Verifiable Side-Channel Security

Vom Promotionsausschuss der
Technischen Universität Hamburg

zur Erlangung des akademischen Grades

Doktor-Ingenieur (Dr.-Ing.)

genehmigte Dissertation (Monographie)

von

Marc Olivier Gourjon

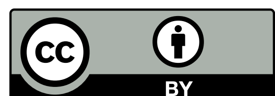
aus
Hamburg

2024

Chair of Examination Board:	Prof. Dr.-Ing. Gerhard Bauch
1st Examiner:	Prof. Dr. techn. Dieter Gollmann
2nd Examiner:	Prof. Dr.-Ing. Thomas Eisenbarth
Date of Examination:	Thursday 16th May, 2024

DOI: https://doi.org/10.15480/882.13604

This work is licensed under a Creative Commons "Attribution 4.0 International" license.



Abstract

The protection of cryptographic implementations against side-channel attacks is essential. Side-channel attacks exploit the secret-dependent information contained in the power consumption or electromagnetic radiation of computations on physical devices. This allows to break implementations of otherwise provably secure cryptography.

Masking is a commonly used countermeasure but manifold pitfalls render theoretically secure masked designs insecure in practice. One reason is the difficulty of implementing masked algorithms as intended instead of just functionally correct. Another one is the rich diversity of device-specific physical side-channel behavior, which mandates additional development steps to mitigate vulnerabilities arising thereof.

This thesis questions the perception that formal verification, due to its inability to adequately model side-channel behavior encountered in practice, is unsuitable for qualitative assessments of physical side-channel resilience. We improve formal methods and show that verification allows for reliable and quick assessments of the side-channel security of software in practice, scaling beyond the limits of physical evaluation.

Central to our results is our concept of explicit leakage. It enables accurate and flexible specifications of device-specific physical side-channel leakage in fine-grained, formal, and intelligible leakage models. We present a tool to verify the security of masked implementations in such models using standard notions of security as well as our extensions. The ability to reliably predict the side-channel security is evaluated by masking and verifying the PRESENT S-Box at 1st and 2nd order in a leakage model for two Arm Cortex M0+ microcontrollers. Physical evaluation indicates that verification reliably predicts practical security when based on leakage models of sufficient quality. Further, verification guides development by providing information on fixes and facilitates optimizations, resulting in implementations with greater confidence in security and lesser overhead.

The applicability to large cryptographic implementations is evaluated by masking the post-quantum cryptography key encapsulation mechanism KYBER at arbitrary security order. We implement two novel and large components, verify their side-channel security, and confirm their practical security by physical evaluation.

We address the inherent dependence of verification on the quality of the used leakage models by proposing contracts. Contracts provably model the complete side-channel leakage of every gate in a processor implementation. We present the first approach to verify completeness between contracts and processors, denoted compliance, and derive a security reduction: any software implementation verified secure in a contract is also secure w.r.t. the leakage caused by each gate of any compliant processor. Hence, contracts restore the instruction set architecture abstraction and permit secure porting of implementations to multiple processors, without demanding the disclosure or accessibility of processor implementations.

Acknowledgements

My deepest gratitude goes to my wife, Friedi, whose support was fundamental throughout this journey. You have been far more than a companion – you have been my steadfast support during every challenge we encountered. Your countless efforts were so significant that I often feel you deserve recognition equal to mine. I am endlessly grateful for your belief in me, your encouragement, and your love, all of which helped me reach this milestone. Without you, none of this would have been possible.

To Fina and Jara, my wonderful children, thank you for your sacrifices along the way. Yet you both stood by me with love and joy, reminding me of what truly matters in life.

I am sincerely grateful to my advisor, Prof. Dieter Gollmann, for giving me the invaluable freedom to carve my own path and discover my strengths and motivation, while always being a source of guidance and support. Your balance of giving space while providing consistent mentorship has shaped this doctorate into something that feels truly mine.

I am also grateful to NXP Semiconductors, who made it possible for me to pursue a doctorate despite the financial challenges of supporting a family. Working within the company provided insights that extended beyond what I could have gained in a purely academic setting. My thanks go to the managers who supported me: Johannes Berg, Georg Menges, Fabian Mackenthun, Stefan Kuipers, and Joppe Bos, as well as to Alejandro Garza for his help in managing the VE-Jupiter project. I am also deeply thankful to Marc Vauclair for his mentorship, but also for his enthusiasm and drive in helping me to transfer the results of this thesis to industrial practice.

Finally, I thank my co-authors, collaborators, friends, family, colleagues, peers, and supporters for making this research journey as enjoyable as it was enlightening. Our discussions – both technical and personal – shaped not only the results we created but also deepened my understanding of science and its human dimensions.

Hamburg, 2024 Marc Gourjon

Contents

Abstract iii

Acknowledgements v

List of Abbreviations xi

List of Algorithms xv

List of Figures xvii

List of Listings xix

List of Tables xxi

List of Definitions and Theorems xxiii

1. Introduction 1
- 1.1. Towards Security in the Presence of Side-Channel 1
- 1.2. Thesis Statement 6
- 1.3. Contributions and Outline 7
  - 1.3.1. Chapter 2 – Side-Channel Resilience 7
  - 1.3.2. Chapter 3 – Verification in Fine-Grained Leakage Models 7
  - 1.3.3. Chapter 4 – Masking Kyber: First- and Higher-Order Implementations 8
  - 1.3.4. Chapter 5 – Verifiable Hardware-Software Contracts . . 10
  - 1.3.5. Chapter 6 – Conclusion 11
- 1.4. Authorship 11

2. Side-Channel Resilience 15
- 2.1. Notation and Preliminaries 15
- 2.2. Physical Side-Channels and Attacks 16
- 2.3. A Framework for Computation in the Presence of Side-Channel 19
  - 2.3.1. Adversarial Models and Provable Resilience 22
  - 2.3.2. Overview on Computation Models 25

2.4. The Masking Countermeasure 27
2.4.1. Concept 27
2.4.2. Encoding Secrets 29
2.4.3. Gadgets and Probing Security 31
2.4.4. Secure Composition and Non-Interference 36
2.5. Verification of Physical Side-Channel Security 40
2.5.1. Leakage Models and Hardened Masking 41
2.5.2. Contracts and Hardware Models 43
2.5.3. Horizontal Resilience 46

3. Verification in Fine-Grained Models 49
3.1. Expressing Semantic and Leakage 49
3.1.1. A Domain Specific Language with Explicit Leakage . . 50
3.1.2. Modeling Execution of Implementations 52
3.1.3. Modeling Instruction Semantics 53
3.1.4. Modeling Leakage 55
3.2. Stateful Strong-Non-interference and Automated Verification . 58
3.2.1. Security Definitions 58
3.2.2. Automated Verification 61
3.2.3. Implementation 65
3.3. Representative Proofs of Efficient Masking 65
3.3.1. Optimized Hardened Masking 66
3.3.2. Case Study: Masking the Present S-Box 67
3.3.3. Physical Resilience of Verified Implementations 70
3.4. Discussion 71

4. Masking Kyber: First- and Higher-Order Implementations 75
4.1. Background 76
4.2. Masking Kyber at Arbitrary Order 78
4.2.1. Higher-Order Masked One-Bit Compression 78
4.2.2. Higher-Order Masked Comparison 85
4.2.3. Higher-Order Masked Decapsulation 89
4.3. Implementation and Evaluation 92
4.3.1. Performance Comparison 93
4.3.2. Verification of Side-Channel Resilience 96
4.3.3. Physical Leakage Assessment 102
4.4. Discussion 103

5. Verifiable Hardware-Software Contracts 107
5.1. Preliminaries 108

5.2. Hardware-Software Contracts 109
5.2.1. Expressing Contracts in Genoa 109
5.2.2. Contract Formalization 111
5.2.3. Hardware Compliance With a Contract 113
5.2.4. End-to-end security 115
5.3. Verifying Hardware Compliance 117
5.3.1. Concept 117
5.3.2. Verifying Model Completeness 119
5.3.3. Implementation 121
5.4. Case Study: A Compliant Contract for Ibex 121
5.4.1. Configuration for Verifying Compliance 122
5.4.2. Complete Power Contract for Ibex 123
5.4.3. Validation of E2E Security 126
5.5. Discussion 127

6. Conclusion and Outlook 131

A. Supporting Material for Section 3 135
A.1. Listings and Algorithms 135

B. Supporting Material for Section 4 139
B.1. Parameters and Algorithms of Kyber 139
B.2. Proof of Theorem 4.1 142
B.3. Proof of Theorem 4.3 143
B.4. Fine-Grained Leakage Model for Arm Cortex M0+ 143

C. Supporting Material for Section 5 153
C.1. Proof of Theorem 5.2 153
C.2. Compliant Contract for the Ibex RISC-V Processor 154
C.3. Ibex Configuration 165

Bibliography 173

List of Abbreviations

A2B arithmetic to Boolean

ADC analog-to-digital converter

ALU arithmetic-logic unit

BB black-box

BNF Backus-Naur form

BSI Federal Office for Information Security

CM0+ Arm Cortex M0+

CM4F Arm Cortex M4F

CMOS complementary metal-oxide-semiconductor

CPA correlation power analysis

DAG directed acyclic graph

DPA differential power analysis

DSL domain specific language

E2E end-to-end

ECC elliptic curve cryptography

EM electromagnetic

FA fault attack

FO Fujisaki-Okamoto

iff if and only if

i.i.d. independent and identically distributed

IND-CPA indistinguishability under chosen-plaintext attack

IND-CCA2 indistinguishability under adaptive chosen ciphertext attack

IOS input output separation

ISA instruction set architecture

ISW Ishai, Sahai and Wagner

LSU load-store unit

KEM key encapsulation mechanism

LUT look-up table

MCU microcontroller

MI mutual information

MIA mutual information analysis

MSB most significant bit

HW Hamming weight

HD Hamming distance

M-LWE module learning-with-errors

R-LWE ring learning-with-errors

MPC multi-party computation

NIST National Institute of Standards and Technology

NTT number-theoretic transform

OCL only computation leaks

OTP one-time pad

RPA random plaintext attacks

RPM robust probing model

PC program counter

PKE public-key encryption

POI point of interest

PQC post-quantum cryptography

SCA side-channel analysis

SD statistical distance

SNR signal-to-noise ratio

SASCA soft analytical side-channel attack

SPA simple power analysis

SRAM static random-access memory

TVLA test vector leakage assessment

TRNG true random number generator

XOR exclusive disjunction

WB write-back

TI threshold implementation

NI non-interference

SNI strong-non-interference

MIMO-SNI multiple-input-multiple-output-strong-non-interference

IL intermediate language

List of Algorithms

2.1. maskedXor gadget. 32
2.2. maskedAnd gadget. 35

3.1. 2nd-order Stateful t–SNI maskedRefresh. 67

4.1. t–SNI Compress$_q$ gadget for Kyber. 80
4.2. t–SNI DecompressedComparison gadget for Kyber. 87

A.1. 2nd-order Stateful t–SNI maskedAnd. 137
A.2. Optimized composition of linear function $d = a \oplus b \oplus c$. 138

B.1. Kyber.CCAKEM.Dec(c, sk): decapsulation. 140
B.2. Kyber.CPAPKE.Enc(pk, m, r): encryption. 141
B.3. Kyber.CPAPKE.Dec(sk, c): decryption. 141

List of Figures

1.1. Development process relying on physical evaluation. 4
1.2. Secure development process combining verification and evaluation. 5

2.1. Concept of side-channel attacks. 17
2.2. Side-channel analysis as information-theoretical communication channel. 22
2.3. Overview of existing and contributed models of leakage. 26
2.4. Comparison of reference and masked circuit side-by-side. 29
2.5. Example of an insecure composition of gadgets. 37

3.1. Syntax of the IL domain specific language. 51
3.2. Block diagram of exemplary processor datapath. 57
3.3. Rules for partial evaluation of IL syntax. 64
3.4. Examples for linear and non-linear gadget composition. 68
3.5. Non-linear layer of PRESENT S-Box. 68
3.6. 1st-order TVLA results of the optimized PRESENT S-Box on CM0+. 71
3.7. Bivariate TVLA results of the optimized PRESENT S-Box on CM0+. 72

4.1. Overview of the decapsulation in KYBER. 78
4.2. Gadget structure of Decode for Theorem 4.1. 83
4.3. Gadget structure of DecompressedComparison for Theorem 4.2. 89
4.4. Structure of the masked KYBER.CCAKEM.Dec for Theorem 4.3. 91
4.5. Univariate 1st and 2nd-order TVLA results of $\mathsf{Compress}_q$. 105
4.6. Univariate 1st and 2nd-order TVLA results of Decode. 106

5.1. Big-steps compliance between contract and hardware. 114
5.2. Instruction lockstep execution for verifying compliance. 119
5.3. Parallel lockstep execution for verifying modeling functions. . . 120

List of Listings

3.1. Example line of assembler code. 53
3.2. Resulting IL code. 53
3.3. IL model of addition with carry and of `adds` instruction. 54
3.4. Assembler code with leakage across instructions. 56
3.5. Leakage model of `adds` instruction. 57
3.6. Excerpt of the simplified IL model $[\![\cdot]\!]^{\text{CM0+}}$ for CM0+. 59
3.7. Execution trace of insecure stateful composition. 60

4.1. Look-up table based arithmetic to Boolean conversion. 94
4.2. Detected leakage behavior during implementation of KYBER. . 100
4.3. IL model of the arithmetic to Boolean conversion in Listing 4.1. 101

5.1. GENOA model of IBEX state. 110
5.2. GENOA model of the step function in IBEX. 110
5.3. GENOA model of RISC-V R-type instructions. 111
5.4. GENOA model of common leakage behavior in IBEX model. . . 123

A.1. License of SCVERIF, presented gadgets and code snippets. . . . 135

B.1. IL model of $[\![\cdot]\!]^{\text{CM0+}}$ used in KYBER. 143

C.1. GENOA model of IBEX contract. 154
C.2. Configuration for verification of hardware-compliance. 165

List of Tables

3.1. Performance of reference and optimized PRESENT S-Box. . . . 69
3.2. Performance impact of hardening in the PRESENT S-Box. . . . 70

4.1. Comparison of gadget configurations in KYBER.CCAKEM.Dec. . 95
4.2. Detailed performance results of masked KYBER.CCAKEM.Dec. 97
4.3. Detailed performance results of masked KYBER.CCAKEM.Dec. 98

5.1. Results of validating end-to-end security. 126

B.1. Recommended parameter sets for KYBER. 139

List of Definitions and Theorems

2.1. Definition (Computation Model $[\![\mathsf{C}]\!]^m$) 20
2.2. Definition (Adversary Model Adv_r^i interacting with $[\![\mathsf{C}]\!]^m$) . . . 23
2.3. Definition (Simulator) 24
2.4. Definition (Gadget G and Computation Model $[\![\mathsf{G}]\!]^{\text{GAD}}$) 31
2.5. Definition (ISW Probing Model $[\![\mathsf{G}]\!]^{\text{ISW}}$) 33
2.6. Definition (t^{th}-order Probing Security of G in $[\![\mathsf{G}]\!]^m$) 33
2.7. Definition (t–NI of G in $[\![\mathsf{G}]\!]^m$) 38
2.8. Definition (t–SNI of G in $[\![\mathsf{G}]\!]^m$) 39
2.9. Definition (Completeness of $[\![\mathsf{C}]\!]^m$ for $[\![\mathsf{C}]\!]^{\text{PHY}}$) 42
2.10. Definition (Hardware Computation Model $[\![(\mathsf{H}, \mathsf{P})]\!]^h$) 45
2.11. Definition (Robust Probing Model $[\![(\mathsf{H}, \mathsf{P})]\!]^{\text{RPM}}$) 46

3.1. Definition (Transition Leakage Effect) 55
3.2. Definition (Revenant Leakage Effect) 56
3.3. Definition (Stateful t–$(\mathsf{S})\mathsf{NI}$ of P in $[\![\mathsf{P}]\!]^m$ under $\pi^{\text{I}}, \pi^{\text{O}}$) 61
3.1. Proposition (Correct Transformation T) 64

4.1. Theorem (t–SNI of Algorithm 4.1) 82
4.2. Theorem (t–SNI of Algorithm 4.2) 89
4.3. Theorem (t–SNI of Decapsulation $\mathsf{G}_{\mathsf{Dec}}$) 91

5.1. Definition (Modeling Function) 108
5.1. Lemma (Modeling Functions Simulate) 108
5.2. Definition (Similar States $\sigma^h \simeq_{\mathcal{V}} \sigma^c$) 113
5.3. Definition (Compliance $[\![\cdot]\!]^h \vdash_{\mathcal{V}} [\![\cdot]\!]^c$) 114
5.4. Definition (Similar Policies $\pi^h \triangleq_{\mathcal{V}} \pi^c$) 115
5.1. Theorem (Model Reduction) 116
5.1. Corollary (Mixed Observations) 117
5.2. Corollary (End-to-end Security) 117
5.2. Theorem (Existence of Modeling Functions) 119

1. Introduction

The need for security and secure connections has become ubiquitous. Growing rates of increasingly private data need to be transmitted securely in diverse and rapidly changing applications. In addition, the applications themselves require secure updates of firmware functionality over the air. Such services rely on cryptography for confidentiality, authenticity, or integrity assurances. However, the strong cryptographic assurances rely on model assumptions which are hard to preserve in settings that allow for physical side-channel attacks.

A preeminent example is the black-box (BB) assumption of many cryptographic security proofs. In BB security proofs the modeled adversary cannot observe intermediate computation results of the internal computation steps performed in a cryptographic algorithm. Any security statement constructed in such a setting depends on this assumption. In reality, however, there is data-dependent information leakage during computation on physical devices. By launching side-channel attacks and performing side-channel analysis (SCA) intermediate computation results can be recovered from covert information. This does not only render the theoretical security claims questionable but allows for practical attacks reconstructing, for example, cryptographic keys to eavesdrop or tamper with otherwise secure connections in practice. Moreover, diverse kinds of side-channel exist which result in a large and rarely well-understood attack vector, especially for software.

Further research on understanding physical side-channel leakage and protecting against SCA is needed to approach the vision of ubiquitous security in the presence of side-channel attacks.

1.1. Towards Security in the Presence of Side-Channel

We motivate this work by discussing problems hindering the broad adoption of countermeasures, especially in the case of software.

Side-channel attacks have been known for more than two decades and countermeasures almost as long [Koc96; KJJ99; Cha+99; ISW03]. Physical devices such as processors or integrated circuits emit information on interme-

diate computation steps in various forms such as execution time, power consumption, or electromagnetic (EM) radiations. Side-channel attacks exploit the information leaked through these channels. They are applicable across a wide range of implementations and increasingly automated [Uen+22; BCS21]. Countermeasures make it harder in terms of effort and likeliness for attackers to succeed but need to be applied carefully to avoid the manifold pitfalls that render protected implementations vulnerable [Cor+14; BCS21; Bec+22; Azo+22b]. The importance of SCA resilience is highlighted by the fact that the side-channel resilience of new ciphers is nowadays considered already during standardization processes, e.g., the National Institute of Standards and Technology (NIST) post-quantum cryptography (PQC) standardization process [Nat17]. Furthermore, an increasing number of ciphers need to be protected since national authorities promote different ciphers, e.g., the German Federal Office for Information Security (BSI) recommending different PQC secure key encapsulation mechanisms (KEMs) than NIST [Fed23].

Masking is a widely deployed and accepted countermeasure to thwart power and EM side-channels, which is the focus of this work. It consists of splitting sensitive data into multiple randomized pieces, denoted *shares*, and computing on the shares instead of the sensitive secrets in such a way that the side-channel reveals little to no information about the sensitive data. However, the solution introduces multiple new design and development problems.

Designing securely masked algorithms, denoted *gadgets*, requires operating on the shares as efficiently as possible but without combining the shares accidentally. This is error-prone since the number of potentially vulnerable combinations increases exponentially with the size of the application to be masked. Further, to achieve higher protection levels the number of shares must be increased, which increases the failure risk during design and development too. Academic works approach this problem by providing pen-and-paper proofs of the security of single gadgets in the abstract *probing model* as well as systematic proofs of compositions of gadgets [ISW03]. However, the necessary expertise might not be available, and human error can lead to incorrect proofs, e.g., [RP10; Cor+14]. Software tools, such as the verification tool MaskVerif, support the principled construction by verifying strong formal security notions of algorithmic designs [Bar+15; Bar+19].

However, a large gap towards SCA resilience in practice is caused by the difficulty of preserving the security of designs during implementation. Theoretically secure designs frequently result in vulnerable implementations [PV17; Bal+14; GPM21]. One reason is the difficulty of adhering to the algorithmic design during implementation, *i.e.*, implementing it as intended instead of just functionally correct. Software compilers cannot be used as they often break security by optimizing and reordering computation steps which are necessary

for security but inefficient from a purely functional viewpoint [DDT20]. Consequently, developers have to write low-level assembly code while adhering to the intentions behind the algorithmic design, which remains error-prone.

The other reason is that physical side-channel behavior is more diverse than what is considered in the probing model and further security *hardening* is needed to mitigate the additional leakages [Bal+14; Cor+12; Fau+18]. In the case of hardware implementations, the physical leakage behavior of complementary metal-oxide-semiconductor (CMOS) gates is well understood and the universal robust probing model (RPM) permits to mitigate the additional leakages systematically and efficiently [Fau+18; MPG05; MPO05; De +17; NRS11]. In the case of software, the leakage is caused by the gates of the processor implementation executing the software, leading to the fact that the same software can leak differently on two dissimilar processors implementing the same instruction set architecture (ISA) [PV17; MOW17; Gou22b; Gou22a; MPW22]. An emerging insight is that for software there is no universal leakage model to securely harden implementations for all processors, instead there is *device-specific* leakage [MPW22; Bar+21b]. This is in stark contrast to the RPM for hardware implementations and severely hinders the development and portability of hardened implementations.

Worse, for no processor a complete or dependable model has been published prior to the co-authored works (Section 1.4) presented in the following. Entire masking schemes have been considered futile due to physical leakage [Gao+19]. The lack of dependable leakage models results in an unpredictable trial-and-error development process, where development and evaluation form an iterative feedback loop. Optimization is hindered by the high cost per iteration which sometimes result in the deployment of inefficient designs on purpose, e.g., [Fri+22; Bal+14; Bel+23b].

Maintaining released applications suffers from similar problems as every modification may break security and require costly re-evaluation. Similarly, the re-use of deployed code in the development of new applications or its porting to other target devices requires re-evaluation against a possibly distinct leakage.

Overcoming these restrictions requires a better understanding of the device-specific leakage and automated software tools continuously providing guidance on the effective security during development.

Physical assessment is one of the most common approaches for the evaluation of side-channel resilience. It is performed after development to confirm the effective security prior release or to detect security bugs that need to be fixed in another development cycle, visualized in Figure 1.1. Security is assessed by conducting attacks, leakage analysis, or statistical evaluation on real side-channel measurements taken from a physical device executing the assessed

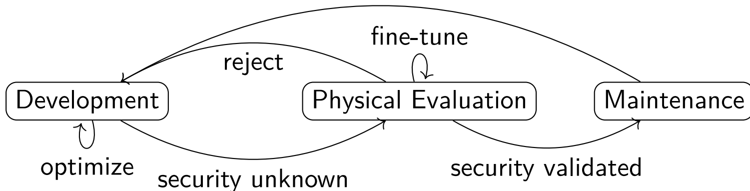


Figure 1.1.: Interaction of physical evaluation within the development process.

implementation [Azo+20; SM15; DSM17; DSV14; DFS15]. Any assurance based on physical evaluation is limited by the quality of the measurements and the scope of the performed analyses. Physical assessment is most beneficial when used to reassure the security of implementations considered secure, e.g., ruling out specific threats by conducting respective attacks.

Emulation-based assessment circumvents the downsides of physical assessment caused by the acquisition of physical measurements. Here, physical side-channel measurements are emulated using generic or device-specific models capturing, for example, the power consumption [Ves14; MOW17; CGD18; GOP22; Baz+21]. Based on the emulated measurements the same analyses can be performed, still introducing development delay due to the computational complexity of emulation & the analyses, and still mandating expert knowledge for fine-tuning analyses. In comparison to physical attacks, the models used for emulation introduce a gap that may result in false claims, either incorrect identification of vulnerabilities or incorrect claims of resilience.

