without giving any input shares when probing an output.

- **Induction Hypothesis:** Assume that any circuit secured by HO-TSM$_{d-1}$ is $(d-1)^{\text{th}}$-order glitch-extended PINI and SNI.
- **Induction Step** (d): We aim to show that if the theorem holds for HO-TSM$_{d-1}$, then it also holds for HO-TSM$_d$. Please refer to Figure 3.13 for reference. To demonstrate d^{th}-order PINI and SNI security, we use a simulation-based argument. First, we categorize all possible probe locations as follows:

  1. Probing the HO-TSM$_{d-1}$ block.
  2. Probing the i^{th} share in the upper register.
  3. Probing the lower register.
  4. Probing the i^{th} output share.

  Next, we construct the simulator $\mathcal{S}$. By the induction hypothesis, the HO-TSM$_{d-1}$ block is $(d-1)^{\text{th}}$-order PINI and SNI secure, and we can utilize its simulator $\mathcal{S}'$. Depending on the probe positions, we provide the following shares to $\mathcal{S}$:

  – Probes in Group 1: We provide $\mathcal{S}$ the same input which would be given to $\mathcal{S}'$.
  – Probes in Group 2: We provide the simulator with nothing.
  – Probes in Group 3: We provide the simulator with x_d.
  – Probes in Group 4: We provide the simulator with nothing.

  With the above input shares, the simulator $\mathcal{S}$ can perfectly simulate the probes as follows:

  – For any probe in Group 1: By the induction hypothesis, $\mathcal{S}'$ perfectly simulates this probe. When d probes are placed in Group 1 (or 2), since the share x_d is excluded, the simulator can be given $(x_0, \ldots, x_{d-1})$ to perfectly simulate the entire HO-TSM$_{d-1}$ block.
  – For any probe in Group 2: A probe in this group views a single output of the HO-TSM$_{d-1}$ block. The simulator $\mathcal{S}'$ can perfectly simulate this output due to its $(d-1)^{\text{th}}$-order PINI and SNI security. The simulator $\mathcal{S}$ also creates the randomness r_i^{d-1}. If $i = d$, $\mathcal{S}$ creates all random shares r_i^{d-1} without needing to simulate any output of the HO-TSM$_{d-1}$ block.
  – For any probe in Group 3: The simulator has access to x_d and creates the randomness r'_d.
  – For any probe in Group 4: A probe in this group views one share of the upper register. Due to the randomness r_i^{d-1}, each value in the upper register can be simulated as uniform randomness unless a probe has already been placed in that position in Group 2 or when a probe was placed on r_d^{d-1}. In such cases, the simulation follows the logic of Group 2. Additionally, a probe in this group also views the lower register value. If a probe was placed in Group 3, the simulator has x_d and can simulate g^d. Furthermore, if d outputs of the HO-TSM$_{d-1}$ block need to be simulated, r'_d is also observed. However, since the adversary has only d probes, no probe could have been placed in Group 4 simultaneously. Otherwise, $r'_d = \sum_{i=0}^{d-1} r'_i$ remains unobserved, and the lower register can be simulated as uniform random, with g^d simulated appropriately.

The above demonstrates that the construction achieves d^{th}-order PINI and SNI security. Since both the base case and the induction step have been established, the theorem holds for all $d \geq 1$. □

Formal Verification. SILVER [93] is a formal verification tool that evaluates the security and composability of masked gadgets against several security notions such as probing security, Non-Interference (NI), Strong Non-Interference (SNI), and Probe-Isolating Non-Interference (PINI). We applied our HO-TSM method to create two small examples for different orders of security: a HO-TSM$_2$ AND gate, introduced in Section 3.3.3, and a HO-TSM$_3$ AND gate. Our two examples successfully pass all second-order and third-order tests, respectively, under all security notions (probing, NI, SNI, PINI) in both the standard and robust probing models. This confirms that our two gadgets are indeed (2-PINI,2-SNI) and (3-PINI,3-SNI) secure.

Application: Second-Order Masked AES S-Box

An unprotected AES S-Box takes eight input bits and produces eight output bits, described as $S : \mathbb{F}_2^8 \rightarrow \mathbb{F}_2^8$. The coordinate functions, when presented in their algebraic normal form, include all terms up to the seventh degree, in total 254 terms. To mask the AES S-Box using HO-TSM$_2$, we begin by computing the sharings for all 254 terms by processing each share individually in three phases, see Figure 3.12. These sharings are then commonly reused across all eight coordinate functions.
In total, for the first stage, we require 254 random bits (r^0) and 508 (254×2) registers. In the second stage, we require 508 (254×2) random bits (r^1) and 762 (254×3) registers. Additionally, 24 random bits (r') are required for the initial input refresh. The total utilization cost for the second-order secure AES S-Box, including the cost for combinatorial logic, is summarized in Table 3.5.

Table 3.5: Utilization results of low-latency second-order masked AES S-Boxes.

Design	Method	Area (kGE)	Random bits	Cycles	Library
This work	**HO-TSM**	**33.7**[a] / **48.6**[b]	**786**	**2**	**NanGate45**
[74]	GLM	57.1	4446	2	UMC90

[a] `compile_ultra -no_autoungroup -no_boundary_optimization`
[b] `compile -exact_map`

The synthesis results are gathered using the NanGate 45nm Open Cell Library [126]. We use Synopsys DC Compiler v2021.06 for Synthesis and provide results for two different sets of options. The first set of options (`compile_ultra -no_autoungroup -no_boundary_optimization`) aims for maximum logic optimization while making sure that no logic is moved across the register layer by separating the registers from logic by placing them in distinct modules. The second set of options (`compile -exact_map`) aims for a direct mapping of our design to logic. The area numbers were gathered by synthesizing our designs with a target frequency of 100 MHz and do not include the area cost of generating the required randomness. Our two-cycle second-order AES S-Box has both lower area as well as randomness cost when compared to the GLM design proposed by Gross *et al.* [74].

3.3.4 Further information

For more information about our higher-order low-latency Boolean masking scheme, an area-latency tradeoff, a case study with a complete AES-128 encryption core, concrete performance results, formal security evaluation, as well as results of a practical leakage assessment on FPGA, we refer the reader to our publication at TCHES 2025 [159].

3.4 Side-channel analysis of three designs in Tiny Tapeout board

3.4.1 Introduction

The primary objective of the work conducted on the Tiny Tapeout board was to perform an in-depth analysis of the discrepancies between theoretical methodologies and physical implementations in hardware security. Our goal was to better understand and reduce the gap between these two worlds. To this end, we focused on the implementation of three cryptographic gadgets that have been extensively studied in the literature from a side-channel analysis perspective, mostly in theoretical terms, with far less emphasis on practical implementation. The final aim is to gain deeper insights into how secure hardware behaves in practice and how design decisions at each level influence vulnerability to side-channel attacks.

Tiny Tapeout 02 project

Tiny Tapeout is a community-driven initiative that enables individuals (including students and hobbyists) to design and fabricate their own custom ASICs (Application-Specific Integrated Circuits) using open-source tools and affordable processes. The project was managed by Matt Venn and supported by Efabless, utilizing the Skywater 130nm **open source** PDK (Process Design Kit) [163].
Participants in the Tiny Tapeout project could create their designs using Wokwi, a graphical editor [113], or hardware description languages (HDL) such as Verilog, Amaranth, and Chisel. This flexibility allowed for a wide range of design approaches, from visual schematics to code-based designs.
The design has *compact constrains*; indeed each submission is required to fit within a 150 x 170 µm area, accommodating approximately 1,000 standard cells. This constraint encourages efficient and innovative designs. Moreover, to manage multiple designs on a single chip, Tiny Tapeout employs a *scan chain method*. This approach allowed each design to be accessed sequentially, facilitating testing and integration. The project fosters a collaborative environment, with participants sharing their designs and experiences.
Tiny Tapeout 02 (TT02) [164] was launched on November 9, 2022, and represented the second round of the Tiny Tapeout project, following the success of the initial run (Figure 3.14).
The easier way to communicate with the Tiny Tapeout board is to manually insert the number of the project to which you are referring and the values of the bits in input. In Figure 3.14, on the left is possible to read the field ”Input” and ”Select Project”, and in correspondence to these labels there are the switches to set these values. The value of the output can be read from the LED on the right of the board: indeed, each segment of the LED is turned on if the value of the correspondent bit is one, turned off if it is zero. See Section 3.4.2 for more information about how we set the board up for our acquisitions.
For TT02 a total of 165 projects were submitted, ranging from simple logic circuits to complex systems like RISC-V processors and programmable sound generators. The designs included both digital and analog components, reflecting the versatility of the platform. The whole project has been developed with the idea to be completely **open source**, and all the projects implemented in it can be found in the chip design repository [51]. On the project’s website [164] is possible to find the *Tiny Tapeout 02 Datasheet*, with the description of all the implemented design and the full Graphic Data Stream (GDS) of the board (Figure 3.15).

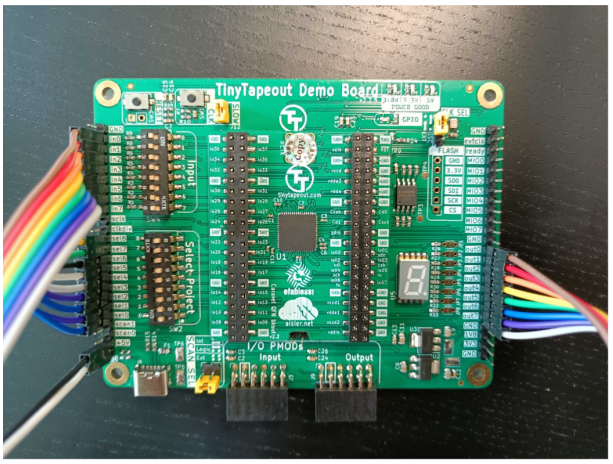


Figure 3.14: Photo of our Tiny Tapeout 02 board.

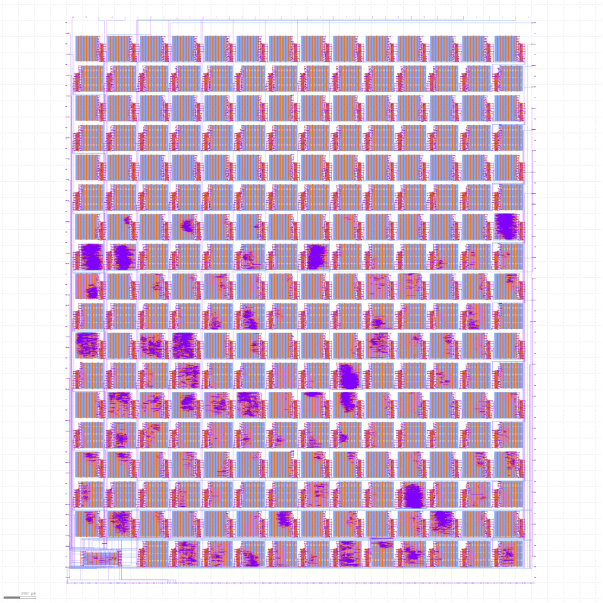


Figure 3.15: Full GDS for TT02, from the TT02 datasheet.

For the project TT02, SEC submitted three designs, called *chiDOM*, *chi2shares* and *chi3shares*. They are simplified versions of the application of three different countermeasures to the nonlinear function χ of Keccak [29, 77, 31]. Function χ (not protected) is applied each time on a subset of three bits of the Keccak row (composed of five bits). The function is described through the follow equation:

$$y_1 = \chi(x_1, x_2, x_3) = x_1 + (not(x_2) \cdot x_3) \tag{3.26}$$

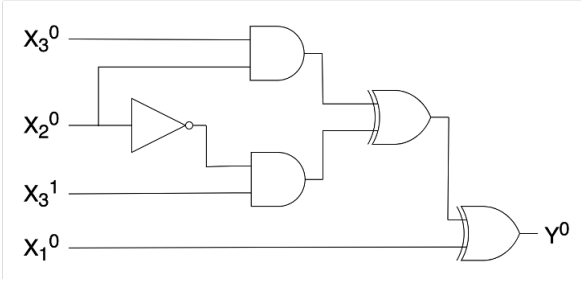
where $+$ is the *xor* operation and $\cdot$ is the *and* logic operation.

chi2shares and GDS

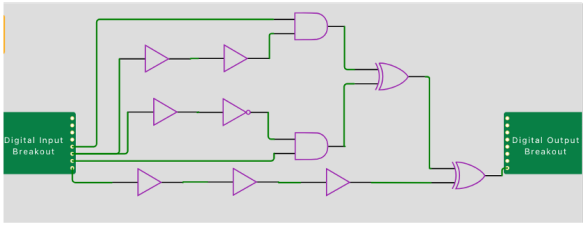
The function in Equation 3.26 is not protected against side channel attacks. As a countermeasure, it is possible to apply the two secret's sharing threshold implementation scheme. This means that each bit x in the Keccak row is split into two shares x^0 and x^1, such that $x = x^0 + x^1$. Then Equation 3.26 becomes:

$$\begin{aligned} y_1^0 &= x_1^0 + (not(x_2^0) \cdot x_3^1 + x_2^0 \cdot x_3^0) \\ y_1^1 &= x_1^1 + (not(x_2^1) \cdot x_3^0 + x_2^1 \cdot x_3^1) \end{aligned} \tag{3.27}$$

However, this sharing does not respect all the conditions listed in the definition of the Threshold Implementation scheme [128]; in particular, it does not respect the **non completeness** property. For that, to achieve independence from native variables, the order in which the operations are executed is important. If the expressions are evaluated from left to right, it can be shown that all intermediate variables are independent of the native variables. Then, although in software it is possible to provide provable resistance against side channel attacks, not always this goal can be reached in hardware. Indeed, signal propagation in circuit can not always be predictable and the order in which the operations are executed cannot always be monitored.
For the TT02 project, SEC submitted the design shown in Figure 3.16 a; it is the design of the first expression in Equation 3.27. In particular, our design was submitted through the Wokwi tool [3], and the design is shown in Figure 3.16 b. Note that we added some latches to the design with the scope to slow down some signals with respect to others, to simulate a case in which a leakage of the sensitive value x_3 could occur.



(a) Scheme of the function χ with two shares.



(b) Scheme of the function χ with two shares.

Figure 3.16: Schemes of χ function with TI 2 shares countermeasure.

GDS file We analysed the GDS file of the chi2shares design, and in Figure 3.19 is shown the circuit describing what has been synthetized on the TT02 chip. We noted that many operations have been replaced by a unique multiplexer, and all the latches have been removed. This level of optimization was unexpected, as the toolchain did not indicate during the submission phase that such optimizations would be applied. Our expectation was to see the design implemented exactly as it was modeled in Wokwi.

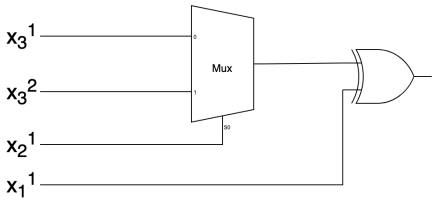


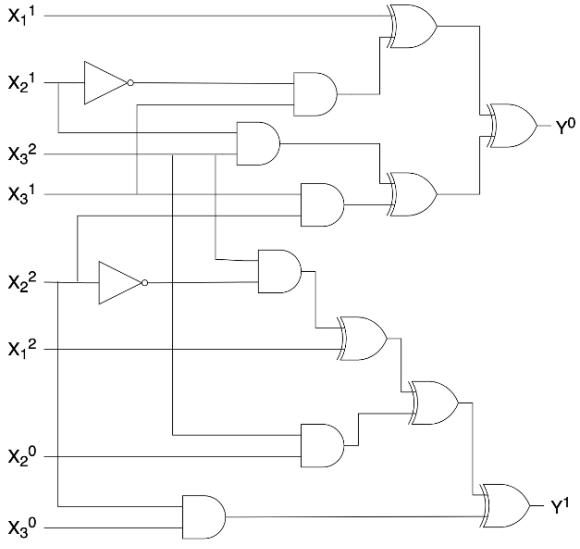
Figure 3.17: Scheme of the function χ with two shares from GDS file.

chi3shares and GDS

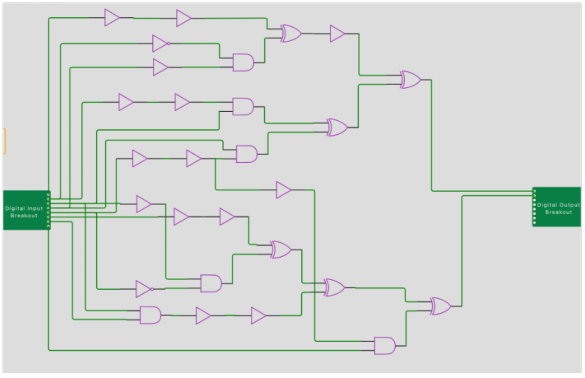
To respect all the conditions of the Threshold Implementation scheme, for χ we need to split all the sensitive variables into three shares. The equations of function χ with three shares are the followed:

$$\begin{aligned} y_1^0 &= x_1^1 + not(x_2^1) \cdot x_3^1 + x_2^1 \cdot x_3^2 + x_2^2 \cdot x_3^1 \\ y_1^1 &= x_1^2 + not(x_2^2) \cdot x_3^2 + x_2^2 \cdot x_3^0 + x_2^0 \cdot x_3^2 \\ y_1^2 &= x_1^0 + not(x_2^0) \cdot x_3^0 + x_2^0 \cdot x_3^1 + x_2^1 \cdot x_3^0 \end{aligned} \tag{3.28}$$

For the TT02 project, SEC submitted the design shown in Figure 3.18 a; it is the design of two expressions in Equation 3.28. In particular, our design was submitted through the Wokwi tool [2], and the design is shown in Figure 3.18 b.



(a) Scheme of the function χ with three shares.



(b) Scheme of the function χ with three shares.

Figure 3.18: Schemes of χ function with TI 3 shares countermeasure.

GDS file We analysed the GDS file of chi3shares design, and in Figure is shown the circuit describing what has been synthetized on the TT02 chi. We noted that two multiplexers have been added, which replace some operations. As for the previous gadget, this level of optimization was unexpected, as the toolchain did not indicate during the submission phase that such optimizations would be applied. Our expectation was to see the design implemented exactly as it was modeled in Wokwi.

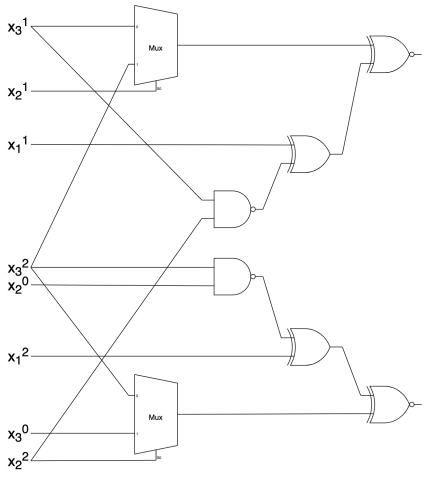


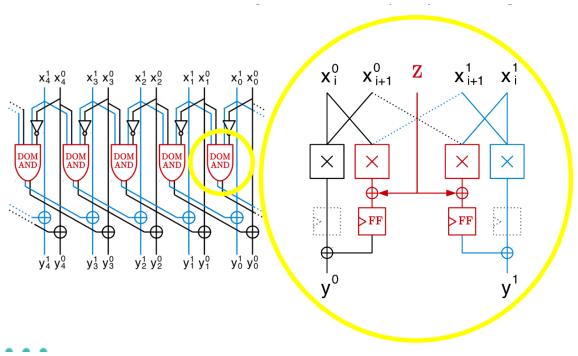
Figure 3.19: Scheme of the function χ with three shares from GDS file.

chiDOM and GDS

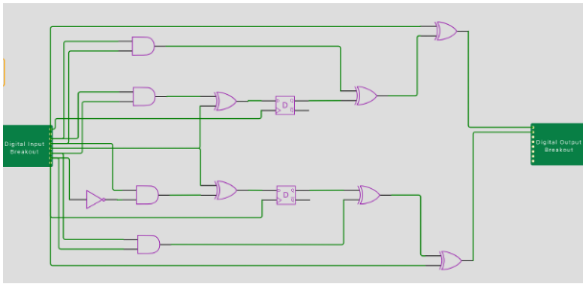
Another gadget that can be used as a countermeasure against side-channel attack is the Domain Oriented Masking scheme (DOM) [76]. In this case, the sensitive variables are split into two shares, and a register is added to store shares coming from different domains. The equation of χ function with DOM is the following:

$$\begin{aligned} y^0 &= x_1^0 \cdot x_2^0 + [x_1^0 \cdot x_2^1 + z] \\ y^1 &= x_1^1 \cdot x_2^1 + [x_1^1 \cdot x_2^0 + z] \end{aligned} \tag{3.29}$$

where z is a random bit and [] means that the values in the brackets are stored in the register. To TT02 project, SEC submitted the design shown in Figure 3.20 a; it is the design of the expressions in Equation 3.29. In particular, our design was submitted through the Wokwi tool [1], and the design is shown in Figure 3.20 b.



(a) Scheme of the function χ with DOM [77].



(b) Scheme of the function χ with DOM.

Figure 3.20: Schemes of χ function with DOM countermeasure.

GDS file We analysed the GDS file of chiDOM design, and in Figure 3.21 is shown the circuit describing what has been synthetized on the TT02 chip. In addition, in this case, the sinthetized version of the circuit does not correspond perfectly to the Wokwi version, because of the optimizations during the synthetization phase. As for the previous gadgets, this level of optimization

was unexpected, as the toolchain did not indicate during the submission phase that such optimizations would be applied. Our expectation was to see the design implemented exactly as it was modeled in Wokwi.

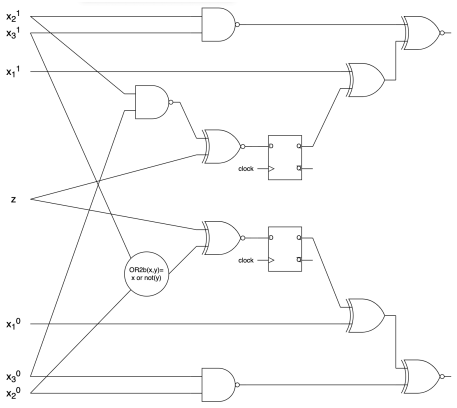
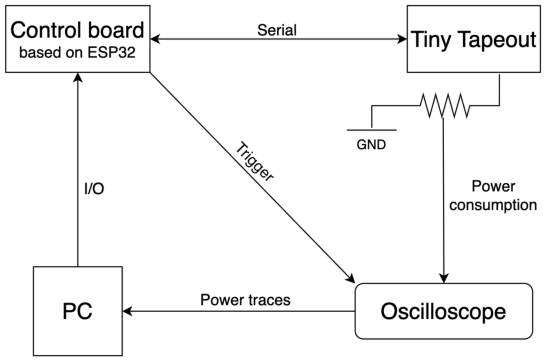


Figure 3.21: Scheme of the function χ with DOM from GDS file.

3.4.2 Acquisition setup

We received the board of Tiny Tapeout 02 in March 2024. Our aim was to verify the gadgets designed in the board via power analysis acquisition. The setup for the acquisition of the traces from the Tiny Tapeout board is shown in Figure 3.22.



(a) Scheme of the setup. [77].

(b) Photo of the setup in our lab.

Figure 3.22: Setup of the tools for the power traces acquisition.

The Tiny Tapeout board, as previously discussed in Sec.3.4.1, can be manually triggered using the onboard switches, which allow the user to select inputs and the project number. In addition to this manual mode, the board also supports serial communication, enabling a more flexible and automated method of interaction. For this purpose, we connected an ESP32 microcontroller to the Tiny Tapeout board via a USB interface. The ESP32 acts as an intermediary, sending input signals to the Tiny Tapeout and reading its outputs over the serial connection. This configuration enables the control of the board, facilitating repeated and precise experiments.
To analyze the power consumption of the board under various input conditions, we used a Teledyne LeCroy WaveSurfer 3000 series oscilloscope [87]. The oscilloscope probe was connected across two parallel resistors of 47Ω each, which are inserted into the power supply line to the

Tiny Tapeout board. This setup allows for accurate measurement of the voltage drop across the resistors, from which the current (and hence the power consumption) can be inferred. A host PC orchestrates the entire process: it sends input stimuli to the ESP32, which forwards them to the Tiny Tapeout board, and simultaneously captures output responses. Additionally, the PC interfaces with the oscilloscope to record the power traces corresponding to each test scenario.
The trigger is generated by the ESP32, so it is unclear when the operations are executed in Tiny Tapeout w.r.t. the trigger. This is why we performed an analysis of the timing of changes observed in the traces.

How to read the figures

In the following sections, we present a series of figures that illustrate the behavior of the power traces acquired using the oscilloscope during our experiments. These visualizations help provide insight into the power consumption patterns observed under various conditions.
All the traces that are compared have been acquired using the same oscilloscope settings and scale, ensuring that comparisons between them are consistent and meaningful. In each figure:

- The x-axis represents the sample index, corresponding to specific points in time during the acquisition process.
- The y-axis displays a value proportional to the voltage levels captured by the oscilloscope, which reflects the instantaneous power consumption of the device under test.

3.4.3 Acquisitions with the LED connected

Preliminary experiments

A primary consideration to keep in mind is that each power trace depends not only on the current input state but also on the previous input state. With this consideration in mind, our initial experiments were conducted under the following conditions.

1. A set of traces acquired with all the current and previous input bits set to zeros (in the next sections called *set 1*).
2. A set of traces acquired with all the current input bits set to ones, while the previous input bits were zeros (in the next sections called *set 2*).
3. A set of traces acquired with the current input bits set an half to zeros and an half to ones, while the previous input bits were zeros (in the next sections called *set 3*).
4. A set of traces acquired with all the current and previous input bits set to random values (in the next sections called *set 4*).

We decided to work on these sets of traces to understand how much the power traces depend on the Hamming distance between the previous and current states.
We performed this analysis on all the three gadgets we designed on the Tiny Tapeout. For each gadget, the first analysis is a Simple power analysis (SPA). This means that we tried to visually inspect some traces or use simple pattern-matching techniques to infer what the device is doing at different times.
The second step we performed for each gadget was to analyse the means of the traces in the four sets listed above.

For each set, we analyzed 1000 traces. All the traces were acquired at bandwidth of 200 Mhz and with a sample rate of 250 kilo samples per second, with a total of 50k samples per trace.
We report the results for all the gadget in the following sections.

chi2shares

The first gadget that we analysed was χ protected with two shares (Sec. 3.4.1).
The first analysis done is an inspection of the traces, as they are. Then in Figure 3.23 two traces are shown:

- In pink there is a trace acquired when the current input bits are set to ones, while the previous input bits were set to zeros;
- In brown there is a trace acquired when the current input bits are set to zeros, while the previous input bits were set to ones.

In Figure 3.23, the two graphs (pink and brown) alternate their voltage transitions. Directly from this very simple analysis we can distinguish almost clearly the two traces, and then from the traces we can understand which inputs generated them.
But not only: in fact, in Figure 3.24 we report the study of the means of the traces acquired in the four different situations previously described.

1. In blue the mean of the traces in the set *all the current and previous input bits set to zeros*.
   - Input bits are $[x_3^1, x_2^0, x_3^0, x_1^0] = [0, 0, 0, 0]$ and the output is $y_1 = 0$.
2. In orange the mean of the traces in the set *all the current input bits set to ones, while the previous input bits were zeros*.
   - Input bits are $[x_3^1, x_2^0, x_3^0, x_1^0] = [1, 1, 1, 1]$ and the output is $y_1 = 0$.
3. In green the mean of the traces in the set *the current input bits set an half to zeros and an half to ones, while the previous input bits were zeros*.
   - Input bits are $[x_3^1, x_2^0, x_3^0, x_1^0] = [1, 1, 0, 0]$ and the output is $y_1 = 1$.
4. In red the mean of the traces in the set *all the current and previous input bits set to random values*.

From these figures we note that:

- Means of traces in set 1 and in set 4 have similar behaviors, the only visible difference is the mean voltage measured, due to the different input and output values (for set 1 they are fixed to all bits zeros, in set 4 they vary).
- Close to the start of the operations both means of the traces in sets 2 and 3 have a gap from the voltage value before and after the start of the operations. The gap in set 2 is greater, almost two times w.r.t. the mean of traces in set 3 (more analysis in subsection *Reasonings on the results with LED connected*).