For physical as well as emulated assessment it remains difficult to decide at which point to stop the evaluation by declaring an implementation secure, or whether to extend the scope of the evaluation. In addition, assessment and debugging effort become problematic for large implementations which is in contrast to the inherent need for higher protection levels for post-quantum secure KEMs [Azo+22c; Uen+22; Azo+22a]. We emphasize the need to scale towards large applications, e.g., the PQC KEM Kyber, to be able to protect them. Profound physical or emulation-based assessment of complete implementations remains costly and therefore undesirable to be repeated. Upcoming standards continue to increase the application size, e.g., hybrid PQC combining schemes like Kyber with classic cryptography like elliptic curve cryptography (ECC) [Bar+22]. These forms of assessment are sub-optimal for iterative development due to the high delay, effort, and needed expert knowledge to acquire & analyze traces profoundly.

Formal verification addresses these weaknesses by exhaustively guaranteeing

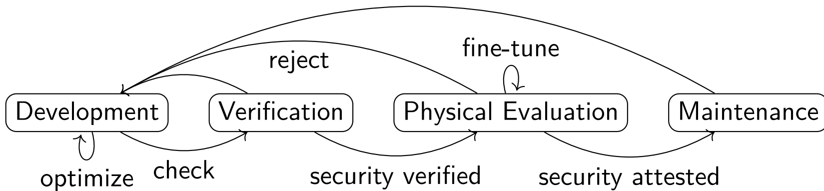


Figure 1.2.: Interaction between development process, pre-silicon and physical evaluation.

resilience against attacks in an automated manner. Here, both attacks as well as side-channel behavior are abstracted into formal models of adversaries and formal leakage models. Security properties are checked against these models using mathematical techniques, which in turn allow proving resilience exhaustively under the assumptions of the model. For the masking countermeasure multiple approaches have been published with reasonable speed for small to medium-sized masked designs [Bay+13; EWS14; Zha+18; Blo+18; Bar+15; Cor18; Bar+19; KSM20; Gig+21; Bel+22]. The model-based nature permits pinpointing the cause of security violations, significantly aiding the subsequent debugging and patching.

However, formal verification for side-channel security is considered unsuitable for achieving practical security due to the inaccuracy of the considered leakage models. Prior to some of our co-authored works, verification tools committed to one or few fixed leakages models and consequently neglected device-specific leakage. But this leakage can drastically reduce or even void the security of the verified countermeasure and renders the benefits of verification questionable [Cor+12; Bal+14; MPW22; Gao+19; Fau+18; GPM21].

If this issue could be overcome then verification would allow evaluating the security continuously during iterative development, providing guidance on fixes, and offering rigorous security claims. Verification could even be considered as a gatekeeper for physical assessment; fast incremental verification allows to quickly reach an implementation considered secure and ready for costly but profound re-evaluation based on physical assessment. In this paired setting of verification and physical assessment, the likelihood of vulnerability findings during costly physical assessment would be minimized. We propose this combination as *secure development process*, depicted in Figure 1.2, as a systematic flow mitigating difficulties hindering the broad adoption of masking.

Both problems, the lack of dependable models and the lack of adequate

tooling make the application of masking difficult and prevent structured optimization & scaling toward larger applications with physical resilience. The resulting inefficiently masked implementations suffer from further degraded practical SCA security since the added instructions increase the exploitable leakage and thereby promote side-channel attacks exploiting many leakages.

We conclude that more widespread adoption of side-channel protections is hindered by the above-mentioned problems. Vice-versa, the problems can be overcome by reliable and fast means to predict side-channel resilience during the development & maintenance of software implementations, especially for large applications.

1.2. Thesis Statement

This work questions the perception that formal verification, due to its inability to adequately model side-channel behavior encountered in practice, is unsuitable for qualitative assessments of physical side-channel resilience. The contrapositive is assumed; formal methods can be extended to capture the physical side-channel phenomena of different processors accurately. Consequently, verification provides the means to reliably and quickly predict the side-channel security of software, overcoming the limits of physical evaluation.

Specifically, two related research questions are investigated:

Q1: Is it possible to use verification techniques to accurately and reliably predict the physical side-channel resilience of masked software?

Q2: Is it possible to ensure the completeness of the leakage models used in the verification of side-channel security?

We introduce the concept of explicit leakage which enables flexible creation of fine-grained models of device-specific leakage. The concept is used in a domain specific language (DSL) for modeling side-channel leakage and associated verification techniques. The resulting tool scVerif verifies side-channel security of implementations in user-supplied leakage models. We empirically show that the verification tool reliably predicts side-channel resilience when given an accurate leakage model, answering **Q1** affirmative. This claim is supported by case studies on a masked Present S-Box and masked Kyber with physical evaluation on an Arm Cortex M0+ (CM0+) processor.

However, the validity of the verification depends on the quality of the leakage model and a systematic approach for constructing *complete* leakage models for other processors is missing. Hardware-software contracts are proposed as

a solution to **Q2**. Contracts are specifications of leakage behavior that can be verified against both hardware and software implementations. We propose verification techniques and an associated tool for proving hardware compliance, *i.e.*, that a contract completely models all leakage potentially caused by the CMOS gates of a processor executing software. Further, we introduce end-to-end (E2E) security, a proven security reduction assuring that any software implementation that is compliant with a contract provably enjoys the same security on any processor that is compliant with that contract too. The methodology is evaluated by applying the tool to the RISC-V processor Ibex and using an independent tool to confirm E2E security on selected case studies.

The qualitative answers to both questions yield the demanded means for unblocking, or at least easing, the protection of software against side-channel attacks. We evaluate the impact and manifold benefits of augmenting software development processes with contracts and verification. The contributions open new avenues towards automated application of masking, e.g., [Mos+12; Abr+21] and automated repair [She+21b; She+21a], as well as secure compilation, which is out of scope of this work.

Eventually, a broader application of the masking countermeasure is fostered and thereby more prevalent security against side-channel attacks.

1.3. Contributions and Outline

A more detailed overview of the contributions by chapter is given.

1.3.1. Chapter 2 – Side-Channel Resilience

Chapter 2 classifies the contributions of this thesis in the respective context for which the necessary background is given. The text is an independent creation.

1.3.2. Chapter 3 – Verification in Fine-Grained Leakage Models

In Chapter 3, we illustrate that automated verification can deliver provably resilient and practically hardened masked implementations with low overhead. The chapter is based on the paper “Masking in Fine-Grained Leakage Models: Construction, Implementation and Verification” by Barthe, Gourjon, Grégoire, Orlt, Paglialonga, and Porth [Bar+21b], presented by Marc Gourjon at the CHES conference in 2021. In addition, the open-source tool scVerif is

presented which has been published as a separate, peer-reviewed artifact at CHES [Bar+21a].

We present a technique for automated verification that flexibly incorporates real side-channel behavior encountered in practice and thereby ensures that security statements provably ensure resilience against the defined device-specific leakage. We introduce a DSL, named IL, which makes leakage explicit to allow expressing fine-grained models of physical side-channel leakage. These formal models accurately describe which side-channel information can be observed during the execution of a program in a human and machine-readable manner. IL is more flexible than prior approaches as it separates the specification of implementation semantics from leakage annotations with a dedicated language construct; the explicit `leak` statement. Next, we develop an approach to automated verification leveraging program analysis and abstract interpretation to compute the leakages exposed during the execution of a program and check whether these leakages adhere to standard notions of probing security, as well as our stateful extensions for execution on real processors.

The approach is evaluated by manually describing the physical leakage of a microcontroller (MCU) in a fine-grained leakage model and applying the verification tool on multiple use-cases ranging from small gadgets to an entire PRESENT S-Box masked at 1st and 2nd order which exhibit no detectable leakage beyond one million physical measurements in test vector leakage assessment (TVLA). We describe generic strategies for optimized leakage mitigation and design. The resulting improvements are evaluated in terms of performance and practical security using physical measurements. The results indicate that fine-grained verification does not only improve the development speed to achieve practical security but also allows to improve the performance of implementations significantly.

In this chapter, we assume that the leakage behavior of the processor foreseen to execute software is sufficiently known to specify an accurate leakage model. This can be done for example using physical characterization. In Chapter 5 we present an alternative approach to the systematic construction of reliable leakage models which avoids the question of whether the model used during verification of software is of sufficient quality, *i.e.*, models all physical leakage.

1.3.3. Chapter 4 – Masking Kyber: First- and Higher-Order Implementations

In Chapter 4 we achieve a complete first- and higher-order masking of the PQC KEM KYBER. The chapter is based on the paper “Masking Kyber: First- and Higher-Order Implementations” by Bos, Gourjon, Renes, Schneider, and van

Vredendaal, presented by Marc Gourjon at the virtual CHES conference 2021 and is the first published higher-order masked implementation of a PQC KEM.

We emphasize the need for higher-order masking; PQC KEMs relying on the Fujisaki-Okamoto (FO) transformation are inherently more susceptible to horizontal side-channel attacks and require higher protection levels [Azo+22c; Uen+22; Azo+22a]. KYBER relies on polynomial arithmetic with prime modulus q as well as compression of ciphertext and message (*i.e.*, the exchanged key) which are challenging to mask when combined. We present concrete masking for two KYBER-specific components with formal security proofs in the probing model for which no masked algorithm was proposed so far, or prior art turns out sub-optimal.

Masked One-Bit Compression We propose a new approach for compressing an arithmetically masked polynomial to a Boolean-masked bit-string. Our approach is based on a bit-sliced binary search and overcomes limitations of prior solutions which are limited to 1st-order masking or power-of-two modulus [Ode+18; Bei+20].

Masked Decompressed Comparison KYBER relies on the FO transform for achieving indistinguishability under adaptive chosen ciphertext attack (IND-CCA2) security. This involves comparing a compressed re-encryption of the decrypted message to the ciphertext. Compressing an arithmetically masked message would introduce a non-negligible overhead due to the prime modulus. Instead, we present a new approach that compares the *uncompressed* masked message to the *compressed* public ciphertext.

We implement and harden the novel modules at first-order for CM0+ with subsequent physical validation of the security order in practice, showing no detectable leakage in TVLA beyond 100 000 measurements. Development and hardening are supported by applying SCVERIF. We show that complete or dependable leakage models are not strictly necessary when verification is paired with physical assessment. Instead, we extend the fine-grained model presented in Chapter 3 to cover additional instruction and adopt it whenever leakage is detected during physical evaluation. In the course, we provide a solution for verifying look-up table (LUT) based arithmetic to Boolean (A2B) conversions with SCVERIF. In addition, we present masked and optimized implementations of the complete KYBER at orders one to three for benchmarking with the pqm4 [Kan+19b] framework on Arm Cortex M4F (CM4F).

As in the previous chapter, we point out the benefit of verifiable side-channel resilience in speeding up development, scaling to larger implementations, and optimizing the performance of hardened implementations.

1.3.4. Chapter 5 – Verifiable Hardware-Software Contracts

The quality of leakage models in terms of completeness has a crucial role in security assessments conducted through automated verification. In Chapter 5 we address this issue systematically by introducing contracts. The chapter is based on the paper "Power Contracts: Provably Complete Power Leakage Models for Processors" by Bloem, Gigerl, Gourjon, Hadzic, Mangard, and Primas, published at CCS 2022.

To begin, we introduce the DSL Genoa, which improves upon IL. Genoa goes beyond simply specifying the exposed leakage of a program instruction; it allows to specify how a processor executes and to compare between this specification and a processor netlist representing the implementation of an instruction at the gate-level. This enables comparisons between the leakage behavior specified in the Genoa model and the leakage emitted by the individual gates of a processor according to established models like the RPM. Consequently, this permits to ensure the completeness of leakage models for software (research question **Q2**) by verifying their completeness against the well-understood leakage behavior of hardware.

The Genoa model specification acts as an interface, referred to as a *contract*, between software programs and hardware implementations of processors capable of executing such programs. It defines a contract because the execution of a program on hardware is guaranteed to be side-channel secure in probing security models whenever the program has been verified to be secure against the specified leakage behavior in the contract and the processor executing the program has been checked to not leak more side-channel information than specified in the same contract. That is, these notions represent sufficient conditions for deriving E2E security; if a software implementation is proven secure against the contract model in terms of specific security notions, then it is guaranteed to execute securely at the *gate-level* of on *any* processor adhering to the contract. We introduce relational security notions between software implementations and the contract, as well as between the contract and the hardware implementation, denoted *compliance*. Security of software against a contract model can be verified using the techniques outlined in Chapter 3. We propose methods to verify compliance of a processor implementation and a contract based on SMT solvers. This provides the first approach to construct provably complete leakage models for software programs.

We implement a proof of concept tool and use it to check compliance of the Ibex processor to a contract, iteratively augmenting the contract with additional `leak` annotations. As a result, we establish a power side-channel contract that provably models all side-channel information exploitable by any adversary probing the CMOS gates of the Ibex processor. We use the result-

ing model to construct, harden, and verify higher-order gadgets against the contract. According to our proven E2E security reduction these implementations are probing secure at the gate level. We use an independent verification tool to validate this by verifying the implementation using the netlist of the Ibex processor.

1.3.5. Chapter 6 – Conclusion

Conclusion and a brief outlook are given in Chapter 6.

1.4. Authorship

A complete list of scientific publications and public presentations conducted during the doctoral studies is given below.

[Abr+21] "Automated Masking of Software Implementations on Industrial Microcontrollers" by Abromeit, Bache, Becker, Gourjon, Güneysu, Jorn, Moradi, Orlt, and Schellenberg, published at DATE 2021.

[Bar+21b] "Masking in Fine-Grained Leakage Models: Construction, Implementation and Verification" by Barthe, Gourjon, Grégoire, Orlt, Pagliaonga, and Porth, presented by Marc Gourjon at the virtual CHES conference 2021.
The open-source tool scVerif presented in this work has been published as a separate, peer-reviewed artifact at CHES [Bar+21a].

[Bos+21] "Masking Kyber: First- and Higher-Order Implementations" by Bos, Gourjon, Renes, Schneider, and van Vredendaal, presented by Marc Gourjon at the virtual CHES conference 2021.

[Blo+22] "Power Contracts: Provably Complete Power Leakage Models for Processors" by Bloem, Gigerl, Gourjon, Hadzic, Mangard, and Primas, published at CCS 2022.

[Ber+23] "Combined Fault and Leakage Resilience: Composability, Constructions and Compiler" by Berndt, Eisenbarth, Faust, Gourjon, Orlt, and Seker, published at CRYPTO 2023.

[Gou19] "Towards Secure Compilation of Power Side-Channel Countermeasures" by Gourjon, presented at PriSC workshop 2019.

[Gou21] "Explicit Leakage: Handling Side-Channel Behavior in Program Rewriting and Analysis" by Gourjon, presented at PriSC workshop 2021.

[Gou22b] "Power Contracts: Provably Complete Power Leakage Models for Secure Execution of Masked Software on Processors" by Gourjon, presented at TASER workshop 2022.

[Gou22a] "Fine-Grained Power Leakage Models and Verification of Software Masking" by Gourjon, presented at VeriSICC seminar 2023.

[Azo+22a] "Surviving the FO-Calypse: Securing PQC Implementations in Practice" by Azouaoui, Bos, Fay, Gourjon, Kuzovkova, Renes, Schneider, and Vredendaal, presented by Schneider at the Real World Crypto Symposium 2023.

Regarding Chapter 3: In [Abr+21] I participated in the characterization of physical leakage behavior and created a measurement setup for physical assessment. This started my research on leakage formalization and secure compilation, e.g., [Gou19]. During a research stage, Benjamin Grégoire and I developed the concept of explicit leakage to formally represent the characterized leakage behavior. Consequently, we developed scVerif, which I later improved in multiple ways. For example, I improved the debug information for guiding fixes, made more security notions available, and added support to analyze RISC-V assembly. Together with Maximilian Orlt and Clara Paglialonga we devised hardened gadgets and verified their security against the aforementioned characterized leakage behavior, including the exploration of optimization strategies. With Gilles Barthe and Benjamin Grégoire, we defined the Stateful t–(S)NI notions and published [Bar+21b]. Benchmarks and physical evaluation of the gadgets, including the purposely built measurement setup, have been primarily done by me. I extended scVerif to verify the output of the automated masking tool presented in [Abr+21], which used the aforementioned gadgets.

Regarding Chapter 4: In [Bos+21] we researched new gadgets for Kyber, which was to a large extent done by Tobias Schneider and me. The implementation and physical evaluation were largely done by me, with optimizations largely done by Joost Renes and Joppe Bos. I extended scVerif and devised the verification of the LUT-based A2B. Among my co-authored publications this work has received the most attention from researchers.

Regarding Chapter 5: Our work on power contracts [Blo+22] was initiated by me and conducted remotely due to the pandemic situation. I contributed Genoa by adapting Sail and its SMT back-end. Formalization of contracts, initial models, and the proofs related to model reduction and E2E resilience are also my work. Together with Barbara Gigerl and Vedad Hadžić, we investigated many approaches to notions of compliance that can be verified efficiently. I proposed the final approach used for checking the existence of

modeling functions, including an initial proof for Theorem 5.2, which was key for the continuation of our activities but remains unpublished. I hardened gadgets and verified them against the contract. Later, I presented our work at [Gou22b] and participated in a panel session.

Together with Sebastian Berndt, Maximilian Orlt, and Okan Seker, we researched polynomial masking and fault resilience. Sebastian and I came up with a novel approach that was completed and proven secure by Maximilian Orlt and Sebastian Berndt [Ber+23]

Finally, I gave a detailed overview of my studies to a greater audience of researchers on masking and side-channel security [Gou22a].

2. Side-Channel Resilience

In this chapter, we revisit computation in the presence of side-channel from an information-theoretic perspective. Further, approaches for establishing rigorous resilience based on verification and the masking countermeasure are discussed in the context of our contributions.

2.1. Notation and Preliminaries

We denote by $\mathbb{R}$ the real numbers, by $\mathbb{Z}$ the set of integers, and by $\mathbb{N}$ the natural numbers excluding 0. The ring of integers modulo $q \in \mathbb{N}$ is denoted by $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$ with shorthand $[q] := \mathbb{Z}_q$. Finite-fields are denoted by $\mathbb{F}_q$ with *order* $q = p^k$ for $p, k \in \mathbb{N}$ and prime p. The quotient rings $\mathbb{Z}[X]/(X^{256}+1)$ and $\mathbb{Z}_q[X]/(X^{256}+1)$ are denoted by $\mathbb{K}$, respectively $\mathbb{K}_q$.

Lower-case characters x in regular font denote variables. Upper-case calligraphic font denotes a set, e.g., $\mathcal{X} = \{x, y, z\}$. The tuple $x = (x_i)_{i \in \mathcal{I}}$ consists of all variables x_i with integer index $i \geq 0$ in *index set* $\mathcal{I}$. For an n-tuple $x = (x_i)_{i \in [n]}$ with $n \in \mathbb{N}$ the shorthand $y \subseteq x$ denotes $y \in \left\{(x_i)_{i \in \mathcal{I}} \,|\forall \mathcal{I} \subseteq [n]\right\}$. The concatenation of n-tuple $x = (x_i)_{i \in [n]}$ and n'-tuple $x' = (x'_i)_{i \in [n']}$ is denoted as $x||x' := (x_0, \ldots, x_{n-1}, {x'}_0, \ldots, {x'}_{n'-1})$.

Symbols $+, -, \cdot$ are used for arithmetic addition, subtraction, respectively multiplication. Bitwise logical conjunction is represented by $\wedge$, bitwise logical disjunction by $\vee$, and $\oplus$ is used for the bitwise logical exclusive disjunction (XOR). We omit $\cdot$ or $\wedge$ if clear from the context. Centered modular reduction for $q \in \mathbb{N}, r \in \mathbb{Z}$ is denoted by $r' = r \bmod^{\pm} q$, where $-\frac{q-1}{2} < r' \leq \frac{q-1}{2}$, and regular reductions by $r' = r \bmod q$ where $0 \leq r' < q$. Rounding to the closest integer is denoted by $\lfloor \cdot \rceil$ (with ties rounded up), rounding up by $\lceil \cdot \rceil$ and rounding down by $\lfloor \cdot \rfloor$.

Upper-case characters X indicate a discrete random variable associated with a sample space $\mathcal{X}$ and a discrete *probability distribution* (probability mass function) $\mathsf{Pr}\left[X = x\right] = p$ defining the probability $p \in \mathbb{R}, 0 \leq p \leq 1$, with $\sum_{x \in \mathcal{X}} \mathsf{Pr}\left[X = x\right] = 1$, for X taking value $x \in \mathcal{X}$. A random variable X is uniformly sampled from $\mathcal{X}$, denoted $X \xleftarrow{\$} \mathcal{X}$, if and only if (iff) $\forall x \in \mathcal{X} : \mathsf{Pr}\left[X = x\right] = \frac{1}{|\mathcal{X}|}$, where $|\mathcal{X}|$ is the cardinality of $\mathcal{X}$. An n-

tuple of (possibly dependent) discrete random variables with index set $\mathcal{I}$ is denoted by upper-case bold font, e.g., $\mathbf{X} = (X_i \in \mathcal{X}_i)_{i\in\mathcal{I}}$. It is equivalently expressed as a (multivariate) random variable X with sample space $\mathcal{X} := \bigtimes_{i\in\mathcal{I}} \mathcal{X}_i$ corresponding to the Cartesian product of the sample spaces of all components. Hence, the *joint probability distribution* $\mathsf{Pr}[\mathbf{X} = x]$ for n-tuple $x = (x_i \in \mathcal{X}_i)_{i\in\mathcal{I}}$ defines the probability for the event that all $X_i \in \mathbf{X}$ take the value of the corresponding $x_i \in x$. Random variables X and Y with sample space $\mathcal{Z}$ are *identically distributed*, denoted $X \equiv Y$, iff their joint probability distributions are (point-wise) *equivalent*, denoted $\mathsf{Pr}[X] \equiv \mathsf{Pr}[Y]$ and defined as $\forall z \in \mathcal{Z} : \mathsf{Pr}[X = z] = \mathsf{Pr}[Y = z]$. The *conditional probability* $\mathsf{Pr}[X \mid Y] \equiv \frac{\mathsf{Pr}[X,Y]}{\mathsf{Pr}[Y]}$ defines the probability for $X = x$ given $Y = y$ with $\mathsf{Pr}[Y = y] > 0$. We say X is *independent of* Y iff $\mathsf{Pr}[X, Y] \equiv \mathsf{Pr}[X]\,\mathsf{Pr}[Y]$ or equivalently $\mathsf{Pr}[X \mid Y] \equiv \mathsf{Pr}[X]$. Random variable $\mathbf{X}$ is *k-wise independent* for $1 < k < |\mathbf{X}|$ iff for every $\mathbf{X}' \subset \mathbf{X}$ with $|\mathbf{X}'| \leq k$ we have $\mathsf{Pr}[\mathbf{X}'] \equiv \prod_{X\in\mathbf{X}'} \mathsf{Pr}[X]$. Random variable $\mathbf{X}$ is *k-wise independent of* Y for $1 \leq k \leq |\mathbf{X}|$ iff for every $\mathbf{X}' \subseteq \mathbf{X}$ with $|\mathbf{X}'| \leq k$ we have $\mathsf{Pr}[\mathbf{X}', Y] \equiv \mathsf{Pr}[\mathbf{X}']\,\mathsf{Pr}[Y]$.

The mutual information (MI) is defined as

$$\mathsf{MI}(X;Y) = \sum_{y\in\mathcal{Y}} \sum_{x\in\mathcal{X}} \mathsf{Pr}[X = x, Y = y] \log_2 \left(\frac{\mathsf{Pr}[X = x, Y = y]}{\mathsf{Pr}[X = x]\,\mathsf{Pr}[Y = y]} \right). \quad (2.1)$$

The statistical distance (SD) (total variation distance) of X, Y with sample space $\mathcal{Z}$ is defined as $\mathsf{SD}(X;Y) = \frac{1}{2} \sum_{z\in\mathcal{Z}} |\mathsf{Pr}[X = z] - \mathsf{Pr}[Y = z]|$. The (Shannon) *entropy* of X is given by $\mathsf{H}[X] = -\sum_{x\in\mathcal{X}} \mathsf{Pr}[X = x] \log_2 \mathsf{Pr}[X = x]$.