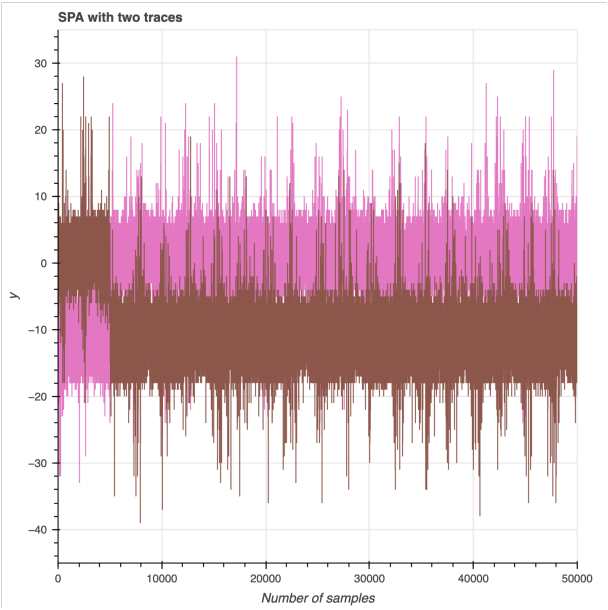
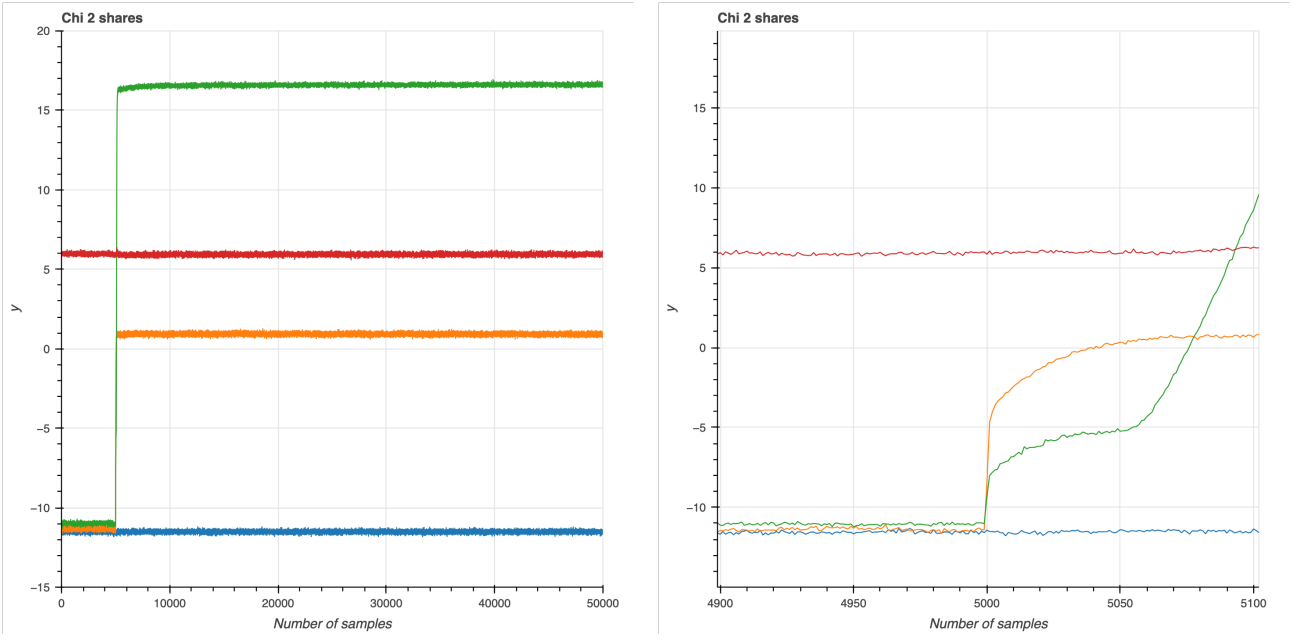


Figure 3.23: Simple Power Analysis for χ with two shares.



(a) (b)

Figure 3.24: Means of the traces in four different sets, with different bits in inputs to the χ gadget with two shares. Whole graph in (a) and a zoom around the start of the operations in (b).

chi3shares

Then we analysed the χ protected with three shares (Sec. 3.4.1).
In this case as well, the SPA is the initial analysis performed. In Figure 3.25 two traces (pink and brown) are shown that represent two opposite input bits transitions, as previously described for chi2shares. As for χ with two shares, also in this case we can clearly distinguish the two traces, and then from the traces we can understand which inputs generated them.
In Figure 3.26 we report the study of the means of the traces acquired in the usual four different situations.

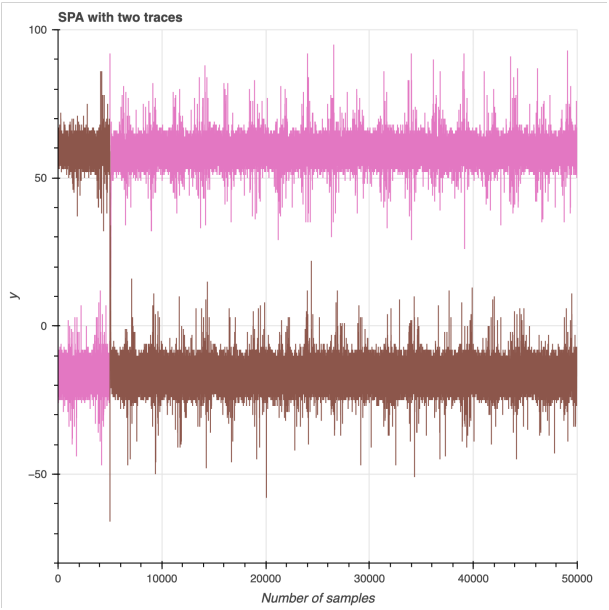


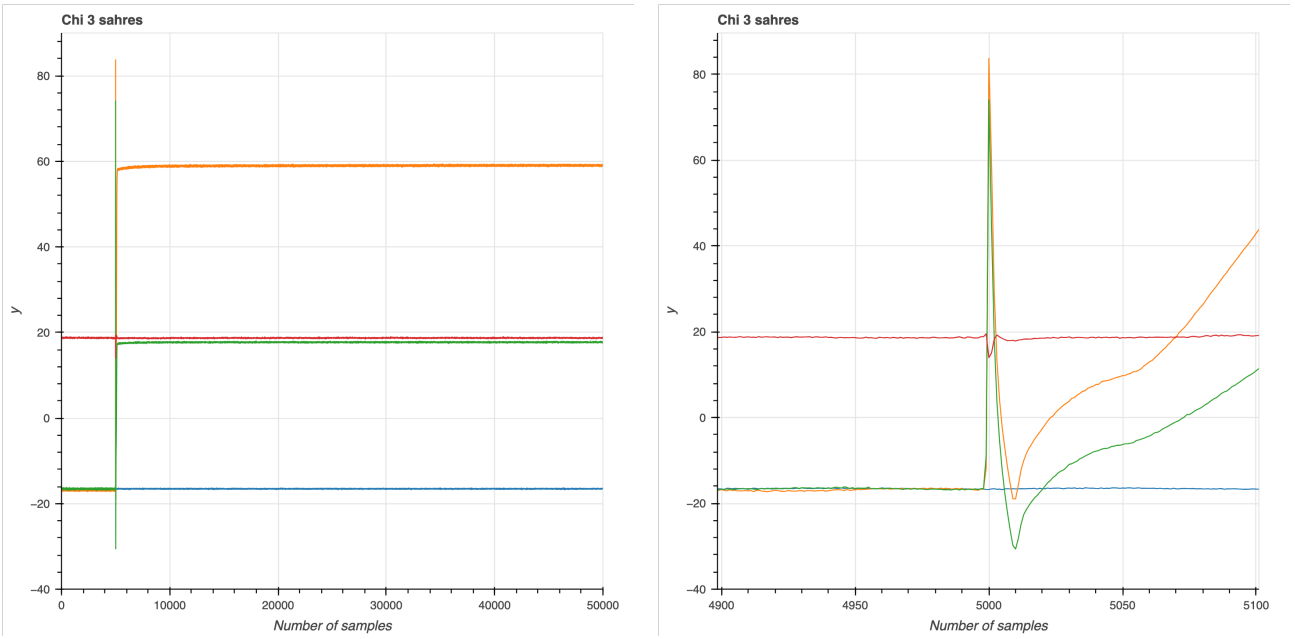
Figure 3.25: Simple Power Analysis for χ with three shares.

1. In blue the mean of the traces in the set *all the current and previous input bits set to zeros*.
   - Input bits are $[x_1^0, x_2^1, x_3^2, x_3^1, x_2^2, x_1^2, x_2^0, x_3^0] = [0, 0, 0, 0, 0, 0, 0, 0]$ and the output is $[y_1^1, y_1^2] = [0, 0]$.
2. In orange the mean of the traces in the set *all the current input bits set to ones, while the previous input bits were zeros*.
   - Input bits are $[x_1^0, x_2^1, x_3^2, x_3^1, x_2^2, x_1^2, x_2^0, x_3^0] = [1, 1, 1, 1, 1, 1, 1, 1]$ and the output is $[y_1^1, y_1^2] = [1, 1]$.
3. In green the mean of the traces in the set *the current input bits set an half to zeros and an half to ones, while the previous input bits were zeros*.
   - Input bits are $[x_1^0, x_2^1, x_3^2, x_3^1, x_2^2, x_1^2, x_2^0, x_3^0] = [1, 1, 1, 0, 0, 0, 0, 0]$ and the output is $[y_1^1, y_1^2] = [0, 1]$.
4. In red the mean of the traces in the set *all the current and previous input bits set to random values*.

From these figures we note that the behavior of the means is similar to that described for chi2shares. Also for this gadget, close to the start of the operations, both the means of the traces in sets 2 and 3 have a gap, but in this case the gap in set 3 is greater, almost two times w.r.t. the mean of traces in set 2 (more analysis in subsection *Reasonings on the results with LED connected*).

chiDOM

Then we analysed the χ protected with The DOM gadget (Sec. 3.4.1).
In this case as well, the SPA is the initial analysis performed. In Figure 3.27 two traces (pink and brown) are shown that represent two opposite input bits transitions, as previously described for the other two gadgets. Also in this case we can clearly distinguish the two traces, and then from the traces we can understand which inputs generated them.



(a) (b)

Figure 3.26: Means of the traces in four different sets, with different bits in inputs to the χ gadget with three shares. Whole graph in (a) and a zoom around the start of the operations in (b).

In Figure 3.28 we report the study of the means of the traces acquired in the usual four different situations.

1. In blue the mean of the traces in the set *all the current and previous input bits set to zeros*.
   - Input bits are $[x_1^0, x_2^0, x_3^0, z, x_3^1, x_2^1, x_1^1] = [0, 0, 0, 0, 0, 0, 0]$ and the output is $[y^0, y^1] = [0, 0]$.
2. In orange the mean of the traces in the set *all the current input bits set to ones, while the previous input bits were zeros*.
   - Input bits are $[x_1^0, x_2^0, x_3^0, z, x_3^1, x_2^1, x_1^1] = [1, 1, 1, 1, 1, 1, 1]$ and the output is $[y^0, y^1] = [0, 1]$.
3. In green the mean of the traces in the set *the current input bits set an half to zeros and an half to ones, while the previous input bits were zeros*.
   - Input bits are $[x_1^0, x_2^0, x_3^0, z, x_3^1, x_2^1, x_1^1] = [1, 1, 1, 0, 0, 0, 0]$ and the output is $[y^0, y^1] = [0, 1]$.
4. In red the mean of the traces in the set *all the current and previous input bits set to random values*.

From these figures, we note that the behavior of the means is similar to that described chi3shares (more analysis in subsection *Reasonings on the results with LED connected*).

Reasonings on the results with LED connected

The first noticeable aspect is the differing gaps in the mean graphs shown in Figures 3.24 a, 3.26 a, and 3.28 a. As previously discussed:

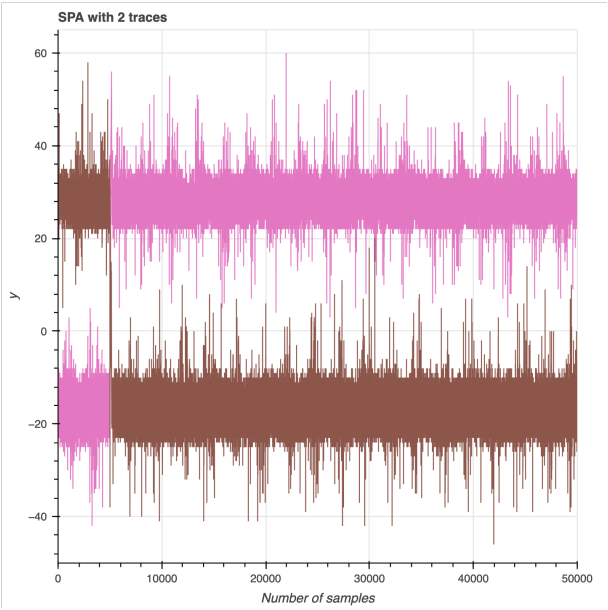
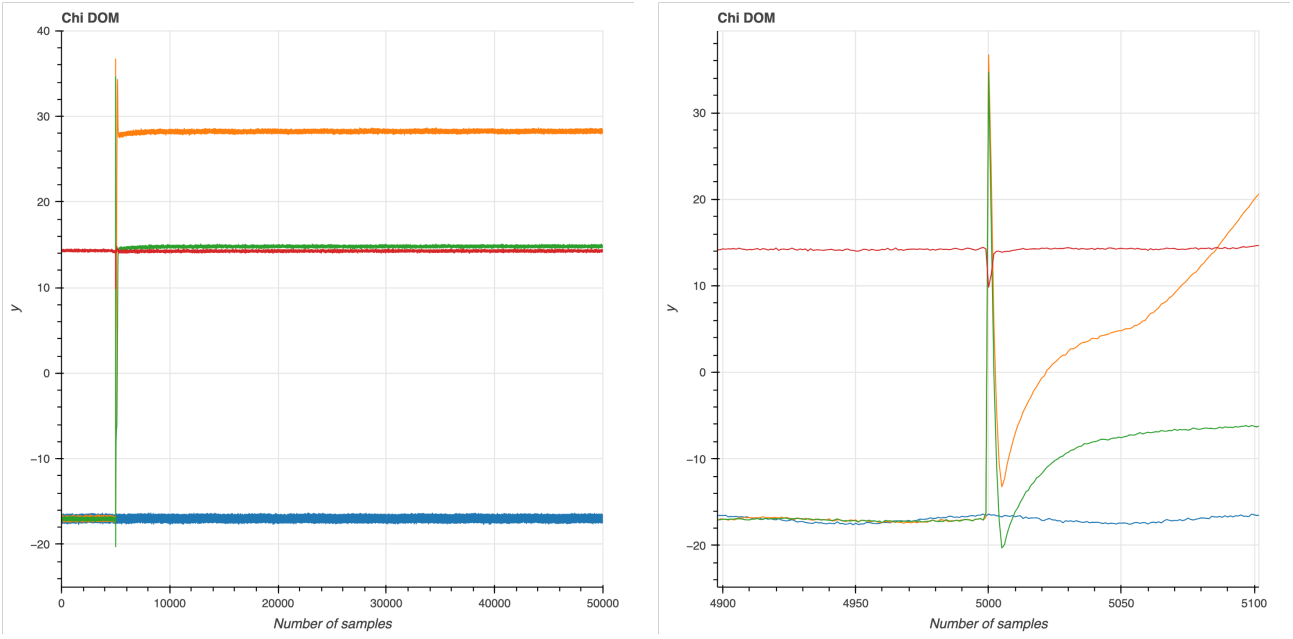


Figure 3.27: Simple Power Analysis for χ with DOM countermeasure.



(a) (b)

Figure 3.28: Means of the traces in four different sets, with different bits in inputs to the χ gadget with DOM. Whole graph in (a) and a zoom around the start of the operations in (b).

- For all gadgets, the mean traces for sets 1 and 4 exhibit a similar linear behavior across the sample range.
- Instead, for sets 2 and 3, all gadgets show a clear gap in the mean power consumption before and after the start of the operations.

But it is interesting to analyse the second case, taking into account the inputs/outputs for each gadget and set of traces:

- For χ with two shares, with half of the input bits value at one and half at zero (graph 3,

green graph), the power consumption gap is higher than in the case of all the input bits at one (graph 2, orange graph).

  - As discussed before, when the set is 2 the input bits are $[x_3^1, x_2^0, x_3^0, x_1^0] = [1, 1, 0, 0]$ and the output is $y_1^1 = 1$. When the set is 3 the input bits are $[x_3^1, x_2^0, x_3^0, x_1^0] = [1, 1, 1, 1]$ and the output is $y_1^1 = 0$.

- For χ with three shares, with half of the input bits value at one and half at zero (graph 3, green graph), the power consumption gap is lower than in the case of all the input bits at one (graph 2, orange graph).

  - As discussed before, when the set is 2 the input bits are $[x_1^0, x_2^1, x_3^2, x_3^1, x_2^2, x_1^2, x_2^0, x_3^0] = [1, 1, 1, 0, 0, 0, 0, 0]$ and the output is $[y_1^1, y_1^2] = [0, 1]$. When the set is 3 the input bits are $[x_1^0, x_2^1, x_3^2, x_3^1, x_2^2, x_1^2, x_2^0, x_3^0] = [1, 1, 1, 1, 1, 1, 1, 1]$ and the output is $[y_1^1, y_1^2] = [1, 1]$.

- For χ with the DOM gadget, with a half of the input bits value at one and a half at zero (set 3, green graph), the power consumption gap is lower than in the case of all the input bits at ones (set 2, orange graph).

  - As discussed before, when the set is 2 the input bits are $[x_1^0, x_2^0, x_3^0, z, x_3^1, x_2^1, x_1^1] = [1, 1, 1, 0, 0, 0, 0]$ and the output is $[y_1^1, y_1^2] = [0, 1]$. When the set is 3 the input bits are $[x_1^0, x_2^0, x_3^0, z, x_3^1, x_2^1, x_1^1] = [1, 1, 1, 1, 1, 1, 1]$ and the output is $[y^0, y^1] = [0, 1]$.

Based on the analysis of the previous results, we can infer that the gaps observed in the mean traces for sets 2 and 3 are strongly influenced by the Hamming weight of the outputs and, to a lesser extent, by the Hamming weight of the inputs. This can be the motivation for which for χ with two shares the difference between the gap in set 2 and the gap in set 3 is greater than the difference for χ with DOM.
Since we observed that the power consumption gaps appear to be strongly influenced by the Hamming weight of the outputs, and considering that in the Tiny Tapeout setup the output values control the LED segments on the board, we decided to cut the tracks connecting the LED. This prevents the LED from switching on in response to changes in the output.

3.4.4 Acquisitions with the LED disconnected

Preliminary experiments

After cutting the traces on the board that powered the LED, we repeated all the preliminary tests previously conducted with the LED actives.
As in the initial experiments, we ran tests on the four sets of power traces, each corresponding to a different combination of previous and current input states. This analysis was carried out for all three gadgets we implemented on the Tiny Tapeout chip. For each gadget, we performed both Simple Power Analysis (SPA) and an examination of the mean values of the acquired traces.
For each set, we analyzed 1.000 traces. All the traces were acquired at bandwidth of 200 mega Hertz and with a sample rate of 250 kilo samples per second, with a total of 50k samples per trace.

chi2shares

As in Section 3.4.3, we started with the analysis of the gadget χ protected with 2 shares TI.

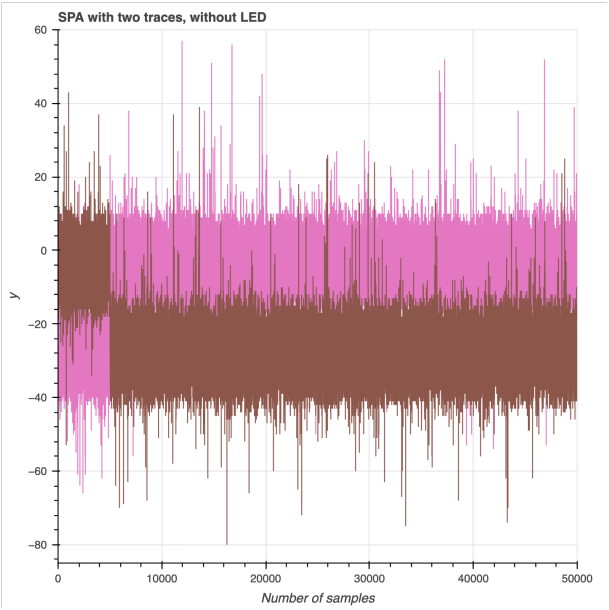


Figure 3.29: Simple Power Analysis for χ with two shares. Situation with LED disconnected.

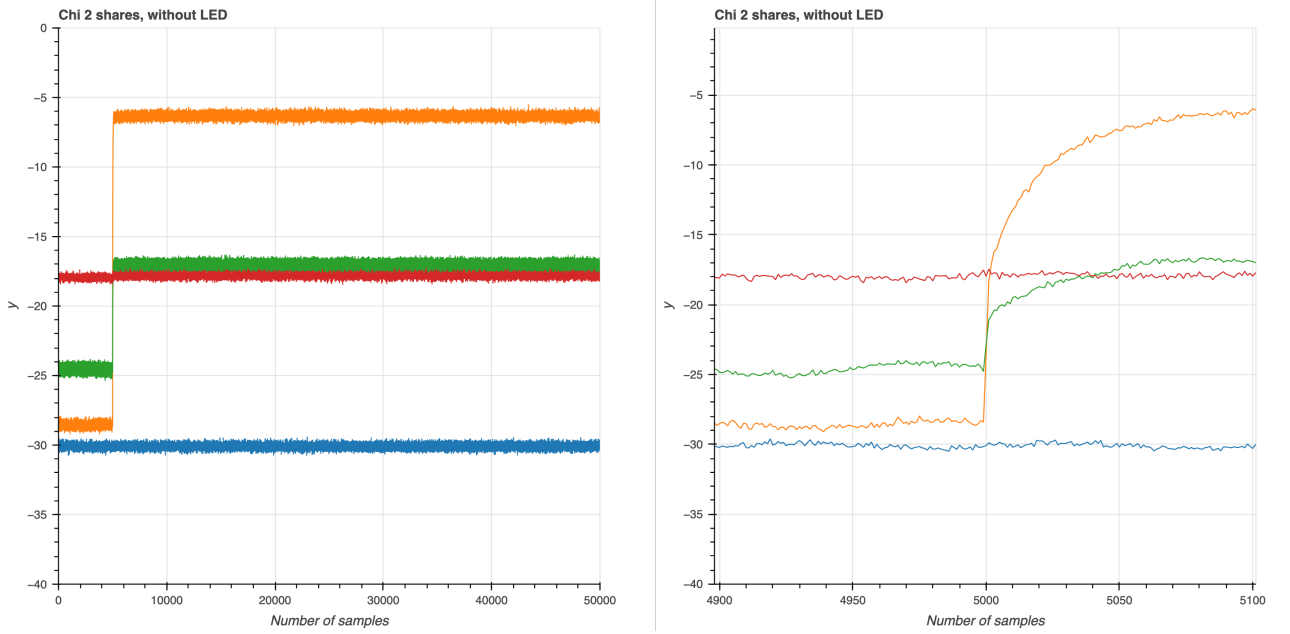
The first analysis done is an inspection of the traces, as they are. Then in Figure 3.29 two traces (pink and brown) are shown that represent two opposite input bits transitions, as previously described in section 3.4.3. Also in this case, similarly as for the case with LED connected, it is possible to distinguish almost clearly the two traces, and then from the traces we can understand which inputs generated them.
As in Section 3.4.3, we also analyzed the means of the traces acquired in the four different sets, as shown in Figure 3.30. The description of the sets and the inputs/outputs for each set is the same as in section 3.4.3. From these figures, we note that the behavior of the means is similar to that described in section 3.4.3, with the difference that the gap of the mean of the traces in set 3 is greater w.r.t. the mean of traces in set 2 (see subsection *Reasonings on the results with LED disconnected* for a deeper analysis).

chi3shares

Then we analysed the χ protected with three shares.
In this case as well, the SPA is the initial analysis performed. Then in Figure 3.31 two traces (pink and brown) are shown that represent two opposite input bits transitions, as previously described in section 3.4.3. Also in this case, similarly as for the case with LED connected, it is possible to distinguish almost clearly the two traces, and then from the traces we can understand which inputs generated them.
As in Section 3.4.3, we also analyzed the means of the traces acquired in the four different sets, as shown in Figure 3.32. The description of the sets and the inputs/outputs for each set is the same as in section 3.4.3. From these figures, we note that the behavior of the means is similar to that described in section 3.4.3 (see subsection *Reasonings on the results with LED disconnected* for a deeper analysis).

chiDOM

Then we analysed the χ protected with The DOM gadget.



(a) (b)

Figure 3.30: Means of the traces in four different sets, with different bits in inputs to the χ gadget with two shares. Whole graph in (a) and a zoom around the start of the operations in (b). Situation with LED disconnected.

In this case as well, the SPA is the initial analysis performed. Then in Figure 3.33 two traces (pink and brown) are shown that represent two opposite input bits transitions, as previously described in section 3.4.3. Also in this case, similarly as for the case with LED connected, it is possible to distinguish almost clearly the two traces, and then from the traces we can understand which inputs generated them.
As in Section 3.4.3, we also analyzed the means of the traces acquired in the four different sets, as shown in Figure 3.34. The description of the sets and the inputs/outputs for each set is the same as in section 3.4.3. From these figures, we note that the behavior of the means is similar to that described in section 3.4.3 (see subsection *Reasonings on the results with LED disconnected* for a deeper analysis).

Reasonings on the results with LED disconnected

The first observation is that, although the gaps in the mean traces for sets 2 and 3 are smaller after disconnecting the LED, they are still noticeable. This confirms that while power consumption is primarily influenced by the Hamming weight of the output, it is also affected by the Hamming weight of the inputs, albeit to a lesser extent.
The most significant result from this analysis emerges when examining the gadget χ with two shares. With the LED active, the gap in the mean trace for set 3 is nearly twice as large as that for set 2. However, after disabling the LED, this relationship is reversed. This change can be explained by considering the Hamming weights in each set:

- In set 2, the output has a Hamming weight of 0, while the input state has a Hamming weight of 4.
- In set 3, the output has a Hamming weight of 1, and the input state has a Hamming weight of 2.

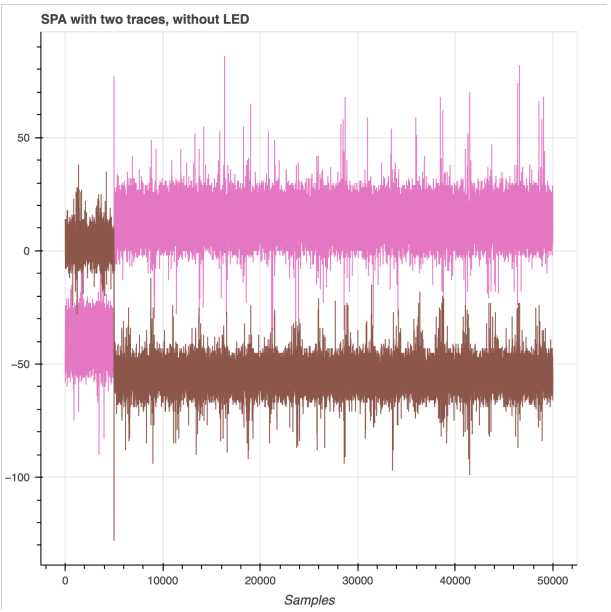
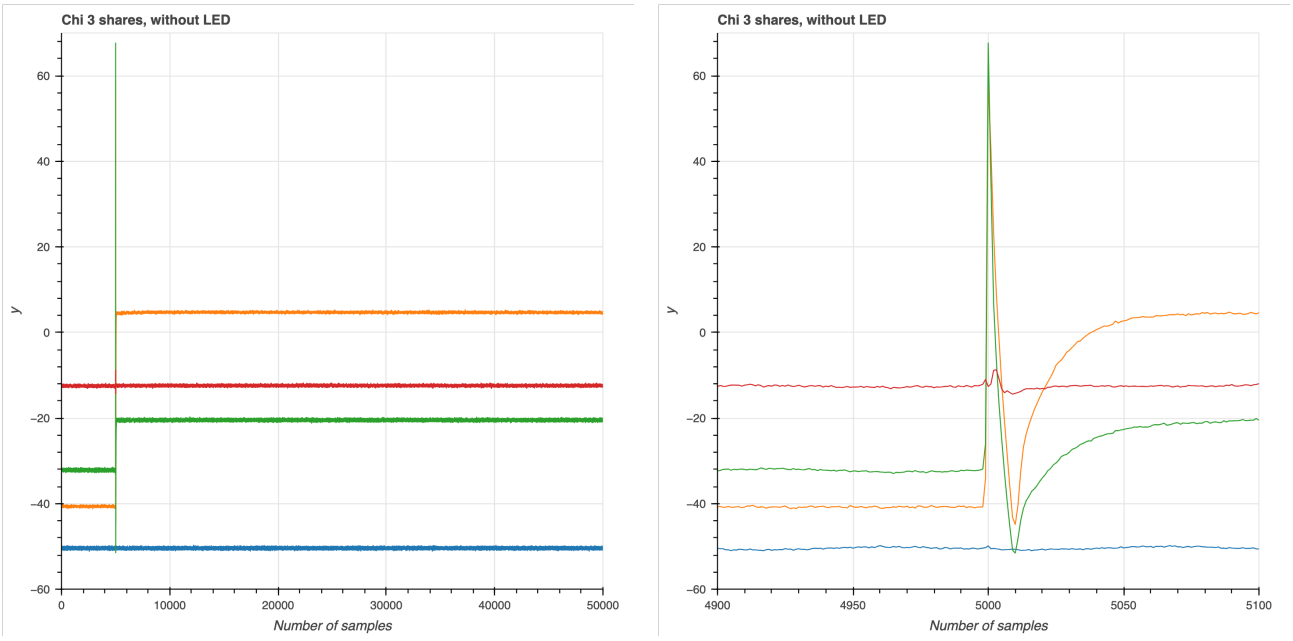


Figure 3.31: Simple Power Analysis for χ with three shares. Situation with LED disconnected.



(a) (b)

Figure 3.32: Means of the traces in four different sets, with different bits in inputs to the χ gadget with three shares. Whole graph in (a) and a zoom around the start of the operations in (b). Situation with LED disconnected.

This analysis reinforces the conclusion that the power traces are predominantly influenced by the Hamming weight of the output, but there is also a secondary dependence on the Hamming weight of the input.