Fraktur font $\mathfrak{v}$ is reserved for the syntax of DSL. Sans-serif font is used for names of functions, algorithms, *etc.*, and $\mathsf{dom}(\cdot)$ is the domain of a function, respectively the sample space of a random variable. Later on, we define *shares* $X_i^{(j)}$ with an optional superscript index $j \in [n_s]$ enclosed in parentheses.

2.2. Physical Side-Channels and Attacks

We provide a high-level view on SCA attacks.

[Koc96] introduces SCA based on timing side-channel, *i.e.*, a side-channel arising in implementations where the computation or response time depends on sensitive values such as cryptographic keys. Later, Kocher, Jaffe, and Jun exploit power side-channel [KJJ99], *i.e.*, the fact that the amount of electric power a computation on a physical device consumes depends on (potentially sensitive) intermediate values involved in the operations to perform the computation. By measuring the power consumption of the operations underlying the computation it is thus possible to learn information on an *intermediate*

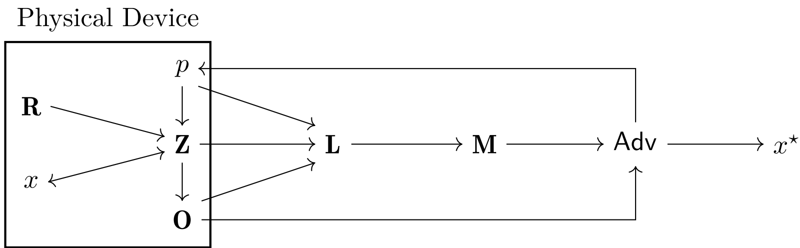


Figure 2.1.: Schematic of side-channel attacks where an implementation on a device is invoked on adversarial controlled public inputs p, producing public outputs $\mathbf{O}$. The execution produces, based on the inputs, a fixed secret state x and randomness $\mathbf{R}$, intermediate values $\mathbf{Z}$ which are subject to physical side-channel leakage $\mathbf{L}$. The adversary Adv may invoke the implementation multiple times, acquiring physical measurements $\mathbf{M}$ of the leakage, inputs and outputs for adapted choices of inputs p, to gain information for an educated guess $x^{\star}$ of the secret x.

value, e.g., an intermediate operand, which is internal to the implementation and otherwise believed unknown to adversaries.

The concept of SCA attacks is highlighted in Figure 2.1 depicting the interaction of an adversary and an attacked implementation executed on a physical device. The computation happening within the implementation involves intermediate operations and intermediate values $\mathbf{Z}$. All operations involving data as well as any data at rest are subject to physical side-channel leakage, producing leakages $\mathbf{L}$. Adversaries try to learn information on the involved secret data (state or secret inputs, if any) by invoking the implementation and taking measurements $\mathbf{M}$ of the emitted side-channel leakage. In general, we consider adaptive adversaries which are able to observe *public* outputs, *i.e.*, values which are declared to be non-confidential and secret-independent, while controlling parts of the input by iteratively choosing specific values prior to execution based on the so-far learned information, similar to IND-CCA2 in classical cryptanalysis.

The success within power SCA attacks transfers to the EM radiation induced by switching transistors and electrical wires [GMO01; QS01; Agr+03]. In general, SCA is considered to be applicable within a wide range of covert channels, such as cache-timing, heat dissipation, and acoustic transmission, even of peripherals such as keyboards [Kuh02; AA04; ZZT05; Lav+21]. They can be exploited remotely when information is radiated by other circuitry on

the same chip [Cam+18] or by simply using a processor's onboard telemetry as found in Intel processors [Lip+21]. We focus on power and EM.

The measurement setup (which can be in software [Lip+21]) is central and yields a *measurement trace* for each invocation of the implementation consisting of small numbers to millions of *sample points*, *i.e.*, measurement quantities over time. Measurement setups can range from low-cost analog-to-digital converter (ADC), measuring the power consumption via a shunt resistor, to professional multi-channel oscilloscopes with multiple giga-samples sampling frequency and multiple GHz bandwidth, as well as precise voltage, respectively EM, measurement and amplification equipment. The available equipment enables or limits an adversary or assessor to detect and subsequently exploit measurement samples $\mathbf{M}$ as the quality, e.g., signal-to-noise ratio (SNR) is directly impacted by the setup. Those sample points are chosen which are considered best for exploitation, denoted points of interest (POIs) with quantity n_{poi}.

Informally, advanced SCA (e.g., differential power analysis (DPA) [KJJ99; Mes00]) computes the expected measurement x' for possible values of the internal state x and input p and then *distinguishes* which values are more likely by comparing the expected measurement with the measured side-channel leakage acquired by iteratively invoking the implementation with input p. Those x' where the measured leakage does not match the expected leakage are considered incorrect guesses and ruled out, or at least considered less likely. Consequently, the entropy of the secret x is reduced and a guess $x*$ for the secret state x is output. The process is repeated on different inputs (and thus multiple traces), or for multiple POIs, e.g., [Mes00], within the same trace until the remaining entropy of secrets can be brute forced or enumerated.

As a guiding example, assume that the power consumption $M \in \mathbf{M}$ at some point in the side-channel measurement corresponds to the number of bits set in the intermediate value $Z \in \mathbf{Z}$ resulting from an operation, *i.e.*, $M = \text{HW}(Z)$, which is a frequent assumption known as the Hamming weight (HW) leakage model. Secret values are unknown to adversaries and hence correspond to realizations of a random variable with high entropy. Let K denote a cryptographic key with sample space $\mathcal{K} = \mathbb{Z}_2^n$ and let $i \in \mathsf{dom}(p)$ be a public input, chosen by the adversary. As a contrived example consider the intermediate $z_{k,i} = k \wedge i$, where $k \in \mathcal{K}$ is a fixed secret key, which is contained in state x and does not change between invocations. The SCA adversary now computes the expected leakage $l_{k^*,i} = \text{HW}(k^* \wedge i)$ for each possible key candidate $k^* \in \mathcal{K}$ and input i to subsequently focus on those candidate keys k^* where the leakage m_i measured when invoking the circuit with input i matches the expected leakage $l_{k^*,i}$. In this setting, at most n traces are needed to fully recover the key as careful choices of input i allow to

leak each key bit individually.

Different leakage models are central to SCA attacks as these represent a hypothetical or empirical definition of the expected side-channel measurement. In the example above, the function determining $l_{k^*,i}$ is a leakage function modeling the leakage of the intermediate $z_{k^*,i}$.

In an *unprofiled attack* common intermediates of the implemented algorithm, e.g., the S-Box outputs of an AES implementation, are assumed to leak under hypothetical leakage functions such as HW or Hamming distance (HD). In a *profiled attack* the expected leakage of the POIs are tailored to the implementation and executing device, covering specific implementation choices. For measurement samples $M_i \in \mathbf{M}$ individual *leakage functions* $f_i(\mathbf{Z}) = M_i$ are used, e.g., by characterizing them using measurements with known secrets.

Different distinguishers exist to cope with measurement noise and (partially) unknown leakage models. The distinguisher may be deterministic as in the original DPA [KJJ99; Mes00], statistical (e.g., Pearson's correlation coefficient in correlation power analysis (CPA) [BCO04]) or information theoretical (e.g., mutual information analysis (MIA) based on MI [Gie+08]).

In a 1st-order attack a single POI is exploited ($n_{poi} = 1$), always corresponding to the same intermediate variable in one or more measurement traces. However, countermeasures allow to ensure that every intermediate variable is independent of the key which may cause first-order attacks to fail. Higher-order attacks exploit the information contained in n_{poi} POIs (multivariate higher-order SCA) or statistical moments of order up to n_{poi} of one POI (univariate higher-order SCA) to defeat countermeasures (e.g., [Mes00]). However, higher-order attacks are susceptible to noise and are not necessarily practical as they may require significantly more measurement traces and computing power. *Horizontal SCA* and deep learning attacks exploit all or a large number of measurement samples [Wal01], *i.e.*, $n_{poi} \gg 1$. Single-trace attacks and simple SCA (e.g., simple power analysis (SPA)) are special cases which mainly target unprotected implementations as only direct leakage of secret keys is exploited [KJJ99]. Notably, soft analytical side-channel attacks (SASCAs) [VGS14] allows to exploit many POIs by leveraging belief propagation and graph-based models of the intermediate values.

2.3. A Framework for Computation in the Presence of Side-Channel

At a high level, side-channel leakage is a physical signal such as the total power consumption or EM field at some position. Both are emitted during compu-

tation on a device and SCA exploits the secret-dependent information carried within the signal. Such information leakage cannot be explained or modeled without incorporating information on the computation, *i.e.*, how computation behaves. Hence, profound protection inherently relies on accurate models of *computation with leakage* which must give qualitative insight on the information available for SCA.

Micali and Reyzin pioneered a generic model of leakage during computation based on a Turing machine [MR04]. They prove that selected cryptographic primitives remain secure in the presence of side-channels with assumptions on the leakage, e.g., that only computation leaks (OCL). Specifically, they allow an adversary to choose among pre-defined leakage functions and subsequently receive the output of the functions evaluated on the internal state of the Turing machine at fixed instances in time. Such approaches are common in *leakage resilient cryptography*, a field concerned with proving the resilience of cryptographic schemes [DP08; KR19]. Among the limitations is that such proofs leave the implementation challenges open since they are tailored to abstract representations of specific schemes and frequently rely on leak-free computations. Further, the inherent approach to allow adversaries to choose leakage function is in contrast to the physical setting where leakage is entirely determined by the computation and the environment. Consequently, such models may demand too high levels of protection [KR19]. On the other hand, the assumed models are frequently too weak as the physical side-channel does not adhere to the model assumptions rendering proven security claims questionable.

We seek to overcome the limitations by proposing models of computation with leakage which are specifically tailored to the masking countermeasure, to the physical leakage behavior of concrete devices, to executable implementations, and generic proofs based on automated verification. Hence, this work proposes new models connecting to existing ones and explores verification and development in such models. We introduce an overarching framework for different models of computation, denoted $[\![\mathsf{C}]\!]^m$, where m is an identifier for distinguishing the models and C is a definition of computation, e.g., executable code for evaluating a function. The purpose is to express the behavior of computation and the resulting side-channel leakage consistently for a broad range of computations C at different levels of abstraction. We define multiple instances of computation models in the sequel and adopt the convention that G (for gadget) indicates high-level algorithms, P software (code), and H hardware circuits (netlists). Note that computation models are extensible, *i.e.*, we define semantic models without leakage and subsequently define additional models on top of the former definition.

Definition 2.1 (Computation Model $[\![\mathsf{C}]\!]^m$)**.** A computation model, denoted $[\![\mathsf{C}]\!]^m$, is a tuple $([\![\mathsf{C}]\!]^m, \mathcal{C}, \mathcal{X}, \mathcal{Y})$ where $\mathcal{X}$ is the finite domain of inputs, respectively $\mathcal{Y}$ for outputs, and $\mathcal{C}$ is a language representing the set of computations which are well-formed and terminating on all inputs, denoted $\mathsf{C} \in \mathcal{C}$. For any computation $\mathsf{C} \in \mathcal{C}$ the computation model is a randomized algorithm mapping an input tuple $x \in \mathcal{X}$ to a deterministic (*i.e.*, reproducible) output tuple $y \in \mathcal{Y}$ and a random n_l-tuple of leakages $\mathbf{L}^m = (L_i^m \in \mathcal{L}_i^m)_{i \in [n_l]}$, referred to as *trace*, with the individual sample spaces $\mathcal{L}_i^m$ and the number of *leakage points* n_l determined by C, denoted

$$[\![\mathsf{C}]\!]_x^m \xmapsto{\mathbf{L}^m} y. \tag{2.2}$$

Model $[\![\mathsf{C}]\!]^m$ is *noiseless* if there is no uncertainty in the leakages, *i.e.*, $\forall \mathsf{C} \in \mathcal{C}, x \in \mathcal{X} : \mathsf{H}[\mathbf{L}^m] = 0$ with $\mathbf{L}^m$ resulting from $[\![\mathsf{C}]\!]_x^m$ and *noisy* otherwise. We say $[\![\mathsf{C}]\!]^m$ is a black-box (BB) model if the trace is an empty n_l-tuple $\mathbf{L}^m = (L_i^m = ())_{i \in [n_l]}$ for any $\mathsf{C} \in \mathcal{C}, x \in \mathcal{X}$.

We focus on masking and the security of masked functions, *i.e.*, the case that C is a masked computation to evaluate function $f(x, p) = (y, o)$ where x, y are *secret* values and p, o are *public* values. This is in line with many cryptographic schemes which involve secret values in the form of cryptographic keys, which need to be protected against SCA, as well as public values, e.g., ciphertext or plaintext, which are independent of secrets and potentially under the control of the adversary. In addition, masking involves random values $\mathbf{R}$ which are by definition uniformly independent and identically distributed (i.i.d.) and neither under the control of nor known by adversaries. Masking and encoding of values are discussed in detail in Section 2.4. For now, consider that $\mathbf{X}$ is a randomized encoding of x, denoted $\mathbf{X} = \mathsf{Enc}(x)$ and that the computation returns a randomized encoding $\mathbf{Y}$ which decodes to the secret output $y = \mathsf{Dec}(\mathbf{Y})$. Most of our computation models have a common input, respectively output, interface, denoted $[\![\mathsf{C}]\!]_{\mathbf{X},\mathbf{R},p}^m \mapsto (\mathbf{Y}, \mathbf{O})$. Here, the input to the computation consists of an encoding $\mathbf{X}$ of a $n_{\mathbf{X}}$-tuple $x \in \mathcal{X}_{\mathsf{C}}^{n_{\mathbf{X}}}$ of inputs, a n_p-tuple $p \in \mathcal{P}_{\mathsf{C}}^{n_p}$ of public inputs, and $n_{\mathbf{R}}$-tuple $\mathbf{R} \in \mathcal{R}_{\mathsf{C}}^{n_{\mathbf{R}}}$ of uniformly i.i.d. random values. The output of the computation consists of an $n_{\mathbf{O}}$-tuple $\mathbf{O} \in \mathcal{O}_{\mathsf{C}}^{n_{\mathbf{O}}}$ of public outputs, and a randomized encoding $\mathbf{Y}$ of a $n_{\mathbf{Y}}$-tuple $y \in \mathcal{Y}_{\mathsf{C}}^{n_{\mathbf{Y}}}$ of secret outputs.

We continue with an information-theoretical discussion of side-channel security.

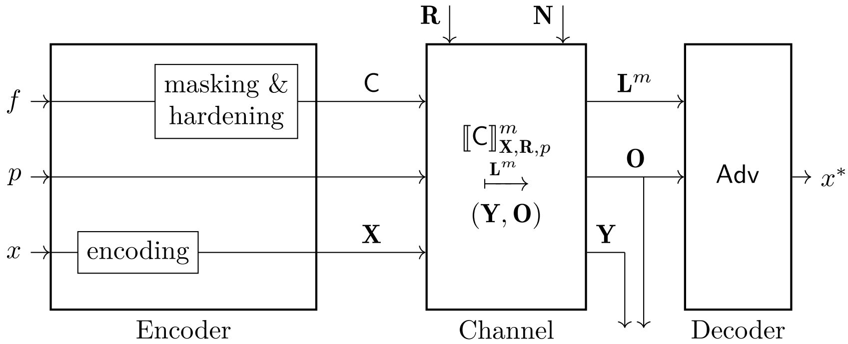


Figure 2.2.: Side-channel represented as an information-theoretic communication channel. The channel is actively performing computations on data, this causes an exploitable signal consisting of leakage and outputs. The adversary tries to recover the data by decoding the signal. Masking corresponds to encoding data and computation statements C to reduce the adversary's chances.

2.3.1. Adversarial Models and Provable Resilience

The relationship between side-channels, SCA adversaries and masking is close to the concept of information-theoretical communication channels, see Figure 2.2 for a schematic discussed next.

The side-channel leakages $\mathbf{L}^m$ and public outputs $\mathbf{Y}$ correspond to a signal containing information on the inputs involved in a computation. The signal is received by an adversary Adv which aims to reconstruct the input value x given the signal. In terms of information-theoretical communication channels, the inputs act as a *source* of information, the computation of the device represents a *channel* and the adversary acts as a *decoder*. Note that the computation may additionally involve noise $\mathbf{N}$ perturbing the leakages. Masking corresponds to the *encoder* choosing an encoding $\mathbf{X}$ of the inputs and a computation C in such a way that the adversary's chance to recover information is reduced or bounded.

The *channel capacity* $C = \mathsf{MI}\left(\mathbf{X}; \left(\mathbf{L}^m, \mathbf{O}\right)\right)$ is an upper bound on the information that can be gained on the encoding $\mathbf{X}$ of secret x. Such a metric may be used to quantify the information loss of a single invocation of a computation if the computation model permits to determine the probability distribution of $\mathbf{L}^m$ and $\mathbf{Y}$ as a function of $\mathbf{X}, \mathbf{R}, p, \mathbf{N}$.

In cryptography, it is common to consider adversaries *repeatedly and adap-*

tively interacting with $[\![\mathsf{C}]\!]^m$. We adopt the *adversarial model* of [DDF19] to computation models. Remark, our definition requires the computation to produce a new secret input for a subsequent invocation if the interaction is *multi-trace*, *i.e.*, the number of invocations n_t is greater than one. This allows modeling *stateful* computations, e.g., computation on concrete devices, where secret inputs are kept in non-volatile storage and need to be refreshed after use. Random plaintext attacks (RPA) can be modeled by a restriction $r = \mathsf{RPA}$ specifying that the public inputs p_j in rounds $j \in [n_t]$ are sampled uniformly i.i.d. and made available to $\mathsf{Adv}^i_{\mathsf{RPA}}$. However, we stick to chosen plaintext attacks. This work is focused on established resilience for $\mathsf{Adv}^i_{n_{poi} \leq t}$.

There are also invasive attacks, denoted fault attack (FA), where an adversary actively manipulates the computation, e.g., [Ber+23], but this is not considered here.

Definition 2.2 (Adversary Model Adv^i_r interacting with $[\![\mathsf{C}]\!]^m$)**.** An adversary Adv^i_r with identifier i and a restriction r is a polynomial-time algorithm that repeatedly and adaptively interacts with the computation $[\![\mathsf{C}]\!]^m$ for a bounded number of rounds $n_t \in \mathbb{N}$ as follows. Before the first round, a secret x is fixed and encoded as $\mathbf{X}_0$ without Adv^r_i gaining information. In round $j \in [n_t]$ the random values $\mathbf{R}_j$ are sampled uniformly i.i.d. without Adv^r_i gaining information. Then, Adv^r_i chooses a public input p_j and the computation $[\![\mathsf{C}]\!]^m_{\mathbf{X}_j,\mathbf{R}_j,p_j}$ is performed yielding $\mathbf{L}^m_j = (L^m_k)_{k \in [n_l]}$ and $\mathbf{Y}_j, \mathbf{O}_j$ with the input for a subsequent invocation $\mathbf{X}_{j+1} \subseteq \mathbf{Y}_j$ if $n_t > 1$. Adv^i_r receives $\mathbf{O}_j$ and trace $\mathbf{L}^m_j$ in part or whole subject to restriction r. After round $n_t - 1$ Adv^i_r outputs a *guess* $a \in \mathcal{A}$. The repeated interaction is denoted by $\mathsf{Adv}^i_r \rightleftharpoons [\![\mathsf{C}]\!]^m$, where $\rightleftharpoons$ denotes interaction. The probability distribution for the adversary outputting a guess $a \in \mathcal{A}$ based on the information received during the rounds is denoted by *out* $\left(\mathsf{Adv}^i_r \rightleftharpoons [\![\mathsf{C}]\!]^m\right)$. A tth*-order adversary* $\mathsf{Adv}^i_{n_{poi} \leq t}$ chooses an index set $\mathcal{I} \subseteq [n_l]$ with $|\mathcal{I}| = n_{poi} \leq t$ prior the first round and receives in every round $j \in [n_t]$ the leakage points of $\mathbf{L}^m_j$ with index $k \in \mathcal{I}$.

The attacks discussed in Section 2.2 are covered by the above definition and we consolidate all these attacks by the tth*-order physical adversary* $\mathsf{Adv}^{\mathrm{PHY}}_{n_{poi} \leq t}$ and the *horizontal physical adversary* $\mathsf{Adv}^{\mathrm{PHY}}$ without restrictions. Both adversaries interact with the *physical computation* $[\![\mathsf{P}]\!]^{\mathrm{PHY}}$ (or *physical execution*) of some software implementation P executed on a processor embedded in a physical device

$$\mathsf{Adv}^{\mathrm{PHY}} \rightleftharpoons [\![\mathsf{P}]\!]^{\mathrm{PHY}}_{\mathbf{X},\mathbf{R},p} \xmapsto{\mathbf{L}^{\mathrm{PHY}}} (\mathbf{Y}, \mathbf{O})\,. \tag{2.3}$$

We give more detail on the computation model $[\![\mathsf{P}]\!]^{\mathrm{PHY}}$, which is noisy, and its connection to our contributions in the next section. In physical attacks,

the number of rounds n_t ranges from multiple thousands to multiple millions and is also referred to as *number of traces*. Adversaries considered in this work cannot tamper with the randomness used by the implementation under attack and cannot change the program code or alter the execution flow apart from providing different public inputs. Hence, randomness used within the execution is i.i.d. among traces. Restrictions in the form of lower bounds on n_t or applicable attacks, e.g., t, may be set by certification authorities or security standards, e.g., [Fed13; Fed16; Fed23].

Based on the definition the advantage of side-channel information is expressed in terms of the SD of an adversary interacting with two similar models as follows. Let computation model $[\![\mathsf{C}]\!]^{a}_{\mathbf{X},\mathbf{R}^{a},p} \overset{\mathbf{L}^{a}}{\longmapsto} (\mathbf{Y}^{a},\mathbf{O}^{a})$ be a computation with leakage and let $[\![\mathsf{C}]\!]^{bb}_{\mathbf{X},\mathbf{R}^{bb},p} \overset{\mathbf{L}^{bb}}{\longmapsto} (\mathbf{Y}^{a},\mathbf{O}^{a})$ be a BB model with empty leakage trace but equivalent semantics, *i.e.*, outputs coincide in both models given the same inputs. Then an upper bound $p \in [0,1]$ for the probability of any Adv^{i}_{r} *distinguishing* within n_t invocations between the information received in either model can be established based on the SD if

$$\forall \mathsf{Adv}^{i}_{r} : \mathsf{SD}\left(out\left(\mathsf{Adv}^{i}_{r} \rightleftharpoons [\![\mathsf{C}]\!]^{bb}\right); out\left(\mathsf{Adv}^{i}_{r} \rightleftharpoons [\![\mathsf{C}]\!]^{a}\right)\right) \le p. \tag{2.4}$$

In the case that $p = 0$ the secret inputs to C remain for any adversary with access to side-channel leakage information-theoretically as private as without access to side-channel leakage. This does not endorse the security of C in general. Instead, it expresses a relation between having access to, or not having access to side-channel leakage, *i.e.*, cryptographically insecure computations can be implemented information-theoretically side-channel secure. The relative approach allows for a focus on side-channel resilience and leaves aside the quite distinct problem of (computational) cryptographic security [DDF19].

Statements as in Eq. (2.4) are commonly proven using simulation-based approaches. This involves the construction of a *simulator* which simulates the information received by an adversary during computation. A definition of simulators is given in Definition 2.3 adapted from co-authored work [Blo+22].

Definition 2.3 (Simulator)**.** Let $\mathbf{C}$ and $\mathbf{H}$ be possibly dependent random variables with sample space $\mathcal{C}$, respectively $\mathcal{H}$, and $\mathbf{R}$ be uniformly i.i.d. sampled from space $\mathcal{R}$. A *simulator* is a randomized algorithm $S : \mathcal{C} \times \mathcal{R} \to \mathcal{H}$ which samples the random variable $\mathbf{R}$ to simulate the distribution of $\mathbf{H}$ from $\mathbf{C}$. We say that simulator S *perfectly simulates*, or *simulates* for short, $\mathbf{H}$ from $\mathbf{C}$ and $\mathbf{R}$ iff $\mathsf{Pr}[S(\mathbf{C},\mathbf{R})] \equiv \mathsf{Pr}[\mathbf{H}]$. Simulator S is explicitly stated to *simulate with error probability* $p \in (0,1]$ for any biased simulation, *i.e.*, $\mathsf{SD}(S(\mathbf{C},\mathbf{R}); \mathbf{H}) \le p$.