Dependencies from the input

Based on the results discussed in Section 3.4.4, we decided to investigate more thoroughly how the power traces depend on both the Hamming weight and the Hamming distance of the input states of the gadgets. Our focus in this deeper analysis is specifically on the χ gadget with two

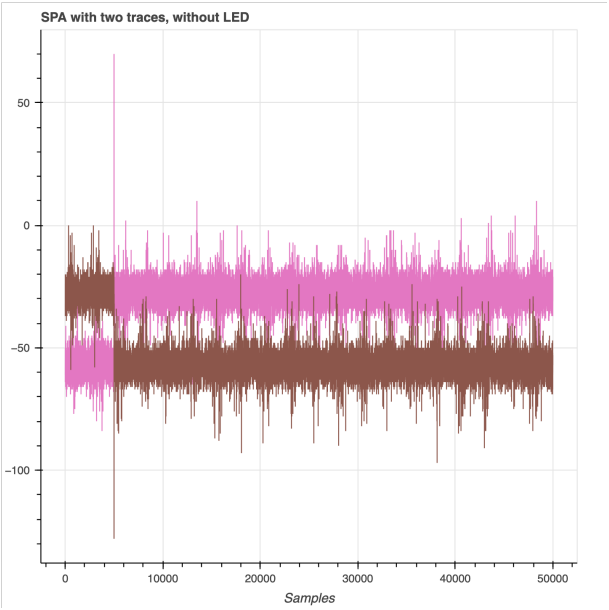
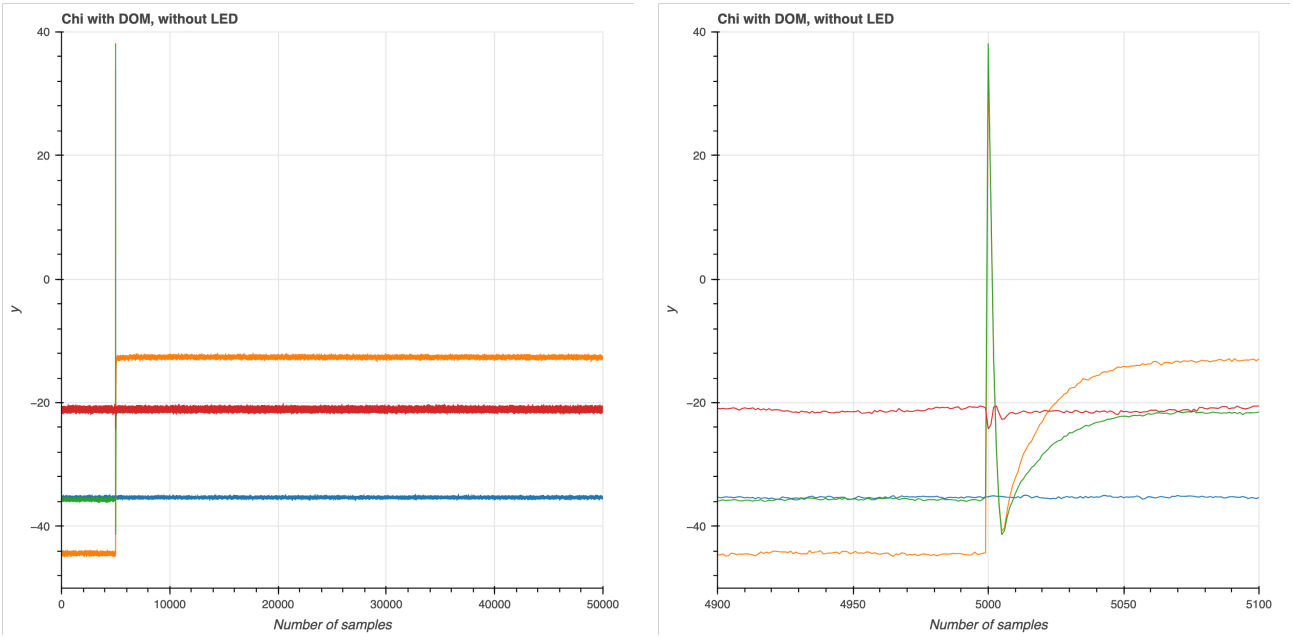


Figure 3.33: Simple Power Analysis for χ with DOM countermeasure. Situation with LED disconnected.



(a) (b)

Figure 3.34: Means of the traces in four different sets, with different bits in inputs to the χ gadget with DOM. Whole graph in (a) and a zoom around the start of the operations in (b). Situation with LED disconnected.

shares.

The first step in this investigation was to analyze the influence of the *Hamming weight of the input state*. To do this, we took a set of traces acquired using random inputs and grouped them into five categories, based on the Hamming weight of the current input state (which consists of four bits):

- Hamming weight 0: 0000
- Hamming weight 1: $1000, 0100, 0010, 0001$
- Hamming weight 2: $1100, 0110, 0011, 1010, 0101, 1001$

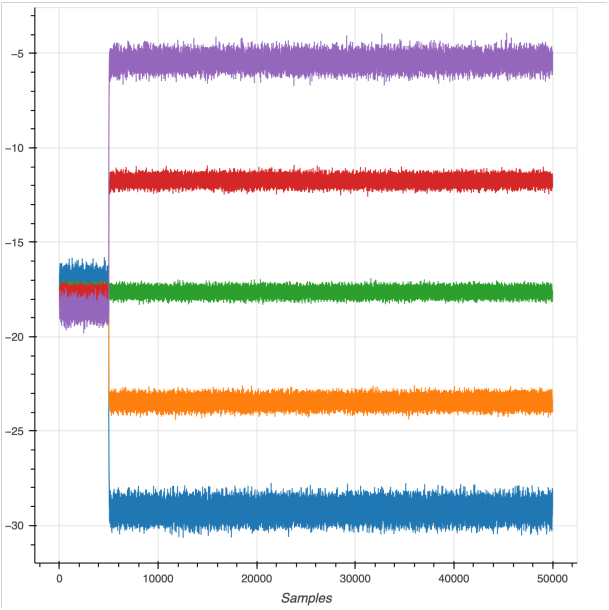


Figure 3.35: Mean of the traces with random inputs divided into five sets, depending on the Hamming weight of the input state. Blue: Hamming weight equal to 0. Orange: Hamming weight equal to 1. Green: Hamming weight equal to 2. Red: Hamming weight equal to 3. Purple: Hamming weight equal to 4. Situation with LED disconnected.

- Hamming weight 3: $1110, 0111, 1011, 1101$
- Hamming weight 4: 1111

Figure 3.35 shows the mean power trace for each of these five sets. It is clearly visible that the mean traces diverge and become distinguishable after the start of the operation, due to the fact that all traces within a set share the same Hamming weight for the current input. In contrast, before the operations begin, the mean traces are nearly identical across all sets. This is because the previous input state is random,even within each set, resulting in an averaged-out effect prior to the start of the operations.
Finally, we extended the analysis to include the Hamming distance. For the input states, this involved dividing the traces into five sets based on the Hamming distance between the current and previous input states:

- Hamming distance 0: previous input was the same as the current one
  - For example, previous input is 0000 and current input is 0000
- Hamming distance 1: current input differs from the previous one of one bit
  - For example, previous input is 0000 and current input is 1000
- Hamming distance 2: current input differs from the previous one of two bits
  - For example, previous input is 0000 and current input is 1100
- Hamming distance 3: current input differs from the previous one of three bits
  - For example, previous input is 0000 and current input is 1110
- Hamming distance 1: current input differs from the previous one of all the bits

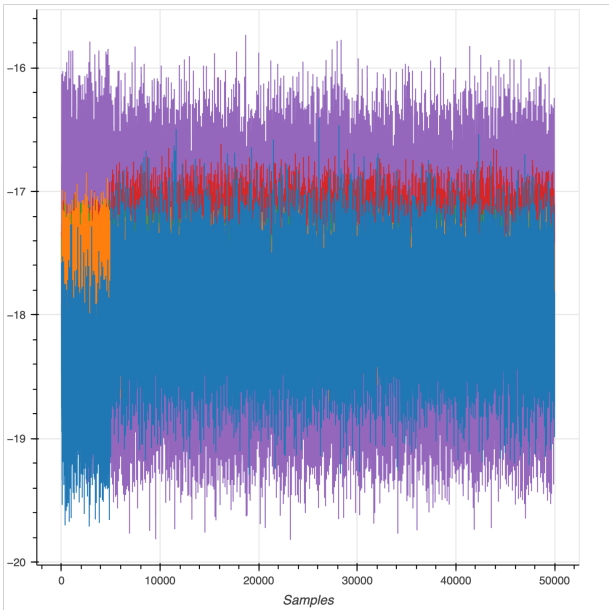


Figure 3.36: Mean of the traces with random inputs divided into five sets, depending on the Hamming distance between the current and previous inputs. Blue: Hamming distance equal to 0. Orange: Hamming distance equal to 1. Green: Hamming distance equal to 2. Red: Hamming distance equal to 3. Purple: Hamming distance equal to 4. Situation with LED disconnected.

  - For example, previous input is 0000 and current input is 1111

In this case, as shown in Figure 3.36, the mean power traces are not clearly distinguishable. This suggests that the power consumption does not significantly depend on the Hamming distance between the current and previous input states.

Deeper analysis of the gadgets with more acquired traces

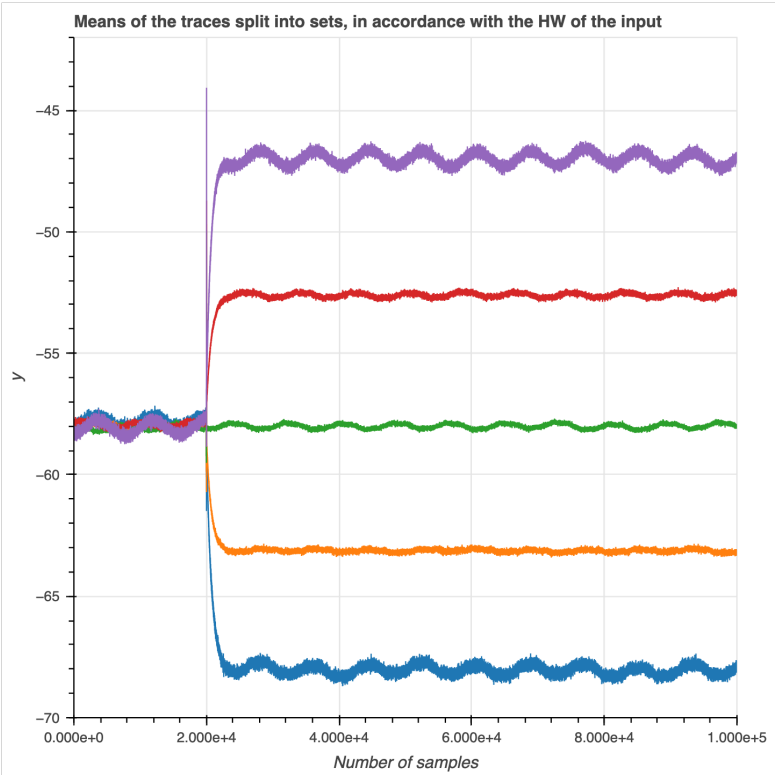
As a final step in our analysis, we collected 15.000 power traces for each gadget using random input values. All the traces were acquired at bandwidth of 200 mega Hertz and with a sample rate of 20 mega samples per second, with a total of 100k samples per trace.
For each gadget, we conducted two types of analysis:

1. Investigated whether the power consumption traces show a dependency on the Hamming weight of the input states.
2. Defined a selection function and assessed whether side-channel analysis techniques could be used to extract information about a secret value (see the corresponding sections for detailed explanations).

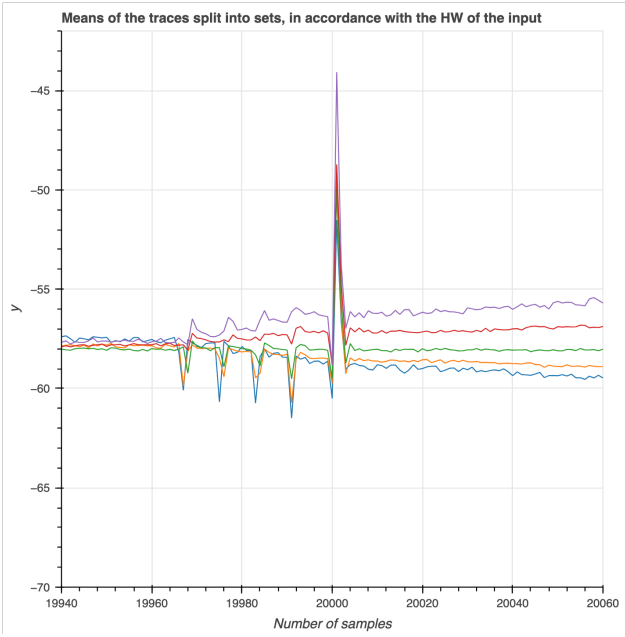
chi2sahres

As in the previous analyses, the first gadget examined is the χ with two shares in a Threshold Implementation (TI) scheme.

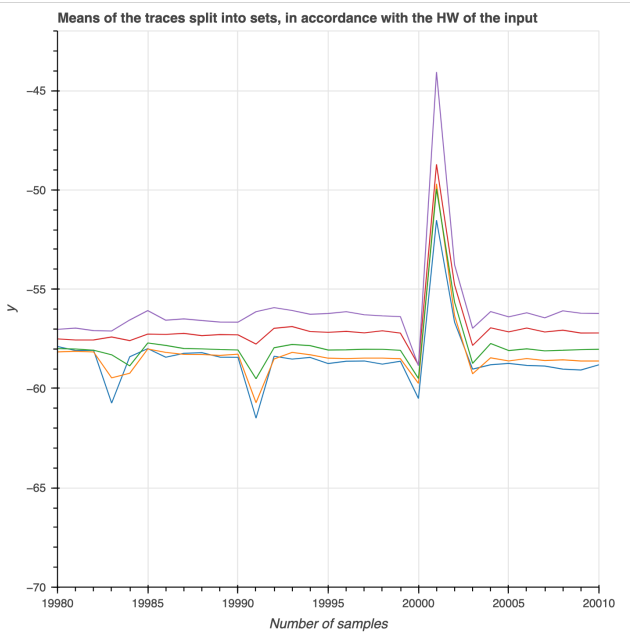
Dependencies from the input We began by investigating the correlation between the Hamming weight of the input states and the corresponding power traces, as shown in Figure 3.37. Also with 15.000 acquisitions, the results remain consistent with those observed in Figure 3.35, confirming a clear relationship between input Hamming weight and power consumption.



(a) Graph on all the samples [0,100.000].



(b) Zoom in the interval of samples [19.940,20.060]



(c) Zoom in the interval of samples [19.980,20.010]

Figure 3.37: Mean of the traces with random inputs divided into five sets, depending on the Hamming weight between the current and previous inputs. Blue: Hamming weight equal to 0. Orange: Hamming weight equal to 1. Green: Hamming weight equal to 2. Red: Hamming weight equal to 3. Purple: Hamming v equal to 4. Situation with LED disconnected. 15.000 traces acquired.

Difference of mean Next, we attempted to extract information about a secret value from the acquired power traces. The first step was to define a suitable selection function. Recalling the expression for our χ gadget with two shares:

$$y_1^0 = x_1^0 + (not(x_2^0) \cdot x_3^1 + x_2^0 \cdot x_3^0) \tag{3.30}$$

From this equation, we observe that:

- One share of x_1 appears: x_1^0
- One share of x_2 is used: x_2^0
- Both shares of x_3 are present: x_3^0 and x_3^1

Given this, we chose as a selection function the XOR of the two shares of x_3, aiming to recover some information about the secret value x_3:

$$g(x_3^0, x_3^1) = x_3^0 + x_3^1$$

We then divided the acquired traces into two sets based on the value of this selection function:

- M_0: traces for which $g(x_3^0, x_3^1) = 0$
- M_1: traces for which $g(x_3^0, x_3^1) = 1$

To assess potential leakage, we computed the Difference of Means (DoM) by subtracting the sample-wise mean of the traces in M_1 from those in M_0.
In Figure 3.38, the mean trace for M_0 is shown in blue, the mean for M_1 in orange, and the DoM in green. While the overall DoM remains close to zero for most samples, a closer look reveals that the mean traces for M_0 and M_1 are consistently separated (figure 3.38 b). Additionally, we observe an high peak in these curves, preceded by four smaller, regular patterns. The higher peak likely reflects the cryptographic computation, since they are close to the trigger; we didn't analyse the smaller peaks before, since we focused on what happens after the trigger.

chi3sahres

The second gadget examined is the χ with three shares in a Threshold Implementation (TI) scheme.

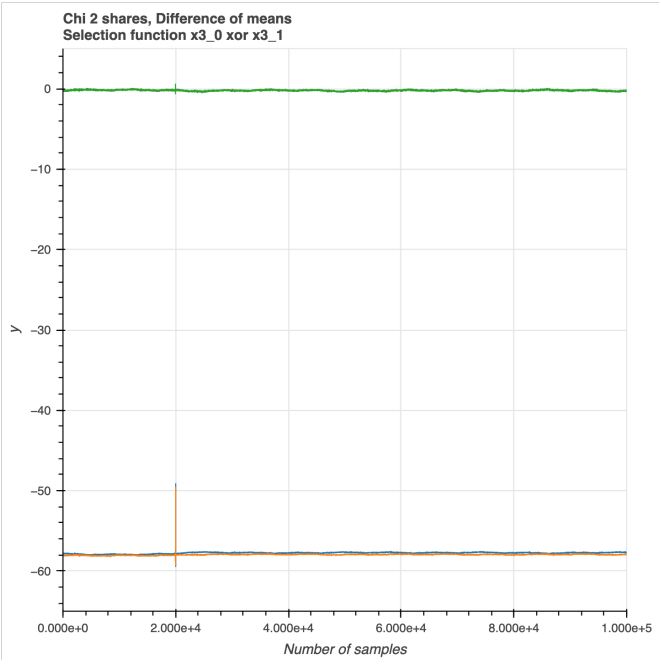
Dependencies from the input We began by investigating the correlation between the Hamming weight of the input states and the corresponding power traces, as shown in Figure 3.39. Similarly to the gadget with two shares, also for this gadget there is a clear relationship between input Hamming weight and power consumption.

Difference of mean Next, we attempted to extract information about a secret value from the acquired power traces. The first step was to define a suitable selection function. Recalling the expression for our χ gadget with three shares:

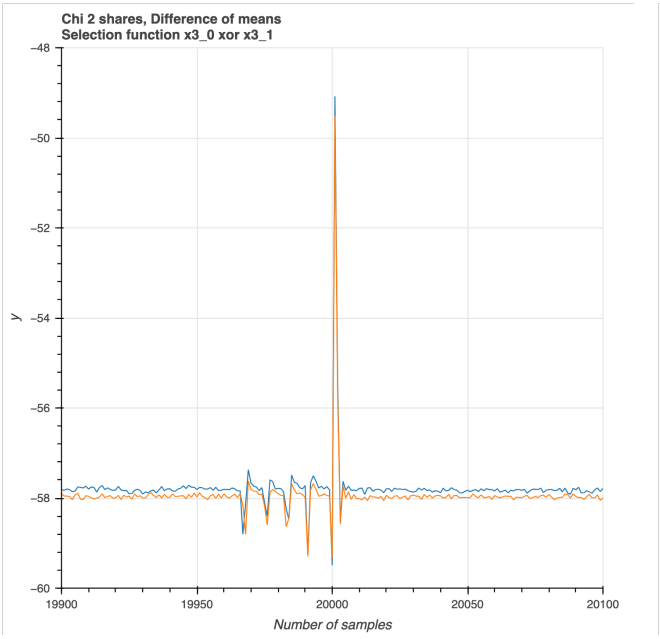
$$\begin{aligned} y_1^1 &= x_1^2 + not(x_2^2) \cdot x_3^2 + x_2^2 \cdot x_3^0 + x_2^0 \cdot x_3^2 \\ y_1^2 &= x_1^0 + not(x_2^0) \cdot x_3^0 + x_2^0 \cdot x_3^1 + x_2^1 \cdot x_3^0 \end{aligned} \tag{3.31}$$

From this equation, we observe that:

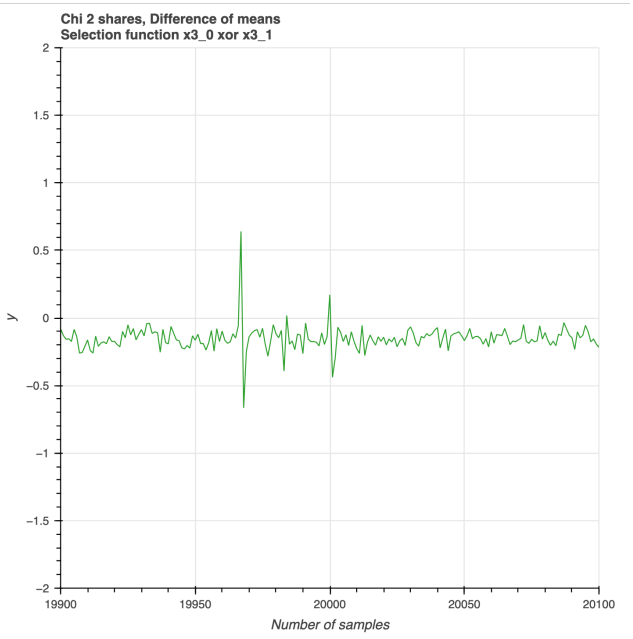
- Two shares of x_1 appears: x_1^0 and x_1^2
- Three shares of x_2 is used: x_2^0, x_2^1 and x_2^2



(a)



(b)



(c)

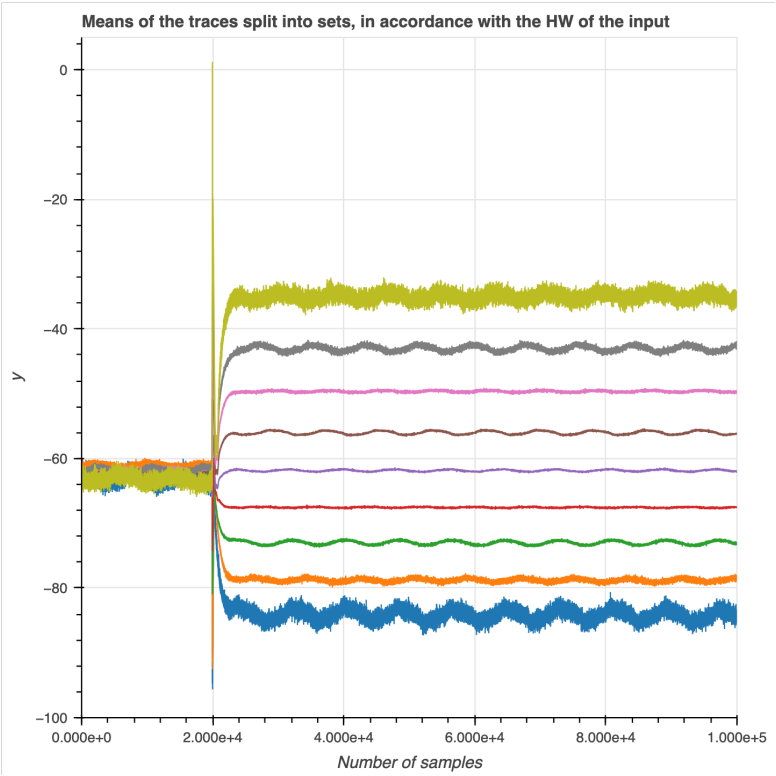
Figure 3.38: Selection function $g(x_3^0, x_3^1) = x_3^0 + x_3^1$. In blue, the mean of the traces in M_0. In orange, the mean of the traces in M_1. In green, the Difference of mean $M_1 - M_0$. Situation with LED disconnected. 15.000 traces acquired.

- Three shares of x_3 is used: x_3^0, x_3^1 and x_3^2

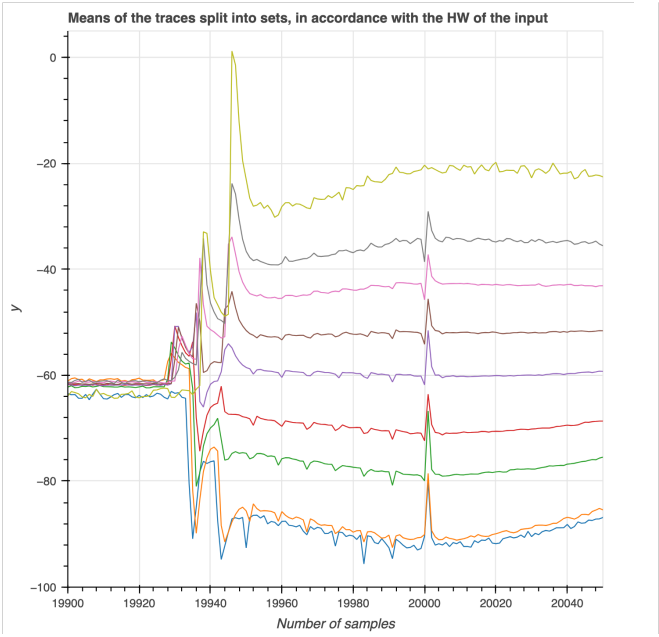
Given this, we chose two selection functions, aiming to recover some information about the secret values x_2 and x_3:

$$\begin{aligned} f_2(x_2^0, x_2^1, x_2^2) &= x_2^0 + x_2^1 + x_2^2 \\ f_3(x_3^0, x_3^1, x_3^2) &= x_3^0 + x_3^1 + x_3^2 \end{aligned} \tag{3.32}$$

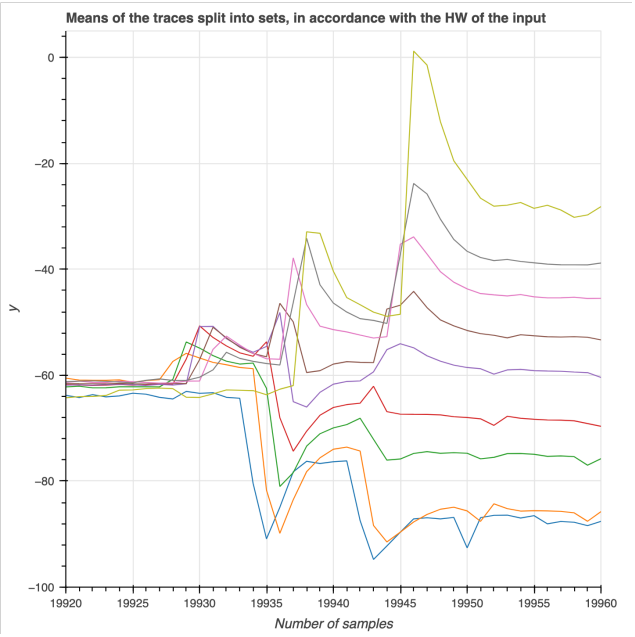
We then divided the acquired traces into two sets based on the value of this selection function:



(a) Graph on all the samples [0,100.000].



(b) Zoom in the interval of samples [19.900,20.040]



(c) Zoom in the interval of samples [19.920,19.960]

Figure 3.39: Mean of the traces with random inputs divided into nine sets, depending on the Hamming weight between the current and previous inputs. Blue: Hamming weight equal to 0. Orange: Hamming weight equal to 1. Green: Hamming weight equal to 2. Red: Hamming weight equal to 3. Purple: Hamming weight equal to 4. Brown: Hamming weight equal to 5. Pink: Hamming weight equal to 6. Grey: Hamming weight equal to 7. Gold: Hamming weight equal to 8. Situation with LED disconnected. 15.000 traces acquired.

- M_0: traces for which is $f_i(x_i^0, x_i^1, x_i^2) = 0$
- M_1: traces for which is $f_i(x_i^0, x_i^1, x_i^2) = 1$

for $i = 2, 3$.

To assess potential leakage, we computed the Difference of Means (DoM) by subtracting the sample-wise mean of the traces in M_1 from those in M_0.
In Figures 3.40 and 3.41, the mean trace for M_0 is shown in blue, the mean for M_1 in orange, and the DoM in green. While the overall DoM remains close to zero for most samples, a closer look reveals that the mean traces for M_0 and M_1 are consistently separated. Additionally, we observe an high peak in these curves, preceded by some smaller, regular patterns, as for the previous gadget, and we reach the same conclusions. TAfter the grater peak, we can note that the means of the traces do not go directly to zero, and this is a different behavior w.r.t. the case for χ with two shares, maybe due to the longer execution of the operations in χ with three shares.

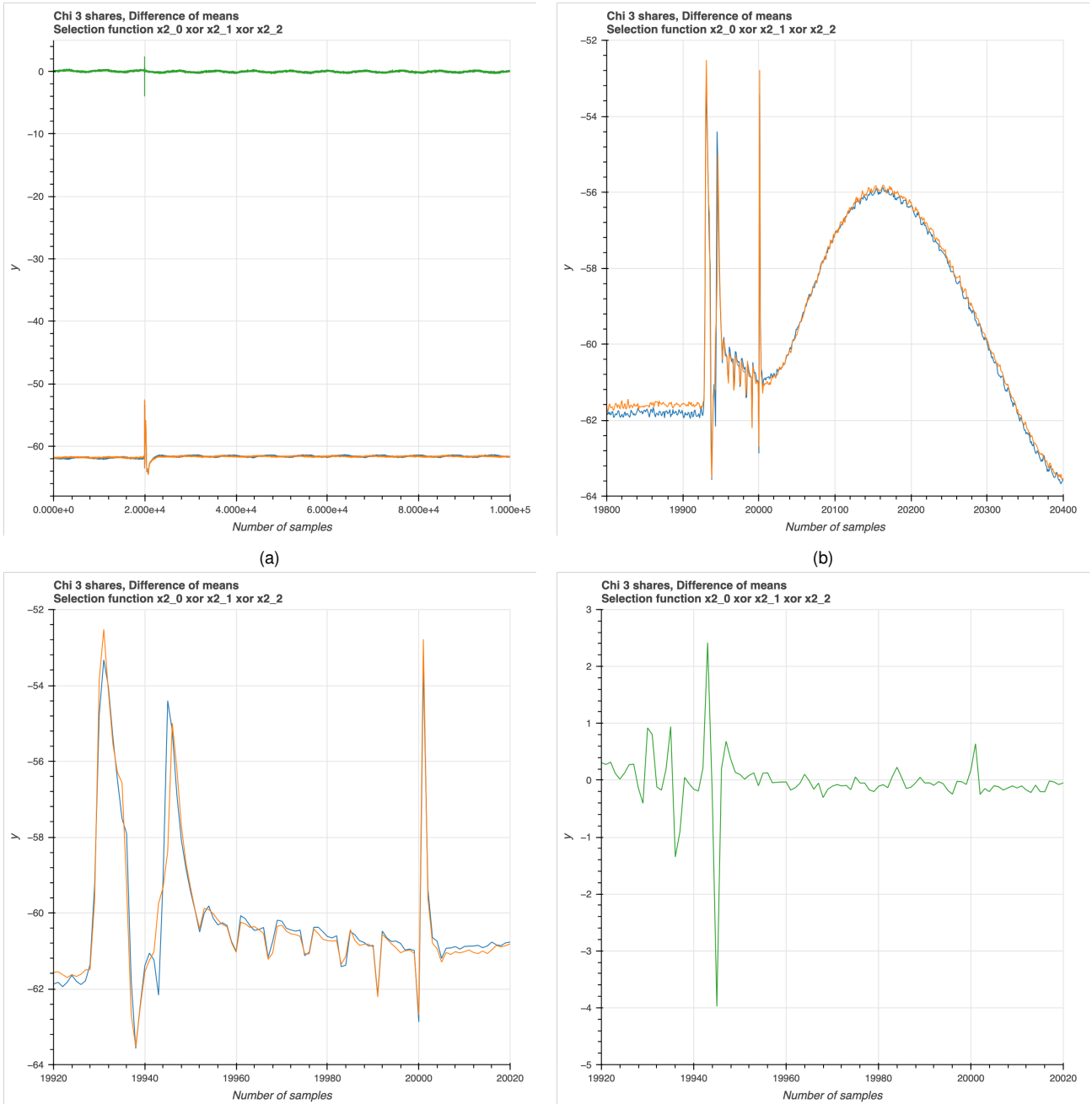


Figure 3.40: Selection function $f_2(x_2^0, x_2^1, x_2^2) = x_2^0 + x_2^1 + x_2^2$. In blue, the mean of the traces in M_0. In orange, the mean of the traces in M_1. In green, the Difference of mean $M_1 - M_0$. Situation with LED disconnected. 15.000 traces acquired.