Model $[\![\mathsf{C}]\!]^a$ and BB model $[\![\mathsf{C}]\!]^{bb}$ remain as defined before. Let $\mathbf{M} = \left(\mathbf{O}^a_j, \mathbf{L}^a_j\right)_{j\in[n_t]}$ denote the entire information received by Adv^i_r interacting with $[\![\mathsf{C}]\!]^a$. A simulation-based proof requires to show for any $\mathsf{Adv}^i_r \rightleftharpoons [\![\mathsf{C}]\!]^a$ the existence of a simulator S capable of simulating the information $\mathbf{M}$ received by Adv^i_r given the public inputs $(p_j)_{j\in[n_t]}$ generated by the adversary and i.i.d. random $\mathbf{R}^s$ by interacting with $[\![\mathsf{C}]\!]^{bb}$. If there exists a simulator to simulate $\mathbf{M}$ with error probability p then Eq. (2.4) can be stated in terms of a simulator

$$\forall\ \mathsf{Adv}^i_r\ \exists\ S : \mathsf{SD}\left(out\left(S \rightleftharpoons [\![\mathsf{C}]\!]^{bb}\right); out\left(\mathsf{Adv}^i_r \rightleftharpoons [\![\mathsf{C}]\!]^a\right)\right) \leq p \tag{2.5}$$

where $out\left(S \rightleftharpoons [\![\mathsf{C}]\!]^{bb}\right)$ corresponds to invoking Adv^i_r on $\mathbf{M}$ and $p = 0$ if S simulates $\mathbf{M}$ perfectly.

The masking countermeasure allows constructing C such that a simulator exists which is capable of perfectly simulating the input of the restricted adversary $\mathsf{Adv}^i_{n_{poi}\leq t} \rightleftharpoons [\![\mathsf{C}]\!]^m$ given access to the corresponding BB model of $[\![\mathsf{C}]\!]^a$ where the security parameter t indicates the *order* at which C needs to be masked. We leave the details of masking to Section 2.4. The security of computations is strictly linked to a specific model as the perfect simulator associated with C is specific to the leakage behavior defined in $[\![\mathsf{C}]\!]^m$. Hence, information-theoretical side-channel security for C in one model does not imply security in another one. Consequently, models must be as close as possible to the physical behavior to improve the confidence in physical security.

2.3.2. Overview on Computation Models

Our contributions are centered around the verification of security in computation models capturing physical leakage accurately. We give an overview of the connection between existing and contributed models which refers to Figure 2.3. An arrow $a \Rightarrow b$ indicates that model a approximately models the leakage considered in model b.

We intend to establish the security of software implementations P against *physically observable leakage* $\mathbf{L}^{\text{PHY}}$. This leakage corresponds to the power consumption or EM radiation of a *device*, which can be observed with measurement equipment. $[\![\mathsf{P}]\!]^{\text{PHY}}$ represents the *physical computation* or *physical execution* on a device with a processor executing software P. We cannot characterize $[\![\mathsf{P}]\!]^{\text{PHY}}$ directly in a manner that accurately considers all attack instances, measurement setups, environment characteristics, *etc.*. Instead, its behavior is approximated by measurements, static analysis, and more abstract computation models. Construction of secure implementations can then be performed in these models to simplify design and initial security assessment. Most gadgets are designed and proven secure in the baseline probing

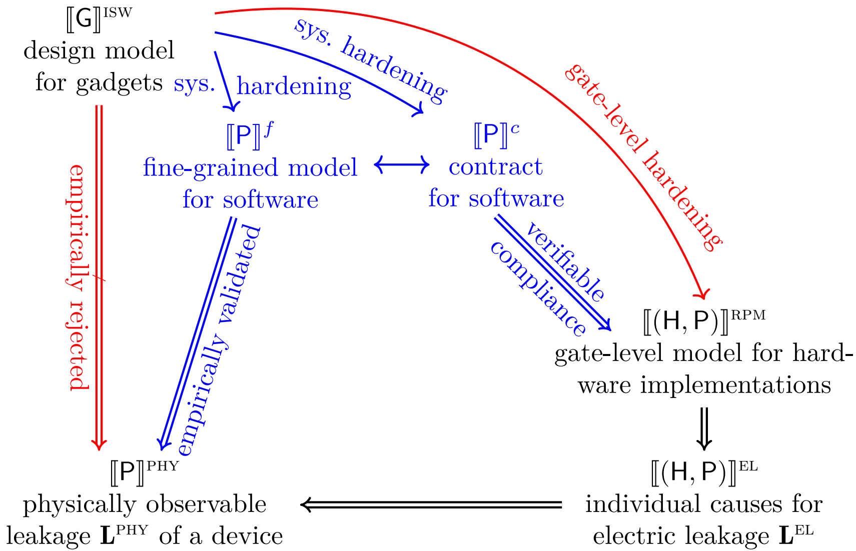


Figure 2.3.: Overview of existing (black) and contributed (blue) computation models to establish resilience against physical leakage (bottom). The probing model $[\![\mathsf{G}]\!]^{\text{ISW}}$ does not capture all physical leakage, hence plain implementations turn out insecure. Fine-grained models permit accurate modeling of physical leakage $\mathbf{L}^{\text{PHY}}$ and systematic hardening of software implementations P attaining physical security. Contracts provably capture the complete gate-level leakage $\mathbf{L}^{\text{EL}}$ modeled by the robust probing model (RPM).

model $[\![\mathsf{G}]\!]^{\text{ISW}}$ [ISW03]. However, plain implementations of such gadgets are insecure when executed on a physical device (red arrow).

There was a lack of reliable approaches to the systematic construction of efficient software implementations with physical security before the co-authored works. We introduce fine-grained models $[\![\mathsf{P}]\!]^{f}$ which permit modeling $\mathbf{L}^{\text{PHY}}$ specifically for some device identified by f based on physical measurements and tests (Chapter 3). The models are intelligible for humans, permit to systematically harden software implementations of gadgets, enable automated verification of security, and, as we show, connect to resilience against $\mathbf{L}^{\text{PHY}}$.

Alternatively, modeling of $[\![\mathsf{P}]\!]^{\text{PHY}}$ can be approached by modeling the underlying physical effects. $\mathbf{L}^{\text{PHY}}$ is widely perceived as a superposition of *elec-*

tric leakage $\mathbf{L}^{\text{EL}}$ corresponding to individual electric effects of semiconductors, resistors, capacitors, *etc.* which are the basis for transistors, CMOS gates, processor implementations, MCUs, and devices. These are modeled at the *gate level* or even *silicon level* based on physical simulations of netlists H and program P, denoted by the model $[\![(\mathsf{H},\mathsf{P})]\!]^{\text{EL}}$. However, low-level and analog simulations are computationally expensive and rather suitable for understanding individual effects than in modeling side-channel of masked software.

The RPM [Fau+18] denoted $[\![(\mathsf{H},\mathsf{P})]\!]^{\text{RPM}}$ is a time-tested model of electric leakage. Its abstract nature permits systematic and efficient hardening of *hardware* implementations. A seasoned approach for developing physically secure hardware implementations is to systematically *harden* gadgets until reaching security in $[\![(\mathsf{H},\mathsf{P})]\!]^{\text{RPM}}$, e.g., by adding gates to mitigate leakage. The recent approach in [GPM21] of verifying software security, based on a netlist and the RPM, requires software developers to understand side-channel vulnerabilities caused at the gate-level of hardware and does not scale to large implementation sizes (Section 5.4.3). Hence, the gate-level model is of limited use for developing software executing on processors due to its low-level nature.

Our contracts $[\![\mathsf{P}]\!]^{c}$ are an extension of fine-grained models and provably capture the gate-level leakage modeled by the RPM but model leakage of instructions instead of low-level gates (Chapter 5). They benefit from the modeling accuracy of the RPM and its security reductions, and at the same time permit for systematic hardening & fast verification. Contracts and fine-grained models serve as systematic approaches to the verification and construction of efficiently masked software implementations resilient against $\mathbf{L}^{\text{PHY}}$ and $\mathbf{L}^{\text{EL}}$.

2.4. The Masking Countermeasure

Masking is a countermeasure applied at the algorithmic level and applicable for protecting diverse applications and cryptographic schemes. We give an introduction to masking as well as the related formal proofs of security for masked algorithms. In Section 2.5 we discuss how masking links to adversarial models.

2.4.1. Concept

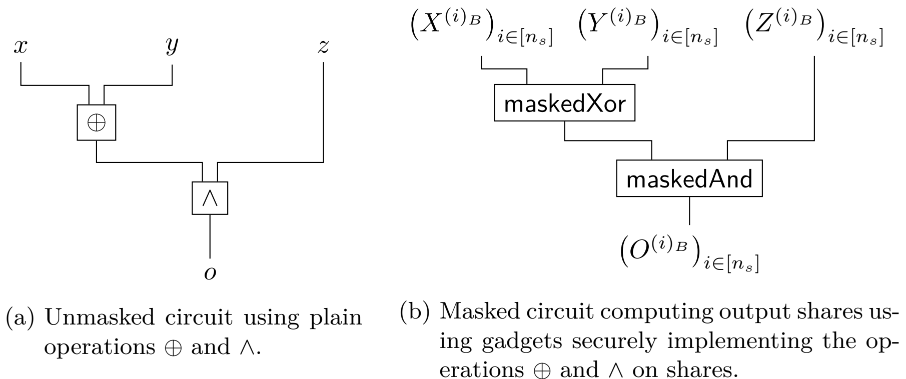
The concept behind masking is to encode all secret values subject to side-channel leakage using $n_s > 1$ randomly distributed values (denoted *shares*) and to replace any operation on such values with *gadgets* which compute on those shares. Gadgets are building blocks that compute an operation on shares

in such a way that an adversary has to retrieve a lower bound of $t + 1 \leq n_s$ shares to recover the sensitive value. In many cases, applying the countermeasure results in an increased runtime scaling asymptotically quadratic in n_s while the difficulty for successful attacks scales exponentially in the security parameter t if sufficient noise is present, which makes the countermeasure appealing [Cha+99; PR13; DDF19; GGS18].

Central to masking is a concept similar to the information-theoretically secure one-time pad (OTP) encryption of Shannon [Sha49]: Learning the value resulting from a modular addition (e.g., XOR) of a secret value $X \in \mathbb{F}_2$ and an independently and uniformly distributed random value $R \in \mathbb{F}_2$ provides information-theoretically no information on X as long as R is unknown, *i.e.*, *perfect secrecy* of secret X can be proven. This enables the strong information-theoretical statements described in Section 2.3.1 and for deriving computational or statistical security claims. Masking is the art of computing on shares in such a way that intermediate values, which are subject to SCA, remain protected by one or more random values (denoted *masks*) and to prove perfect, computational, or statistical security. An inherent assumption is that masks are required to be *fresh*, *i.e.*, they must be sampled i.i.d. in every invocation of a gadget or implementation thereof.

Masking shares a lot of similarities with *secret sharing* and *threshold* schemes as proposed by [Bla79; Sha79] in a more general cryptographic and multi-party computation (MPC) context since a lower-bounded number of more than t values need to be known to reconstruct a secret value. In MPC multiple parties securely compute a function by exchanging intermediate computation results via an insecure channel or where entire parties can be corrupted. Masking is related in that a function is securely computed knowing that intermediate computations results leak. However, all parties are in the same location (e.g., device), respectively there is only one "player" executing the protocol of each party. Masked algorithms are thus optimized for randomness consumption instead of the amount of interactions.

Masking can be applied systematically by representing the algorithm or implementation to be protected as a *circuit*, *i.e.*, as a directed acyclic graph where vertices represent operations, inputs, random masks or outputs, and edges represent intermediate values. We use the term *implementation* for the executable implementation of a *circuit*, both of them are definitions of computation. In their seminal work [ISW03] Ishai, Sahai, and Wagner defined, and proved secure, a *circuit compiler* which produces a masked circuit given a reference circuit. In a nutshell, masking a reference circuit results in a masked circuit where input and output vertices are replaced by n_s vertices encoding the respective unmasked value, and vertices representing operations are replaced by gadgets computing the same operation on the encoded values



(a) Unmasked circuit using plain operations $\oplus$ and $\wedge$.

(b) Masked circuit computing output shares using gadgets securely implementing the operations $\oplus$ and $\wedge$ on shares.

Figure 2.4.: Masking a circuit computing $o = (x \oplus y) \wedge z$.

in a secure manner [MR04; Cha+99; GP99; ISW03] as depicted in Figure 2.4. The process of securely combining gadgets computing a function or algorithm is denoted *composition*.

We detail the inherent steps to masking in the following. The choice of *encoding* (*sharing*) for values in Section 2.4.2. The design of gadgets securely computing on shares in Section 2.4.3. The composition of masked circuits from gadgets in Section 2.4.4.

2.4.2. Encoding Secrets

Encoding a secret value into n_s randomly distributed (dependent) values can be achieved using different encoding schemes. We denote the algorithm to encode a value x which secret X can take, into shares $X^{(0)_\mathsf{e}}, \dots, X^{(n_s-1)_\mathsf{e}}$ by $\mathsf{eEnc}_t^{n_s}(x) = \left(X^{(i)_\mathsf{e}}\right)_{i\in[n_s]}$ where t is the *security order* or *threshold* and e is an optional identifier to distinguish the schemes. The secret value can be reconstructed from the shares using $\mathsf{eDec}_t^{n_s}\left(X^{(0)_\mathsf{e}}, \dots, X^{(n_s-1)_\mathsf{e}}\right) = x$. The algorithms are applied component-wise in the case of tuples with indices in $\mathcal{I}$, *i.e.*, $\mathsf{eEnc}_t^{n_s}\left((x_i)_{i\in\mathcal{I}}\right) = \left(\mathsf{eEnc}_t^{n_s}(x_i)\right)_{i\in\mathcal{I}}$, respectively $\mathsf{eDec}_t^{n_s}\left((\mathbf{X}_i)_{i\in\mathcal{I}}\right) = \left(\mathsf{eDec}_t^{n_s}(\mathbf{X}_i)\right)_{i\in\mathcal{I}}$.

The shares must be t-wise independent of X. Enc and Dec have to be performed in a secure environment not susceptible to leakage, *i.e.*, gadgets and masked implementations expect all sensitive inputs and internals to be (fresh and independently) encoded before execution starts such that adversaries cannot observe leakage of unmasked sensitive values. Security scales with the security order whereas performance cost largely grows with the num-

ber of shares. Hence, a general design intent is to optimize towards the least amount of shares for a given security order, *i.e.*, $n_s = t + 1$, which we assume unless explicitly stated otherwise.

First-order masking is the case where $t = 1$. Higher-order masking considers the case where $t > 1$ and thus at least three shares are used.

We revisit common masking schemes for $n_s = t + 1$:

Boolean masking [Cha+99; ISW03; RP10] operates in fields $\mathbb{F}_2^k$ of $k \geq 1$ bits and relies on the bitwise logical XOR:

$$\mathsf{BEnc}_{k,t}^{t+1}(x) := \left(X^{(i)_B} \xleftarrow{\$} \mathbb{F}_2^k\right)_{i\in[t]} || \left(X^{(t)_B} = x \oplus \bigoplus_{i\in[t]} X^{(i)_B}\right) \quad (2.6)$$

$$\mathsf{BDec}_t^{n_s}\left(\left(X^{(i)_B}\right)_{i\in[n_s]}\right) := \bigoplus_{i\in[n_s]} X^{(i)_B} = x \quad (2.7)$$

Arithmetic masking operates in fields $\mathbb{F}_k$, respectively $\mathbb{Z}_q$, and is based on arithmetic addition:

$$\mathsf{AEnc}_{k,t}^{t+1}(x) := \left(X^{(i)_A} \xleftarrow{\$} \mathbb{F}_k\right)_{i\in[t]} || \left(X^{(t)_A} = x - \sum_{i\in[t]} X^{(i)_A} \pmod{k}\right) \quad (2.8)$$

$$\mathsf{ADec}_{k,t}^{n_s}\left(\left(X^{(i)_A}\right)_{i\in[n_s]}\right) := \sum_{i\in[n_s]} X^{(i)_A} \pmod{k} = x \quad (2.9)$$

Frequently, the considered field sizes are a power of two but the NIST PQC standardization amplified research on masking with prime modulus k since operations in some PQC schemes are performed using a prime modulus [Sch+19; Bos+21; Fri+21].

Polynomial masking also known as *Secret Shamir Sharing* is a special case where the sensitive value is encoded in a random polynomial $f(x) = f_0 + \sum_{i=1}^{t} F_i x^i$ of degree t with coefficients $F_1, ..., F_t$ sampled uniformly i.i.d. at random in $\mathbb{F}_k$. Coefficient f_0 equals the sensitive value to be masked and the shares correspond to the evaluations of the polynomial at fixed and pairwise different *support points* $\alpha_0, ..., \alpha_{n_s-1}$. The encoded value can be retrieved from the shares using polynomial interpolation, requiring at least $t + 1$ shares. Polynomial sharing is attractive as it allows to protect against combined SCA and FA by choosing redundant shares, *i.e.*, $n_s \geq t + 1$ [Sek+18; Ber+23]. In contrast to linear sharing

schemes such as Boolean and arithmetic masking the redundant shares allow to detect if a computation has been tampered.

The presented masking schemes are actively used in protecting software implementations and are sufficient to evaluate the contributions of this thesis.

For each encoding, dedicated gadgets have to be designed to be able to compute operations in a masked manner. The choice of encoding and available gadgets determines the overhead involved in masking certain operations. For example, it is straightforward and efficient to mask a Boolean operation like XOR using Boolean masking but the same task becomes less obvious when the inputs and outputs are masked using arithmetic masking. Secure *conversions* between many masking schemes exist (in the form of gadgets implementing the identity function), which allows one to switch between schemes without revealing the secret. Which encoding to choose is up to the designer and usually depends on the application to be protected and the available gadgets.

2.4.3. Gadgets and Probing Security

Gadgets are central in masking as they represent the basic building blocks to compose applications. Research continuously proposes improved gadgets or new gadgets for the protection of custom operations or multiple operations at once. This allows masking applications more efficiently. In Chapter 4 two custom gadgets are designed for the PQC scheme Kyber. Here, we give insight into the principles and goals in designing gadgets that are tightly linked to formal security notions such as probing security, which we explain too.

Gadgets are abstract algorithms or implementations thereof that compute some function $f(\cdot)$ on the shares of one or more encoded inputs and return the shares of one or more encoded outputs. Gadgets have to be *functionally correct*, *i.e.*, compute the shares of each output such that they encode the correct result as well as *secure*, *i.e.*, adhere to a specific security definition, restricting the possible ways of computing on the shares. A formal definition of gadgets is given in Definition 2.4. This definition extends the original notions of Ishai, Sahai and Wagner (ISW) by adding public inputs and outputs to gadgets. In Chapter 3 new security notions are introduced which use the additional in- and outputs for modeling the state of masked implementations (e.g., non-volatile memory).

Definition 2.4 (Gadget G and Computation Model $[\![\mathsf{G}]\!]^{\text{GAD}}$)**.** A gadget G is a specification for computing outputs $\mathbf{Y}, \mathbf{O}$ and intermediates $\mathbf{Z}$ from inputs $\mathbf{X}, \mathbf{R}, p$. It operates on

- an encoded input $\mathbf{X} = \mathsf{Enc}_t^{n_s}(x)$ encoding an $n_{\mathbf{X}}$-tuple of secrets $x \in \mathcal{X}_{\mathsf{G}}^{n_{\mathbf{X}}}$,

Algorithm 2.1 $\mathsf{maskedXor}$ gadget.

Input: Boolean shares $\left(X^{(i)} \in \mathbb{F}_2^k\right)_{i\in[n_s]}$, with $\mathsf{BDec}_t^{n_s}\left(\left(X^{(i)}\right)\right) = x$ and $\left(Y^{(i)} \in \mathbb{F}_2^k\right)_{i\in[n_s]}$ with $\mathsf{BDec}_t^{n_s}\left(\left(Y^{(i)}\right)\right) = y$.

Output: Boolean shares $\left(Z^{(i)} \in \mathbb{F}_2^k\right)_{i\in[n_s]}$ s.t. $\mathsf{BDec}_t^{n_s}\left(\left(Z^{(i)}\right)\right) = x \oplus y$.

1: **for** $i = 0$ **to** $n_s - 1$ **do**
2: $\quad Z^{(i)} \leftarrow X^{(i)} \oplus Y^{(i)}$
3: **return** $\left(Z^{(i)}\right)_{i\in[n_s]}$

- a $n_{\mathbf{R}}$-tuple of uniformly i.i.d. random values $\mathbf{R} \in \mathcal{R}_{\mathsf{G}}^{n_{\mathbf{R}}}$, and
- a n_p-tuple of public inputs $p \in \mathcal{P}_{\mathsf{G}}^{n_p}$,

where $\mathcal{X}_{\mathsf{G}}^{n_{\mathbf{X}}}, \mathcal{R}_{\mathsf{G}}^{n_{\mathbf{R}}}, \mathcal{P}_{\mathsf{G}}^{n_p}, n.$ and t are specific to G. It defines for $\mathcal{O}_{\mathsf{G}}^{n_{\mathbf{O}}}, \mathcal{Y}_{\mathsf{G}}^{n_{\mathbf{Y}}}$ specific to G

- a z-tuple $\mathbf{Z} = \left(Z_i = f_{Z_i}\left(\mathbf{E}_{Z_i}\right) \mid \mathbf{E}_{Z_i} \subseteq \left(\mathbf{X}, \mathbf{R}, p, ||_{j\in[i]} Z_j\right)\right)_{i\in[z]}$ of intermediate computation results where f_{Z_i} is an arithmetic or Boolean operation and $\mathbf{E}_{Z_i}$ are the respective *operands*,
- a $n_{\mathbf{O}}$-tuple of public outputs $\mathbf{O} \in \mathcal{O}_{\mathsf{G}}^{n_{\mathbf{O}}}$, and
- a tuple of masked outputs $\mathbf{Y} = \left(Y_i^{(j)}\right)_{i\in[n],j\in[n_s]}$ encoding a $n_{\mathbf{Y}}$-tuple of secret outputs $y \in \mathcal{Y}_{\mathsf{G}}^{n_{\mathbf{Y}}}$ where $Y_i^{(j)} = Z_{z_{i,j}}$ for $z_{i,j} \in [z]$.

The noiseless BB computation model $[\![\mathsf{G}]\!]^{\text{GAD}}_{\mathbf{X},\mathbf{R},p} \xmapsto{\mathbf{L}^{\text{GAD}}} (\mathbf{Y}, \mathbf{O})$ produces outputs according to above definition and an empty trace $\mathbf{L}^{\text{GAD}}$.

Consider as a first guiding example a gadget computing the XOR operation $f(x, y) = x \oplus y = z$ on shares $X^{(i)} = \mathsf{BEnc}_{1,1}^2(x)$ and $Y^{(i)} = \mathsf{BEnc}_{1,1}^2(y)$ encoding secret inputs $x, y \in \mathbb{F}_2$ in two shares each. A first-order gadget has to compute output shares $Z^{(0)}, Z^{(1)} \in \mathbb{F}_2$ encoding the result, *i.e.*, $z = x \oplus y = \mathsf{BDec}_{1,1}^2\left(Z^{(0)}, Z^{(1)}\right)$. A functionally correct solution is to compute output share $Z^{(0)} = X^{(0)} \oplus Y^{(0)}$, respectively $Z^{(1)} = X^{(1)} \oplus Y^{(1)}$, and to output both, constituting a first-order $\mathsf{maskedXor}$ gadget. Functions which are affine over the encoding can be applied *share-wise*, e.g., $Z^{(i)} = f\left(X^{(i)}, Y^{(i)}\right)$ for $i \in [n_s]$. A generic algorithm for arbitrary values of t is depicted in Algorithm 2.1.

Functional correctness follows immediately.

$$\begin{aligned}\mathsf{BDec}^2_{1,1}\left(\left(Z^{(i)}\right)_{i\in[2]}\right) &= Z^{(0)} \oplus Z^{(1)} = \left(X^{(0)} \oplus Y^{(0)}\right) \oplus \left(X^{(1)} \oplus Y^{(1)}\right)\\ &= \left(X^{(0)} \oplus X^{(1)}\right) \oplus \left(Y^{(0)} \oplus Y^{(1)}\right)\\ &= \mathsf{BDec}^2_1\left(\left(X^{(i)}\right)_{i\in[2]}\right) \oplus \mathsf{BDec}^2_1\left(\left(Y^{(i)}\right)_{i\in[2]}\right)\\ &= x \oplus y = z = f\left(x, y\right)\end{aligned} \tag{2.10}$$

It remains to show that the $\mathsf{maskedXor}$ gadget satisfies a notion of side-channel security.