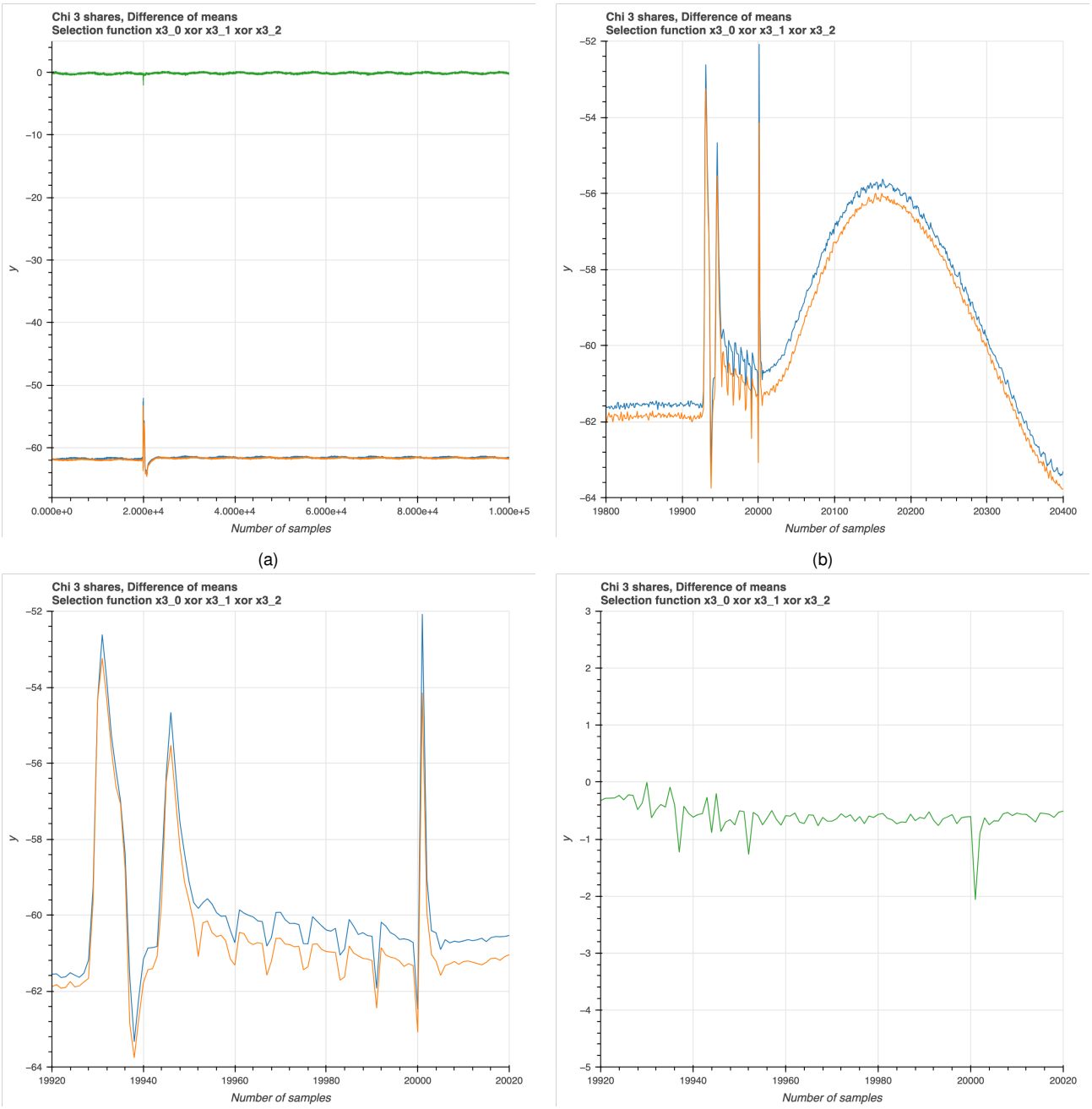
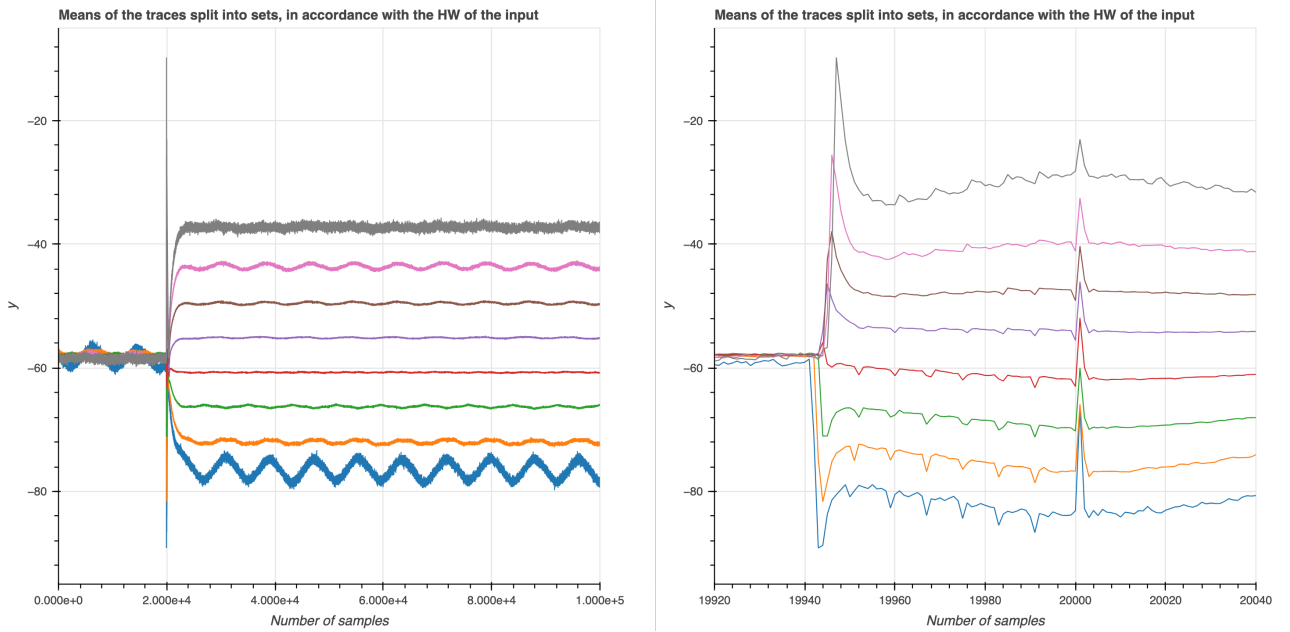


Figure 3.41: Selection function $f_3(x_3^0, x_3^1, x_3^2) = x_3^0 + x_3^1 + x_3^2$. In blue, the mean of the traces in M_0. In orange, the mean of the traces in M_1. In green, the Difference of mean $M_1 - M_0$. Situation with LED disconnected. 15.000 traces acquired.

chiDOM

Last gadget examined is the χ with DOM countermeasure.

Dependencies from the input We began by investigating the correlation between the Hamming weight of the input states and the corresponding power traces, as shown in Figure 3.42. Similarly to the other two gadgets, also for this gadget there is a clear relationship between input Hamming weight and power consumption.



(a) Graph on all the samples, i.e. in the interval [0,100.000]

(b) Zoom in the interval of samples [19.920,20.040]

Figure 3.42: Mean of the traces with random inputs divided into nine sets, depending on the Hamming weight between the current and previous inputs. Blue: Hamming weight equal to 0. Orange: Hamming weight equal to 1. Green: Hamming weight equal to 2. Red: Hamming weight equal to 3. Purple: Hamming weight equal to 4. Brown: Hamming weight equal to 5. Pink: Hamming weight equal to 6. Grey: Hamming weight equal to 7. Situation with LED disconnected. 15.000 traces acquired.

Difference of mean Next, we attempted to extract information about a secret value from the acquired power traces. The first step was to define a suitable selection function. Recalling the expression for our χ gadget with DOM countermeasure:

$$\begin{aligned} y^0 &= x_1^0 \cdot x_2^0 + [x_1^0 \cdot x_2^1 + z] \\ y^1 &= x_1^1 \cdot x_2^1 + [x_1^1 \cdot x_2^0 + z] \end{aligned} \tag{3.33}$$

From this equation, we observe that:

- Two shares of x_1 appears: x_1^0 and x_1^1
- Two shares of x_2 is used: x_2^0 and x_2^1

Given this, we chose two selection functions, aiming to recover some information about the secret values x_1 and x_2:

$$\begin{aligned} h_1(x_1^0, x_1^1) &= x_1^0 + x_1^1 \\ h_2(x_2^0, x_2^1) &= x_2^0 + x_2^1 \end{aligned} \tag{3.34}$$

We then divided the acquired traces into two sets based on the value of this selection function:

- M_0: traces for which is $h_i(x_i^0, x_i^1) = 0$
- M_1: traces for which is $h_i(x_i^0, x_i^1) = 1$

for $i = 1, 2$.
To assess potential leakage, we computed the Difference of Means (DoM) by subtracting the sample-wise mean of the traces in M_1 from those in M_0.

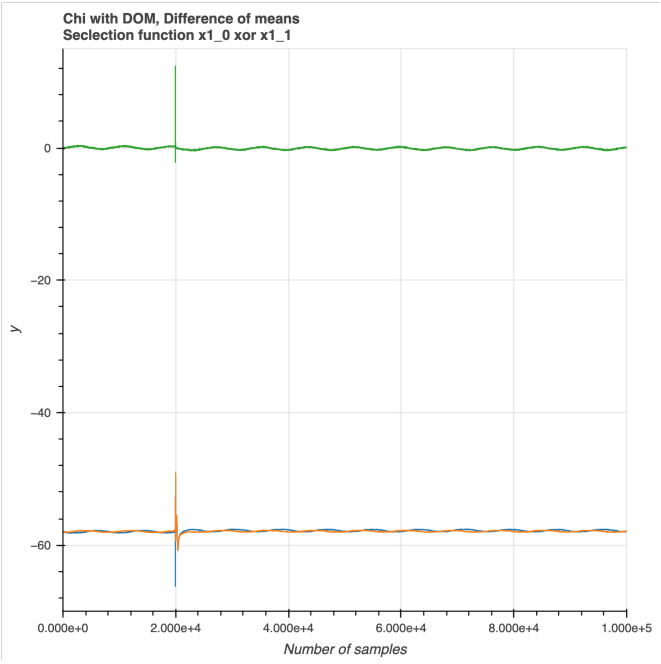
In Figures 3.43 and 3.44, the mean trace for M_0 is shown in blue, the mean for M_1 in orange, and the DoM in green. While the overall DoM remains close to zero for most samples, a closer look reveals that the mean traces for M_0 and M_1 are consistently separated. Additionally, we observe an high peak in these curves, preceded by some smaller, regular patterns as for the previous gadgets, and we reach the same conclusions. Also in this case, as for the χ with 3 shares, it seems that the activities after the start of the operations are longer than for the χ with two shares TI.
However, in this case, the most interesting behavior can be read in figure 3.43. In fact, here the mean of the traces in M_0 has a high negative peak around sample 19.950, which is completely vice versa of the behavior of the mean of the traces in M_1. This results in a very high peak in the difference of means, which can be assumed to be a leakage of value x_1. This is left to future investigations.

3.4.5 Conclusions and Future works

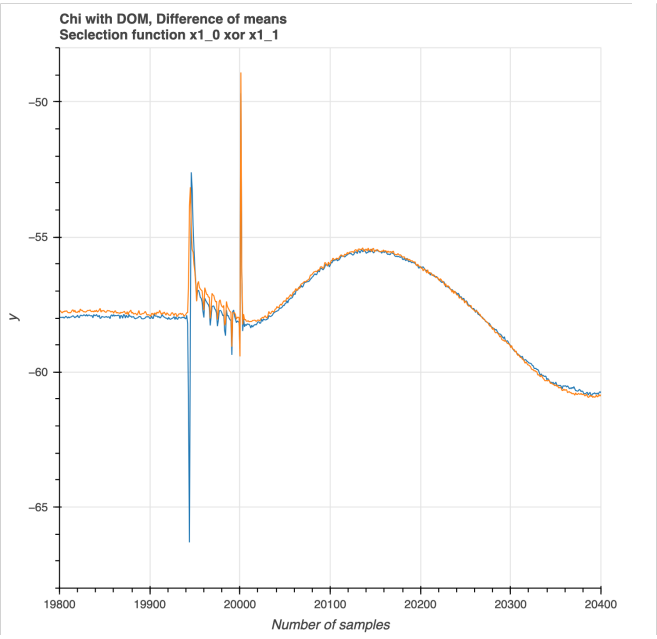
Thanks to the open-source project Tiny Tapeout, we had the opportunity to study the behavior of three cryptographic gadgets implemented in silicon: the χ function with a 2-share threshold implementation countermeasure, the χ function with a 3-share threshold implementation countermeasure, and the χ function protected using a domain-oriented masking (DOM) scheme.
Our investigation highlighted some points:

- During the synthesis phase, certain design optimizations were automatically applied by the toolchain. As a result, the implemented gadgets are functionally equivalent to the original designs, but they differ in structure and internal behavior. Due to these changes, some of the tests we had previously planned could not be applied directly.
- In both operating modes (i.e., with the on-board LED enabled or disabled), we observed leakage of information related to secret values across all three gadget implementations. Based on our observations, we can state the following:
  - Due to the optimizations applied by the toolchain during synthesis, the designs of the χ function with two shares and with three shares were significantly altered. As a result, theoretical inferences based on the original designs no longer hold.
  - In contrast, the χ function with DOM underwent far fewer modifications. The implemented design closely resembles the one presented in [77] and illustrated in [1]. This suggests that the leakage observed in the difference of means analysis in section 3.4.4 may indeed correspond to a sensitive value, making the results potentially meaningful from a side-channel perspective.

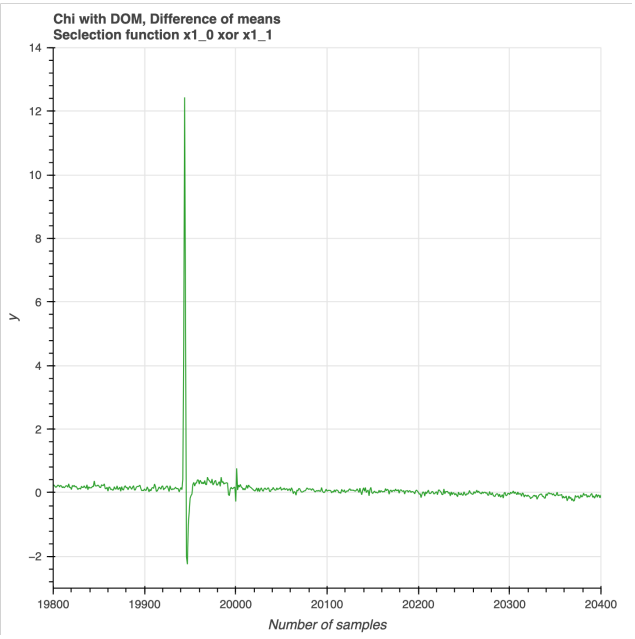
  However, as of the time of writing, we do not yet have a definitive interpretation of these results.
- For this reason, further investigation is required. We plan to conduct additional testing and analysis to better understand the implications of the observed leakage and to refine our methodology for evaluating side-channel resistance in real-world silicon implementations.



(a)



(b)



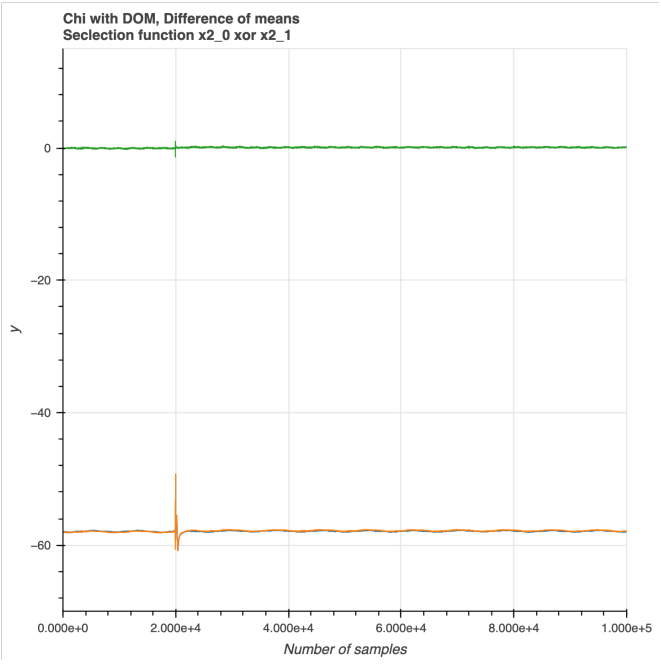
(c)

Figure 3.43: Selection function $h_1(x_1^0, x_1^1) = x_1^0 + x_1^1$. In blue, the mean of the traces in M_0. In orange, the mean of the traces in M_1. In green, the Difference of mean $M_1 - M_0$. Situation with LED disconnected. 15.000 traces acquired.

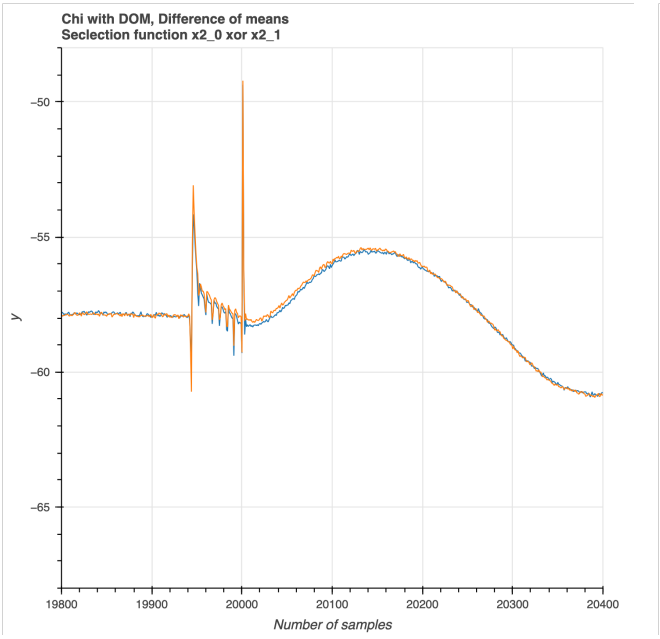
3.5 Leakage assessment of some implementations of Ascon with countermeasures

3.5.1 Introduction

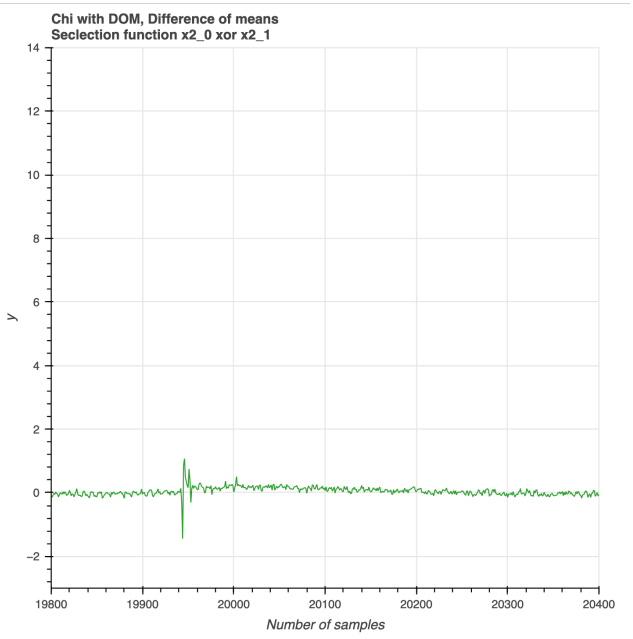
In the context of side-channel attacks (SCAs), one of the most effective defense strategies involves the implementation of *countermeasures* such as Domain-Oriented Masking (DOM) and Threshold Implementations (TI). These techniques aim to protect the intermediate values pro-



(a)



(b)



(c)

Figure 3.44: Selection function $h_2(x_2^0, x_2^1) = x_2^0 + x_2^1$. In blue, the mean of the traces in M_0. In orange, the mean of the traces in M_1. In green, the Difference of mean $M_1 - M_0$. Situation with LED disconnected. 15.000 traces acquired.

cessed by cryptographic algorithms, thereby reducing the correlation between physical leakage (e.g., power consumption) and sensitive data.
In this work, we focused on the analysis of three implementations of the **lightweight cipher Ascon**:

- a baseline version without any countermeasures,
- a version protected using the DOM scheme,
- one secured through threshold implementation.

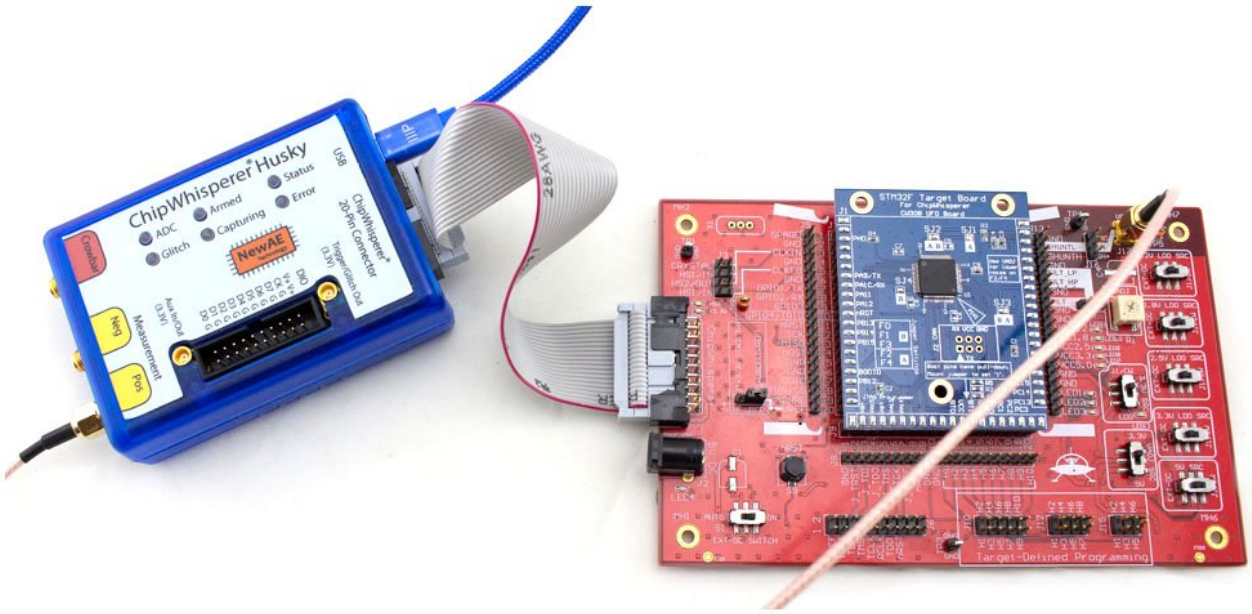


Figure 3.45: ChipWhisperer-Husky connected with a ribbon cable to the ChipWisperer CW313, on which is placed the target board.

To carry out this analysis, we collected power traces using the **ChipWhisperer Husky**, an open-source hardware platform developed by NewAE Technology specifically for side-channel analysis and embedded hardware security research.
Our evaluation leveraged both classical and modern *leakage assessment techniques*. On one hand, we used Test Vector Leakage Assessment (TVLA), a well-established statistical method for detecting leakages. On the other hand, we employed Deep Learning-based Leakage Assessment (DL-LA), a more recent and (maybe) powerful technique that leverages neural networks to uncover subtle patterns in side-channel traces. By applying both methods, we aimed to compare their detection capabilities and gain a better understanding of the strengths and limitations of each approach in evaluating the security of masked implementations of Ascon.

Chipwhisperer-Husky

The ChipWhisperer-Husky is a compact, open-source hardware tool developed by NewAE Technology for side-channel power analysis and fault injection. It is designed to assist researchers and security professionals in evaluating the robustness of embedded systems against attacks such as Differential Power Analysis (DPA) and fault injection [85, 86].
The ChipWhisperer-Husky is a open-source tool. In fact, while not fully OSHW certified, the FPGA logic, microcontroller firmware, and software are **open source**, promoting transparency and customization.
More than that, the ChipWhisperer-Husky has high-speed sampling, it captures fine-grained power traces essential for detailed analysis, thanks to a 12-bit ADC sampling at 200 MS/s. It also offers voltage glitching with two crowbar sizes and clock glitching with sub-nanosecond resolution, enabling precise fault injection. It can stream data at over 20 MS/s, facilitating long-duration captures and real-time analysis. It features a 20-pin header, additional data and clock lines, and supports JTAG/SWD programming, enhancing flexibility in interfacing with various targets.

3.5.2 State of the art

Side-channel attacks and countermeasures

In the field of cybersecurity, cryptographic algorithms are often designed to be mathematically secure. However, real-world systems rarely operate in ideal conditions. When cryptographic computations are implemented in hardware or software, they produce physical effects, such as power consumption, timing variations, or electromagnetic emissions. These physical phenomena can unintentionally leak sensitive information. Exploiting such leaks is the basis of side-channel attacks. Indeed, unlike traditional cryptanalysis, which tries to break the algorithm itself, side-channel attacks focus on observing how the system behaves during execution.
To defend against this class of threats, designers employ what are known as side-channel countermeasures. These are techniques or strategies specifically aimed at minimizing or neutralizing the leakage of information through physical channels. The goal is to make the observable behavior of a system independent (or at least statistically unrelated) to the sensitive data being processed.
Side-channel countermeasures can take many forms. It's important to note that side-channel countermeasures are not universally effective; they must be carefully tailored to the specific threat model, implementation platform, and attack vector. Furthermore, improperly implemented countermeasures can give a false sense of security, as even subtle design flaws or leakage paths can be exploited by attackers.

Domain Oriented Masking scheme (DOM)

DOM is a masking scheme specifically designed to be robust against side-channel leakages while remaining practical for hardware implementations. The fundamental idea of this scheme is to split the sensitive data in shares, and maintain a domain separation during non linear operations [76]: the computations are organized in such a way that the interaction between shares is strictly controlled. This is crucial because side-channel leakage often arises not just from the values themselves, but from the unintended combinations of signals. By carefully separating the domains of operands and applying fresh randomness where needed, DOM minimizes the opportunities for such leakages.
One of the key advantages of DOM is that it is hardware-friendly. In fact, DOM tends to require fewer resources in terms of area and randomness. This efficiency makes it especially attractive for lightweight or embedded cryptographic devices where resource constraints are critical.

Threshold implementation (TI)

The core idea behind side-channel countermeasures like TI is to mask sensitive values, splitting them into multiple randomized shares such that no single piece reveals useful information on its own. However, what makes TI distinct from simpler masking schemes is its rigorous approach to maintaining security even in the presence of hardware-specific challenges, such as glitches [128]. The Threshold Implementation approach ensures security through three main properties:

- Correctness: The combination (e.g., XOR) of all shares must reconstruct the correct value of the sensitive variable.
- Non-completeness: No intermediate function during computation should depend on all shares of a given variable. This prevents information from being fully exposed through any internal node.

- Uniformity: The output shares must remain uniformly distributed when the inputs are masked and uniformly distributed. This is crucial to prevent statistical biases that an attacker could exploit.

Side-channel leakage assessment

In Side-Channel Analysis, leakage assessment refers to the process of determining whether the measurements collected from a device under test (DUT) contain any input-dependent information. It serves as a preliminary phase that can provide information on the feasibility of the attack. In fact, in case no information correlated with secret data is found, one can conclude that the DUT is sufficiently secure, without even performing an actual key-recovery attack. However, detection of leakage does not imply that an attack can be carried out immediately, but is a warning sign that exploitable information may exist.

One of the most used leakage assessment methods is TVLA [72], which stands for Test Vector Leakage Assessment. TVLA is a statistical approach that, rather than attempting to extract a key or perform an actual attack, simply answers the question: *Does this implementation leak information in a measurable way?*
The strength of TVLA lies in its formal, quantifiable, and implementation-independent methodology. It uses hypothesis testing, typically a t-test, to assess whether two sets of measurements come from the same distribution. The most common scenario compares traces collected under two conditions: one set where the input (or key) is held constant, another where the input (or key) varies randomly.
If the implementation is secure and free of leakage, the physical traces under both conditions should look statistically indistinguishable. If there is leakage, the test will detect significant differences between the two sets of measurements.
A commonly accepted threshold in TVLA is a $|t|$ score of 4.5, which corresponds to a confidence level above 99.999%. If the absolute t-value exceeds this threshold at any point in time, it is considered evidence of leakage. However, it's important to understand that TVLA does not quantify how much leakage there is, nor whether the implementation can be practically attacked. It simply flags that leakage is present, prompting further investigation or countermeasures.
However, while TVLA is widely used, it is not without limitations. It assumes proper experimental setup, including sufficient sample size, proper alignment of traces, and stable environmental conditions. A poorly configured test may either miss real leakage or produce false positives. Additionally, TVLA is typically performed at the implementation or post-silicon validation stage, meaning it is a detection tool rather than a preventative one.