In their seminal work Ishai, Sahai, and Wagner introduce the formal security notion *perfect privacy* and the *probing model* to prove resistance against higher-order SCA for their gadget constructions [ISW03]. In the probing model, the *threshold probing adversary* is allowed to observe any set of t intermediate values computed by a circuit, motivated by a physical adversary putting measurement needles on the electrical wires of a hardware implementation. More formally, a gadget's circuit is represented as directed acyclic graph (DAG), and the probing adversary, knowing the circuit structure, chooses the values of public inputs and a set of edges (representing wires) she wants to learn and subsequently retrieves the value of each edge resulting from evaluating the circuit under given inputs. Based on this we define a computation model (Definition 2.5) where the leakage corresponds to the plain intermediate values, without any perturbation by noise.

Definition 2.5 (ISW Probing Model $[\![\mathsf{G}]\!]^{\text{ISW}}$)**.** The *ISW probing model* is a noiseless computation model $[\![\mathsf{G}]\!]^{\text{ISW}}$ and extends $[\![\mathsf{G}]\!]^{\text{GAD}}$ (Definition 2.4) by defining the leakage $\mathbf{L}^{\text{ISW}}$ of computation $[\![\mathsf{G}]\!]^{\text{ISW}}_{\mathbf{X},\mathbf{R},p}$ as the z-tuple of intermediates Z of gadget G

$$\mathbf{L}^{\text{ISW}} = \left(L_i = Z_i\right)_{i\in[z]}.$$

The definition which is today associated with *probing security* is given by Rivain and Prouff in [RP10] w.r.t. the probing model of [ISW03] but denoted *t-th order SCA security.* Definition 2.6 is adapted to define probing security relating to the leakage of any computation model $[\![\cdot]\!]^m$, corresponding to $[\![\mathsf{G}]\!]^{\text{ISW}}$ unless noted. Remark that the computation is treated as a randomized algorithm defining distributions of $\mathbf{L}^m, \mathbf{Y}, \mathbf{O}$ as a function of random variables passed as input and internal noise, if any.

Definition 2.6 (t^{th}-order Probing Security of G in $[\![\mathsf{G}]\!]^m$)**.** A gadget G executed in model $[\![\cdot]\!]^m$ for randomly distributed secret $X \in \mathcal{X}^{n_\mathrm{X}}_{\mathsf{G}}$ encoded as input $\mathbf{X} = \mathsf{Enc}\left(X\right)$ and randomly distributed public input $p \in \mathcal{P}^{n_p}_{\mathsf{G}}$ with

$$[\![\mathsf{G}]\!]^m_{\mathbf{X},\mathbf{R},p} \xmapsto{\mathbf{L}^m} \left(\mathbf{Y}, \mathbf{O}\right)$$

is t probing secure iff $\forall \mathbf{E} \subseteq \mathbf{L}^m$ with $|\mathbf{E}| \leq t$ tuple $\mathbf{E}$ is independent of X.

In the case of the exemplary maskedXor gadget it is straightforward to show that every set of $t = 1$ leakages $\mathbf{L}^{\text{ISW}}$ of $[\![\mathsf{maskedXor}]\!]^{\text{ISW}}$ is independent of the secrets X and Y. This is because of the underlying uniformly i.i.d. random mask. The distribution of $X^{(1)} = X \oplus X^{(0)}$ is independent of X because of the random mask $X^{(0)}$ and the operation $\oplus$ which is a bijection in $\mathbb{F}_2$. It follows immediately that observing any physical leakage $f_{phy}\left(X^{(1)}\right)$, which is a function of $X^{(1)}$ only, e.g., the HW, provides no information on secret X. Similar reasoning holds for $X^{(0)}, Y^{(0)}, Y^{(1)}, Z^{(0)}, Z^{(1)}$.

We conclude that the sensitive values X, Y and Z cannot be learned by recovering t or less leakages of $\mathbf{L}^{\text{ISW}}$. Ensuring the independence of t-tuples is one of the core principles in designing gadgets and is often performed in a principled fashion with hand-written proofs that are valid for any order t. Probing security and $[\![\mathsf{G}]\!]^{\text{ISW}}$ are well suited for the design of gadgets as the leakages that need to be considered exactly correspond to the intermediates, easing to prove constructions alongside functional design.

We continue with maskedAnd, a gadget of the function $f(x, y) = x \wedge y = z$. Non-linear operations are more involved as multiple shares of an input need to be combined to compute the correct output. A first-order maskedAnd must compute $Z^{(0)}, Z^{(1)}$ such that:

$$\begin{aligned} Z^{(0)} \oplus Z^{(1)} = x \wedge y \quad &= \left(X^{(0)} \oplus X^{(1)}\right) \wedge \left(Y^{(0)} \oplus Y^{(1)}\right) \\ &= X^{(0)}Y^{(0)} \oplus X^{(0)}Y^{(1)} \oplus X^{(1)}Y^{(0)} \oplus X^{(1)}Y^{(1)}. \end{aligned} \tag{2.11}$$

For any functionally correct gadget is it not possible to split the product terms $X^{(i)}Y^{(j)}$ such that both output shares $Z^{(0)}, Z^{(1)}$ are computed from just one share of each input, which is due to the cross-products $X^{(0)}Y^{(1)}$ and $X^{(1)}Y^{(0)}$. Any sum depending on both shares of one input would leak information in $[\![\mathsf{maskedAnd}]\!]^{\text{ISW}}$ on the secret input value. Non-linear gadgets therefore require the use of additional random values to mask the combination of multiple shares of one encoding. A probing secure variant is presented in Algorithm 2.2 adapted from [RP10]. This algorithm is known as the ISW *multiplication* gadget of [ISW03], which was extended to larger fields by Rivain and Prouff [RP10]. In Line 5 fresh, i.i.d. masks are sampled to protect the intermediate computation results starting from Line 7.

For the first-order implementation with $n_s = t + 1 = 2$ all leaking inputs, intermediates, and outputs have to be considered. The leakages $X^{(0)}$, $X^{(1)}$, $Y^{(0)}$, $Y^{(1)}$, $X^{(0)} \wedge Y^{(0)}$, $X^{(1)} \wedge Y^{(1)}$, $X^{(0)} \wedge Y^{(0)} \oplus R_{0,1}$, $X^{(1)} \wedge Y^{(1)} \oplus R_{0,1}$ until Line 6 as well as the cross-products $X^{(0)} \wedge Y^{(1)}$, $R_{0,1} \oplus X^{(0)} \wedge Y^{(1)}$ and $X^{(1)} \wedge Y^{(0)}$, $R_{i,j} \oplus X^{(1)} \wedge Y^{(0)}$ in Line 7 depend on at most one share

Algorithm 2.2 maskedAnd gadget.

Input: Boolean shares $\left(X^{(i)} \in \mathbb{F}_2^k\right)_{i\in[n_s]}$ with $\mathsf{BDec}_t^{n_s}\left(\left(X^{(i)}\right)\right) = x$ and $\left(Y^{(i)} \in \mathbb{F}_2^k\right)_{i\in[n_s]}$ with $\mathsf{BDec}_t^{n_s}\left(\left(Y^{(i)}\right)\right) = y$.

Output: Boolean shares $\left(Z^{(i)} \in \mathbb{F}_2^k\right)_{i\in[n_s]}$ s.t. $\mathsf{BDec}_t^{n_s}\left(\left(Z^{(i)}\right)\right) = x \wedge y$.

1: **for** $i = 0$ **to** $n_s - 1$ **do**
2: $\quad Z^{(i)} \leftarrow X^{(i)} \wedge Y^{(i)}$
3: **for** $i = 0$ **to** $n_s - 1$ **do**
4: $\quad$ **for** $j = i + 1$ **to** $n_s - 1$ **do**
5: $\quad\quad R_{i,j} \xleftarrow{\$} \mathbb{F}_2^k$
6: $\quad\quad Z^{(i)} \leftarrow Z^{(i)} \oplus R_{i,j}$
7: $\quad\quad R \leftarrow \left(R_{i,j} \oplus X^{(i)} \wedge Y^{(j)}\right) \oplus X^{(j)} \wedge Y^{(i)}$
8: $\quad\quad Z^{(j)} \leftarrow Z^{(j)} \oplus R$
9: **return** $\left(Z^{(i)}\right)_{i\in[n_s]}$

of each input. Hence, the distribution of these leakages can be perfectly simulated by returning a random value sampled uniformly from $\mathbb{F}_2$. The remaining leakages $\left(R_{0,1} \oplus \left(X^{(0)} \wedge Y^{(1)}\right)\right) \oplus \left(X^{(1)} \wedge Y^{(0)}\right)$ and $X^{(1)}Y^{(1)} \oplus \left(\left(R_{0,1} \oplus \left(X^{(0)} \wedge Y^{(1)}\right)\right) \oplus \left(X^{(1)} \wedge Y^{(0)}\right)\right)$ in Line 7 onward are a linear combination involving $R_{0,1}$ additively, meaning that the individual probability distribution of each of these intermediates can be perfectly simulated by returning a random value sampled uniformly from $\mathbb{F}_2$ and concluding the independence of X and Y. At higher orders, multiple random variables and t-tuples of intermediates need to be considered. In [ISW03; RP10] respective proofs are provided which enumerate and examine every tuple systematically. However, this is unpractical for software development and can be performed by automated verification tools instead.

The multiplication gadget has an asymptotic complexity $\mathcal{O}\left(n_s^2\right)$. The asymptotic performance of gadgets implementing function f can be $\mathcal{O}\left(n_s\right)$ or lower if f is affine over the encoding, *i.e.*, $f\left(x\right) = \mathsf{Dec}\left(f\left(X^{(0)}\right), \ldots, f\left(X_{n_s-1}\right)\right)$. The complexity of Algorithm 2.1 is $\mathcal{O}\left(n_s\right)$, a logical negation on Boolean shares can be as low as $\mathcal{O}\left(1\right)$ [ISW03].

Multiple multiplication gadgets have been proposed, some are more efficient, satisfy different security notions, reduce the randomness consumption, or have better practical SCA characteristics, supporting arbitrary security orders and have been proven by pen-and-paper [ISW03; Bel+16; CS18; Bat+16; CS19; Bar+16]. Research on new gadgets is often but not necessarily connected to masking a concrete application, as generic gadgets can be reused among

different applications.

In general, there can be multiple gadgets implementing the same function but under different encoding or with different properties and optimizations. In [Cor+14] a monolithic gadget computing $a \wedge g(a)$ for any linear function g is presented, avoiding refreshing before the multiplication and thus being more efficient than a composition of individual gadgets. Similarly, a core routine in the PQC key-exchange Kyber can be more efficiently masked using custom gadgets and the works [Fri+21; Bos+21; Cor+21] each propose a different variant with varying performance characteristics at first- and higher-order.

Thus, there is a great variety of gadgets that may need to be implemented securely, depending on the application to be protected. Choosing the right set of gadgets to mask an application is a non-trivial design problem considered during composition.

2.4.4. Secure Composition and Non-Interference

Secure composition is the process of systematically selecting and connecting gadgets such that they securely and correctly implement the functionality of an unprotected reference circuit. The process relies on the security properties of the used gadgets that have been proven beforehand. Circuit compilers are algorithmic representations of this process, often proven to produce secure and correct masked circuits for specific notions of security and gadgets [ISW03; RP10; DDF19; Bar+16].

The seminal work [ISW03] provides a simple circuit compiler; by representing a reference circuit in $\mathbb{F}_2$ with field operations $\oplus$ and $\wedge$ only it is sufficient to replace any operation in the algorithm simply with the respective gadget of [ISW03]. However, to achieve t probing security each input to the circuit must be encoded using $n_s = 2t + 1$ shares and in such a way that the shares are $2t$-wise independent [ISW03]. This choice of encoding causes a significant increase in the masked circuit size, operation count, and randomness consumption.

Rivain and Prouff reduce the requirement to the least number of shares and therefore improve the performance, but at the cost of an additional assumption on the maskedAnd [RP10]. Namely, a circuit following the composition strategy of [ISW03] but using $n_s = t + 1$ shares instead is t probing secure if for all gadgets with encoded inputs $\left(X_i^{(\cdot)}\right)_{i \in \mathcal{I}}$ each pair $i, i' \in \mathcal{I}, i \neq i'$ of *encodings is independent*, *i.e.*, $\forall \mathcal{J}, \mathcal{J}' \subset [n_s], |\mathcal{J}| \leq |\mathcal{J}'| \leq t$ the tuple of $2t$ shares $\left(X_i^{(j)}, X_{i'}^{(j')}\right)_{j \in \mathcal{J}, j' \in \mathcal{J}'}$ is t-wise independent. The property ensures that each random mask in each masked operand must be uniformly distributed

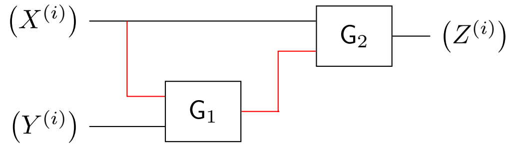


Figure 2.5.: Composition of two gadgets where $(X^{(i)})$ is used in both gadgets. If G_1 is t–SNI this composition is secure, otherwise it might be insecure.

and not used in the other input's shares. In case the circuit does not satisfy this property the operand must be *refreshed* by securely re-encoding it.

Many applications re-use intermediates in multiple operations, as depicted in Figure 2.5, and adhering to this property proved to be difficult. Many pitfalls exist, e.g., in [RP10] a pen-and-paper proof for a securely composed t probing secure AES is presented but later shown to achieve $\lceil \frac{t}{2} \rceil + 1$ order security only due to a composition flaw [Cor+14]. Consider as an example, that the above 1st-order multiplication gadget (Algorithm 2.2) retrieves as input $X^{(0)} = X \oplus X^{(1)}$ with $X^{(1)}$ sampled uniformly from random and dependent share $Y^{(1)} = X^{(1)}$ of the second input Y re-using this randomness, *i.e.*, $Y^{(0)} = Y \oplus Y^{(1)}$. In this case, the first intermediate computed in Line 7 would resolve to $X^{(0)} \wedge Y^{(1)} = X^{(0)} \wedge X^{(1)}$, depending on X, breaking the probing security of the composition.

In these cases, it is necessary to refresh the encoding before re-use with the help of a special *refresh* gadget [RP10]. Refresh gadgets implement the identity function but internally apply fresh randomness to re-encode the masked value such that encodings of input and output are independent, as defined above. Refresh gadgets are detrimental to performance and should be avoided unless necessary. Deciding where to insert and where to omit a refresh gadget requires taking into account the overall circuit and poses an additional, non-trivial design problem.

Barthe et al. propose in [Bar+15; Bar+16] the security notions non-interference (NI) and strong-non-interference (SNI) which simplify composition. The notions reformulate threshold probing security into a probabilistic variant of the relational *non-interference* property [GM82]. The probabilistic property requires the distribution of a gadget's leakage to be equal when executed twice with different values for secrets, *i.e.*, that secrets do not interfere with the distribution of side-channel leakage. Specifically, for each $\mathbf{E} \subseteq \mathbf{L}$ of up to t leakages the existence of a set of at most t shares of each encoded secret is mandated, s.t. the probability distribution of the leakages is equal for

(and hence independent of) any choice of the remaining input shares. This proves that the value of encoded secrets does not interfere with the leakage, or vice-versa, that the encoded secrets are independent of the leakage.

The notions can be equivalently expressed as simulation-based properties by requiring a simulator or simulation procedure that simulates the probability distribution of every subset of t leakages given a subset of at most t input shares. The notions can be checked mechanically with the help of verification routines in an automated manner, though the simulation-based property is the preferred style for written security proofs of gadgets [Bar+15; Bar+16]. We state the notions based on a simulator as presented in [BCZ18; Cor17] in Definition 2.7 and 2.8. These are restricted to single input, single output gadgets for simplicity. NI is equivalent to threshold probing security (Definition 2.6) but can be extended for multilevel security circuits with public in- and outputs, which is necessary for concrete implementations. An extension of t–SNI to multiple outputs denoted multiple-input-multiple-output-strong-non-interference (MIMO-SNI) is presented in [CS20]. In Section 3.2.1 the *Stateful* t–$(\mathsf{S})\mathsf{NI}$ extensions are introduced which capture the stateful execution on processors.

Definition 2.7 (t–NI of G in $[\![\mathsf{G}]\!]^m$)**.** Let G be a gadget taking as input shares $\left(X^{(i)}\right)_{i\in[n_s]}$ with leakage $\mathbf{L}^m$ defined by computation model $[\![\mathsf{G}]\!]^m$. Gadget G is t–NI iff for every set consisting of $t_{\mathsf{G}} \leq t$ leakages of $\mathbf{L}^m$, denoted *observations*, there exists a subset $\mathcal{I} \subset [n_s]$ of input indices with $|\mathcal{I}| \leq t_{\mathsf{G}}$, such that all observations can be perfectly simulated from $\left(X^{(i)}\right)_{i\in\mathcal{I}}$.

The notions enable systematic arguments about the security of a composition: a threshold probing adversary can distribute probes among all gadgets but in total at most t probes and $t_i \leq t$ probes on each t–$(\mathsf{S})\mathsf{NI}$ gadget can be simulated knowing at most t_i input shares of the respective gadget. Hence, it is sufficient to check that for any encoded intermediate at most t shares are needed. These are secret independent for any t-wise independent encoding. Unlike prior approaches this permits to iterate backward through a masked circuit, starting from the outputs and to check that the sum of shares which are required to simulate the leakage of each gadget computing on the shares does not exceed t.

t–SNI implies t–NI but is stronger as it ensures that output shares are t-wise independent of the inputs by mandating that output shares can be simulated without access to input shares. This is beneficial for composition as the probes on output shares (and subsequent gadgets) do not add to the sum of required input shares, hence the propagation of probes can be stopped with t–SNI gadgets. By reasoning about the structure of a composition (*i.e.*, how gadgets are connected) and the properties the gadgets fulfill, but not relying on particular

constructions, one can deduce the security of the entire composition without the need to consider the exponential number of combinations of probes within two or more gadgets. Building threshold probing secure circuits thus becomes significantly easier.

Definition 2.8 (t–SNI of G in $[\![\mathsf{G}]\!]^m$)**.** Let G be a gadget taking as input shares $\left(X^{(i)}\right)_{i\in[n_s]}$ and outputting shares $\left(Y^{(i)}\right)_{i\in[n_s]}$ with leakage $\mathbf{L}^m$ defined by computation model $[\![\mathsf{G}]\!]^m$. Gadget G is t–SNI iff for every set consisting of $t_{\mathsf{G}} \leq t$ leakages of $\mathbf{L}^m$, denoted *observations*, and any set $\mathcal{O} \subset [n_s]$ of output indices, s.t. $t_{\mathsf{G}} + |\mathcal{O}| \leq t$, there exists a subset $\mathcal{I} \subset [n_s]$ of input indices with $|\mathcal{I}| \leq t_{\mathsf{G}}$, s.t. all observations and the output shares $\left(Y^{(i)}\right)_{i\in\mathcal{O}}$ can be perfectly simulated from $\left(X^{(i)}\right)_{i\in\mathcal{I}}$.

The gain in ease of composition comes with a loss in accuracy since gadgets in general can be threshold probing secure but not t–SNI, hence a circuit might not be securely composable without the t–SNI property yet satisfy t–NI. Vice versa, composing a t–NI circuit might require the usage of t–SNI gadgets and refreshing when following the t–$(\mathsf{S})\mathsf{NI}$ composition rules and thus result in an artificially increased cost in randomness or performance. In [Bel+20b] an automated solution to fix the inaccuracy is provided, resulting in masked circuits without any overhead from needless t–SNI gadgets such as refreshes. However, its tightness property only holds when using a predefined set of gadgets, limiting adoption to other use cases and leveraging purposely built gadgets such as those introduced for Kyber in Chapter 4.

Cassiers and Standaert introduce the alternative PINI notion to t–$(\mathsf{S})\mathsf{NI}$ which allows arbitrary compositions of gadgets, requiring essentially no effort or expertise to prove the security of a masked application [CS20]. However, the effort to prove PINI and the performance cost for adhering to this notion for non-linear gadgets is again higher than for t–SNI. Threshold implementation (TI) is a special case focusing on hardware implementations that use a non-optimal number of shares $n_s > t + 1$ in conjunction with additional requirements on the gadgets. TI gadgets enjoy an inherent resilience against some physical leakages of hardware circuits and a partial resistance against FA by trading-off performance.

Another potential performance issue is the encoding for shares. The chosen encoding determines which gadgets can be composed without converting between encodings which often introduces significant overhead [Sch+19; Cor+21]. As each gadget requires a specific input encoding and enforces a specific output encoding it is necessary to make a choice during composition. Determining the performance (penalty) of different approaches is non-trivial and may require evaluating and benchmarking different compositions.

The randomness consumption poses a dedicated challenge as the generation of good entropy is computationally intense and can only in part be circumvented by efficient gadgets. At the level of an application, it is possible to reuse randomness securely as analyzed by Faust et al. [Fau+18]. However, the reuse of randomness is easy to get wrong in larger applications as the sampled values have to be cached and used exactly at the intended occasions without accidental uses. Accidental reuse of randomness cannot be detected by functional testing and thus requires specialized testing effort or formal verification to be detected.

The process of applying the masking countermeasure to produce protected implementations can be automated by implementing circuit compilers. The tools presented in [Mos+12; Abr+21] produce executable implementations whereas other tools like TORNADO and MASKCOMP extend to higher-order masking [Bar+16; Bel+20b]. However, the fixed nature limits the potential for optimization, e.g., by leveraging the structure of a circuit or specialized gadgets, which is why manual composition remains in widespread use. Despite the provable security of circuit compilers it remains an important activity to verify the security of compositions. Both manual composition by humans, as well as implementations automating the composition process, may be flawed and result in insecure circuits. Software implementations frequently inline gadgets and reorder computation steps to minimize memory accesses and register usage.

Formal security notions are not only inherent to gadgets but are important for composing applications from gadgets too. Proofs mandate additional work during gadget design but also represent an auditable argumentation chain and (ideally) a clear transcript of assumptions that can be checked independently at any point, which is in stark contrast to physical evaluations performed in specific settings. Stricter notions also ease the application of masking as gadgets are proven once and their use during composition then, e.g., in the case of PINI, does not require additional proof work.

2.5. Verification of Physical Side-Channel Security

Our inherent goal is to use verification for establishing rigorous security against the physical adversary $\mathsf{Adv}^{i}_{n_{poi} \leq t}$ interacting with a device, as in Eq. (2.3).

We discuss our contributions and the role of the models, security notions, and the masking order in protecting against physical attacks. In the remainder, program P is a functionally equivalent implementation of gadget G, and in all computation models P and G yield the same outputs given the same inputs.

2.5.1. Leakage Models and Hardened Masking

Verification permits to prove t–$(\mathsf{S})\mathsf{NI}$ in the ISW probing model $[\![\mathsf{G}]\!]^{\text{ISW}}$. This ensures the perfect simulation of all sets of t leakages of $\mathbf{L}^{\text{ISW}}$ given only BB access. Consequently, for any restricted adversary $\mathsf{Adv}^{i}_{n_{poi} \leq t}$ there exists a simulator S perfectly simulating the adversary's inputs for any number of invocations n_t with

$$\forall \; \mathsf{Adv}^{i}_{n_{poi} \leq t} \; \exists \; S : \mathsf{SD}\left(out \left(S \rightleftharpoons [\![\mathsf{G}]\!]^{\text{ISW}} \right); out \left(\mathsf{Adv}^{i}_{n_{poi} \leq t} \rightleftharpoons [\![\mathsf{G}]\!]^{\text{ISW}} \right) \right) = 0 \tag{2.12}$$

provided that G is t–SNI in $[\![\mathsf{G}]\!]^{\text{ISW}}$ and the next round input $\mathbf{X}_{j+1} \subseteq \mathbf{Y}_j$ in round $j \in [n_t]$ is encoded with $n_s \geq 2t$ shares which are $2t$-wise independent and the sole output of a $\mathsf{Refresh}$ gadget. The latter condition ensures that probes of round $j+1$ can be simulated in addition to t probes in round j. Remark that this condition does not require to mask the entire circuit at order $2t$ but only the accesses to $\mathbf{X}$. We omit a proof and refer to [DDF19] for a proof in a model similar to $[\![\mathsf{G}]\!]^{\text{ISW}}$. Hence, $\mathsf{Adv}^{i}_{n_{poi} \leq t}$ has no advantage in observing side-channel.

However, the leakage $\mathbf{L}^{\text{ISW}}$ modeled in $[\![\mathsf{G}]\!]^{\text{ISW}}$ and many others, such as the OCL of [MR03] are too restricted and do not sufficiently cover physical leakages $\mathbf{L}^{\text{PHY}}$, see e.g., [Cor+12]. A simulation of $\mathsf{Adv}^{\text{PHY}}_{n_{poi} \leq t} \rightleftharpoons [\![\mathsf{G}]\!]^{\text{PHY}}$ would fail for any n_{poi} physical leakages $\mathbf{L}^{\text{PHY}}$ which cannot be simulated from t leakages in $\mathbf{L}^{\text{ISW}}$.

For example, in a software implementation P of $\mathsf{maskedXor}$ (Algorithm 2.1) the two outputs $Z^{(0)}, Z^{(1)}$ may be computed using the same register `rD` of the processor architecture. During the computation of $[\![\mathsf{P}]\!]^{\text{PHY}}$ it thus may be the case that `rD` contains $Z^{(0)}$ but transitions to $Z^{(1)}$. The underlying electric circuit will consequently alter the physical charges depending on the values of $Z^{(0)}$ and $Z^{(1)}$ which is well observable in side-channel measurements. Hence, the physical leakage $\mathbf{L}^{\text{PHY}}$ allows to observe a *transition leakage*, *i.e.*, the HD between the initial and subsequent value of `rD` corresponding to $Z^{(0)}, Z^{(1)}$. The resulting difference between the physical and modeled leakage is a *gap* in $[\![\cdot]\!]^{\text{ISW}}$, respectively any derived security statement.

A central contribution of this work is to lift verification of t–$(\mathsf{S})\mathsf{NI}$ security to executable implementations P and fine-grained, device-specific computation models $[\![\mathsf{P}]\!]^{f}$ which allow capturing rich physical leakage behavior accurately for any device f. Under the same assumptions as for Eq. (2.12) the restricted adversary can be simulated perfectly but for a flexible and more accurate definition of leakages $\mathbf{L}^{f}$

$$\forall \; \mathsf{Adv}^{i}_{n_{poi} \leq t} \; \exists \; S : \mathsf{SD}\left(out \left(S \rightleftharpoons [\![\mathsf{G}]\!]^{f} \right); out \left(\mathsf{Adv}^{i}_{n_{poi} \leq t} \rightleftharpoons [\![\mathsf{G}]\!]^{f} \right) \right) = 0. \tag{2.13}$$

We model the data dependency of physical leakage instead of precisely modeling the leakage functions. The physically observable leakage $L_i^{\text{PHY}} \in \mathbf{L}^{\text{PHY}}$ follows, by assumption, a function $f_i(\mathbf{E}_i, u, \mathbf{N})$ with $\mathbf{E}_i \subset (\mathbf{X}||\mathbf{R}||p)$ being the *explanatory variables* and the auxiliary data u and noise $\mathbf{N}$ of the computation model $[\![\mathsf{P}]\!]^{\text{PHY}}$ are independent of $\mathbf{X}, \mathbf{R}, p$. This assumption is backed by other works [CRR03; PR13; SLP05]. The intention behind fine-grained models is to define leakages $\mathbf{L}_j^f \in \mathbf{L}^f$ corresponding to the implementation and device-specific explanatory variables $\mathbf{E}_i$. We do not model the physical leakage function f_i and instead assure, with the help of verification, that all sets of t explanatory variables are independent of secrets. This is comparable to the RPM where adversaries probe n-tuples of intermediates, referred to as *extended probes* for $n > 1$ [ISW03; Fau+18]. However, our explanatory variables extend over algorithmic intermediates and incorporate additional device or processor-specific variables (more on this in Chapter 3).

The ideal of modeling is *exhaustive completeness*, outlined in Definition 2.9, where all physical leakage can be modeled from at most one modeled leak. Consequently, simulator S in Eq. (2.13) exists, albeit with small error probability, and we say that t–(S)NI security is *preserved*. Fine-grained models aim for *empirical completeness*, where Eq. (2.14) is validated for some P empirically and conjectured to others.

Definition 2.9 (Completeness of $[\![\mathsf{C}]\!]^m$ for $[\![\mathsf{C}]\!]^{\text{PHY}}$)**.** Computation model $[\![\mathsf{C}]\!]^m$ models the leakage of $[\![\mathsf{C}]\!]^{\text{PHY}}$ *completely* and is said to be *complete* if for any computation C in the language of $[\![\mathsf{C}]\!]^{\text{PHY}}$ with computations on equal inputs and $c \in [0,1]$ negligible

$$[\![\mathsf{C}]\!]^m_{\mathbf{X},\mathbf{R},p} \xmapsto{\mathbf{L}^m = \left(\mathbf{L}_i^m\right)_{i \in \left[n_l^m\right]}} (\mathbf{Y}, \mathbf{O}) \quad \text{and} \quad [\![\mathsf{C}]\!]^{\text{PHY}}_{\mathbf{X},\mathbf{R},p} \xmapsto{\mathbf{L}^{\text{PHY}} = \left(\mathbf{L}_j^{\text{PHY}}\right)_{j \in \left[n_l^{\text{PHY}}\right]}} (\mathbf{Y}, \mathbf{O}),$$

where n_l^m, n_l^{PHY} are the respective number of leakages, it holds that

$$\forall\, \mathbf{L}_j^{\text{PHY}} \in \mathbf{L}^{\text{PHY}}\ \exists\, \mathbf{L}_i^m \in \mathbf{L}^m, f_i\ : \mathsf{SD}\left(\mathbf{L}_i^{\text{PHY}}; f_i\left(\mathbf{L}_j^m, u, \mathbf{N}\right)\right) \le c. \tag{2.14}$$

We stress the benefit of the abstraction; instead of modeling specific leakage functions such as HW, HD, *etc.*, one abstract leak $\mathbf{L}_j^f$ represents any leakage function $\lambda\left(\mathbf{L}_j^f\right)$. Consequently, verification in such a model proves resilience against *any* leakage function $\lambda\left(\mathbf{L}_j^f\right)$. In contrast to emulation-based approaches, our verification approach does not require knowledge of the exact physical leakage functions, removing the need for precise physical characterization.

Hardening is the process of protecting an implementation against device-specific leakage. Software can be hardened by re-ordering instructions, choosing different register allocations, or adding *dummy* instructions. In the above example on maskedXor, allocating $Z^{(1)}$ to a different register or adding an instruction to clear the contents of `rD` permits to mitigate the transition leakage. Additional masks could be introduced or the masking order be increased alternatively. Any change increases the divergence from the blueprint algorithm with proven security. Thus, hardening an implementation should be paired with verification of security.

In Chapter 3 we overcome limitations in the modeling of all prior approaches; we augment BB computation models with dedicated `leak` annotations. These annotations define the leakage points, e.g., `leak` $\{E, E'\}$ defines a leakage point $(E, E') \in \mathbf{L}^f$, where E, E' are intermediate variables in the computation model. The representation is forwarded into a verification tool to assist development and hardening by flagging every t-tuple of leakages that is not independent with precise debug information. The annotations can be freely adopted to cover physical behavior without mandating to change tooling, solving a so-far open engineering problem.

Our concept significantly narrows the gap between verified security and practical resilience for $\mathsf{Adv}^{\text{PHY}}_{n_{poi} \leq t}$. It allows predicting resilience against single- and multi-trace t^{th}-order SCA. This is because our concept enables (I) representing diverse physical leakage behavior $\mathbf{L}^{\text{PHY}}$ as formal models with slight abstractions and reduced assumptions on physical leakage behavior and (II) preserving t–(S)NI security to the physical setting for models with empirical completeness. For the CM0+ processor, we provide an empirically complete model which preserves security in the physical setting. We validate this claim by physically evaluating security against $\mathsf{Adv}^{\text{PHY}}_{n_{poi} \leq t}$ using statistical evaluation on multiple implementations.

To summarize, our contributions ease the protection of applications with the masking countermeasure and extend to higher orders of security which is necessary to thwart ongoing improvements and automation of such attacks. With our contributions applications can be protected against all attacks up to order t by implementing and verifying masking at the optimal order $t = n_s - 1$. The method significantly eases the hardening processes and, due to the fast automated security checks, permits the exploration of optimizations resulting in large efficiency gains.

2.5.2. Contracts and Hardware Models

An issue of our verification approach remains the questionable model quality which renders the preservation of verified security in physical settings

questionable. Also, no systematic approach to the engineering problem of constructing empirically complete models is available. The sole approach being trial-and-error hypothesis testing paired with physical measurements, e.g., [PV17; MOW17; MPW22]. This is problematic for secure software since the leakage behavior of processors might be missed or incorrectly modeled, e.g., special behavior caused by rare stalling, conditional forwarding, or instruction pipelining.

To systematically incorporate processor-specific behavior we consider the *gate-level leakage* $\mathbf{L}^h$ of the processor implementation. The gate-level leakage is determined by a noiseless gate-level computation model $[\![(\mathsf{H}, \mathsf{P})]\!]^h$, where h is the identifier of the used leakage model and H is the netlist of a processor implementation, referred to as *microarchitecture* and P machine-code of an implementation.

We exemplify the methodologies for the RPM, *i.e.*, $h = \textsc{rpm}$ but emphasize the generic nature of our approach. Our approach seamlessly extends to technology-specific gate-level leakage models of technology-mapped netlists with physical layout, *i.e.*, the result after synthesis, technology mapping, and place & route.

Processors and digital hardware circuits can be modeled as labeled directed graphs, we adapt the definition of co-authored work [Blo+22]. A hardware circuit $\mathsf{H} : (\mathcal{G}, \mathcal{W}, \mathcal{L})$ consists of a set of gates $\mathcal{G}$, and a set of wires $\mathcal{W} \subseteq \mathcal{G} \times \mathcal{G}$ connecting the gates. In addition, the labeling $\mathcal{L} : \mathcal{G} \to \mathcal{T}$ defines the (technology specific) type $\tau \in \mathcal{T}$ of each gate $g \in \mathcal{G}$. We distinguish gates for inputs τ_{in}, outputs τ_{out}, registers τ_{reg}, and combinatorial gates, e.g., $\tau_+, \tau_-, \tau_\cdot, \tau_\wedge, \tau_\vee, \tau_\oplus$. Notation $g \in \tau_{\mathsf{x}}$ is a shorthand for $g \in \mathcal{G}\ s.t.\ \mathcal{L}(g) = \tau_{\mathsf{x}}$. For simplicity, we assume that each input and output of a gate corresponds to a single Boolean bit and gates have at most fan-out one. Input gates have fan-in zero and output a value determined by the environment in every cycle. Output gates have fan-out zero and represent an output calculated by the circuit in every cycle. We assume circuits to be well-defined, *i.e.*, all inputs and outputs of gates are connected and loops in the graph are separated by a register.

The state σ_i^h of a hardware circuit in cycle i is determined by the values of all inputs and registers, denoted *locations* with $\mathcal{V}^h = \{g \in \mathcal{G} \mid \mathcal{L}(g) \in \{\tau_{\mathsf{in}}, \tau_{\mathsf{reg}}\}\}$. Hence, $\sigma_i^h \in \mathbb{F}_2^{\left|\mathcal{V}^h\right|}$ and a location $v^h \in \mathcal{V}^h$ represents one bit of the state. Any gate $g \in \mathcal{G} \setminus \mathcal{V}^h$ is a function of the state $g := \mathbb{F}_2^{\left|\mathcal{V}^h\right|} \to \mathbb{F}_2$ according to its type.

Let $\sigma_i^h \xrightarrow{\mathbf{L}_i^h} \sigma_{i+1}^h$ denote a small-steps semantics, evaluating the ith clock cycle starting in state σ_i^h and defining the next state σ_{i+1}^h as well as a fixed

number of n_{cl} leakage points, corresponding to the empty tuple unless overridden. Each location in σ_{i+1}^h either corresponds to an input gate with a value defined by the environment or to a register gate g. Register gates have a sole input wire $(g', g) \in \mathcal{W}$ connected to gate g' and output value $g'\left(\sigma_i^h\right)$ in the next cycle.

Definition 2.10 (Hardware Computation Model $[\![(\mathsf{H},\mathsf{P})]\!]^h$)**.** The computation model $[\![(\mathsf{H},\mathsf{P})]\!]^h$ is defined as the m fold application of the small-steps semantics $\sigma_i^h \xrightarrow{\mathbf{L}_i^h} \sigma_{i+1}^h$

$$[\![(\mathsf{H},\mathsf{P})]\!]^h_{\sigma_0^h} \xmapsto{\mathbf{L}^h=\mathbf{L}_0^h||\mathbf{L}_1^h||\ldots||\mathbf{L}_m^h} \sigma_m^h \;:=\; \sigma_0^h \xrightarrow{\mathbf{L}_0^h} \sigma_1^h \xrightarrow{\mathbf{L}_1^h} \ldots \xrightarrow{\mathbf{L}_{m-1}^h} \sigma_m^h,$$

with an initial state $\sigma_0^h \in \mathbb{F}_2{}^{\left|\mathcal{V}^h\right|}$ containing machine-code P, a final state $\sigma_m^h \in \mathbb{F}_2{}^{\left|\mathcal{V}^h\right|}$ with $\mathcal{V}^h$ specific to processor netlist $\mathsf{H} := (\mathcal{G}, \mathcal{W}, \mathcal{L})$ and m determined by P.

The RPM defines $\mathbf{L}_i^h$ for $h = \textsc{rpm}$ in the form of multiple kinds of leakages representing the different physical leakage effects $\mathbf{L}^{\text{EL}}$. The aforementioned transition leakage is modeled in the RPM as an extended probe, permitting an adversary to observe the initial and next value of a register. This abstraction permits modeling of common physical impurities, respectively transistor characteristics, which otherwise permit to distinguish not only if a register changes the value, e.g., HW leakage, but also whether it changed from low to high or high to low. In addition, it covers common value-based leakage models and to some degree static leakage currents [Mor14].

A major component of glitch leakage is related to early signal propagation due to the timing characteristic of physical gates [MPG05]. A wire or gate may take multiple different temporary values, denoted *unstable* signals, before taking the correct *stable* value. The RPM models any of these leakages by a single observation per gate which allows an adversary to learn the value of any location determining the gate's value. Let $\mathsf{glitch}\left(\sigma_i^h, g\right)$ return the input and register values used to compute the value of gate g (2.15).

$$\mathsf{glitch}\left(\sigma_i^h, g\right) = \begin{cases} g\left(\sigma_i^h\right) & \text{if } g \in \{\tau_{\mathsf{in}}, \tau_{\mathsf{reg}}\} \\ \left(\mathsf{glitch}\left(\sigma_i^h, g'\right)\right)_{\forall g' : (g', g) \in \mathcal{W}} & \text{otherwise} \end{cases} \tag{2.15}$$

Finally, the RPM models coupling leakage $\mathbf{L}_{\mathsf{coup}}^{\text{RPM}}$ corresponding to a potential inductive coupling of adjacent electrical wires with changing charges [De +17]. Let $\mathsf{adjacent}(\mathsf{H}, w, w') \rightarrow \mathbb{F}_2$ be a Boolean relation returning $\top$ if wire w' is adjacent to w in H. Further, for some wire $w = (g, g') \in \mathcal{W}$ let $w\left(\sigma_i^h\right) = g\left(\sigma_i^h\right)$ be a shortcut to denote the output value of the unique gate g driving this wire.

Definition 2.11 (Robust Probing Model $[\![(\mathsf{H},\mathsf{P})]\!]^{\text{RPM}}$)**.** The RPM extends $[\![(\mathsf{H},\mathsf{P})]\!]^{h}$ with $h = \text{RPM}$ by defining the leakage $\mathbf{L}_i^{\text{RPM}}$ in the small-steps semantics $\sigma_i^h \xrightarrow{\mathbf{L}_i^h} \sigma_{i+1}^h$ of (H,P) in cycle i as

$$\mathbf{L}_i^{\text{RPM}} = \mathbf{L}_{\mathsf{trans},i}^{\text{RPM}} || \mathbf{L}_{\mathsf{glitch},i}^{\text{RPM}} || \mathbf{L}_{\mathsf{coup},i}^{\text{RPM}} \qquad \text{with}$$

$$\begin{aligned}
\mathbf{L}_{\mathsf{trans},i}^{\text{RPM}} &:= \left(\left(g\left(\sigma_i^h\right), g\left(\sigma_{i+1}^h\right)\right)\right)_{\forall g \in \tau_{\mathsf{reg}}}, \\
\mathbf{L}_{\mathsf{glitch},i}^{\text{RPM}} &:= \left(\mathsf{glitch}\left(\sigma_i^h, g\right)\right)_{\forall g \in \tau_{\mathsf{out}}}, \\
\mathbf{L}_{\mathsf{coup},i}^{\text{RPM}} &:= \left(\left(w\left(\sigma_i^h\right)\right) || \left(w'\left(\sigma_i^h\right)\right)_{\forall w' \in \mathcal{W}: \mathsf{adjacent}(\mathsf{H}, w, w')}\right)_{\forall w \in \mathcal{W}}.
\end{aligned}$$

Stricter models incorporate also glitches on the inputs of registers and extend glitches with the last value of the previous cycle to cover transitions in combinatorial logic, e.g., [CS21]. Others refine the leakages [DBR19]. The accuracy of the models is compelling for verification of security.

We define and verify in Chapter 5 *compliance* (Definition 5.3). Compliance is a formal property which ensures that all gate-level leakages and semantic outputs of some processor H executing P can be modeled (respectively simulated) from *contract* $[\![\mathsf{P}]\!]^c$ in a generic manner for any P. Contracts $[\![\mathsf{P}]\!]^c$ and fine-grained models $[\![\mathsf{P}]\!]^f$ differ in the underlying DSL but are otherwise similar. It goes beyond completeness as probing security and t–$(\mathsf{S})\mathsf{NI}$ verified in $[\![\mathsf{P}]\!]^c$ also hold at the gate-level $[\![(\mathsf{H},\mathsf{P})]\!]^h$ for any P in compliant models. This corresponds to a security reduction which we denote as E2E resilience.

In combination with scVerif this permits to verify gate-level probing security and t–$(\mathsf{S})\mathsf{NI}$ based on a contract. The abstract contract model leads to significant speed gains that outperform prior art for verification of gate-level probing resilience and is conjectured to scale to higher orders, larger implementations P, and processors H. Contracts can be shared after proving compliance, allowing users of contracts to prove gate-level security without requiring access to the netlist, which are confidential assets in the industry and unlikely to be shared. Besides, it permits for the first time to securely port implementations across different processors by designing one contract and verifying compliance for multiple processors. Any implementation secure in the contract is without any further effort gate-level probing secure when executed on any of the compliant processors.

2.5.3. Horizontal Resilience

This work is focused on establishing higher-order resilience for constrained adversaries $\mathsf{Adv}_{n_{poi} \leq t}$. Protecting against $\mathsf{Adv}^{\text{PHY}}$ exploiting $n_{poi} > t$ leakage points violates the verified t–$(\mathsf{S})\mathsf{NI}$ security notion. Attacks exploiting many

points, such as in horizontal SCA, deep learning, or SASCA, are therefore not covered.

We argue that our contributions have an indirect effect on horizontal resilience. First, it becomes easier to implement higher-order masking which increases the difficulty of attacks in noisy settings [Cha+99; PR13; DDF19; GGS18]. Second, the verification enables the exploration of optimizations to reduce the runtime of software implementations and thereby reduce the number of exploitable leakage points.

Duc, Dziembowski, and Faust show that n_{poi} *noisy leakages* can be simulated from t leakages for $n_{poi} > t$ with information-theoretical bounds [DDF19]. The bounds remain theoretical as unpractical security orders are needed for the bound to become meaningful [KR19]. Remark, that the bound depends on the number of leakages, hence verification and optimization enable to improve the theoretical security bounds [DDF19, Lemma 5 and 6].

Recent works on the random probing model provide alternative reductions with better parameters. The *random probing model* is an intermediate model used in the reduction of [DDF19]. Belaïd et al. show that the notions input output separation (IOS) and free-t–SNI connect well to this model [Bel+23a]. A straightforward extension of our fine-grained verification and contract-based E2E resilience would be to incorporate these notions into a respective verification tool and to support the preservation of these notions to concrete implementations. Other works propose constructions directly secure in the random probing model as well as verification tools to confirm the desired security level [Bel+20a; Cas+21a]. Combining such tools with fine-grained models or contracts that have been augmented with leakage probabilities appears interesting for future work.

3. Verification in Fine-Grained Models

We present the technical contributions for verifying the security of masked software P in user-defined fine-grained computation models $[\![\mathsf{P}]\!]^f$ subject to leakage $\mathbf{L}^f$

$$[\![\mathsf{P}]\!]^f_{\mathbf{X},\mathbf{R},p} \xmapsto{\mathbf{L}^f} (\mathbf{Y}, \mathbf{O})\,. \tag{3.1}$$

For this, we build a framework for partial evaluation of assembly code in user-specified IL models of assembly instructions. By abstract interpretation, we represent leakages and outputs as a function of the annotated inputs and verify the security of this representation.

First, a DSL called intermediate language (IL) is introduced, which implements the novel concept of explicit leakage for representing executable software. This concept permits separate modeling of semantic and leakage behavior, yet expresses both in a single model, which we exemplify for the CM0+ processor. Second, in Section 3.2, the extended security notions Stateful t–$(\mathsf{S})\mathsf{NI}$ are defined as well as verification methods to check implementations represented in IL against these notions. In Section 3.3, we use the resulting tool scVerif to support the development of multiple gadgets and evaluate whether the flow delivers physical security. The results are discussed in Section 3.4 w.r.t. research question **Q1**.

The following sections are based on the co-authored paper [Bar+21b], with major textual improvements, and adopt definitions from another co-authored work [Blo+22].

3.1. Expressing Semantic and Leakage

Verification of side-channel resilience requires a suitable representation of the implementation under assessment. This representation must not only express the functional semantic of an implementation but also its observable leakage.

3.1.1. A Domain Specific Language with Explicit Leakage

Already at CHES 2013 Bayrak et al. pointed out the difficulty of expressing arbitrary side-channel leakage behavior yet providing a “good interface” to users willing to specify custom side-channel characteristics [Bay+13]. This is caused by the standard approach of deeply embedding the specification of side-channel leakage behavior in the constructs of the programming language used for defining the model. In such a setting, the code for adding two variables

```
1 c ← a + b;
```

implicitly models information observable by an adversary. For example, the assignment operator ← is in [Bar+19] defined to semantically assign the sum to variable `c` and at the same time allows the adversary to observe the sum. The leakage behavior is thus embedded into the language semantics. Modeling different leakage behaviors, e.g., allowing the adversary to observe the value of operand `a` or `b` instead of the sum, requires adapting the language semantics and as such modifying any tool operating on it. This hampers the flexible adoption of leakage characteristics and is the reason why verification tools so far supported only a few fixed leakage models.