Deep Learning Leakage Assessment (DL-LA)

In recent years, growing interest in deep learning techniques applied to side-channel analysis has yielded promising approaches, due to their ability to attack countermeasure-protected targets and to learn directly from raw data, bypassing lengthy preprocessing steps even in case of misaligned traces or multivariate leakage. Here is where Deep Learning Leakage Assessment comes into play. Introduced by Moos et al. in 2021 [121], DL-LA aims to improve the leakage detection capabilities of traditional methods such as TVLA [72], especially in the most complex scenarios (noisy or misaligned traces, multivariate or horizontal leakage).
In DL-LA, a neural network is trained to classify two groups of side-channel traces. Similarly to TVLA, DL-LA claims that a leakage is present if we are able to distinguish between those two

groups of traces, that is, if the neural network is able to learn from the training data. Concerning the acquisition of the traces, Moos et al. suggest the use of a fixed-versus-fixed input test (specific test), as it should guarantee a larger difference between the two groups of traces, thus helping the neural network in the classification task.
The DL-LA methodology can be summarized in the following steps. First, the set of traces is standardized by computing

$$X_i^j := \frac{X_i^j - \mu_i}{\sigma_i} \tag{3.35}$$

Then, the traces are split into a training set N and a validation set M, the latter of which will be used only to evaluate the trained model on an unseen set of traces. Hence, the validation traces are still extracted from our initial trace set, but they do not contribute to the learning process.
In this scenario, the null hypothesis H_0 refers to the case in which the neural network did not learn anything, that is, it functions as a random classifier. Consequently, the number of correct classifications can be modeled as a random variable following a binomial distribution:

$$H_0 : X \sim Binom(M, 0.5) \tag{3.36}$$

Adhering to the terminology presented by Moos et al. [121], we will instead call $s_M = v \cdot M$ the number of correct classifications resulting from our trained model, where v is its validation accuracy. In order to reject the null hypothesis and hence claim that a leakage is present, DL-LA aims to prove the following:

$$P(X \geq s_M) \leq p_{th} \tag{3.37}$$

Where p_{th} is a user-defined threshold (e.g., $p_{th} = 10^{-5}$).
This probability (that we will refer to as *confidence value*, or *p value*) can then be computed as the probability density function of the binomial distribution:

$$P(X \geq s_M) = \sum_{k=s_M}^{M} \binom{M}{k} 0.5^k 0.5^{M-k} = 0.5^M \sum_{k=s_M}^{M} \binom{M}{k} \tag{3.38}$$

Furthermore, DL-LA allows to determine which are the samples that contributed to the leakage by means of a sensitivity analysis. This could come in handy, when preparing a key-recovery attack, to understand which phases of the algorithm expose a leakage in the examined implementation.
One of the most important features of DL-LA is that, unlike TVLA, it considers all samples of a trace when assigning it to a group. Therefore, it is capable of detecting multivariate or horizontal leakage without requiring any kind of preprocessing or samples combination, as well as dealing with misaligned and noisy traces. This also results in a lower number of false positives and is particularly important when dealing with algorithms such as Ascon.

Ascon

ASCON is a family of lightweight cryptographic algorithms designed for authenticated encryption and hashing, optimized for efficiency on resource-constrained devices like IoT nodes and embedded systems. It was selected as the primary recommendation in the NIST Lightweight Cryptography standardization process in 2023 [59].
ASCON is known for its:

- Simplicity: Uses a sponge construction with a permutation-based core.
- Efficiency: Performs well on both hardware and software platforms.

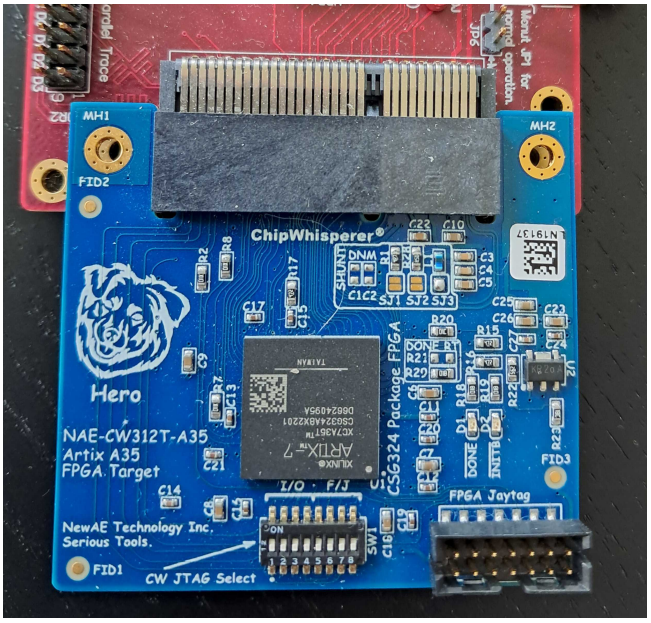


Figure 3.46: The target Artix A35.

- Security: Offers strong protection against known cryptographic and side-channel attacks.
- Flexibility: Includes multiple variants (e.g., ASCON-128, ASCON-128a) for different use cases.

Its balance of performance and security makes ASCON a modern choice for authenticated encryption in constrained environments.
In our experiments, we synthesized three versions of Ascon:

- The first one was without countermeasures, which was from the github repository [129].
- The second one with DOM countermeasure, which was from the github repository, version V1 [130].
- The last one with TI countermeasure, which was from the github repository [62].

3.5.3 Experiments

This section presents preliminary results obtained by applying TVLA and DL-LA to both protected and unprotected hardware implementations of the Ascon cipher, providing an initial assessment of its effectiveness under varying acquisition strategies and levels of countermeasures.
Before getting to the experimental verification, we will briefly detail the applied methodology and the choices we made in each step of the procedure.

Measurement setup

We used a CW Husky board (see sec. 3.5.1), with target a NAE CW312T A35 (figure 3.46). The target board support is a ChipWhisperer CW313.
We acquired traces at a frequency of 7.37 MHz. For Ascon without countermeasures, we acquired 4 sample per cycle, for a total of 500 samples; for Ascon with DOM countermeasure, we acquired 8 sample per cycle, for a total of 1300 samples; for Ascon with DOM countermeasure, we acquired 4 sample per cycle, for a total of 1300 samples.

Acquisition of the trace sets

For the traces acquisition phase, we opted for fixed-versus-random input test (non-specific test), also varying the input data in multiple ways to assess whether this could result in higher the differences among the two groups of traces. Throughout the process, we followed to the guidelines of the leakage assessment methodology, as reported in [149].

Neural Network and confidence value

Concerning the choice of the neural network for the DL-LA, we use a Multi-Layer Perceptron similar to the one proposed by Moos et al. It consists of four dense layers of 120, 90, 50 and 1 output neurons. All layers use a *Rectified Linear Unit (ReLU)* as an activation function, except for the final one, which uses the Sigmoid activation function. The four layers are interleaved with a *BatchNormalization* layer. Moreover, the model uses Binary Cross Entropy (BCE) as loss function - since we deal with a binary classification task - and Adam as optimizer.
Once we compute the Welch's t-test values, in order to compare them to DL-LA's confidence value, we also estimate the confidence *p* to accept the null hypothesis via the Student's t probability density function [149][121]:

$$p = 2 \int_{|t|}^{\infty} f(t, v)\, dt \tag{3.39}$$

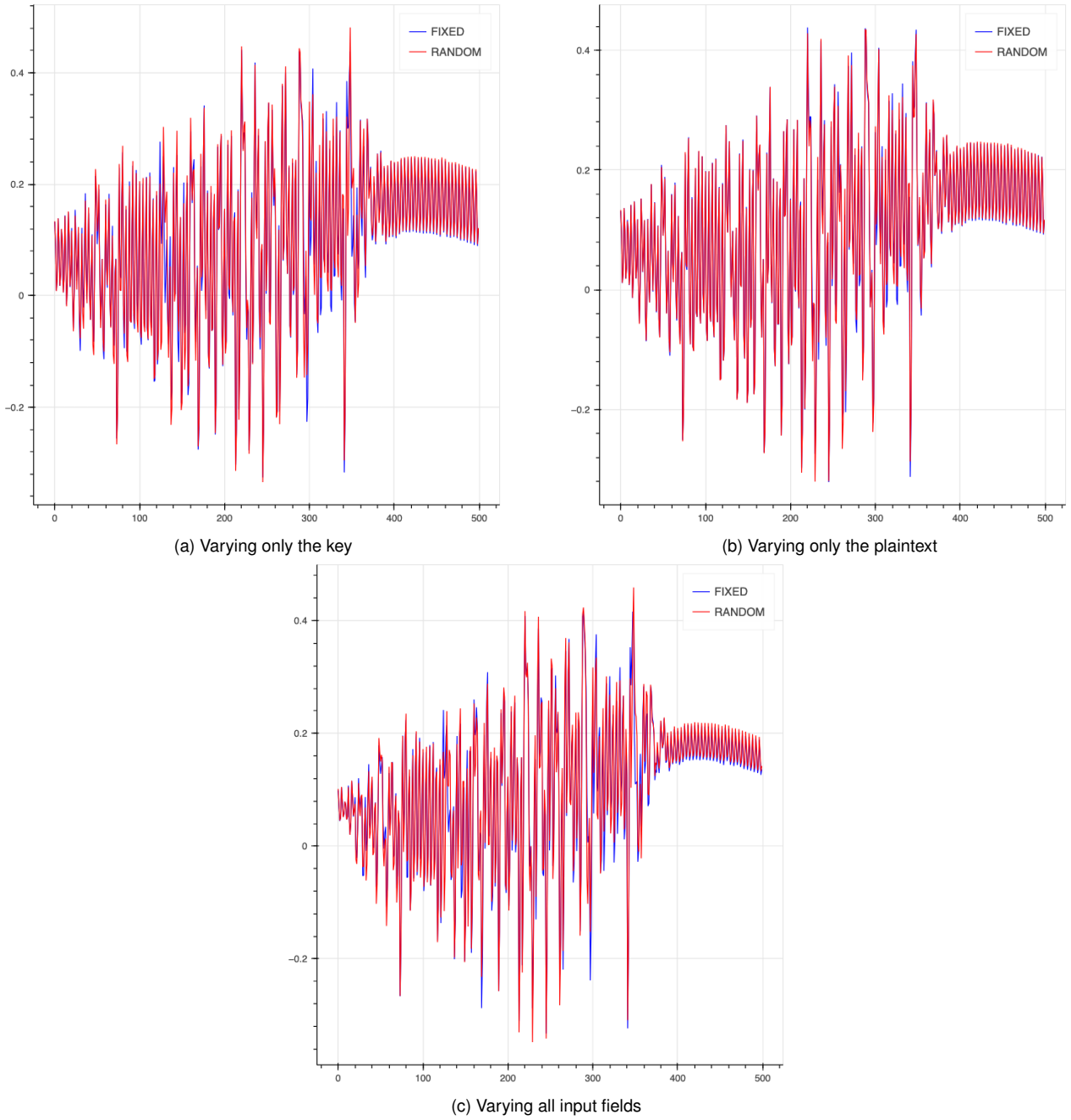
$$f(t, v) = \frac{\Gamma\left(\frac{v+1}{2}\right)}{\sqrt{\pi v}\Gamma\left(\frac{v}{2}\right)} \left(1 + \frac{t^2}{v}\right)^{-\frac{v+1}{2}} \tag{3.40}$$

Where *t* is the t-test value of a given point in time, *v* are the degrees of freedom and $\Gamma(\cdot)$ is the Gamma function.
It is important to point out that, unlike DL-LA, the two-tailed p-values represent each sample individually, that is, we have as many p-values as number of samples in the traces. Therefore, we decided to take the minimum among all p-values as representative of all the points in time. In other words, in the case of TVLA, we consider only the sample that corresponds to the highest leakage.
Finally, to determine the number of traces required by each of the two techniques to expose a leakage, we performed multiple executions of both methods, each time varying the number of considered traces. Note that, in the case of DL-LA, what varied is the number of *training* traces, while the number of *validation* has been fixed accordingly.

Application of TVLA

Ascon without countermeasure

Concerning the unprotected hardware implementation of Ascon, three main acquisitions have been performed: the first varying only the key of the random input data and setting the rest to 0. The second with the same approach but varying the plaintext instead of the key. The last one varying all fields of the random input, namely key, plaintext, nonce and associated data. In all cases, we acquired 2000 traces (half with fixed input and half with random input) of 500 samples each, covering all the authenticated encryption phases of the Ascon cipher. In the following, we will address the outcome of the experiments based on the traces.
We begin our analysis by inspecting the mean and variance of each sample across all traces, as shown in figures 3.47 and 3.48. In particular, the differences present in the variance values anticipate a possible leakage exposed by the implementation.

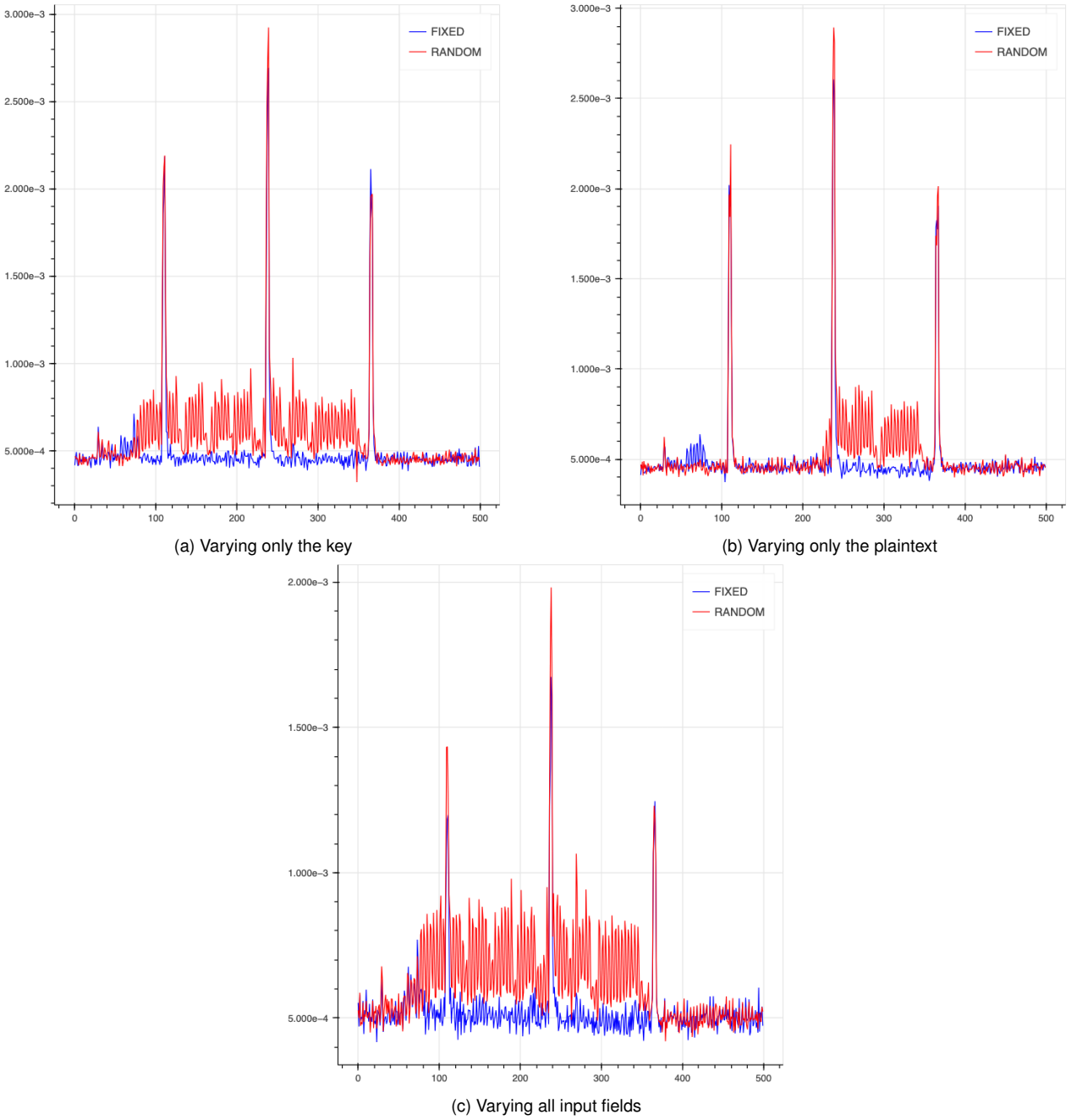


(a) Varying only the key

(b) Varying only the plaintext

(c) Varying all input fields

Figure 3.47: Mean of the traces acquired by varying the key (a), the plaintext (b) and all input fields (c)



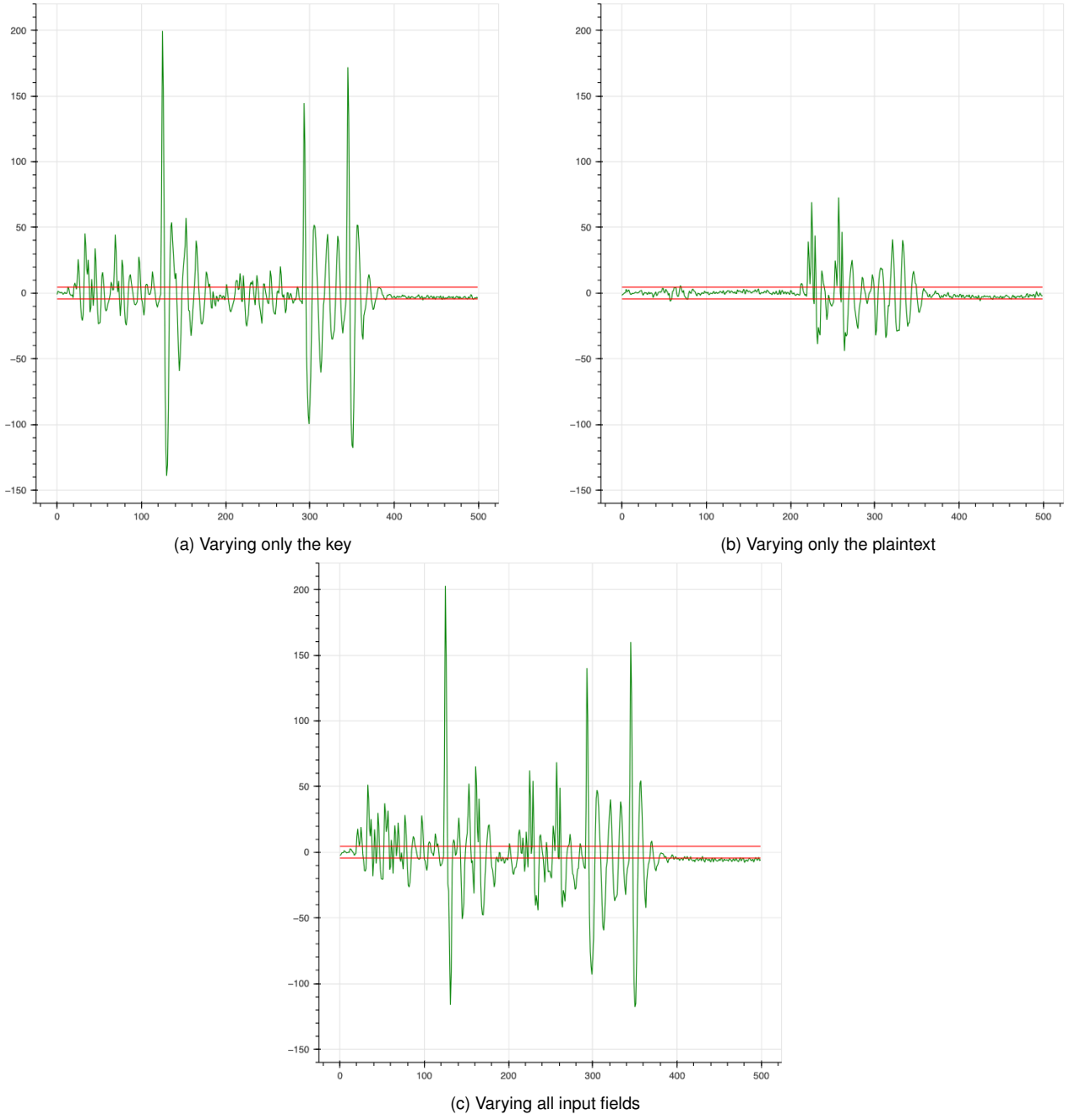
(a) Varying only the key

(b) Varying only the plaintext

(c) Varying all input fields

Figure 3.48: Variance of the traces acquired by varying the key (a), the plaintext (b) and all input fields (c)

As expected, the Welch's t-test performed on the given sets of traces, reveal that the predefined threshold is exceeded in all three cases, and by multiple samples (Fig. 3.49).



(a) Varying only the key

(b) Varying only the plaintext

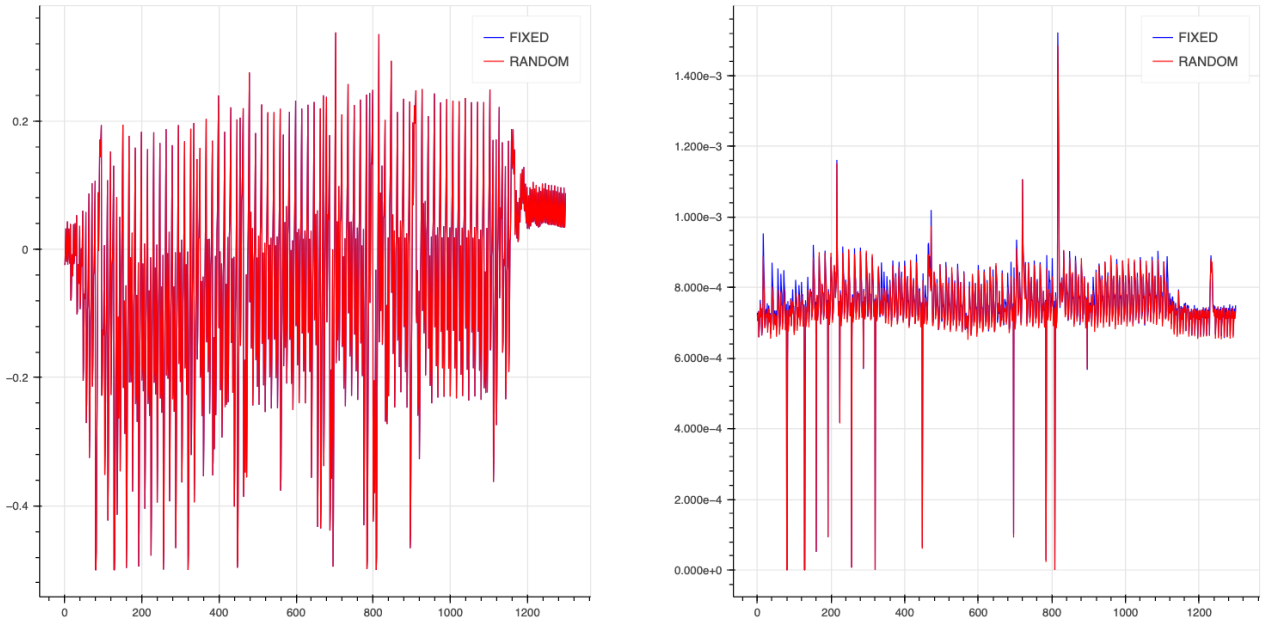
(c) Varying all input fields

Figure 3.49: T-test of the traces acquired by varying the key (a), the plaintext (b) and all input fields (c)

Ascon with DOM

We integrated a hardware implementation of Ascon protected with Domain Oriented Masking (DOM). Due to a memory restriction, the implementation takes in input randoms different from zero only for the first permutation, the one operating in the inizializaiton phase. Here we focus our attention on the case in which the shares of all Ascon's input fields (PT, AD, key and nonce)

are randomly varied at each execution for both the random and fixed sets, such that for the fixed set the xor of the shares gives as result an array of 16 zeros bytes. We acquired in this way a total of 100k traces (half with fixed input and half with random input), containing 1300 samples each.
In Fig. 3.50 are reported the usual mean and variance across all traces.



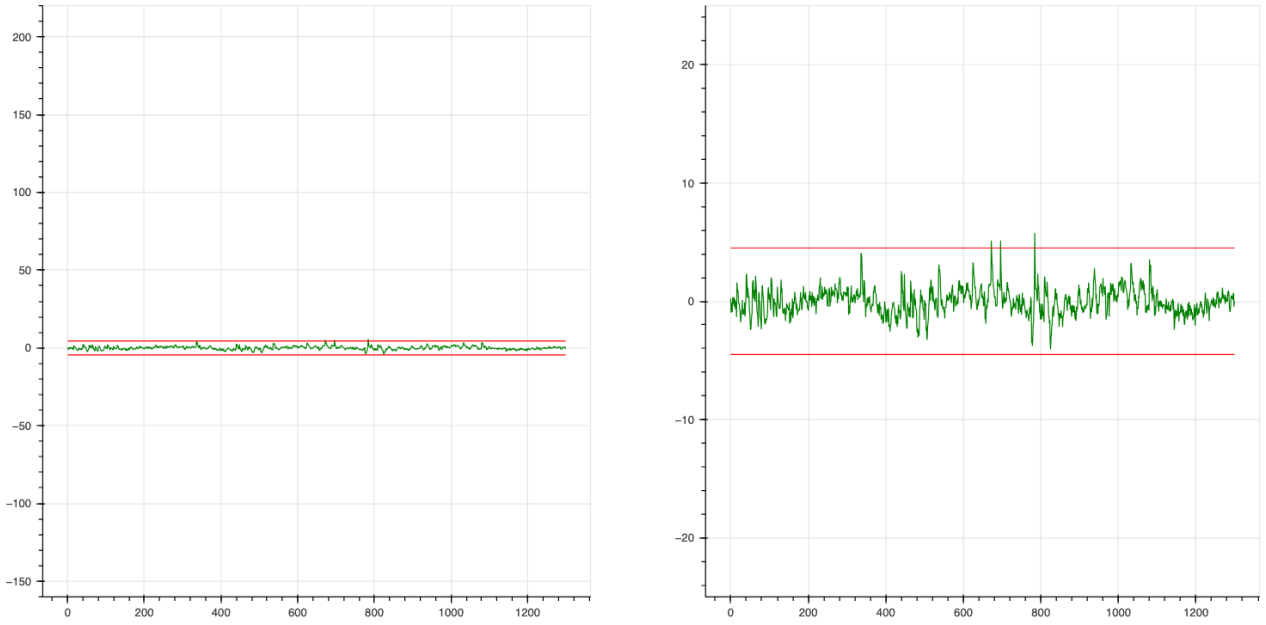
(a) Mean (b) Variance

Figure 3.50: Mean (a) and variance (b) of the traces acquired by varying all shares of the input fields, Ascon with DOM

We also computed the t-test on this set of traces, and the result is in figure 3.51. We noted some peaks that go out from the threshold. However, as explained before, the random are effectively random only for the first permutation of Ascon, during the inizialization phase. The peaks that we see in figure b are in the following phases, leaving the doubt that they appear because of it.