Our concept of *explicit leakage* instead separates concerns; it mandates explicit statements about the emitted side-channel information using a dedicated language construct. We present the IL language which implements this concept. All but one of its language constructs express no observable side-channel behavior, *i.e.*, their execution provides no side-channel information to adversaries. Only the dedicated statement `leak` provides information, in the form of a leakage point, to an adversary but has no semantic otherwise. This allows to model physical leakages by defining a leakage point consisting of the set of explanatory variables (Section 2.5.1). The previously given example can now be stated with an explicit annotation of leakage.

```
1 c ← a + b;
2 leak {a + b};
```

This has two important benefits: First, the specification of side-channel behavior becomes more flexible in that a diverse set of complex side-channels can be expressed and altered without effort. Second, the verification and representation of programs can be decoupled to become two independent tasks. The `leak` statement can be perceived as an undefined function with no side-effect on state, arguments, *etc.* but with special meaning for program analysis. We remark that leakage functions have been considered before, e.g., in [MR04], but were so far not explored in expressing detailed physical behavior.

$$
\begin{array}{lcl}
\mathfrak{s} & ::= & \mathfrak{v} \mid \mathfrak{v}\,[\mathfrak{e}] \mid \langle \mathfrak{e} \rangle \\
\mathfrak{e} & ::= & \mathfrak{s} \mid \mathfrak{n} \in \mathbb{Z} \mid \mathfrak{l} \mid \mathfrak{o}\,(\mathfrak{e}_1, \ldots, \mathfrak{e}_j) \\
\mathfrak{i} & ::= & \mathfrak{s} \leftarrow \mathfrak{e} \mid \mathfrak{m}\,(\mathfrak{e}_1, \ldots, \mathfrak{e}_j) \\
 & & \mid \texttt{leak "mnemonic" } \{\mathfrak{e}_1, \ldots, \mathfrak{e}_j\} \\
 & & \mid \texttt{if } \mathfrak{e} \texttt{ then } \mathfrak{i} \texttt{ else } \mathfrak{i} \mid \texttt{while } \mathfrak{e} \texttt{ do } \mathfrak{i} \\
 & & \mid \texttt{label } \mathfrak{l} \mid \texttt{goto } \mathfrak{e} \mid \mathfrak{i}; \mathfrak{i} \\
\mathfrak{g} & ::= & \texttt{macro } \mathfrak{m}\,(\mathfrak{v}_1, \ldots, \mathfrak{v}_j)\ \mathfrak{v}_1, \ldots, \mathfrak{v}_k\ \{\mathfrak{i}\} \\
 & & \mid \texttt{var } \mathfrak{s}
\end{array}
$$

Figure 3.1.: Simplified syntax of IL where $\mathfrak{n}$ ranges on integers, $\mathfrak{v}$ on identifiers of variables, $\mathfrak{m}$ on macro identifiers, $\mathfrak{o}$ on symbols of operations and $\mathfrak{l}$ on label identifiers.

IL is designed to be small, yet has specific features to simplify the representation of low-level software, *i.e.*, assembly. A Backus-Naur form (BNF) representation is given in Figure 3.1. Its building blocks are state elements $\mathfrak{s}$, expressions $\mathfrak{e}$, statements $\mathfrak{i}$ and blocks of multiple statements $\mathfrak{i}*$. Any IL model is a collection of global statements $\mathfrak{g}$ defining global variables and macros. Macros with identifier $\mathfrak{m}$ declare $k \geq 0$ local variables $\mathfrak{v}_1, \ldots, \mathfrak{v}_k$ and $j \geq 0$ input parameters $\mathfrak{v}_1, \ldots, \mathfrak{v}_j$ for $j, k \in \mathbb{Z}$.

The semantic and leakage behavior of a processor executing an implementation is represented by global IL variables modeling the processor state and IL macros modeling each of the processor instructions used in the implementation. Further, we use macros to represent implementations, which are explained in detail after defining the remainder of the language.

A state element $\mathfrak{s}$ is either a variable with symbol $\mathfrak{v}$, an element $\mathfrak{v}\,[\mathfrak{e}]$ of an array with symbol $\mathfrak{v}$ accessed at index $\mathfrak{e}$, or a location in memory $\langle \mathfrak{e} \rangle$. Memory is modeled as disjoint regions identified by a unique symbol $\mathfrak{l}$ which is used in expressions for determining the accessed memory address $\mathfrak{e}$. Modeling memory as disjoint regions simplifies formal verification.

Expressions are built from state elements $\mathfrak{s}$, constant integers $\mathfrak{n}$, unique labels $\mathfrak{l}$, and operators $\mathfrak{o}$ applied to expressions. In the following we represent IL code using the previously defined operators for clarity, *i.e.*, $+, -, \cdot, \wedge, \vee, \oplus$ and `!` for bitwise negation as well as logical left shift `<<`, respectively right shift `>>`. In our implementation operators and state elements are typed, e.g., `int` for signed integer, `uint` for unsigned integer, `bool` for Boolean bits, and `w32` for bit-vectors of 32 bits. This allows catching specification errors early as operators and parameters need to have the correct type or need to be cast to

the correct type, e.g., `(w32) 1` to cast the integer 1 to a representation with 32 bits.

Allowed statements $\mathfrak{i}$ consist of assignments $\mathfrak{s} \leftarrow \mathfrak{e}$ as well as explicit leaks `leak` $\{\mathfrak{e}_1, \ldots, \mathfrak{e}_j\}$ of one or more expressions with an optional mnemonic improving readability and debugging during verification. Additionally, statement $\mathfrak{m}(\mathfrak{e}_1, \ldots, \mathfrak{e}_j)$ invokes the macro with identifier $\mathfrak{m}$ and binds the macro's input variables to expressions. Statements for `if` conditionals and `while` loops are supported as well.

Labels $\mathfrak{l}$ are also used to represent the execution of processors which is based on the address of an instruction. They are defined by the dedicated `label` statement, enabling the execution of IL models to jump to the IL instruction subsequent to this label using `goto` $\mathfrak{e}$ where expression $\mathfrak{e}$ resolves to a defined label $\mathfrak{l}$.

In Section 3.2.2 we build a partial evaluator for IL.

3.1.2. Modeling Execution of Implementations

We explain how IL permits modeling the semantic of implementations and the leakage behavior caused by a processor executing it.

In short, the implementation's code is transformed into an IL `macro` such that the execution of this IL code changes global variables that represent the architectural state of the processor. This requires a correct model of the architectural state and the instructions, *i.e.*, for each instruction a corresponding `macro` needs to be provided which alters the state accordingly. Users have to specify such a model or re-use our published models. Our framework is agnostic of particular models and provides an automated translation of assembly to an IL macro invoking `macro` definitions of the user-supplied model.

Put simply, a processor executes a program by reading a binary machine code instruction from memory and executing it. The address of the instruction to fetch and execute is determined by the program counter (PC) which is in turn incremented or altered during the execution of instructions. Vice-versa, executable code consists of binary machine code instructions and associated addresses determining the location of the code within the instruction memory of the processor.

We detail the decoding of user-provided programs. Programs need to be provided as assembler code, denoted assembly for brevity. Compiled code can be disassembled using readily available off-the-shelf tools. Consider as an example the line of assembly in Listing 3.1. Here, the `adds` instruction located at address `0x16E` operates on the registers `r0` and `r1`.

Listing 3.1: Example line of assembler code.

```
1 0x16E: adds r0, r1
```

Listing 3.2: Resulting IL code.

```
1 label 0x16E;
2 adds(r0, r1);
```

This line is straight-forward represented by two IL statements as depicted in Listing 3.2. The behavior of `adds` is represented by calling the `adds` `macro` with the two registers as parameters as in the assembly. Line 1 declares a label with the address. This allows instructions, respectively their IL model, to alter the control flow by jumping to the label using `goto` statements. scVerif automates the representation of implementations by representing each line of assembly as a label declaration representing the address and a macro call to the model of the respective instruction. An assembly file may contain multiple implementations separated in *sections*, which are used to define corresponding macros.

Our generic approach to verifying side-channel resilience involves three stages: (I) modeling the behavior of instructions (Section 3.1.3 and 3.1.4), (II) representing an implementation using such a model, which is automated following the above strategy, and (III) verifying the representation based on annotations (Section 3.2).

We stress that verification and representation become separate concerns, *i.e.*, automated verification is now defined over the semantic of our DSL, and the leakage model of step (I) can be freely modified or replaced entirely without altering the work-flow of stages (II) and (III). In particular, scVerif allows the user to provide such a leakage specification in conjunction with an implementation for verification of side-channel resilience.

3.1.3. Modeling Instruction Semantics

We continue by explaining how IL permits to accurately model instruction semantics. The approach is exemplified by constructing the model $[\![\mathsf{P}]\!]^{\textsc{cm0+}}$ for CM0+ processors, *i.e.*, $f = \textsc{cm0+}$ in Eq. (3.1). These are augmented with leakage in Section 3.1.4. Likewise, the DSL enables the construction of semantic & leakage models for other architectures or programming languages.

The instructions of the CM0+ ISA operate on a set of globally accessible registers and flags, denoted *architecture state*. IL enables expressing architecture flags, carry bits, unsigned/signed operations, cast between data types, bit operations, control flow, *etc.* in close correspondence to ISA specifications. These are modeled as global variables in IL: `var r0`, `var r1`, . . ., `var pc`, `var apsrc` (carry flag), `var apsrv` (overflow flag), `var apsrz` (zero flag), `var apsrn` (negative flag).

Listing 3.3: IL model of addition with carry and of `adds` instruction.

```
1  // see license in Listing A.1
2  macro AddWithCarry (w32 x, w32 y, bool carry_in, w32 result,
3                      bool carry_out, bool overflow)
4    int unsigned_sum, int signed_sum
5  {
6    unsigned_sum ← (uint) x + (uint) y + (uint) carry_in;
7    signed_sum   ←  (int) x +  (int) y + (uint) carry_in;
8    result       ← (w32) unsigned_sum;
9    carry_out    ← !((uint) result == unsigned_sum);
10   overflow     ← !( (int) result ==   signed_sum);
11 }
12 macro adds (w32 rd, w32 rn, w32 rm) {
13   AddWithCarry(rn, rm, false, rd, apsrc, apsrv);
14   apsrn ← (rd >>w32 31) ==w32 (w32) 1;
15   apsrz ← (uint) rd == 0;
16   if (rd ≃n pc) {
17     goto rd;
18   }}
```

Addition is used in the `adds` instruction as well as instructions operating on pointers such as `ldr` (load) and `str` (store). Expressing the semantic of addition with carry requires casting 32 bit values to unsigned, respective signed values and comparing the results of addition to assign the carry and overflow flags correctly. The IL model of `adds` is expressed in Listing 3.3, closely following the Arm ISA specification [ARM18] with six parameters for inputs, output, carry and overflow flags. Note that macros are expanded during processing, *i.e.*, the definition of `adds` modifies the variables passed as arguments. `unsigned_sum` and `signed_sum` are local variables (Line 4). The `adds` instruction is modeled by calling the macro and expressing the side-effect on global flags. A special case of addition to `pc` requires issuing a branch to the resulting address (represented as a label). The operator $\simeq_n$ is used to compare whether the parameter `rd` is equal to the register with the name `pc` and conditionally issues a branch.

Sampling randomness, e.g., in the form of queries to random number generators, can be expressed by reading from a tape of pre-sampled randomness in global state.

3.1.4. Modeling Leakage

The `leak` statement allows generic specification of *multi-variate leakage*. The benefit in terms of physical resilience against $\mathsf{Adv}^{\mathrm{PHY}}_{n_{poi} \leq t}$ of modeling the data-dependency of leakages rather abstractly was already outlined in Section 2.5.1. We explain different leakage effects and augment the instruction model of `adds` (Listing 3.5) with `leak` statements that explain the power side-channel. The expressions inside a `leak` statement correspond to explanatory variables that explain a specific physical leakage without defining the concrete physical leakage function.

The arithmetic addition of the `adds` instruction is performed by electrical gates with a data-dependent power consumption. This is modeled by a simple univariate `leak` statement in Line 3 allowing adversaries to observe the result.

Updating a register content causes data-dependent physical leakage and thereby allows learning a function of the original and updated contents. This is modeled by a `leak rd, sum` in Line 4 where `rd` is expanded to the global variable representing the register. It is common to specify HD leakage but the 2-extended probe has the advantage of covering any function of the explanatory variables `rd` and `sum`, specially weighted or partial sums of the bits, e.g., observed in [MOW17; Gao+19]. A formal pattern for defining transition leakage is given in Definition 3.1. The order of execution matters, thus leakage must be specified before assigning `rd`.

Definition 3.1 (Transition Leakage Effect)**.** The transition leakage effect provides an observation on state `x` and the expression `e` to be assigned.

```
1 leak "transition" {x, e};
```

Consider the small contrived processor in Figure 3.2, where the arithmetic-logic unit (ALU) retrieves two operands from the read ports rpA and rpB of the register file and writes the result to the register file. Each module may contain registers in a pipelined design and each register is again subject to transition leakage (Definition 2.11). Hence, the execution of consecutive instructions may be subject to leakage combing operands, intermediates, or results of two or more instructions. In [CGD18] the *operand leakage* is described, a combination of current and previous operands of two instructions, e.g., parameters to `adds`. To model such behavior we introduce an additional state, denoted *leakage state*.

During our measurements on the CM0+ we observe that many instructions leak a combination of the current and previous value of the first, respectively second, operand of an instruction, see e.g., Listing 3.4. Hence, two leakage states `opA` and `opB` are defined globally and their content leaked in a single (worst-case approximation) `leak` in Line 6. In Line 7 f. the propagation of

Listing 3.4: A sequence of assembler code with an indication of some leakages across instructions.

```
1 ldr  rA, rB      // load valA
2 adds rC, rD, rE // opA <- rD; opB <- rE;
3 adds rF, rG, rH // observable leakage between rD, rG and rE,
    ↪ rH
4 ldr  rI, rJ      // observable leakage between read value, valA
```

the current operands into the microarchitectural state is modeled by updating the leakage state.

We denote this effect as *revenant leakage* because, as we observe, leakage may also span across distant instructions. For example, during the execution of the second `ldr` (load) in Line 10 of Listing 3.4 a combination of the value currently read from memory and the previously loaded value is leaking. We observe distinct behavior between loads and stores and use distinct leakage states to model the last read value (`opR`) and last written value (`opW`).

In pipelined designs, revenant leakage might be caused by a register stage, e.g., between read ports rpA or rpB in Figure 3.2. The quite general design pattern, expressed in Definition 3.2, models the microarchitectural register by leakage state (*i.e.*, a global variable) and adds a `leak` statement whenever the leakage state is altered, e.g., during loads. Anomalies are common. Especially in the load-store unit (LSU) the behavior is device specific and subject to conditions [MPW22; PV17; MOW17]. These can be expressed easily in IL and leakage across multiple instructions or more complicated behavior can be modeled with additional leakage states.

We remark that adding `leak` statements and leakage behavior is sometimes detrimental to performance and should be avoided. Especially for memory-related revenant leakage the impact on performance is relatively high and it is beneficial to model the least necessary leakage behavior to keep momentum for optimizations, which is in contrast to worst-case models.

Definition 3.2 (Revenant Leakage Effect)**.** The "revenant" leakage effect releases a transition leakage prior to updating some leakage state `x` $\leftarrow$ `e`.

```
1 leak "revenant" {x, e};
2 x ← e;
```

The concept of explicit leakage permits treating the modeling of leakage and semantics separately. Line 10 of the `adds_leak` definition calls the semantic model. This step is automated and omitted in further listings. The overall leakage model for a simplified ISA is depicted in Listing 3.6, it corresponds

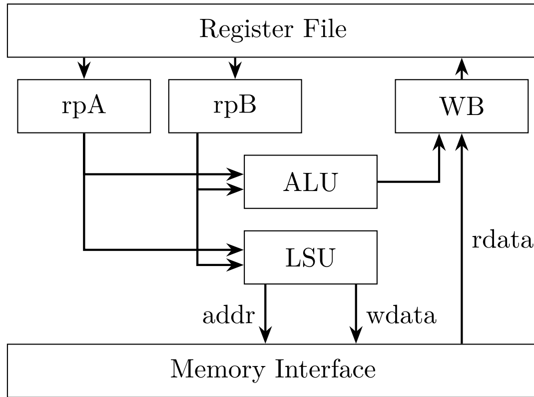


Figure 3.2.: Datapath of a contrived processor with read ports rpA, rpB, write-back (WB) unit, arithmetic-logic unit (ALU) and load-store unit (LSU). The electrical wires and gates cause well-understood leakage but the diverse processor architectures, varying registers placed in the datapath, and special corner cases in pipelined instruction execution do not permit to devise one universal software leakage model.

Listing 3.5: Leakage model of `adds` instruction.

```
1  // see license in Listing A.1
2  macro adds_leak(w32 rd, w32 rn, w32 rm {
3    leak "addsCompResult" (rn +w32 rm);
4    leak "addstransition" (rd, rn +w32 rm);
5
6    leak "addsRevenant" (opA, rn, opB, rm);
7    opA ← rn; // update leakage state according to the
8    opB ← rm; //   revenant leakage
9
10   adds(rd, rn, rm); /* call semantic model */ }
```

to the model $[\![\mathsf{G}]\!]^{\text{CM0+}}$ used for CM0+ assembly. The full model is provided in combination with scVerif in [Bar+20; Bar+21a]. The variables which explain the physical leakage of an instruction are instruction-specific. Some effects have been refined to match the behavior encountered in practice, e.g., an unexpected propagation of the destination register (which is semantically not required) in `load` instructions.

The empirical construction of this model was conducted in cooperation and is described in [Abr+21]. The model was initiated by testing whether the leakage effects discovered in prior work are detectable on our platform by constructing small first-order test cases as described in [PV17; MPW22]. In an iterative process, multiple gadgets with provable security in the model were constructed and assessed by physical leakage detection using TVLA with up to one multiple million traces. Every detected leak was analyzed and added to the model, again guided by tests analog to [PV17; MPW22], until no leakage was detectable. In Chapter 5.2 we present a systematic approach to model construction while in Chapter 4 heuristic ad-hoc extensions of the model are described. Given the amount of test-cases and physical evaluations we conclude that the model is empirically complete.

The DSL in combination with the concept of explicit leakage enables modeling all leakage effects known to us such that verification of threshold probing security becomes aware of these additional leakages. Our effect definitions can serve as building blocks to construct fine-grained models like in Listing 3.6 or can be heuristically augmented during development as we explore in Chapter 4. In [Bar+21b] we detail how rare effects like *neighboring* leakage observed in [PV17] can be modeled easily. We did not observe this effect in any experiment. The expressiveness of modeling appears not to be limited except in that IL is too simple. In Chapter 5 we present the industry-grade DSL Genoa with greater expressiveness which simplifies modeling further.

3.2. Stateful Strong-Non-interference and Automated Verification

In this section, we lay the foundations for proving the security of IL implementations. We first define the stateful security notions, used for IL and Genoa gadgets as well as contracts. Then, we present an effective method for verifying whether an IL gadget satisfies one of these notions.

3.2.1. Security Definitions

We first start with a brief explanation of the need for a new security definition.

Listing 3.6: Excerpt of the simplified IL model $[\![\cdot]\!]^{\text{CM0+}}$ for CM0+.

```
// see license in Listing A.1
var r0; var r1; … var r12; var pc; // Global CM0+ registers
var opA; var opB; var opR; var opW; // Global leakage state
macro xor (w32 op1, w32 op2) {
  leak "xorCompResult" (op1 ^w32 op2);
  leak "xorTransition" (op1, op1 ^w32 op2);
  // worst-case combination of revenants:
  leak "xorOperand"    (opA, op1, opB, op2);
  opA ← op1; opB ← op2;
  op1 ← op1 ^w32 op2; // semantic
}
macro and (w32 op1, w32 op2) {
  leak "andCompResult" (op1 &w32 op2);
  leak "andTransition" (op1, op1 &w32 op2);
  // worst-case combination of revenants:
  leak "andOperand"    (opA, op1, opB, op2);
  opA ← op1; opB ← op2;
  op1 ← op1 &w32 op2; // semantic
}
macro load (w32 dst, w32 ptr, int i)
  w32 val
{
  val ← [w32 mem (int) (ptr +w32 (w32) (i * 4))];
  leak "loadTransition" (dst, val);
  leak "loadOperand" (opA, ptr, opB, i);
  opA ← ptr; // mixed mapping
  leak "loadOperandB" (opB, dst); // dest. register propagates
  opB ← dst;
  leak "loadMemOperand" (opR, val); // read revenant
  opR ← val;
  dst ← val; // semantic
}
macro store (w32 src, w32 ptr, int i) {
  leak "storeOperand" (opA, ptr, opB, i);
  leak "storeOperandA" (opA, ptr);
  leak "storeOperandB" (opB, src);
  opA ← ptr; opB ← src;
  leak "storeMemOperand" (opW, src); // write revenant
  opW ← src;
  [w32 mem (uint) (ptr +w32 (w32) (i * 4))] ← src; // semantic
}
```

Listing 3.7: The execution trace of two t–(S)NI gadgets indicates a vulnerability due to residue in register r6 causing order-reducing leakage in the subsequent gadget. Stateful t–(S)NI mitigates such issues.

```
1 xor(r6, r5);       // compute output share C^(2) then store in
    ↪ mem
2 store(r6, r3, 2); //   see Lines 67 f. of Listing A.1
3 ... // maskedAnd returns control-flow to caller
4 ... // caller invokes maskedRefresh on shares C^(·)
5 load(r4, r3, 0);  // Line 1 ff. of Listing 3.1
6 load(r6, r1, 0);  // loading share C^(0) into r6 leaks C^(2), C^(0)
```

At a high level, the security of stateful computations requires dealing with residual effects on the state. Indeed, when a gadget is executed on a processor, it does not only return the computed output but it additionally leaves *residue* in registers, memory, or leakage state. Code subsequently executed might produce leakages combining these residues with output shares, breaking secure composability. As an example, consider the composition of two gadgets which correspond to refreshing the output shares $C^{(\cdot)}$ of a maskedAnd using a refresh gadget maskedRefresh, *i.e.*, $\mathsf{G}_{\mathsf{Refr}}(\mathsf{G}_{\mathsf{And}}(\mathbf{X}, \mathbf{Y}))$. In the case of non-stateful gadgets, if maskedAnd is t–NI and maskedRefresh is t–SNI, such a composition is t–SNI. However, if the gadgets, respectively their implementations, are stateful this is not necessarily anymore the case. Consider in Listing 3.7 the excerpt of the execution trace corresponding to the sequential execution of the two gadgets shown in Listing A.1 and 3.1 without the hardening introduced later. After the maskedAnd the output share $C^{(2)}$ remains in register r6. The register is in Line 6 overwritten by $C^{(0)}$ causing a transition leak, captured in Line 24 of the CM0+ model in Listing 3.6. This composition is vulnerable; one probe allows observing two shares, the transition leak is *order-reducing*.

Our stateful notions permit declaring public outputs which are required to be secret-independent. Consequently the Stateful t–SNI maskedAnd is hardened with additional instructions to clear residue in Lines 69 ff. in Listing A.1 to mitigate the described vulnerability. The benefit of our Stateful t–(S)NI is that it allows the systematic mitigation of leakage from residue in architectural & leakage state and therefore permits composing gadget implementations following the same rules as standard t–(S)NI. In [Bar+21b] it is proven that a slightly less formal variant of the stateful notions is composable like standard t–(S)NI. We give a more formal variant in Definition 3.3 adopted from the variant published in [Blo+22].

Our more formal definition includes *policies*. Gadgets are specified w.r.t.

input and output variables which are located in registers or memory during execution of implementations which is modeled as state σ. Policies map between the more abstract perspective taken by gadgets and the concrete locations of values in implementations. They correspond to a user-provided specification determining which location of the state belongs to which gadget variable. Input policy $\pi^{\mathrm{I}} : (\mathbf{X}, \mathbf{R}, p) \leftrightarrow \sigma_0$ constructs an initial state given values of variables for input shares, random and public variables but also the converse; extracting the values of gadget inputs given a state. Output policy $\pi^{\mathrm{O}} : (\mathbf{Y}, \mathbf{O}) \leftrightarrow \sigma_n$ maps between the values of public outputs and output shares of a gadget and the state σ_n resulting from an execution. Using policies we can define Stateful t–(S)NI security of gadgets, respectively their implementation in software or hardware depending on the execution of their concrete implementation P w.r.t. the big-steps semantics of computation model $[\![\cdot]\!]$. The definition applies to IL, contracts, and hardware.