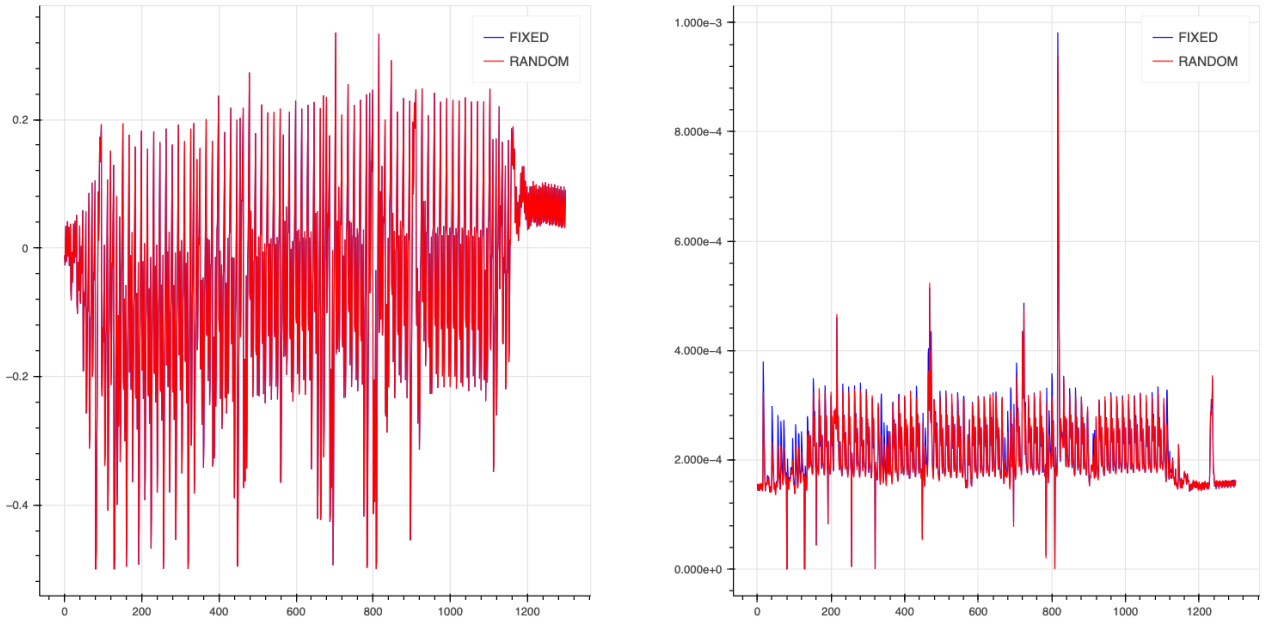
When the random are badly implemented We performed the same analysis in the case in which all the random used to implement the threshold implementation protection are zeros. We acquired in this way a total of 100k traces (half with fixed input and half with random input).
In Fig. 3.52 are reported the usual mean and variance across all traces.
We also performed a t-test on this set of traces, and the results are presented in Figure 3.53. In this case, we observe a greater number of peaks compared to the previously analyzed scenario, and these peaks appear throughout all phases of the Ascon algorithm.
Interestingly, some peaks now exceed the threshold even before the start of the actual Ascon operations (something that did not occur when the random values were correctly implemented). However, during these early cycles, the input is being transmitted, and thus we would not expect the behavior to depend on the random values, which affect the non-linear parts of the permutations. To investigate this unexpected behavior and rule out any possibility of key leakage during these initial cycles, the Pearson's correlation between the values of the traces at each sample and a leakage prevision based on the Hamming Weight or the Hamming Distance, applied to some intermediate values computed by a software simulation of the algorithm. In particular, our previsions were (see figure 3.54):

- The Hamming Weight of the 32-bit words of the key (four words in total);



(a) (b)

Figure 3.51: T-test of the traces acquired by varying all shares of the input fields (a), and zoom on the y axes (b), Ascon with DOM



(a) Mean (b) Variance

Figure 3.52: Mean (a) and variance (b) of the traces acquired by varying all shares of the input fields, Ascon with DOM and all the randoms zero

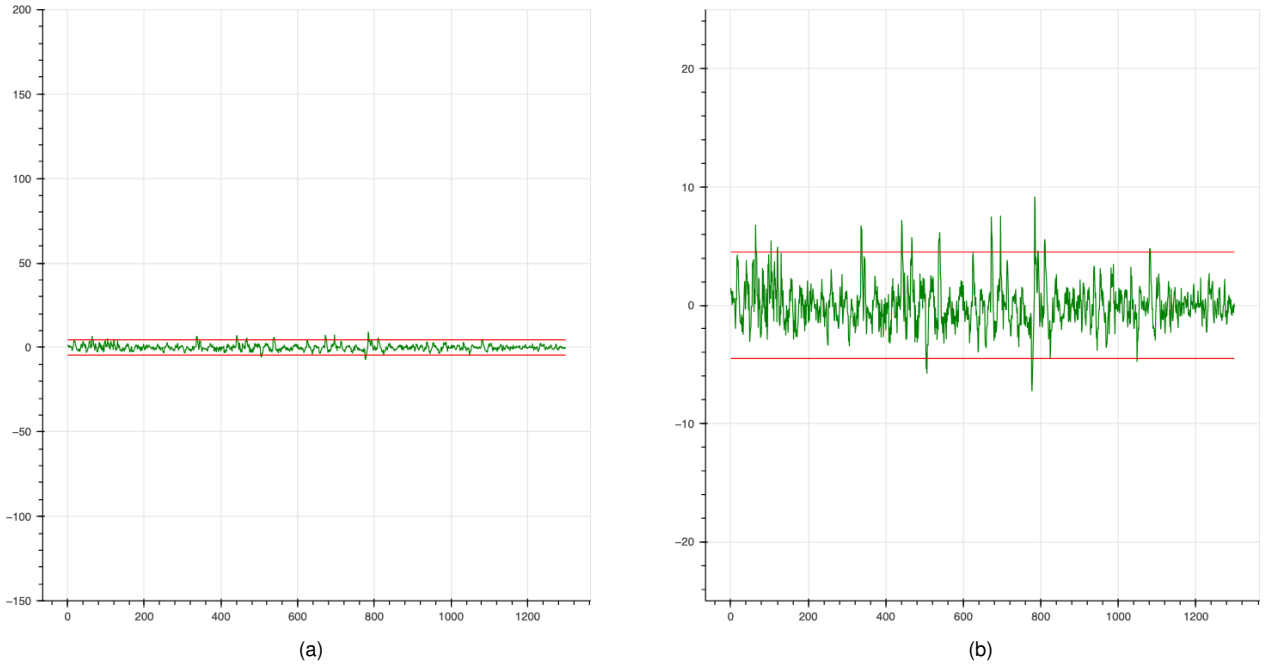


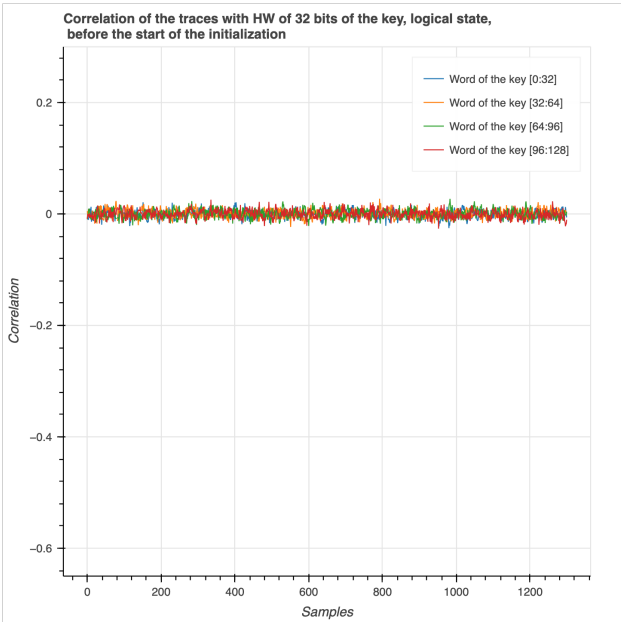
Figure 3.53: T-test of the traces acquired by varying all shares of the input fields (a), and zoom on the y axes (b), Ascon with DOM and all the randoms zero

- The Hamming Distance between these 32-bit key words;
- The Hamming Weight of the 32-bit words of the first share of the key K_1 concatenated with the corresponding 32-bit words of the second share of the key K_2.

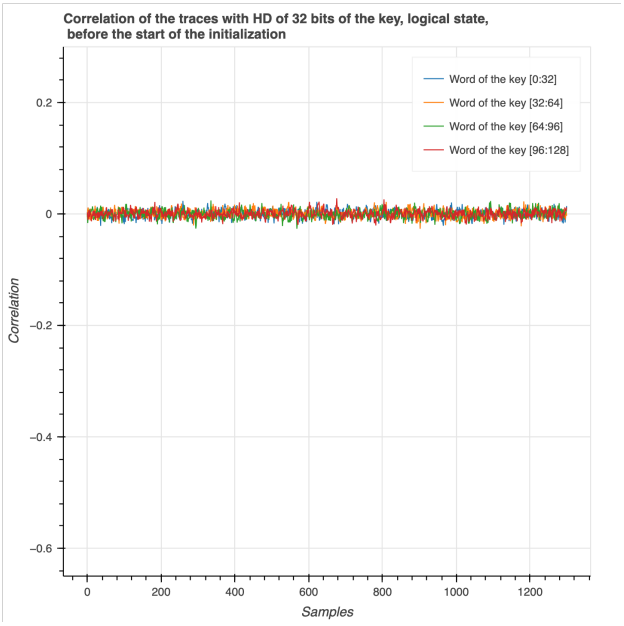
In the first two cases (figures a and b), the results were negligible, showing no significant peaks in the correlation. However, in the third case (figure c), we observe that the Hamming Weight of the 32 bits from the first share of the key K_1, concatenated with the corresponding 32 bits from the second share K_2, shows some correlation with the traces, highlighting the moments when this information are transmitted.
When we acquired the traces, we also saved the inputs that produced them, and the outputs. Here, we have a Python code that is able to compute all the intermediate values starting from the saved inputs, and we performed some correlations between the traces and this kind of data. Let S_1 be the state at some point in the computations of the first share, and S_2 for the second share. In particular, we computed the correlation between the traces and:

- the Hamming Distance between the concatenation of the states S_1 and S_2 before (input) and after (output) of round r;
- the Hamming Weight of the concatenation of the states S_1 and S_2 in input to round r;
- the Hamming Weight of the concatenation of the states S_1 and S_2 in output to round r.

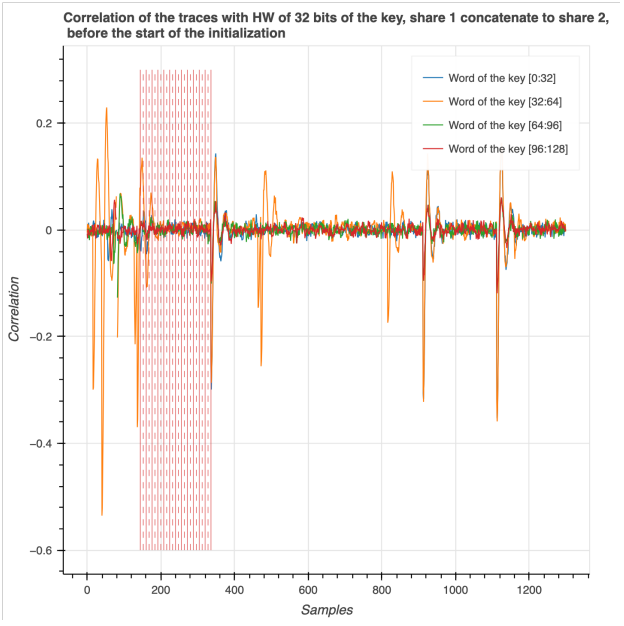
In particular, we analyzed round 0, round 1, and the final round of the permutation during the initialization phase, with the corresponding results shown in Figure 3.55. The figure reveals a strong correlation in the case of the Hamming distance, clearly indicating both when the internal states reach the computed values and the exact clock cycles in which the permutation rounds are executed. We don't see the same kind of peaks in the two cases with Hamming Weight.



(a) Correlation with the Hamming Weight of the logical state of 32-bits words of the key.

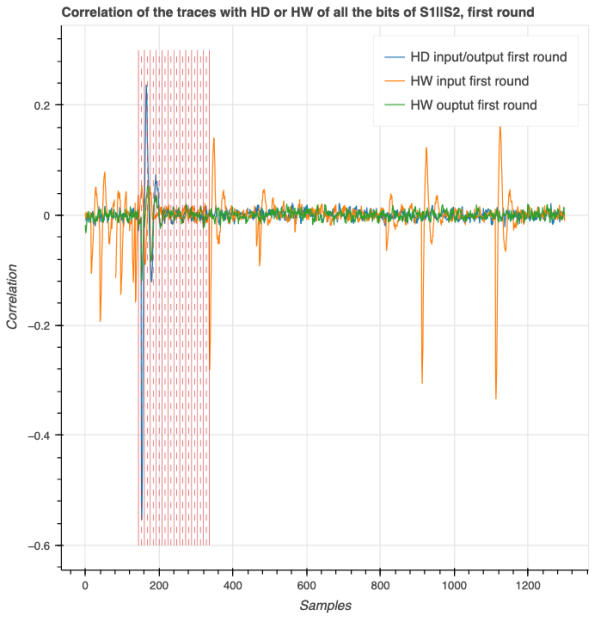


(b) Correlation with the Hamming Distance of the logical state of 32-bits words of the key.

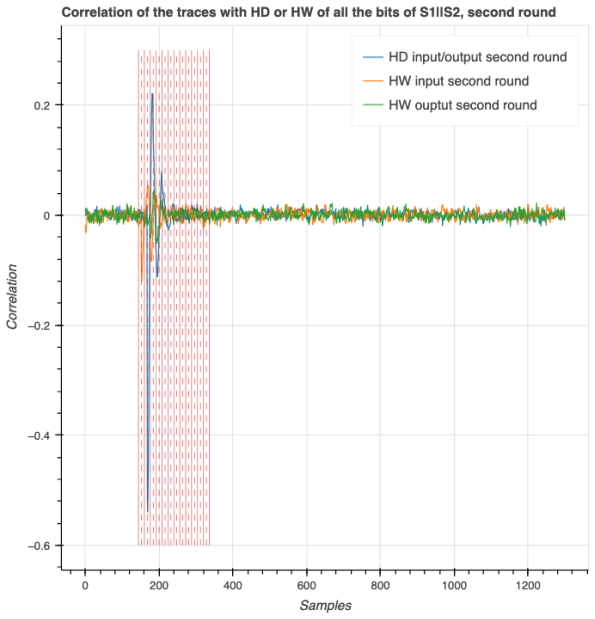


(c) Correlation with the Hamming Weight of 32-bits words of the first shares of the key, concatenated with the corresponding 32-bits word of the second share of the key.

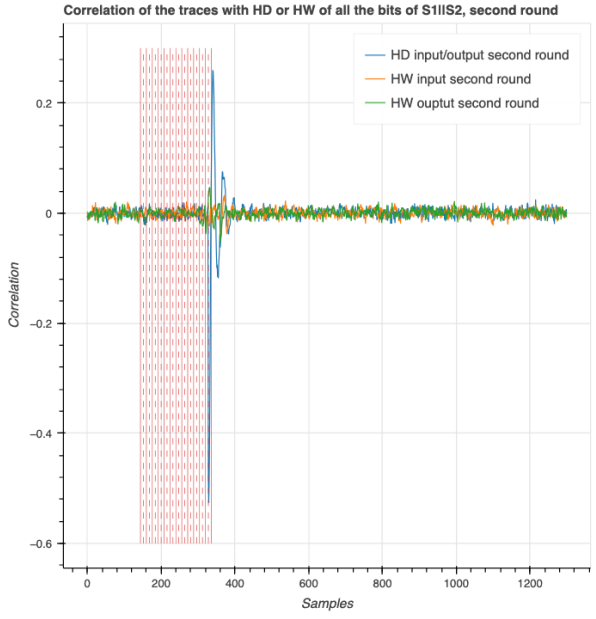
Figure 3.54: Correlations between the traces and some leakages previsions (Hamming Weight and Hamming Distance of the initial state of the inizialization state).



(a) Round 0 of the initialization phase.



(b) Round 1 of the initialization phase.

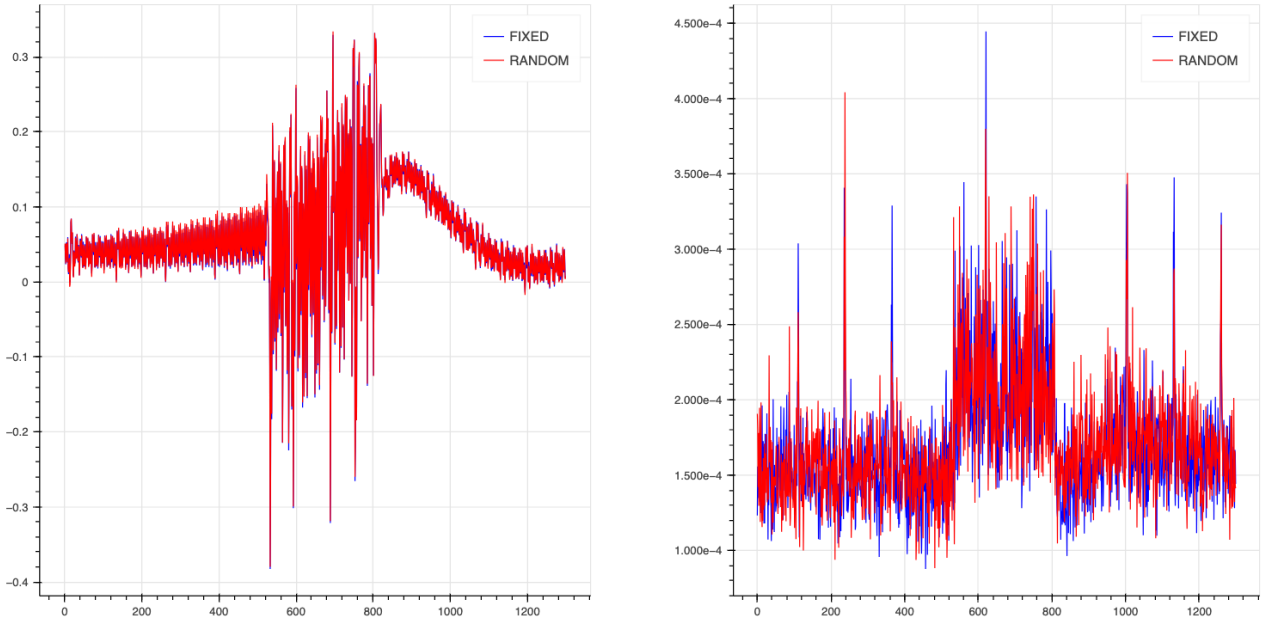


(c) Last round of the initialization phase.

Figure 3.55: In blue the correlation between the traces and the Hamming Distance input/output of the round; in orange the correlation between the traces and the Hamming Weight of the input of the round; in green the correlation between the traces and the Hamming Weight of the output of the round. The red vertical solid lines represent the rounds of the permutation during the inizialization phase. The red dashed lines represent the cycles (each round is performed in two cycles).

Ascon with TI

We also implemented and synthetized the Ascon algorithm with the Threshold Implementation countermeasure. As for the case of Ascon with DOM, also in this case the shares of all Ascon's input fields (PT, AD, key and nonce) are randomly varied at each execution for both the random and fixed sets, such that for the fixed set the xor of the shares gives as result an array of 16 zeros bytes. For this analysis we used only 200 traces (half fixed input and half with random input), because we noted some relevant deviations in the acquired voltage among traces with far time of acquisition.
In figure 3.56 we reported the mean and the variance across all traces.



(a) Mean

(b) Variance

Figure 3.56: Mean (a) and variance (b) of the traces acquired by varying all shares of the input fields, Ascon with TI

The Welch's t-test (figure 3.57) performed on the sets of traces reveals some possible leakages for this specific implementation. However, we note that the peaks that go out of the threshold are all concentrated before the beginning of the permutations of Ascon.

When the random are badly implemented We performed the same analysis in the case in which all the random used to implement the threshold implementation protection are zeros. Also in this case we used 200 traces (half fixed input and half with random input).
In figure 3.58 we reported the mean and the variance across all traces.
In an initial analysis of the means and variances, two key differences can be observed between the cases of Ascon with threshold implementation, the one using correct random values and the other with all randoms set to zero:

- In both mean plots (Figures 3.56 a and 3.58 a), a clear distinction can be seen between two phases: an initial input preparation phase and a subsequent phase where the Ascon operations take place. In the preparation phase, Figure 3.56 a shows a noticeable increasing trend in the traces leading up to the start of the operations. This trend is absent in Figure 3.58 a. A possible explanation is that, in the first case, the absorbed random values differ from each other, resulting in increased power consumption. In contrast, when all randoms are zero, this variation is not present.

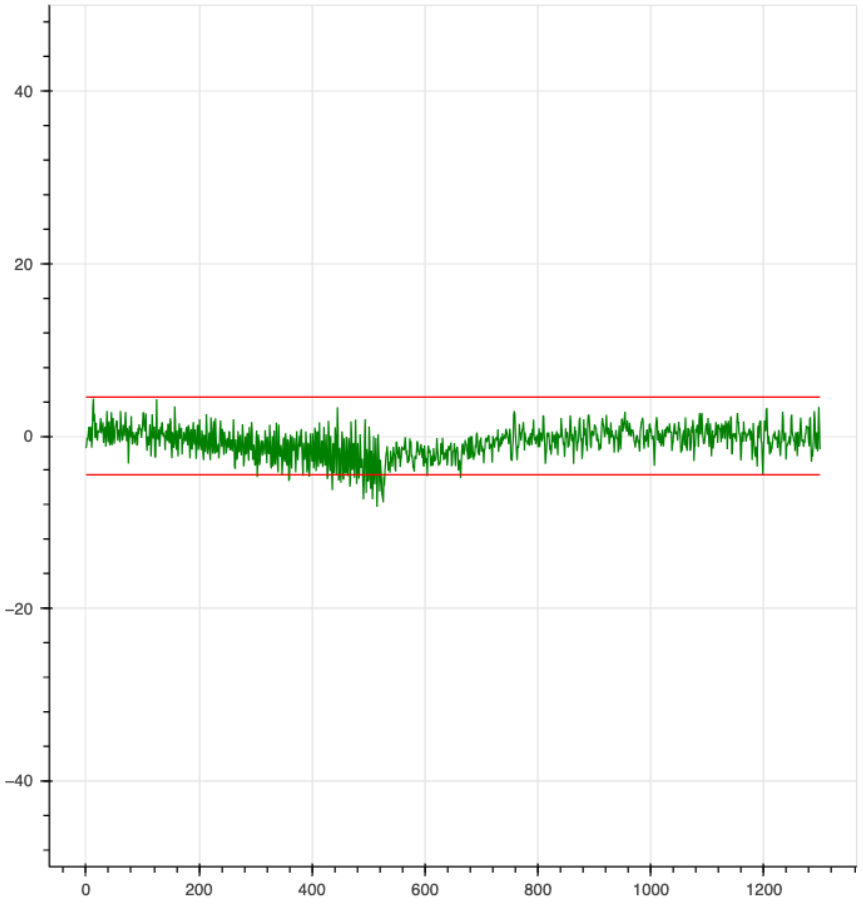
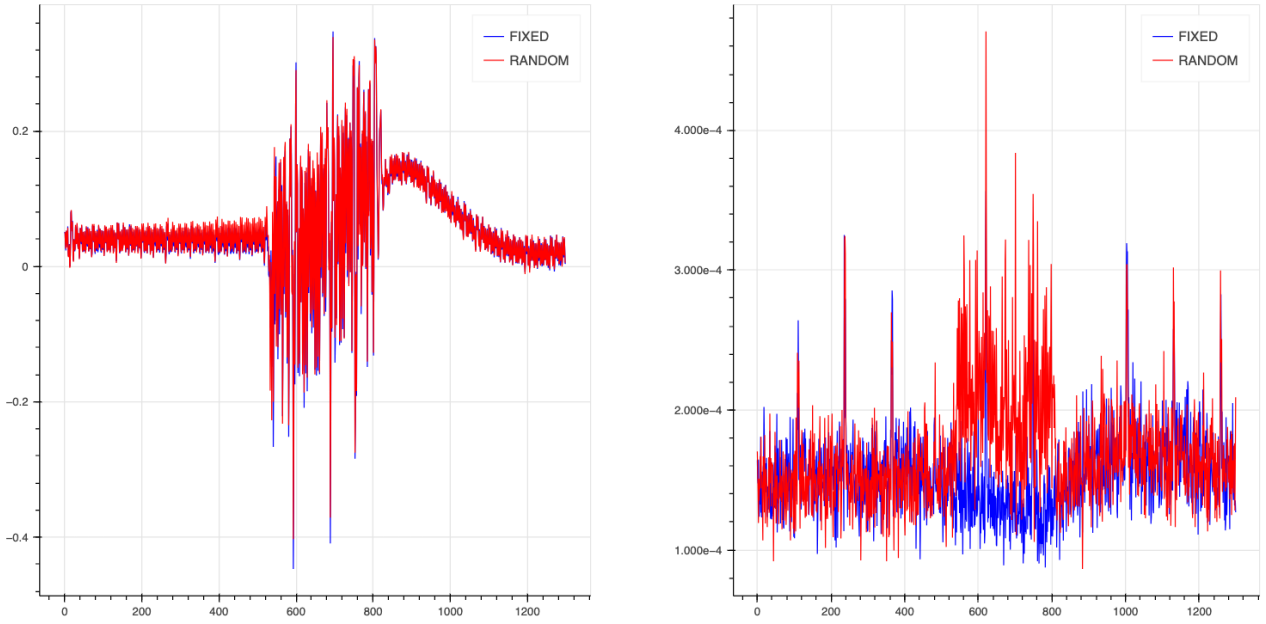


Figure 3.57: T-test of the traces acquired by varying all shares of the input fields, Ascon with TI



(a) Mean

(b) Variance

Figure 3.58: Mean (a) and variance (b) of the traces acquired by varying all shares of the input fields, Ascon with TI with randoms for the countermeasure set to zero

- Also noteworthy is the variance shown in Figure 3.58 b, where the behavior of the trace variance in the set with fixed inputs significantly differs from that in the set with random inputs, especially in the phase of the crypthographic operations.

The Welch's t-test (figure 3.59) performed on the sets of traces reveals some possible leakages for this specific implementation. Unlike in the previous case, we note that the peaks that go out of the threshold are not only before the beginning of the cryptographic operations but also after it. Moreover, it is really visible that the t-test peaks are much higher than those in the previous case.

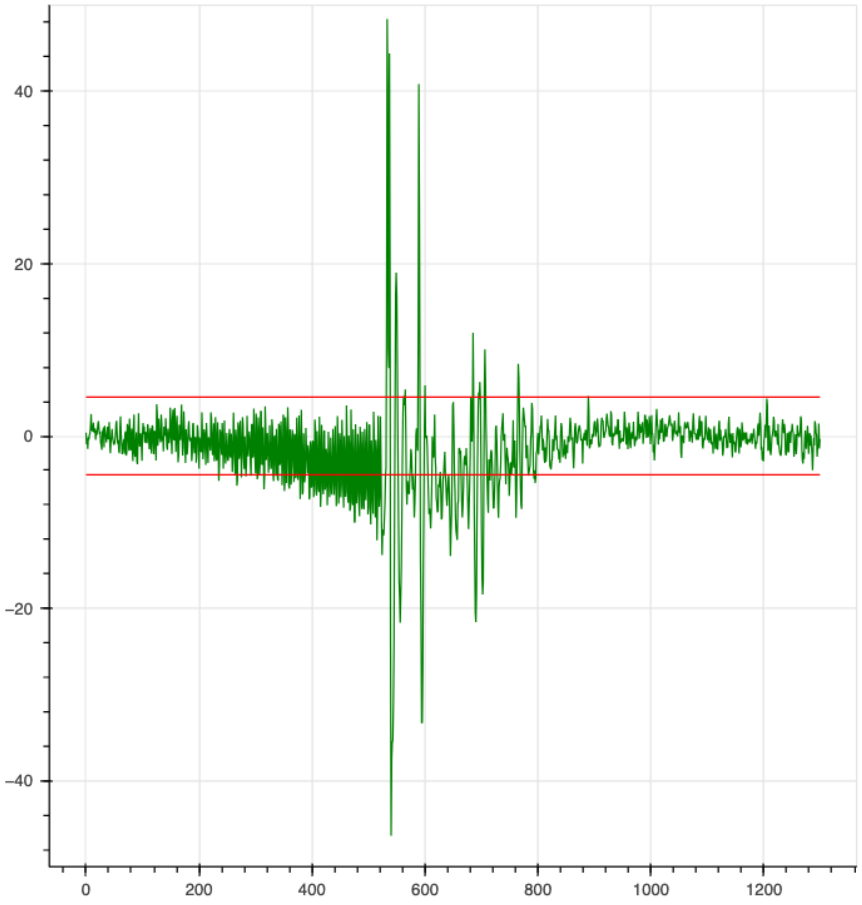
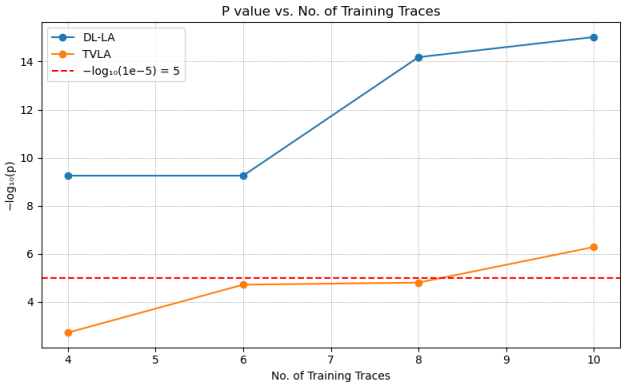


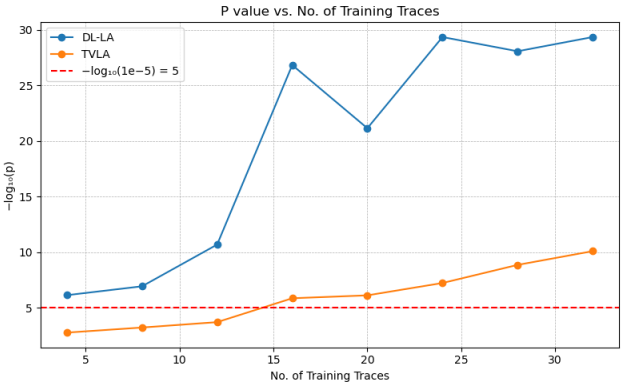
Figure 3.59: T-test of the traces acquired by varying all shares of the input fields, Ascon with TI with randoms for the countermeasure set to zero

Application of DL-LA and comparison with TVLA

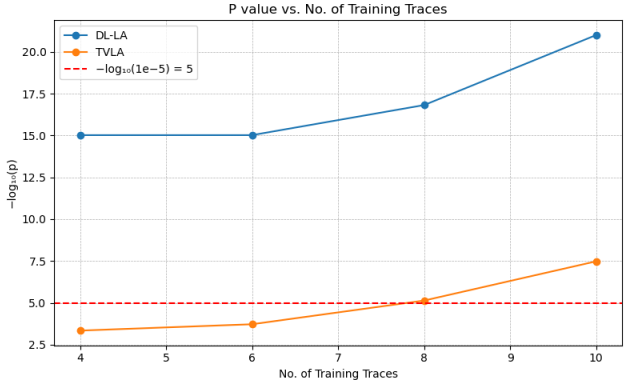
Not surprisingly, very few traces were enough for both TVLA and DL-LA to find a leakage in this Ascon implementation without countermeasures, as shown in figure 3.60. For both the *key-varying* trace set and the one varying all input fields, the validation set was fixed to only 100 traces, while for the *pt-varying* traces set it was fixed to 300 traces, because a few more training traces were needed to better generalize the results of the MLP, so the validation set was increased as well.
In this scenario, DL-LA seems to perform better than TVLA, requiring less traces to find a leakage; however, it is important to point out that in this comparison for DL-LA method only training traces are considered in the count. Obviously, if one performs the comparison considering the total count of traces (hence executing TVLA with both the training and validation traces), then TVLA would



(a) Varying only the key (M=100)



(b) Varying only the plaintext (M=300)



(c) Varying all input fields (M=100)

Figure 3.60: P-values varying the number of (training) traces for both TVLA and DL-LA, in all the three different kinds of acquisitions

produce confidence values consistently higher than those obtained by DL-LA. However, here we want to highlight the fact that only the training traces are actually used by DL-LA to *observe* the leakage, as the validation traces only serve the purpose of evaluating the model.
We also tried to apply our DL-LA method to the implementation of Ascon with DOM countermeasure. For it, at the current stage, the available traces have proven insufficient for the MLP to detect significant leakage or to achieve better results than those obtained through TVLA.

3.5.4 Conclusions and Future works

There are several promising directions for extending this work.
To begin with, we have initiated a preliminary comparison of different countermeasure implementations for the Ascon algorithm, with the aim of assessing their robustness. This represents an initial step toward a broader and more systematic evaluation. While our t-test analyses have revealed some leakages in both Domain-Oriented Masking (DOM) and Threshold Implementation (TI) schemes, we were not yet able to exploit these leakages to successfully mount an attack. This indicates that, although leakages seem to be present, their practical exploitability remains uncertain and requires deeper investigation.
Additionally, considerable work remains in the area of Deep Learning-based Leakage Assessment (DL-LA). In this direction:

- So far, we have employed a neural network architecture closely aligned with models commonly used in the literature. However, we believe that a network specifically tailored to the characteristics of the Ascon cipher could significantly enhance the effectiveness of DL-LA in detecting subtle leakages. Designing and evaluating such models will be a key focus of our future work.

- Moreover, there is substantial room for improvement in the study of protected implementations of Ascon within the DL-LA framework. Further exploration of these aspects (including deeper architectural tuning, extended training methodologies, and broader dataset usage) is also planned as part of our future research efforts.

3.6 Side-channel leakages analysis with VoLPE

3.6.1 Introduction

Although in the case in which the security of the algorithms used in a cryptosystem is well established, the physical security of their implementation remains a significant concern. An attacker with physical access to a device can extract information about the secret key and internal computational state by measuring the device's power consumption and electromagnetic emissions. These measurements can be made using metal probes placed on internal circuit wires.
Such attacks, which exploit physical side effects of computation, are known as Side Channel Attacks (SCA). When the attack specifically targets information leaked through power consumption, it is referred to as a Power Analysis Attack. These leakages often arise from glitches, unintended signal transitions during computation, caused by differences in signal propagation times within the circuit.
A highly effective strategy for defending against SCAs involves the use of masking schemes, which aim to decorrelate the computation from the sensitive data being processed. This is typically achieved by XORing the circuit inputs with random values. An additional layer of protection

can be implemented using threshold implementations. This technique splits each sensitive variable v into $n + 1$ random, independent shares, whose XOR equals v.
Non-linear gates are particularly susceptible to side-channel leakage. To mitigate this, these gates are replaced by gadgets, which are composite gate structures designed to perform the same logic operation, but in a way that incorporates masking and threshold implementation techniques.
To evaluate the effectiveness of these countermeasures, security models are created to emulate the capabilities of a potential attacker. These models define what information an attacker could extract from each measurement. However, they inherently approximate the real-world behavior of the physical circuit, and thus introduce some degree of inaccuracy in the verification process.

State of the art

In the literature, various tools and techniques have been proposed to assess the level of protection a circuit (or parts of it) offers against side-channel attacks. Many of these methods are based on the concept of probing security, as discussed in works such as [22, 27, 119].
Among the available tools, some rely on hardware simulation to estimate characteristics like timing and area during the pre-silicon design phase. For instance, CASCADE [181] is a framework that integrates a largely automated, full-stack standard-cell design flow with state-of-the-art side-channel analysis techniques. It enables efficient evaluation of side-channel leakage before chip fabrication.
In the literature, some works incorporate toggle-based analysis within their evaluation workflows, similar to the approach used in our tool. For example, [144] presents a leakage model for pre-silicon power analysis in cryptographic designs. However, unlike our method, they rely exclusively on proprietary tools, and their use of toggle count differs slightly from our approach.
In this context, some works investigate the accuracy of design time power estimation tools in assessing the security level of a device against differential power attacks [19]. For example, these models and tools can lack precision. Consider SPICE, a transistor-level simulator that generates analog waveforms reliable enough to validate timing accuracy. While its high fidelity suggests that it could be useful for side-channel leakage evaluation after layout, the study in [120] shows that the statistical variation in power noise amplitude produced by SPICE is inconsistent and not always accurate for side-channel analysis.

Motivations

In ORSHIN Work Package 3, one of our objectives was to explore new and effective tools to simulate circuits and analyze their behavior, including in the presence of glitches. Our specific goal was to define a methodology for modeling the power consumption of circuits using only **open-source tools**. This effort led to the development of VoLPE, our dedicated tool for side-channel analysis of digital circuits, particularly those protected by techniques such as secret sharing or by replacing vulnerable parts with gadgets.

3.6.2 Workflow and Exploited tools

The aim of this work was to create a tool for the analysis of the side-channel attack resistance for gadgets. All the tools used in the workflow are open source tools, and in particular we developed a new one for the analysis phase that is called VoLPE (Verification of Leakages Propagation Escalation).

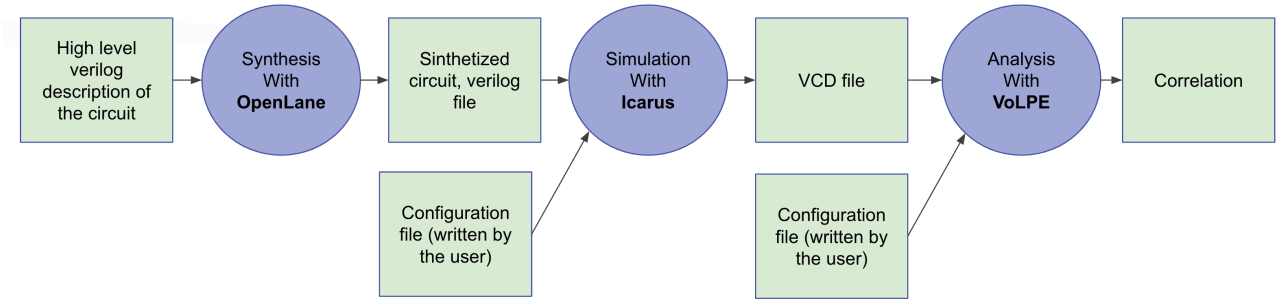


Figure 3.61: Workflow followed in this work and described in section 3.6.2.

Designing a digital circuit is a complex process that goes far beyond simply writing hardware description language (HDL) code. Particularly in security applications such as cryptographic hardware, the **workflow** includes not only design and verification but also implementation steps that affect the physical characteristics of the circuit, characteristics that can inadvertently leak information through side channels.

Synthesis

The general workflow begins with the *synthesis phase*. In this step, the HDL (Hardware Description Language) code is transformed into a netlist, which represents the circuit in terms of gates and wires. During synthesis, various optimization techniques can be applied to improve performance, reduce area, or lower power consumption. The input to the synthesis tool is a high level Verilog description of the circuit, and output a verilog description of the synthetized circuit.

OpenLane

In our case, we perform the synthesis phase through OpenLane. OpenLane is an open source tool that takes a circuit description written in Verilog and produces a GDSII (or GDS2) file as output [70, 148]. This GDSII file is a binary representation detailing the circuit's various layers, which is the format required by semiconductor foundries for physical fabrication.
To create a GDSII file, OpenLane follows a structured design flow consisting of six main steps. The process begins with **synthesis**, where OpenLane translates the logical Verilog description into a netlist, that means in a comprehensive list of the circuit's components and how they are interconnected. These components are chosen from a standard cell library provided within OpenLane. The synthesis step is carried out using an external tool called *Yosys*, which generates the netlist [4]. This is followed by a static timing analysis performed by another tool, OpenSTA [71], to evaluate the timing performance of the netlist.
Next, in the **floorplanning stage**, OpenLane determines how much space is needed to accommodate all the components on the chip. This is followed by the **placement phase**, which involves positioning the components within the chip area. Placement is done in two stages: a rough (coarse) placement to estimate general locations, and a fine placement to determine their exact positions. During this step, OpenLane ensures there is no overlap between components.
The workflow then proceeds to **Clock Tree Synthesis (CTS)**. The clock signal is vital for the operation of many gates in a digital circuit, and this step ensures that the clock is distributed with appropriate strength and timing to all required parts of the design.
Following CTS is the **routing phase**, where the actual physical connections such as signal paths, power lines, and ground connections, are established between components.

The final stage is **signoff**, where various checks are performed to validate the design before the GDSII file is finalized. During both the CTS and routing phases, OpenLane may make changes to the netlist in response to new design requirements. To verify that these changes haven't altered the intended functionality of the circuit, a **Logic Equivalence Check (LEC)** is performed using Yosys. This ensures that the modified netlist is functionally equivalent to the original one produced during synthesis.
Throughout the entire process, OpenLane relies on a **Process Design Kit (PDK)**, which is a collection of foundry-specific files essential for chip design. The PDK includes the standard cell library used during synthesis, as well as design rule checks and other data necessary for the signoff stage. This ensures that the resulting design adheres to the foundry's manufacturing constraints and specifications.

Simulation

Once synthesis is complete, the process moves on to *simulation*. This phase is crucial for understanding how the circuit will behave under different conditions, such as estimating power consumption, without the need to physically manufacture the design. Simulation helps verify functionality and can provide insight into potential weaknesses or inefficiencies. In this case, the input is the output of the sinthetization tool, and the output is a vcd file.

Icarus

An open source tool commonly used for simulation is Icarus Verilog. It compiles Verilog source files into a .vvp executable, which can then be run to carry out the simulation. Alongside the circuit description, a testbench is required—this is a separate file created by the user that defines the sequence of input signals to apply during the simulation. As the simulation runs, Icarus generates a .vcd (Value Change Dump) file, which records the changes in signal values over time. This file can be viewed using waveform visualization tools like GTKWave, allowing users to observe how each wire in the circuit behaves throughout the simulation.

Analysis

Finally, *analysis* is performed. This involves examining the simulation outputs to detect any unintentional information leaks or vulnerabilities that could be exploited, for example, through power analysis or timing attacks. This step is especially important in security-sensitive applications, as it allows designers to address potential risks before fabrication. We perform this step exploiting our new tool VoLPE, which takes as input the vcd file in output from the simulation tool, and gives to use information about the correlation between the number of toggles counted for each input and the Hamming weight or the Hamming distance of the corresponding input state.

3.6.3 Structure of VoLPE

The objective of this work is to develop a tool able of realistically analyzing a circuit's vulnerability to side-channel attacks (SCA). The resulting tool, named VoLPE (Verification of Leakages Propagation Escalation), achieves this by simulating a synthesized circuit, produced using the OpenLane digital design flow, and computing the correlation between the circuit's inputs and a power consumption model. This correlation value provides insight into the extent to which internal

signal transitions leak information, thus helping to assess the circuit's resistance to side-channel leakage.
The overall structure of the workflow has been outlined in section 3.6.2 and figure 3.61. Once the user has created a high-level description of the gadget in Verilog, the next step is to generate its synthesized version using the OpenLane digital design flow. The resulting Verilog file, which contains the synthesized netlist, is then provided as input to VoLPE, along with a user-defined configuration file. Simulation is handled by a script named sim.sh, which executes the simulation process and returns the resulting logs to VoLPE's main module. VoLPE then processes these logs to compute the relevant metrics, such as correlation values, and exports the results into an Excel file for further analysis. From this point forward, the term *circuit* refers specifically to the synthesized version produced by OpenLane.

Configuration file

Before running the tool, some essential information about the circuit to be simulated must be provided. This is done via a **configuration file**, where the user fills in specific fields that define the simulation parameters and circuit features. Below is a description of each field in the configuration file:

- *full*: This field specifies whether the simulation should be exhaustive or partial. Setting this field to "y" enables exhaustive simulation, where all possible input combinations are tested. Setting it to "n" enables partial simulation, using a subset of the input space. A more detailed explanation is provided in next sessions.
- *sim*: This field defines the total number of simulations to be performed. If full is set to y, the number of simulations must be exactly 2^{2*x}, where x corresponds to the value of *in_size* (the number of input bits). If full is set to n, then x can be any integer from 0 up to *in_size*, and the corresponding number of simulations must be specified manually.
- *clk*: This field indicates whether the circuit includes a clock. Use y if a clock is required for simulation, and n if the circuit is combinatorial (i.e., does not use a clock).
- *period*: This field sets the clock period (in nanoseconds or the simulation time unit). It also defines how frequently the input values are updated during simulation. The value must be large enough to allow the circuit to complete processing the current input before the next input is applied. If, for example, the circuit's longest propagation delay is 5 ns, the period must be set to a value greater than 5 ns. To determine the appropriate period, the maximum propagation delay must be calculated by identifying the slowest path through the circuit, i.e. the path with the highest cumulative delay based on gate delays.
- *cycles*: This field indicates the number of clock cycles the circuit requires to complete its computation. It should be equal to the number of register stages in the design plus one.
- *in_size* / *out_size* / *rand_size*: These fields define the number of input bits, output bits, and random bits, respectively, used in the circuit. Random bits may be used for masking or other security techniques.
- *in_name* / *out_name*: These fields specify the names of the variables representing the circuit's inputs and outputs, as defined in the Verilog code.

Table 3.6: Sample of a configuration file.

sim:	576
full:	n
clk:	y
period:	5
cycles:	2
in_size:	6
in_name:	in
rand_size:	2
out_size:	4
out_name:	out
input a0	0.02
gate XNOR_DELAY	0.10
gate NAND_DELAY	0.20
gate XOR_DELAY	0.50
gate AND_DELAY	0.30
gate OR_DELAY	0.40

- *input delayed_input_name*: These fields allow the user to define input arrival delays (in nanoseconds). Each input bit can be assigned a delay value; if no delay is present, it should be set to 0.
- *delayed_gate_name*: This field is used to define the propagation delays associated with each type of gate in the synthesized circuit (in nanoseconds). These delay values are used in computing signal propagation and for timing-related analysis during simulation.

In table 3.6 is shown a sample of configuration file.

Some notes about simulations

Generally speaking, a simulated circuit will have an initial state. During simulation, a new input is applied, potentially causing a change in the output. The simulation continues until the circuit reaches a new stable state, at which point it is considered complete.
The first step executed by the tool involves generating the set of inputs and corresponding initial states required for each simulation. These values are stored in a dedicated file to be used during the simulation phase.

Exhaustive or partial simulation

As previously mentioned, the circuit can be simulated in two modes: exhaustively, where all possible input combinations are tested, or partially, using only a selected subset of inputs, depending on the configuration provided by the user. The partial simulation option was introduced to address the exponential growth in the number of required simulations as the number of input bits increases. Since the number of possible input combinations doubles with each additional bit, exhaustive simulation can quickly become computationally infeasible for larger circuits. Partial simulation helps mitigate this by significantly reducing computation time, making the tool more practical for analyzing complex designs.

Ideal or delayed propagation

After generating the initial states and input vectors, the tool proceeds with the simulation phase. For each circuit under analysis, the tool supports three distinct types of simulation runs, each modeling different timing scenarios.

- **Circuit with no delays**: In this mode, the circuit is treated as ideal, meaning all gate and input delays are assumed to be zero. As a result, no glitches can occur during simulation. This allows the tool to isolate and evaluate correlations that stem solely from the algorithmic behavior of the circuit, rather than from its physical implementation.
- **Circuit with gate delays**: In this run, gate-level delays are taken into account. The inclusion of these delays can lead to the appearance of glitches, which are temporary signal transitions caused by differing gate propagation times. This simulation models a circuit with synchronized inputs that all arrive simultaneously, allowing the structural effects of the implementation to be studied.
- **Circuit with gate and input delays**: This mode extends the previous one by also incorporating input arrival delays, simulating scenarios where inputs reach the circuit asynchronously or at different times. This provides a more realistic representation of how the circuit might behave in a real-world environment, especially when driven by unsynchronized components or external sources.

Power consumption model

Once all simulations are completed, **power consumption** is estimated by analyzing the logs generated during these runs. For each simulation, power usage is modeled based on how many *bit toggles* occur in the output before it stabilizes into a new state. A toggle is a change in a single bit's value, either from 0 to 1 or from 1 to 0.
The rationale behind using toggles as a power model is that each toggle roughly corresponds to a switching activity, which in real digital circuits consumes dynamic power. Therefore, counting toggles gives a reasonably accurate estimate of power usage.
This analysis step results in a list, often referred to as *tl* (toggle list), where each entry corresponds to the number of toggles observed in a single simulation. This list can then be used for further analysis, such as computing average power, peak power, or comparing power efficiency across different designs or inputs.

Consumption model

After generating a power consumption profile for each simulation, the next step is to identify the operation responsible for that consumption. To achieve this, we introduce two distinct modeling approaches, each implemented through a dedicated function:

- **Input Consumption Model** (I-CS): Captures the relationship between power consumption and the input values that triggered it.
- **Input-State Consumption Model** (IS-CS): Captures the relationship between power consumption and the combination of input values and the system's initial state.

Since users may only be interested in specific bits or combinations of input and initial state bits, we provide a selection function for each model. This function enables users to specify which bits should be considered when building the corresponding consumption model. Once the relevant bits are selected, a second function generates the respective model based on this selection. Users can redefine these functions to suit their needs.

3.6.4 Results

In the final stage of the analysis, Pearson's correlation coefficient is calculated to measure the correlation between the power consumption values and the outputs of each consumption model.

3.6.5 Testing and results

Following the description of the developed tool provided in Section 3.6.3, this section presents some of the results obtained through its application. We analyze the performance of the tool, discuss the insights derived from the data it generated, and evaluate its effectiveness in modeling and correlating power consumption based on the defined methodologies.

Analysis of some gadgets

In this section, we analyze the results obtained by applying the VoLPE tool to three cryptographic gadgets, which were also examined in Section 3.4. The three gadgets under evaluation are: the χ operation implemented with a threshold implementation using two shares, the χ operation with a threshold implementation using three shares, and the χ operation employing a domain-oriented masking scheme.
For all the following examples, we considered three scenarios: one in which there are no delays, neither in the inputs nor in the gates, one in which only the gates experience delays, and the other in which both inputs and gates experience delays. In the latter case, these delays are randomly specified in the configuration file.
Note that without countermeasures, the means and maximum values of the correlations with random delays for both consumption models are approximately one (compared to the results in tables 3.9, 3.12, 3.15).

χ with two shares

In the χ function, the inputs are three bits x_1, x_2 x_3, and the output is one bit. When the threshold implementation with two shares countermeasure is applied, each input bits is split into two shares x_i^0, x_i^1 (sec.3.4.1).

Single execution To facilitate the understanding of the analysis presented in Section 3.6.5, we begin with an example illustrating a single execution of the tool. Table 3.7 displays the randomly assigned delays used in this example, for all the shares of the three bits that serve as inputs to the χ function, as well as the delays associated with all possible gates (noting that only a subset of these gates is actually used in the gadget under analysis). Table 3.8 presents the correlation results of the consumption computed counting the toggles, as explained in section 3.6.3, and the consumption models based on Hamming Weight and Hamming Distance for each of the three inputs to the χ function. This correlation is computed under three different scenarios: (i) no delay, (ii) delays applied to the gates, and (iii) delays applied to both gates and inputs.

Table 3.7: Used delays for inputs and gates in the example, χ with two shares.

Input	Used delay (ns)
x_1^0	0.57
x_1^1	0.26
x_2^0	1
x_2^1	0.70
x_3^0	0.64
x_3^1	0.36

Gate	Used delay (ns)
XNOR	0.32
NAND	0.91
XOR	0.41
AND	0.57
OR	0.38
NOR	0.58

Table 3.8: Correlation results for the example, χ with two shares.

	Hamming weight			Hamming distance		
	x_1	x_2	x_3	x_1	x_2	x_3
no del.	~ 0	~ 0	0	~ 0	~ 0	0
gate del.	~ 0	~ 0	0	~ 0	~ 0	0
gates/inputs del.	0,21	~ 0	~ 0	0,02	~ 0	~ 0

Analysis of more executions After that, we did 100 execution of our tool in the case in which both inputs and gates have some delays, with everytime these delays randomically computed. Then we compute the mean and extract the maximum value for both the consumption model cases (Hamming weight and Hamming distance) and for all the χ inputs (x_1, x_2 and x_3) (table 3.9).
We can note that the value of mean correlation for input x_1 is quite high w.r.t. the others, which can tell us a possible leakage of this input value; this is highlighted in the both consumption model cases, with Hamming weight and Hamming distance.

χ with three shares

When the threshold implementation with three shares countermeasure is applied to function χ, each input bits is split into three shares x_i^0, x_i^1, x_i^2 (sec.3.4.1).

Single execution To facilitate the understanding of the analysis presented in Section 3.6.5, we begin with an example illustrating a single execution of the tool. Table 3.10 displays the randomly assigned delays used in this example, for all the shares of the three bits that serve as inputs to the χ function, as well as the delays associated with all possible gates (noting that only a subset of these gates is actually used in the gadget under analysis). Table 3.11 presents the correlation results of the consumption computed counting the toggles, as explained in section 3.6.3, and the consumption models based on Hamming Weight and Hamming Distance for each of the three inputs to the χ function. This correlation is computed under three different scenarios: (i) no delay, (ii) delays applied to the gates, and (iii) delays applied to both gates and inputs.

Table 3.9: Mean and Max of the correlation results on 100 executions of VoLPE, χ with two shares.

	Hamming weight			Hamming distance		
	x_1	x_2	x_3	x_1	x_2	x_3
Mean with delays	0,034	~ 0	~ 0	-0,001	~ 0	~ 0
Max with delays	0,378	~ 0	~ 0	0,059	~ 0	~ 0

Table 3.10: Used delays for inputs and gates in the example, χ with three shares.

Input	Used delay (ns)
x_1^0	0.81
x_1^1	0.26
x_1^2	0.95
x_2^0	0.27
x_2^1	0.62
x_2^2	0.76
x_3^0	0.25
x_3^1	0.64
x_3^2	0.14

Gate	Used delay (ns)
XNOR	0.06
NAND	0.87
XOR	0.92
AND	0.27
OR	0.57
NOR	0.59

Table 3.11: Correlation results for the example, χ with three shares.

	Hamming weight			Hamming distance		
	x_0	x_1	x_2	x_0	x_1	x_2
no del.	~ 0	~ 0	0	~ 0	~ 0	0
gate del.	~ 0	~ 0	~ 0	~ 0	~ 0	~ 0
gates/inputs del.	~ 0	~ 0	~ 0	~ 0	~ 0	~ 0

Analysis of more executions After that, we did 100 execution of our tool in the case in which both inputs and gates have some delays, with everytime these delays randomically computed. Then we compute the mean and extract the maximum value for both the consumption model cases (Hamming weight and Hamming distance) and for all the χ inputs (x_1, x_2 and x_3) (table 3.12).
Unlike the case of the function χ with two shares, in this scenario even the row reporting the maximum correlation values across all executions shows consistently low correlation values (tending to zero), different from what was observed for χ with two shares.

χ with DOM

When DOM countermeasure is applied to function χ, each input bits is split into two shares x_i^0, x_i^1 (sec.3.4.1).

Single execution Table 3.13 displays the randomly assigned delays used in this example, for all the shares of the three bits that serve as inputs to the χ function, as well as the delays associated with all possible gates (noting that only a subset of these gates is actually used in the gadget under analysis). Table 3.14 presents the correlation results of the consumption computed counting the toggles, as explained in section 3.6.3, and the consumption models based on Hamming Weight and Hamming Distance for each of the three inputs to the χ function. This correlation is

Table 3.12: Mean and Max of the correlation results on 100 executions of VoLPE, χ with three shares.

	Hamming weight			Hamming distance		
	x_1	x_2	x_3	x_1	x_2	x_3
Mean with delays	~ 0	~ 0	~ 0	~ 0	~ 0	~ 0
Max with delays	~ 0	~ 0	~ 0	~ 0	~ 0	~ 0

Table 3.13: Used delays for inputs and gates in the example, χ with DOM scheme.

Input	Used delay (ns)
x_1^0	0.57
x_1^1	0.29
x_2^0	0.03
x_2^1	0.43
x_3^0	0.94
x_3^1	0.34

Gate	Used delay (ns)
XNOR	0.40
NAND	0.19
XOR	0.65
AND	0.77
OR	0.28
NOR	0.31

Table 3.14: Correlation results for the example, χ with DOM scheme.

	Hamming weight			Hamming distance		
	x_1	x_2	x_3	x_1	x_2	x_3
no del.	~ 0	~ 0	~ 0	~ 0	~ 0	0.004
gates del.	0,025	~ 0	~ 0	0,025	~ 0	0,329
gates/inputs del.	0,030	~ 0	~ 0	0,030	~ 0	0,429

computed under three different scenarios: (i) no delay, (ii) delays applied to the gates, and (iii) delays applied to both gates and inputs.

Analysis of more executions After that, we did 100 execution of our tool in the case in which both inputs and gates have some delays, with everytime these delays randomically computed. Then we compute the mean and extract the maximum value for both the consumption model cases (Hamming weight and Hamming distance) and for all the χ inputs (x_1, x_2 and x_3) (table 3.15).
In this case, it should be noted that the mean correlation values for the inputs x_1 and x_3 are not particularly low and do not tend to zero. This observation suggests the possibility of leakage associated with these inputs, although a more in-depth analysis is required to draw definitive conclusions.

Analysis of the S-Box of AES

For the AES S-Box, we chose to conduct our tests on two distinct implementations. The first implementation employs a lookup table to perform the byte substitution. The second implementation achieves the same functionality using a sequence of MUX gates. Unlike the first, this implementation supports both encryption and decryption operations, with the mode determined by a configuration bit: setting the bit to 0 enables encryption, while setting it to 1 enables decryption. To explore the behavior of this design, we synthesized the circuit twice using OpenLane, once with the configuration bit fixed to 0 (indicating encryption only), and once without fixing the bit value. When the bit is fixed to 0, OpenLane recognizes that the decryption logic is unused and

Table 3.15: Mean and Max of the correlation results on 100 executions of VoLPE, χ with DOM scheme.

	Hamming weight			Hamming distance		
	x_1	x_2	x_3	x_1	x_2	x_3
Mean with delays	-0,036	~ 0	~ 0	0,036	~ 0	0,125
Max with delays	0,146	~ 0	~ 0	0,146	~ 0	0,583

Table 3.16: Mean and Max of the correlation results on 100 executions of VoLPE, S-Box of AES implemented with lookup table.

	Hamming Weight		**Hamming Distance**	
	1 bit	8 bits	1 bit	8 bits
Mean with delays	0.047	0,0189	0,3236	0,7792
Max with delays	0,1128	0,0485	0,3836	0,8504

Table 3.17: Mean and Max of the correlation results on 100 executions of VoLPE, S-Box of AES implemented with MUX - only encryption.

	Hamming Weight		**Hamming Distance**	
	1 bit	8 bits	1 bit	8 bits
Mean with delays	0,0356	0,0197	0,1555	0,1645
Max with delays	0,117	0,0398	0,2977	0,3181

removes it during synthesis. As a result, we obtained two different synthesized versions of the second implementation.
In total, we tested three circuit variants: the lookup table-based implementation, and both synthesized versions of the MUX-based design. For each circuit, we computed the correlation first for each input bit individually, and then for all input bits considered collectively (see Section 4.1.4).
Note that in these cases the s-boxes are not protected by countermeasures, and in the next sections it is clear how for no protected nonlinear functions our tool reveals some leakages of the input values.

AES S-Box lookup table

Table 3.16 presents the results for the scenario where the AES S-Box is implemented using a lookup table. In this case, we observe that the correlation values do not tend toward zero, neither the maximum values nor the means. For each of the considered power consumption models, two correlation values are reported: one calculated with respect to a single input bit, and another computed with respect to all eight input bits of the S-Box.

AES S-Box MUX only encryption

Table 3.17 presents the results for the scenario in which the AES S-Box is implemented using MUX gates, able to perform only encryption. In this case also, we observe that the correlation values do not tend toward zero, neither the maximum values nor the means, and we can reach the same conclusions as in section 3.6.5.

AES S-Box MUX encryption and decryption

Table 3.18 presents the results for the scenario in which the AES S-Box is implemented using MUX gates, able to perform both encryption and decryption. In this case also, we observe that the correlation values do not tend toward zero, neither the maximum values nor the means, and we can reach the same conclusions as in sections before.

Table 3.18: Mean and Max of the correlation results on 100 executions of VoLPE, S-Box of AES implemented with MUX - encryption and decryption.

	Hamming Weight		**Hamming Distance**	
	1 bit	8 bits	1 bit	8 bits
Mean with delays	0,0339	0,0148	0,1931	0,1993
Max with delays	0,1536	0,0511	0,3298	0,4529

3.6.6 Conclusions and Future works

We developed a tool called VoLPE, designed to quantify the power consumption leakage of a circuit by analyzing the toggle activity observed during its simulation. Our workflow relies exclusively on open-source tools. Because our analysis is based on the simulation of a synthesized netlist, it provides more realistic results compared to approaches that evaluate higher-level descriptions of cryptographic components.
Our methodology consists of the following key steps.

- **Modeling power consumption**: We identified a meaningful way to model a circuit's power consumption during computation. Since dynamic power consumption primarily arises from switching activity, i.e., changes in signal values from 0 to 1 and vice versa, we chose to model power based on the number of such transitions, commonly referred to as **toggles**.
- **Defining input-related metrics**: To correlate input data with power consumption, we defined a consumption model. Specifically, we considered metrics such as Hamming weight and Hamming distance of the inputs.

The most notable result concerns the countermeasures applied to the nonlinear function χ. We analyzed three protection schemes: a two shares threshold implementation, a three shares threshold implementation, and the DOM (Domain-Oriented Masking) scheme. As discussed in Section 3.6.5, when examining the mean correlation values for each gadget, the three shares threshold implementation of χ shows correlations that tend toward zero, indicating strong protection. In contrast, our tool detects potential leakage in the other two cases (the two shares implementation and the DOM-protected version). While leakage in the two shares implementation is expected due to its lower security order, the result for the DOM-protected χ is more surprising and warrants further investigation.
There are several potential improvements to be made in the implementation of the tool. In its current state, only one power consumption model has been developed and integrated, along with a limited set of selection functions and consumption models. Future enhancements could include the incorporation of additional power models, selection functions, and consumption models, as well as the ability for users to select the desired configuration before running computations.
Currently, VoLPE computes correlations only between the inputs and the power consumption observed at the outputs of the gadget. A valuable extension would be the ability to simulate probes at internal nodes of the circuit, enabling correlation analysis between inputs and power consumption within specific sub-circuits.

Chapter 4

Summary, conclusion and outlook

The work in ORSHIN WP3 ”Models for formal verification” has been very successful. Both the task on micro-architectural side channels as well as the task on physical side channels have produced significant results.
In the area of protection against micro-architectural side channels three hardware models were developed. First, ProSpeCT is a processor model that formally specifies the security guarantees that a processor gives during speculative execution. Based on this model, we have formally proven that constant-time code running on a ProSpeCT processor is not vulnerable to Spectre attacks. To show the feasibility of the proposed model, we have implemented a ProSpeCT compliant processor as an extension to an open-source RISC-V processor. This result has been published at the prestigious Usenix Security conference [55].
The second and third models (AMi and Libra) follow the same model-driven approach to handle other classes of micro-architectural attacks, more specifically control-flow leakage attacks. In particular, AMi extends a processor with architectural features to handle control-flow leakage attacks more efficiently using linearization and balancing, and Libra adds architectural features that make balancing possible on a wider range of processors. We have again implemented both models as an extension to an open-source RISC-V processor. AMi was published at IEEE S&P [171] and Libra was published at ACM CCS [170].
Security and implementation cost in particular are critical for IoT devices, which are the focus of the ORSHIN project. Security does not come for free and will always require some overhead. It is therefore important to understand how much this overhead can be reduced without sacrificing security. With regards to physical side channels we worked in several directions.
We designed, implemented and manufactured a real silicon chip featuring three case studies of state-of-the-art countermeasures, in order to examine gaps between security guarantees provided by theoretical models and practical implementations. We also performed comparative experiments with state-of-the-art countermeasures on FPGA. In both cases our goal was to gain deeper insight into discrepancies and help bridge the gap between theory and practice, which is a primary objective of the ORSHIN project.
We have developed and implemented an open-source tool capable of analyzing hardware designs for potential side-channel leakage. The entire workflow leading up to the use of the tool is carried out using open-source electronic design automation tools, aligning with the objectives of the ORSHIN project.
We have also developed several new countermeasures. The first countermeasure challenges an assumption that is frequently made in current models for formal verification, is secure in practice, and leads to reduced implementation cost. In other words we have shown that current models make an unnecessary assumption and this leads to bloated protected implementations.

This result has been published at the DATE 2023 conference [102]. An extended version of the DATE paper was published in the journal IEEE Transactions on Information Forensics and Security [103].
The second countermeasure is tailored for applications with a strict requirement for low latency. In such applications low latency is prioritized at the cost of greater chip area or higher randomness cost, but they remain secondary design goals. We also implemented and evaluated our countermeasure. It offers first-order security, is provable secure, and leads to reduced implementation cost. Our prototype circuits are formally verified and secure in practice. This result has been published at TCHES 2024 [158].
The third countermeasure is an extension of the second countermeasure to higher security orders. Also here we designed the countermeasure and implemented and evaluated prototype circuits in practice. The countermeasure provides provable higher-order security, and reduced implementation cost compared to the state-of-the-art. Our prototype circuits are formally verified and secure in practice. This result was published at TCHES 2025 [159].
Several prototype implementations of two countermeasures are available under an open-source license and served as basis for the ORSHIN demonstrators reported in D3.2.

Bibliography

[1] Wokwi: chi with dom design, 2023. https://wokwi.com/projects/341614374571475540.

[2] Wokwi: chi with three shares design, 2023. https://wokwi.com/projects/341608574336631379.

[3] Wokwi: chi with two shares design, 2023. https://wokwi.com/projects/341589685194195540.

[4] Yosys open synthesis suite, 2025. https://yosyshq.net/yosys/.

[5] Johan Agat. Transforming out timing leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 40–53. ACM, 2000.

[6] Sam Ainsworth and Timothy M. Jones. MuonTrap: Preventing cross-domain spectre-like attacks by capturing speculative state. In *ISCA*, pages 132–144. IEEE.

[7] Nadhem J. Al Fardan and Kenneth G. Paterson. Lucky thirteen: Breaking the tls and dtls record protocols. In *S&P*, 2013.

[8] Martin R. Albrecht and Kenneth G. Paterson. Lucky microseconds: A timing attack on amazon's s2n implementation of tls. In *EUROCRYPT*, 2016.

[9] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *S&P*, 2019.

[10] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 53–70, 2016.

[11] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *CCS*, pages 1807–1823. ACM.

[12] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *USENIX Security Symposium*, pages 53–70. USENIX Association.

[13] AMD. Security Analysis of AMD Predictive Store Forwarding. https://www.amd.com/system/files/documents/security-analysis-predictive-store-forwarding.pdf.

[14] Nadav Amit, Fred Jacobs, and Michael Wei. JumpSwitches: Restoring the performance of indirect branches in the era of spectre. In *USENIX Annual Technical Conference*, pages 285–300. USENIX Association.

[15] Victor Arribas, Zhenda Zhang, and Svetla Nikova. LLTI: low-latency threshold implementations. *IEEE Trans. Inf. Forensics Secur.*, 16:5108–5123, 2021.

[16] Qinkun Bao, Zihao Wang, Xiaoting Li, James R. Larus, and Dinghao Wu. Abacus: Precise side-channel analysis. In *ICSE*, pages 797–809. IEEE.

[17] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. SpecShield: Shielding speculative data from microarchitectural covert channels. In *PACT*, pages 151–164. IEEE.

[18] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks. In *USENIX Security*, August 2022. Intel Bounty Reward.

[19] Alessandro Barenghi, Guido Bertoni, Fabrizio De Santis, and Filippo Melzani. On the efficiency of design time evaluation of the resistance to power attacks. In *14th Euromicro Conference on Digital System Design, Architectures, Methods and Tools, DSD 2011, August 31 - September 2, 2011, Oulu, Finland*, pages 777–785. IEEE Computer Society, 2011.

[20] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François-Xavier Standaert. maskverif: Automated verification of higher-order masking in presence of physical defaults. In Kazue Sako, Steve A. Schneider, and Peter Y. A. Ryan, editors, *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I*, volume 11735 of *Lecture Notes in Computer Science*, pages 300–318. Springer, 2019.

[21] 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 *CCS*, pages 116–129. ACM, 2016.

[22] Gilles Barthe, Sonia Belaïd, Francois Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In *23rd ACM Conference on Computer and Communications Security*, volume October 2016, pages 116–129, United States, October 2016. Association for Computing Machinery (ACM).

[23] Gilles Barthe, Sunjay Cauligi, Benjamin Grégoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. High-assurance cryptography in the spectre era. In *IEEE Symposium on Security and Privacy*, pages 1884–1901. IEEE.

[24] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time". In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 328–343. IEEE, 2018.

[25] Gilles Barthe, Tamara Rezk, and Martijn Warnier. Preventing timing leaks through transactional branching instructions. *Electronic Notes in Theoretical Computer Science*, 153(2):33–55, 2006.

[26] Arthur Beckers, Lennert Wouters, Benedikt Gierlichs, Bart Preneel, and Ingrid Verbauwhede. Provable secure software masking in the real-world. In Josep Balasch and Colin O'Flynn, editors, *Constructive Side-Channel Analysis and Secure Design - COSADE 2022*, volume 13211 of *LNCS*, pages 215–235. Springer, 2022.

[27] Sonia Belaïd, Darius Mercadier, Matthieu Rivain, and Abdul Rahman Taleb. IronMask: Versatile verification of masking security. Cryptology ePrint Archive, Paper 2021/1671, 2021.

[28] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *LATINCRYPT*, volume 7533 of *Lecture Notes in Computer Science*, pages 159–176. Springer.

[29] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Keccak. Cryptology ePrint Archive, Paper 2015/389, 2015.

[30] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting speculative execution through port contention. In *CCS*, pages 785–800. ACM.

[31] Begül Bilgin, Joan Daemen, Ventzislav Nikov, Svetla Nikova, Vincent Rijmen, and Gilles Van Assche. Efficient and first-order dpa resistant implementations of keccak. In Aurélien Francillon and Pankaj Rohatgi, editors, *Smart Card Research and Advanced Applications*, pages 187–199, Cham, 2014. Springer International Publishing.

[32] Alex Biryukov, Daniel Dinu, Yann Le Corre, and Aleksei Udovenko. Optimal first-order boolean masking for embedded iot devices. In Thomas Eisenbarth and Yannick Teglia, editors, *Smart Card Research and Advanced Applications - CARDIS 2017*, volume 10728 of *LNCS*, pages 22–41. Springer, 2017.

[33] Marton Bognar, Jo Van Bulck, and Frank Piessens. Mind the gap: Studying the insecurity of provably secure embedded trusted execution architectures. In *S&P*, 2022.

[34] Marton Bognar, Hans Winderix, Jo Van Bulck, and Frank Piessens. Microprofiler: Principled side-channel mitigation through microarchitectural profiling. In *EuroS&P*, 2023.

[35] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. Constantine: Automatic side-channel resistance using efficient control and data flow linearization. In *CCS*, 2021.

[36] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut T. Kandemir. CaSym: Cache aware symbolic execution for side channel detection and mitigation. In *IEEE Symposium on Security and Privacy*, pages 505–521. IEEE.

[37] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *USENIX Security Symposium*, pages 991–1008. USENIX Association, 2018.

[38] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking transient execution through microarchitectural load value injection. In *IEEE Symposium on Security and Privacy*, pages 54–72. IEEE.

[39] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security Symposium*, pages 249–266. USENIX Association.

[40] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security*, 2019.

[41] Gaëtan Cassiers, Benjamin Grégoire, Itamar Levi, and François-Xavier Standaert. Hardware private circuits: From trivial composition to full verification. *IEEE Trans. Computers*, 70(10):1677–1690, 2021.

[42] Gaëtan Cassiers and François-Xavier Standaert. Trivially and efficiently composing masked gadgets with probe isolating non-interference. *IEEE Trans. Inf. Forensics Secur.*, 15:2542–2555, 2020.

[43] Sunjay Cauligi, Craig Disselkoen, Klaus von Gleissenthall, Dean M. Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Constant-time foundations for the new spectre era. In *PLDI*, pages 913–926. ACM.

[44] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. FaCT: A flexible, constant-time programming language. In *SecDev*, pages 69–76. IEEE Computer Society.

[45] Chandler Carruth. Speculative Load Hardening. https://llvm.org/docs/SpeculativeLoadHardening.html.

[46] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99*, volume 1666 of *LNCS*, pages 398–412. Springer, 1999.

[47] Kevin Cheang, Cameron Rasmussen, Sanjit A. Seshia, and Pramod Subramanyan. A formal approach to secure speculation. In *CSF*, pages 288–303. IEEE.

[48] Yun Chen, Lingfeng Pei, and Trevor E. Carlson. Afterimage: Leaking control flow data and tracking load operations via the hardware prefetcher. In *ASPLOS*, 2023.

[49] Rutvik Choudhary, Jiyong Yu, Christopher W. Fletcher, and Adam Morrison. Speculative privacy tracking (SPT): Leaking information from speculative execution without compromising privacy. In *MICRO*, pages 607–622. ACM.

[50] Md Hafizul Islam Chowdhuryy and Fan Yao. Leaking secrets through modern branch predictor in the speculative world.

[51] Many contributors. Github repository: Tinytapeout/tinytapeout-02, 2022. https://github.com/TinyTapeout/tinytapeout-02.

[52] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *2009 30th IEEE Symposium on Security and Privacy*, pages 45–60. IEEE, 2009.

[53] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Binsec/Rel: Efficient relational symbolic execution for constant-time at binary-level. In *IEEE Symposium on Security and Privacy*, pages 1021–1038. IEEE.

[54] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Hunting the haunter - efficient relational symbolic execution for spectre with haunted RelSE. In *NDSS*. The Internet Society.

[55] Lesly-Ann Daniel, Marton Bognar, Job Noorman, Sébastien Bardin, Tamara Rezk, and Frank Piessens. Prospect: Provably secure speculation for the constant-time policy. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 7161–7178. USENIX Association, 2023.

[56] Lesly-Ann Daniel, Marton Bognar, Job Noorman, Sébastien Bardin, Tamara Rezk, and Frank Piessens. Prospect: Provably secure speculation for the constant-time policy (extended version), 2023.

[57] Florian Dewald, Heiko Mantel, and Alexandra Weber. AVR processors as a platform for language-based security. In *European Symposium on Research in Computer Security*, pages 427–445. Springer, 2017.

[58] Siemen Dhooghe, Svetla Nikova, and Vincent Rijmen. Threshold implementations in the robust probing model. In Begül Bilgin, Svetla Petkova-Nikova, and Vincent Rijmen, editors, *Proceedings of ACM Workshop on Theory of Implementation Security, TIS@CCS 2019, London, UK, November 11, 2019*, pages 30–37. ACM, 2019.

[59] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2. Submission to Round 1 of the NIST Lightweight Cryptography project, 2019.

[60] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A tool for the static analysis of cache side channels. In *USENIX Security Symposium*, pages 431–446. USENIX Association.

[61] Goran Doychev and Boris Köpf. Rigorous analysis of software countermeasures against cache attacks. In *PLDI*, pages 406–421. ACM.

[62] Aneesh Kandi et Al. ascon-hw-public, 2024. https://github.com/ascon/ascon-hardware.

[63] 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. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):89–120, 2018.

[64] F.Pizlo. What spectre and meltdown mean for WebKit. https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/.

[65] Jacob Fustos, Michael Garrett Bechtel, and Heechul Yun. SpectreRewind: Leaking secrets to past instructions. In *ASHES@CCS*, pages 117–126. ACM.

[66] Jacob Fustos, Farzad Farshchi, and Heechul Yun. SpectreGuard: An efficient data-centric defense mechanism against spectre attacks. In *DAC*, page 61. ACM.

[67] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001*, volume 2162 of *LNCS*, pages 251–261. Springer, 2001.

[68] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. 8(1):1–27.

[69] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1):1–27, 2018.

[70] GitHub. Openlane, 2025. https://github.com/The-OpenROAD-Project/OpenLane/tree/master.

[71] GitHub. opensta, 2025. https://github.com/The-OpenROAD-Project/OpenSTA.

[72] Gilbert Goodwill, Benjamin Jun, Joshua Jaffe, and Pankaj Rohatgi. A testing methodology for side-channel resistance validation. In *NIST Non-Invasive Attack Testing Workshop*, 2011.

[73] Louis Goubin and Jacques Patarin. DES and differential power analysis (the ”duplication” method). In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES'99*, volume 1717 of *LNCS*, pages 158–172. Springer, 1999.

[74] Hannes Groß, Rinat Iusupov, and Roderick Bloem. Generic low-latency masking in hardware. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(2):1–21, 2018.

[75] Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. In Begül Bilgin, Svetla Nikova, and Vincent Rijmen, editors, *Theory of Implementation Security - TIS@CCS 2016*, page 3. ACM, 2016.

[76] Hannes Gross, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. Cryptology ePrint Archive, Paper 2016/486, 2016.

[77] Hannes Gross, David Schaffenrath, and Stefan Mangard. Higher-order side-channel protected implementations of keccak. In *2017 Euromicro Conference on Digital System Design (DSD)*, pages 205–212, 2017.

[78] Hannes Groß, Ko Stoffelen, Lauren De Meyer, Martin Krenn, and Stefan Mangard. First-order masking with only two random bits. In Begül Bilgin, Svetla Petkova-Nikova, and Vincent Rijmen, editors, *Theory of Implementation Security - TIS@CCS 2019*, pages 10–23. ACM, 2019.

[79] Roberto Guanciale, Musard Balliu, and Mads Dam. InSpectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis. In *CCS*, pages 1853–1869. ACM.

[80] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. Spectector: Principled detection of speculative information flows. In *IEEE Symposium on Security and Privacy*, pages 1–19. IEEE.

[81] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. Hardware-software contracts for secure speculation. In *IEEE Symposium on Security and Privacy*, pages 1868–1883. IEEE.

[82] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. SpecuSym: Speculative Symbolic Execution for Cache Timing Leak Detection.

[83] Shaobo He, Michael Emmi, and Gabriela F. Ciocarlie. Ct-fuzz: Fuzzing for timing leaks. In *ICST*, pages 466–471. IEEE.

[84] Jann Horn. Speculative execution, variant 4: Speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.

[85] NewAE Technology Inc. Newae technology inc., chipwhisperer husky, 2020. https://www.newae.com/product-page/chipwhisperer-husky.

[86] NewAE Technology Inc. Chipwhisperer-husky starter kit, 2023. https://chipwhisperer.readthedocs.io/en/latest/Capture/ChipWhisperer-Husky.html.

[87] Teledyne LeCroy Inc. Operator's manual, wavesurfer 3000 oscilloscopes. User Manual, November 2014.

[88] Yuval Ishai, Amit Sahai, and David A. Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, volume 2729 of *LNCS*, pages 463–481. Springer, 2003.

[89] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. "they're not that hard to mitigate": What cryptographic library developers think about timing attacks. In *S&P*, 2022.

[90] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael B. Abu-Ghazaleh. SafeSpec: Banishing the spectre of a meltdown with leakage-free speculation. In *DAC*, page 60. ACM.

[91] Sungkeun Kim, Farabi Mahmud, Jiayi Huang, Pritam Majumder, Neophytos Christou, Abdullah Muzahid, Chia-Che Tsai, and Eun Jung Kim. ReViCe: Reusing victim cache to prevent speculative cache leakage. In *SecDev*, pages 96–107. IEEE.

[92] David Knichel and Amir Moradi. Low-latency hardware private circuits. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 1799–1812. ACM, 2022.

[93] David Knichel, Pascal Sasdrich, and Amir Moradi. SILVER - statistical independence and leakage verification. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part I*, volume 12491 of *Lecture Notes in Computer Science*, pages 787–816. Springer, 2020.

[94] David Knichel, Pascal Sasdrich, and Amir Moradi. Generic hardware private circuits towards automated generation of composable secure gadgets. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):323–344, 2022.

[95] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy*, pages 1–19. IEEE.

[96] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, 1999.

[97] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[98] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.

[99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer.

[100] Boris Köpf and Heiko Mantel. Transformational typing and unification for automatically correcting insecure programs. *International Journal of Information Security*, 6(2-3):107–131, 2007.

[101] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael B. Abu-Ghazaleh. Spectre returns! Speculation attacks using the return stack buffer. In *WOOT @ USENIX Security Symposium*. USENIX Association.

[102] Dilip S. V. Kumar, Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. Low-cost first-order secure boolean masking in glitchy hardware. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2023, Antwerp, Belgium, April 17-19, 2023*, pages 1–2. IEEE, 2023.

[103] S. V. Dilip Kumar, Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. Low-cost first-order secure boolean masking in glitchy hardware. *IEEE Trans. Inf. Forensics Secur.*, 20:2437–2449, 2025.

[104] Adam Langley. ImperialViolet - Checking that functions are constant time with Valgrind. https://www.imperialviolet.org/2010/04/01/ctgrind.html.

[105] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 557–574, 2017.

[106] Andrew J. Leiserson, Mark E. Marson, and Megan A. Wachs. Gate-level masking under a path-based leakage metric. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, volume 8731 of *Lecture Notes in Computer Science*, pages 580–597. Springer, 2014.

[107] Peinan Li, Lutan Zhao, Rui Hou, Lixin Zhang, and Dan Meng. Conditional speculation: An effective approach to safeguard out-of-order execution against spectre attacks. In *HPCA*, pages 264–276. IEEE.

[108] Mikko H. Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *MICRO*, pages 226–237. ACM/IEEE Computer Society.

[109] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value locality and load value prediction. In *ASPLOS*, pages 138–147. ACM Press.

[110] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. Platypus: Software-based power side-channel attacks on x86. In *S&P*, 2021.

[111] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium*, pages 973–990. USENIX Association, 2018.

[112] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography. *ACM Comput. Surv.*, 2021.

[113] CodeMagic LTD. Wokwi: World's most advanced esp32 simulator, 2019. https://wokwi.com/.

[114] Giorgi Maisuradze and Christian Rossow. Ret2spec: Speculative execution using return stack buffers. In *CCS*, pages 2109–2122. ACM.

[115] Stefan Mangard, Thomas Popp, and Berndt M. Gammel. Side-channel leakage of masked CMOS gates. In Alfred Menezes, editor, *Topics in Cryptology - CT-RSA 2005*, volume 3376 of *LNCS*, pages 351–365. Springer, 2005.

[116] Ross McIlroy, Jaroslav Sevcík, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. abs/1902.05178.

[117] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. CopyCat: Controlled Instruction-Level attacks on enclaves. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[118] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *International Conference on Information Security and Cryptology*, pages 156–168. Springer, 2005.

[119] Maria Chiara Molteni, Jürgen Pulkus, and Vittorio Zaccaria. On robust strong-non-interferent low-latency multiplications. *IET Information Security*, 16(2):127–132, November 2021.

[120] Kazuki Monta, Makoto Nagata, Josep Balasch, and Ingrid Verbauwhede. On the unpredictability of spice simulations for side-channel leakage verification of masked cryptographic circuits. In *Proceedings of the 60th Annual ACM/IEEE Design Automation Conference*, DAC '23, page 1–6. IEEE Press, 2025.

[121] Thorben Moos, Felix Wegener, and Amir Moradi. Dl-la: Deep learning leakage assessment: A modern roadmap for sca evaluations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(3):552–598, Jul. 2021.

[122] Amir Moradi and Tobias Schneider. Side-channel analysis protection and low-latency in action - - case study of PRINCE and midori -. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 517–547, 2016.

[123] Nicolai Müller and Amir Moradi. PROLEAD A probing-based hardware leakage detection tool. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4):311–348, 2022.

[124] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *S&P*, 2020.

[125] Rishub Nagpal, Barbara Gigerl, Robert Primas, and Stefan Mangard. Riding the waves towards generic single-cycle masking in hardware. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4):693–717, 2022.

[126] NANGATE. The nangate 45nm open cell library. https://www.nangate.com.

[127] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *Information and Communications Security - ICICS 2006*, volume 4307 of *LNCS*, pages 529–545. Springer, 2006.

[128] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *Information and Communications Security*, pages 529–545, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[129] Hardware Design Group: Institute of Applied Information Processing and Austria Communications, Graz. ascon-hardware, 2023. https://github.com/ascon/ascon-hardware.

[130] Hardware Design Group: Institute of Applied Information Processing and Austria Communications, Graz. ascon-hardware-sca, 2023. https://github.com/ascon/ascon-hardware-sca.

[131] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In *Topics in Cryptology – CT-RSA*, 2006.

[132] Emmanuel Pescosta, Georg Weissenbacher, and Florian Zuleger. Bounded model checking of speculative non-interference. In *ICCAD*, pages 1–9. IEEE.

[133] Frank Piessens. Security across abstraction layers: old and new examples. In *EuroS&PW*, 2020.

[134] Josh Poimboeuf. [PATCH v2 0/4] Static calls [LWN.net]. https://lwn.net/ml/linux-kernel/cover.1543200841.git.jpoimboe@redhat.com/.

[135] Sepideh Pouyanrad, Jan Tobias Mühlberg, and Wouter Joosen. SCF-MSP: static detection of side channels in MSP430 programs. In *15th International Conference on Availability, Reliability and Security (ARES)*, pages 1–10, 2020.

[136] Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan Capkun. Frontal attack: Leaking Control-Flow in SGX via the CPU frontend. In *USENIX Security*, 2021.

[137] Zhenxiao Qi, Qian Feng, Yueqiang Cheng, Mengjia Yan, Peng Li, Heng Yin, and Tao Wei. SpecTaint: Speculative Taint Analysis for Discovering Spectre Gadgets. In *NDSS*. The Internet Society.

[138] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In *Smart Card Programming and Security*, 2001.

[139] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): measures and counter-measures for smart cards. In Isabelle Attali and Thomas P. Jensen, editors, *Conference on Research in Smart Cards - E-smart 2001*, volume 2140 of *LNCS*, pages 200–210. Springer, 2001.

[140] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *USENIX Security Symposium*, pages 1451–1468. USENIX Association.

[141] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital Side-Channels through obfuscated execution. In *24th USENIX Security Symposium (USENIX Security 15)*, August 2015.

[142] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M Tullsen, and Ashish Venkat. I see dead μops: Leaking secrets via intel/amd micro-op caches. page 14.

[143] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating masking schemes. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015*, volume 9215 of *Lecture Notes in Computer Science*, pages 764–783. Springer, 2015.

[144] Rajat Sadhukhan, Paulson Mathew, Debapriya Basu Roy, and Debdeep Mukhopadhyay. Count your toggles: a new leakage model for pre-silicon power analysis of crypto designs. *J. Electron. Test.*, 35(5):605–619, October 2019.

[145] Gururaj Saileshwar and Moinuddin K. Qureshi. CleanupSpec: An ”Undo” approach to safe speculation. In *MICRO*, pages 73–86. ACM.

[146] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. Efficient invisible speculative execution through selective delay and value prediction. In *ISCA*, pages 723–735. ACM.

[147] Pascal Sasdrich, Begül Bilgin, Michael Hutter, and Mark E. Marson. Low-latency hardware masking with application to AES. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(2):300–326, 2020.

[148] Sneh Saurabh. *RTL to GDS Implementation Flow*, page 69–82. Cambridge University Press, 2023.

[149] Tobias Schneider and Amir Moradi. Leakage assessment methodology - a clear roadmap for side-channel evaluations. Cryptology ePrint Archive, Paper 2015/207, 2015.

[150] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. ConTExT: A generic approach for mitigating spectre. In *NDSS*. The Internet Society.

[151] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read arbitrary memory over network. In *ESORICS (1)*, volume 11735 of *Lecture Notes in Computer Science*, pages 279–299. Springer.

[152] Laurent Simon, David Chisnall, and Ross Anderson. What you get is what you c: Controlling side effects in mainstream c compilers. In *EuroS&P*, 2018.

[153] Luigi Soares, Michael Canesche, and Fernando Magno Quintão Pereira. Side-channel elimination via partial control-flow linearization. *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 2023.

[154] Mohammadkazem Taram, Ashish Venkat, and Dean M. Tullsen. Context-sensitive fencing: Securing speculative execution via microcode customization. In *ASPLOS*, pages 395–410. ACM.

[155] Jan Philipp Thoma, Jakob Feldtkeller, Markus Krausz, Tim Güneysu, and Daniel J. Bernstein. BasicBlocker: ISA redesign to make spectre-immune CPUs faster. In *RAID*, pages 103–118. ACM.

[156] Elena Trichina. Combinational Logic Design for AES SubByte Transformation on Masked Data. http://eprint.iacr.org/2003/236, 2003.

[157] Paul Turner. Retpoline: A software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886.

[158] Dilip Kumar S. V., Siemen Dhooghe, Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. Time sharing - A novel approach to low-latency masking. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(3):249–272, 2024.

[159] Dilip Kumar S. V., Siemen Dhooghe, Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. Higher-order time sharing masking. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2025(2):235–267, 2025.

[160] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 178–195. ACM, 2018.

[161] Daan Vanoverloop, Hans Winderix, Lesly-Ann Daniel, and Frank Piessens. Compiler support for control-flow linearization using architectural mimicry. 2024.

[162] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kici, Ranjit Jhala, Dean M. Tullsen, and Deian Stefan. Automatically eliminating speculative leaks from cryptographic code with blade. 5:1–30.

[163] Matt Venn. From idea to chip design in minutes!, 2022. https://tinytapeout.com/.

[164] Matt Venn. Tiny tapeout 2, 2022. https://tinytapeout.com/runs/tt02/.

[165] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. KLEESpectre: Detecting information leakage through speculative cache attacks via symbolic execution. 29(3):14:1–14:31.

[166] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. Oo7: Low-overhead Defense against Spectre Attacks via Program Analysis.

[167] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. NDA: Preventing speculative execution attacks at their source. In *MICRO*, pages 572–586. ACM.

[168] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MicroWalk: A framework for finding side channels in binaries. In *ACSAC*, pages 161–173. ACM.

[169] Jan Wichelmann, Florian Sieck, Anna Pätschke, and Thomas Eisenbarth. Microwalk-CI: Practical side-channel analysis for JavaScript applications. abs/2208.14942.

[170] Hans Winderix, Marton Bognar, Lesly-Ann Daniel, and Frank Piessens. Libra: Architectural Support For Principled, Secure And Efficient Balanced Execution On High- End Processors. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2024.

[171] Hans Winderix, Marton Bognar, Job Noorman, Lesly-Ann Daniel, and Frank Piessens. Architectural mimicry: Innovative instructions to efficiently address control-flow leakage in data-oblivious programs. In *S&P*, 2024.

[172] Hans Winderix, Jan Tobias Mühlberg, and Frank Piessens. Compiler-assisted hardening of embedded software against interrupt latency side-channel attacks. In *EuroS&P*, 2021.

[173] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. Eliminating timing side-channel leaks using program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 15–26, 2018.

[174] Meng Wu and Chao Wang. Abstract interpretation under speculative execution. In *PLDI*, pages 802–815. ACM.

[175] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. InvisiSpec: Making speculative execution invisible in the cache hierarchy. In *MICRO*, pages 428–441. IEEE Computer Society.

[176] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, 2014.

[177] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. Data oblivious ISA extensions for side channel-resistant and high performance computing. In *NDSS*, 2019.

[178] Jiyong Yu, Trent Jaeger, and Christopher Wardlaw Fletcher. All your pc are belong to us: Exploiting non-control-transfer instruction btb updates for dynamic pc extraction. In *ISCA*, 2023.

[179] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. Speculative taint tracking (STT): A comprehensive protection for speculatively accessed data. In *MICRO*, pages 954–968. ACM.

[180] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *CCS*, pages 1789–1806. ACM.

[181] Danilo Šijačić, Josep Balasch, Bohan Yang, Santosh Ghosh, and Ingrid Verbauwhede. Towards efficient and automated side channel evaluations at design time. In Lejla Batina, Ulrich Kühne, and Nele Mentens, editors, *PROOFS 2018. 7th International Workshop on Security Proofs for Embedded Systems*, volume 7 of *Kalpa Publications in Computing*, pages 16–31. EasyChair, 2018.