Definition 3.3 (Stateful t–(S)NI of P in $[\![\mathsf{P}]\!]^m$ under $\pi^{\mathrm{I}}, \pi^{\mathrm{O}}$)**.** Let P be a software or hardware implementation and $[\![\mathsf{P}]\!]^m$ be a big-steps semantic determining the outputs and leakage $\mathbf{L}^m$ of P under policies $\pi^{\mathrm{I}}, \pi^{\mathrm{O}}$ and small-steps execution $\rightarrow$ where

$$\text{initial state} \quad \boldsymbol{\sigma}_0^m \overset{\pi^{\mathrm{I}}}{\longleftrightarrow} (\mathbf{X}, \mathbf{R}, p) \quad \text{and output state} \quad \boldsymbol{\sigma}_n^m \overset{\pi^{\mathrm{O}}}{\longleftrightarrow} (\mathbf{Y}, \mathbf{O}), \quad (3.2)$$

$$[\![\mathsf{P}]\!]^m_{\mathbf{X},\mathbf{R},p} \xmapsto{\mathbf{L}^m = \bigcup_{i=0}^{n-1} \mathbf{L}_i^m} (\mathbf{Y}, \mathbf{O}) := \boldsymbol{\sigma}_0^m \xrightarrow{\mathbf{L}_0^m} \boldsymbol{\sigma}_1^m \xrightarrow{\mathbf{L}_1^m} \dots \xrightarrow{\mathbf{L}_{n-1}^m} \boldsymbol{\sigma}_n^m, \quad (3.3)$$

with $\mathbf{X} = \left(X_i^{(j)}\right)_{i \in [n_{\mathbf{X}}], j \in [n_s]}$, $\mathbf{Y} = \left(Y_i^{(j)}\right)_{i \in [n_{\mathbf{Y}}], j \in [n_s]}$. P is Stateful t–(S)NI if

- there exists a deterministic function $F : \mathsf{dom}(p) \rightarrow \mathsf{dom}(\mathbf{O})$ modeling public outputs $\mathbf{O}$ from public inputs p such that $\mathbf{O} = F(p)$ and
- for any tuple $\mathbf{E}$ consisting of any number of shares $Y_i^{(j)}$ and $t_{\mathsf{P}} \leq t$ leakages of $\mathbf{L}^m$ with $|\mathbf{E}| \leq t$, there exists a tuple $\mathbf{S} = \left(X_i^{(j)}\right)_{i \in [n_{\mathbf{X}}], j \in \mathcal{I}_i}$ of input shares, with $\forall i \in n_{\mathbf{X}} : \mathcal{I}_i \subseteq [n_s] \wedge |\mathcal{I}_i| \leq t'$, a uniformly i.i.d. random variable $\mathbf{R}'$ and a simulator $S : \mathsf{dom}(\mathbf{S}, \mathbf{R}', p) \rightarrow \mathsf{dom}(\mathbf{E})$ perfectly simulating observations $\mathbf{E}$ from the shares $\mathbf{S}$, random $\mathbf{R}'$ and public inputs p such that $\mathsf{Pr}[S(\mathbf{S}, \mathbf{R}', p)] \equiv \mathsf{Pr}[\mathbf{E}]$.

For t–NI $t' = |\mathbf{E}|$ while the stricter t–SNI notion requires $t' = |t_{\mathsf{P}}|$.

3.2.2. Automated Verification

In this section, we consider the problem of formally verifying that an IL program is secure at order t for $t \geq 1$.

The obvious angle for attacking this problem is to extend existing formal verification approaches to IL. However, there are two important caveats. First, some verification approaches make specific assumptions on the programs, e.g., [BGR18] assumes that gadgets are built from ISW core gadgets. Such assumptions are reasonable for more theoretical models but are difficult to transpose to a more practical model; besides they defeat the purpose of our approach, which is to provide programmers with a flexible environment to build verified implementations. Second, re-implementing a t–SNI and t–NI checker for IL is a very significant engineering endeavor. We follow an alternative method: we define a transformation T that maps IL programs P into a fragment of IL that coincides with the core language of MaskVerif, *i.e.*, we *lower* IL to the MaskVerif fragment. Then, we reuse the verification algorithm of MaskVerif [Bar+15; Bar+19] with minor adaptions for checking the transformed program.

The transformation is explained below and satisfies correctness and precision. Specifically, the transformation T is correct: if $\mathsf{T}(\mathsf{P})$ is secure at order t then P is secure at order t (where security is either threshold probing security, t–$(\mathsf{S})\mathsf{NI}$, stateful t–$(\mathsf{S})\mathsf{NI}$ or PINI). The transformation T is also precise: if P is secure at order t and $\mathsf{T}(\mathsf{P})$ is defined then $\mathsf{T}(\mathsf{P})$ is secure at order t. Thus, the sole concern with the approach is the partial nature of the transformation T. While our approach rejects legitimate programs, it works well on a broad range of examples.

Target language The core language of MaskVerif is a subset of IL:

$$
\begin{array}{lcl}
\mathfrak{s} & ::= & \mathfrak{v} \mid \mathfrak{v}[\mathfrak{n}] \\
\mathfrak{e} & ::= & \mathfrak{s} \mid \mathfrak{n} \mid \mathfrak{o}(\mathfrak{e}_1, \ldots, \mathfrak{e}_j) \\
\mathfrak{i} & ::= & \mathfrak{s} \leftarrow \mathfrak{e} \mid \mathfrak{i}; \mathfrak{i} \mid \texttt{leak}\{\mathfrak{e}_1, \ldots, \mathfrak{e}_j\}
\end{array}
$$

The main difference between IL and MaskVerif is that the latter does not have memory accesses, macros, and control-flow instructions and limits array accesses to constant indices. Our program transformation proceeds in two steps: first, all macros are inlined; then the expanded program is partially evaluated.

Partial evaluation The partial evaluator takes as input an IL program. Additional annotations of policy π^{I} define the initial state. It returns another IL program which is equivalent to the original program w.r.t. functionality and leakage, under some mild assumptions about the policy, explained below.

Our partial evaluator manipulates abstract values and tuples of abstract values, and abstract memories. An abstract value $\mathfrak{a}$ can be either a base

value corresponding to concrete base values like Boolean $\mathfrak{b} \in \{t, f\}$, integer $\mathfrak{n}$, some label $\mathfrak{l}$ representing an abstract code pointer (e.g., for indirect jumps) or abstract pointers $\langle \mathfrak{v}, \mathfrak{n} \rangle$. The latter is an abstraction of real pointers. Formally, the BNF syntax of values is defined by:

$$\mathfrak{a} \quad ::= \quad \mathfrak{b} \mid \mathfrak{n} \mid \mathfrak{l} \mid \langle \mathfrak{v}, \mathfrak{n} \rangle \mid \bot$$

Initially, the abstract memory is split into different (disjoint) regions modeled by fresh arrays with maximal offset that do not exist in the original program. Those regions are what we call the memory layout of policy π and are defined by the user. A base value $\langle \mathfrak{v}, \mathfrak{n} \rangle$ represents a pointer to the memory region $\mathfrak{v}$ with integer offset $\mathfrak{n}$. This encoding is helpful to deal with pointer arithmetic and avoid aliasing problems. The following code gives an example of region declarations:

```
1  region mem w32 a[0:1]
2  region mem w32 b[0:1]
3  region mem w32 c[0:1]
4  region mem w32 rnd[0:0]
5  region mem w32 stack[-4:-1]
```

This defines an initial memory `mem` which is split into 5 distinct regions `a`, `b`, `c`, `rnd`, `stack`, where `a` is an array of size 2 with index 0 and 1. Annotations are assumptions and not checked by the tool. Another part of the memory layout provides some initialization for IL variables, in this case initializing registers.

```
6  init r0 [rnd, 0]
7  init r1 [c, 0]
8  init r2 [a, 0]
9  init r3 [b, 0]
10 init sp [stack, 0]
```

Here, a part of π^{I} is specified; the assumption that register `r0` is initially a pointer to the region `rnd`, hence mapped to a variable in p. Some extra annotations indicate which regions or globals initially map to random values, or correspond to input, respectively output, shares.

We give an overview of the partial evaluator $(\!|\mathsf{P}|\!)^{\mathsf{il}}_{\mu,\rho}$. The partial evaluator is parameterized by a state $\langle \mathsf{P}, \mathfrak{i}, \mu, \rho, \mathsf{P}' \rangle$, where P is the original IL program, $\mathfrak{i}$ is the current statement to be reduced, μ a mapping from P's variables to their abstract value, ρ a mapping from the variable corresponding to memory region to their abstract value, and P' is the sequence of statements that have been partially executed forming the transformed program. In this sense P' is a symbolic execution trace augmented with leakage. The partial evaluator

$$\frac{}{(\!|\mathsf{P}|\!)^{\mathsf{il}}_{\mu,\rho} = (\mu(\mathfrak{v}), \mathfrak{v})}\ \textsc{Var} \qquad \frac{\forall i \in [1,n] \quad (\!|\mathsf{P}|\!)^{\mathsf{il}}_{\mu,\rho} = (\mathfrak{a}_i, \mathfrak{e}'_i)}{(\!|\mathsf{P}|\!)^{\mathsf{il}}_{\mu,\rho} = (\tilde{\mathfrak{o}}(\mathfrak{a}_1, \ldots, \mathfrak{a}_n), \mathfrak{o}\,(\mathfrak{e}'_1, \ldots, \mathfrak{e}'_n))}\ \textsc{Op}$$

$$\frac{(\!|\mathsf{P}|\!)^{\mathsf{il}}_{\mu,\rho} = (\mathfrak{n}, \mathfrak{e}')}{(\!|\mathsf{P}|\!)^{\mathsf{il}}_{\mu,\rho} = (\mu(\mathfrak{v})[\mathfrak{n}], \mathfrak{v}[\mathfrak{n}])}\ \textsc{Array} \qquad \frac{(\!|\mathsf{P}|\!)^{\mathsf{il}}_{\mu,\rho} = (\langle \mathfrak{v}, \mathfrak{n}\rangle, \mathfrak{e}')}{(\!|\mathsf{P}|\!)^{\mathsf{il}}_{\mu,\rho} = (\rho(\mathfrak{v})[\mathfrak{n}], \mathfrak{v}[\mathfrak{n}])}\ \textsc{Mem}$$

$$\frac{\begin{array}{c}\mathfrak{i}_1 = \mathfrak{s} \leftarrow \mathfrak{e} \quad \mathfrak{i}'_1 = \mathfrak{s}' \leftarrow \mathfrak{e}' \\ (\!|\mathsf{P}|\!)^{\mathsf{il}}_{\mu,\rho} = (\mathfrak{a}', \mathfrak{s}') \quad (\!|\mathsf{P}|\!)^{\mathsf{il}}_{\mu,\rho} = (\mathfrak{a}, \mathfrak{e}') \qquad (\mu,\rho)\{\mathfrak{s}' \leftarrow \mathfrak{a}\} = (\mu', \rho')\end{array}}{\langle \mathsf{P}, \mathfrak{i}_1; \mathfrak{i}_2, \mu, \rho, \mathsf{P}'\rangle \rightsquigarrow \langle \mathsf{P}, \mathfrak{i}_2, \mu', \rho', \mathsf{P}'; \mathfrak{i}'_1\rangle}\ \textsc{Assign}$$

$$\frac{\forall i \in [1,n] \quad (\!|\mathsf{P}|\!)^{\mathsf{il}}_{\mu,\rho} = (\mathfrak{a}_i, \mathfrak{e}'_i)}{\langle \mathsf{P}, \texttt{leak}\,\{\mathfrak{e}_1, \ldots, \mathfrak{e}_j\}\,; \mathfrak{i}, \mu, \rho, \mathsf{P}'\rangle \rightsquigarrow \langle \mathsf{P}, \mathfrak{i}, \mu, \rho, \mathsf{P}'; \texttt{leak}\,\{\mathfrak{e}'_1, \ldots, \mathfrak{e}'_j\}\rangle}\ \textsc{Leak}$$

$$\frac{(\!|\mathsf{P}|\!)^{\mathsf{il}}_{\mu,\rho} = (\mathfrak{l}, \mathfrak{e}') \quad \mathsf{P}_{\mathfrak{l}} = \mathfrak{i}'}{\langle \mathsf{P}, \texttt{goto}\ \mathfrak{e}; \mathfrak{i}, \mu, \rho, \mathsf{P}'\rangle \rightsquigarrow \langle \mathsf{P}, \mathfrak{i}', \mu, \rho, \mathsf{P}'\rangle}\ \textsc{Goto}$$

$$\frac{\mathfrak{i} = \texttt{if}\ \mathfrak{e}\ \mathfrak{i}_t\ \mathfrak{i}_f \quad (\!|\mathsf{P}|\!)^{\mathsf{il}}_{\mu,\rho} = (\mathfrak{b}, \mathfrak{e}')}{\langle \mathsf{P}, \mathfrak{i}; \mathfrak{i}', \mu, \rho, \mathsf{P}'\rangle \rightsquigarrow \langle \mathsf{P}, \mathfrak{i}_{\mathfrak{b}}; \mathfrak{i}', \mu, \rho, \mathsf{P}'\rangle}\ \textsc{If}$$

$$\frac{\mathfrak{i} = \texttt{while}\ \mathfrak{e}\ \mathfrak{i}_w \quad \mathfrak{i}' = (\texttt{if}\ \mathfrak{e}\ \mathfrak{i}_w; i); \mathfrak{i}''}{\langle \mathsf{P}, \mathfrak{i}; \mathfrak{i}'', \mu, \rho, \mathsf{P}'\rangle \rightsquigarrow \langle \mathsf{P}, \mathfrak{i}', \mu, \rho, \mathsf{P}'\rangle}\ \textsc{Loop}$$

Figure 3.3.: Selected inference rules for the partial evaluation of IL syntax.

iteratively propagates values, removes branching instructions, and replaces memory accesses with variable accesses (or constant array accesses). Figure 3.3 provides selected rules for the partial evaluator, which are explained in more detail in [Bar+21b]. The rules MEM and ARRAY apply the partial evaluation rules on the expression for the address, respectively index, and try to lower these to an integer to rewrite memory, respectively array, accesses to array access with constant offset. T is successful if all expressions fall into the MASKVERIF fragment.

Proposition 3.1 (Correct Transformation T)**.** *The transformation* T *is sound because it preserves the joint probability distribution of leakages and outputs for equal inputs. Let* G *be any IL gadget and* $\mathsf{G}' = \mathsf{T}(\mathsf{G})$ *be the successful*

transformation with provided annotations,

$$\llbracket \mathsf{G} \rrbracket^f_{\mathbf{X},\mathbf{R},p} \xmapsto{\mathbf{L}} (\mathbf{Y}, \mathbf{O}) \qquad \textit{and} \qquad \llbracket \mathsf{G}' \rrbracket^f_{\mathbf{X},\mathbf{R},p} \xmapsto{\mathbf{L}'} (\mathbf{Y}', \mathbf{O}')$$

$$\textit{then} \qquad \Pr\left[\mathbf{Y}, \mathbf{O}, \mathbf{L}\right] \equiv \Pr\left[\mathbf{Y}', \mathbf{O}', \mathbf{L}'\right].$$

Proposition 3.1 could be proven by defining formal small-steps and big-steps semantics $\llbracket \mathsf{G} \rrbracket^f$ for IL and proving that the transformation steps of the partial evaluator rules of transformation T preserve probability distributions.

3.2.3. Implementation

We have implemented the partial evaluator as a front-end to MaskVerif, named scVerif. Since the input language of MaskVerif did not include a `leak` construction, we have modified the MaskVerif input language to provide direct access to the `leak` statement as well as leak-free assignment. Users are now able to write MaskVerif gadgets with custom leakage by using explicit `leak` and leak-free statements.

Moreover, we have extended the input language of MaskVerif so that inputs and outputs can be declared as public. This allows expressing the presented notions of stateful t–SNI and t–NI. The extended checker verifies that public outputs only depend on constants and public inputs, *i.e.*, they neither depend on the private or shared input variables, nor the random variables used by the gadget. Automated representation of programs in custom leakage models is done by the scVerif front-end that generates an equivalent MaskVerif program (see Proposition 3.1).

Users can write leakage models, annotations, and programs in IL or provide programs in assembly code. If the output program lies in the MaskVerif fragment, then verification starts with user-specified parameters such as security order or which property to verify. Else, the program is rejected. The tool also applies to bit- and n-sliced implementations. Sharesclicing is not yet supported as the scheme is questioned in [Gao+19] and insecure in our CM0+ models.

3.3. Representative Proofs of Efficient Masking

We describe the construction and optimization of gadgets that do not exhibit vulnerable leakage according to a fine-grained model.

3.3.1. Optimized Hardened Masking

We design gadgets first in the simplified IL model depicted in Listing 3.6. Designing in IL is more flexible since shortcuts such as leakage-free operations and abstract countermeasures are available. Once the security is verified, the IL code can be trivially translated to assembly and verified again before continuing to physical validation.

Any gadget G t–(S)NI in $[\![G]\!]^{ISW}$ is secure against computation leakage. Therefore it can be secured by clearing the architecture and leakage state at selected locations within the code. The reason is that every `leak` is defined over locations of the state and clearing these locations prior to the instruction causing such `leak` mitigates the ability to observe the sensitive data.

We use macros as abstract countermeasures `scrub(rD)` and `clear(opX)`; for assigning a public value to `rD`, respectively `opX`. In assembly these need to be substituted by available instructions which is trivial for architectural state. Clearing leakage state is specific to the microarchitecture, e.g., clearing `opA` or `opB` is done by `ands r0, r0` since `r0` is a public memory address in our implementations. Clearing `opR`, respectively `opW`, requires to execute `load`, respectively `store`, but the side-effects of both instructions require additional care. Sometimes multiple countermeasures can be combined in assembly, e.g., `loadpub(rD)` clears both `rD` and `opR` and can be implemented using a single `ldr`.

At the end of a gadget we clear all residue, *i.e.*, microarchitectural state and all the registers and memory temporarily involved in computations. This is a general approach to fix the composition problem outlined in Section 3.2.1. In our model, this corresponds to locations `opA`, `opB`, `opR`, `opW` and `stack`, unless policy π^{O} specifies shares or randomness at a location. Let `fclear` denote an abstract macro to clear all such locations, its implementation in assembly is specific to the model and available instructions.

Hardening a gadget by inserting instructions at the end and within the code can be expensive. In the 2nd-order maskedRefresh in Listing 3.1 more than 50% of the instructions correspond to clearings and scrubs, severely affecting performance. In [Bar+21b] many more Stateful t–(S)NI implementations, verified by scVerif, are presented. We describe optimization strategies to overcome the performance drawback.

In [Bar+21b] we present an approach to efficiently implement compositions of linear gadgets, depicted in Figure 3.4a. In short, linear functions F_1 and F_2 can be implemented share-wise, commonly first $\mathsf{F}_1\left(A^{(i)}, B^{(i)}\right) = M^{(i)}$ on all shares, then clearing of residue, then $\mathsf{F}_2\left(M^{(i)}, C^{(i)}\right)$ on the outputs. Instead, we execute them jointly share-wise to save on the clearing of residue, *i.e.*, $\mathsf{F}_2\left(\mathsf{F}_1\left(A^{(i)}, B^{(i)}\right), C^{(i)}\right)$.

Algorithm 3.1 2nd-order Stateful t–SNI maskedRefresh [Bar+21b]

Input: Registers `r1` pointing to shares $\mathsf{BEnc}^{3}_{32,2}(a) := \left(A^{(\cdot)_B}\right)$ and `r3` to randomness R_0, R_1, R_3.

Output: Register `r0` pointing to shares $\left(C^{(\cdot)_B}\right) = \mathsf{BEnc}^{3}_{32,2}(a)$, such that

$$C^{(0)_B} = A^{(0)_B} \oplus R_0 \qquad C^{(1)_B} = A^{(1)_B} \oplus R_1 \qquad C^{(2)_B} = A^{(2)_B} \oplus R_0 \oplus R_1$$

```
1: load(r4, r3, 0);
2: load(r6, r1, 0);
3: clear(opR);
4: load(r5, r3, 1);
5: xor(r6, r4);
6: store(r6, r0, 0);
7: clear(opW);
8: loadpub(r6);
9: load(r7, r1, 1);
10: clear(opA);
11: xor(r7, r5);
12: store(r7, r0, 1);
13: clear(opW);
14: clear(opB);
15: xor(r4, r5);
16: loadpub(r5);
17: clear(opB);
18: load(r5, r1, 2);
19: xor(r5, r4);
20: store(r5, r0, 2);
21: loadpub(r4);
22: loadpub(r5);
23: loadpub(r6);
24: loadpub(r7);
25: clear(opA);
26: clear(opB);
27: clear(opR);
28: clear(opW);
```

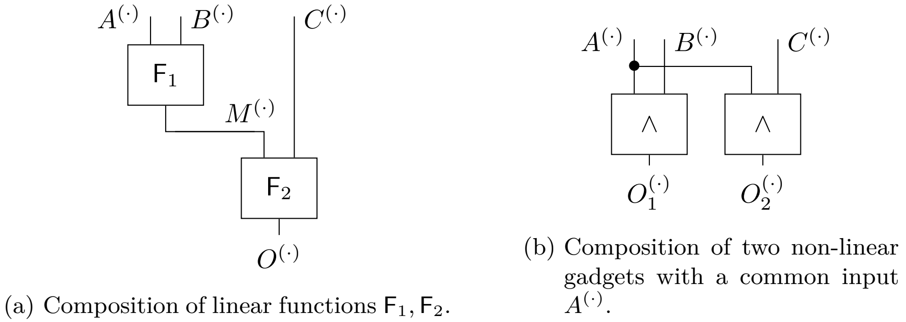
An example of this approach is given in Algorithm A.2. This method allows us to save on the number of `clear`, `load`, and `store` operations. We provide a generic proof for this strategy in [Bar+21b].

The second scenario is when two non-linear gadgets sharing one of the inputs are composed, depicted in Figure 3.4b. The number of loadings and clearings can be reduced by re-using the common shares and avoiding storing and loading the intermediate output. Most importantly, it is possible to replace intermediate clearings with (independent) computations on the unrelated share. For example, to mitigate transition leakage caused by transitioning between two shares $B^{(\cdot)}$ the computation can be changed such that the transitioning happens between another share of $C^{(\cdot)}$. Hence, meaningful computation can be used to mitigate leakage instead of overhead producing dummy operations. The complexity of getting this right is non-negligible and verification with scVerif is of great help.

3.3.2. Case Study: Masking the Present S-Box

We apply both strategies in masking the Present S-Box at the 1st and 2nd order to evaluate the impact.

The S-Box consists of a non-linear function depicted in Figure 3.5 and two



(a) Composition of linear functions $\mathsf{F}_1, \mathsf{F}_2$.

(b) Composition of two non-linear gadgets with a common input $A^{(\cdot)}$.

Figure 3.4.: We propose optimization strategies for linear compositions (a) and non-linear composition with one common input (b).

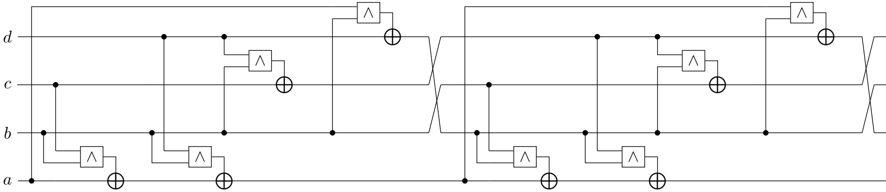


Figure 3.5.: Our optimized composition strategies apply to the non-linear layer of the PRESENT S-Box resulting in efficient and hardened masking at 1st and 2nd-order.

share-wise functions [CFE16]. We create masked variants `calcA`, `calcB` of the linear and `calcG` of the non-linear function by composing maskedAnd and maskedXor gadgets. In addition, we create optimized, monolithic gadgets `calcA_opt`, `calcB_opt`, and `calcG_opt` using the aforementioned strategies and SCVERIF. The code of all IL gadgets and assembly implementations is provided in [Bar+21b; Bar+21a; Bar+20].

With the help of SCVERIF it is possible to quickly check if randomness can be re-used, as proposed in [FPS17]. Our 1st and 2nd-order PRESENT S-Box require 7, respective 26 words of entropy which can be reduced to 3, respective 18 (2nd-order) words by re-using randomness, independent of the optimizations.

The performance results are shown in Table 3.1, highlighting the number of instructions, scrubbing, and clearings as well as the overall cycle count. Optimization at both orders yields gadgets with equivalent probing security in our fine-grained model but at a much lesser count of dummy operations.

		1st-order			2nd- order		
	unprot.	*comp.*	*opt.*	*opt./comp.*	*comp.*	*opt.*	*opt./comp.*
`xor`	21	57	59	1.035	133	142	1.07
`and`	18	24	24	1	54	54	1
`load`	18	129	60	0.47	251	136	0.54
`store`	4	82	40	0.49	93	72	0.77
`scrub`	-	95	16	0.17	211	54	0.26
`clear(opA)`	-	35	12	0.34	67	20	0.30
`clear(opB)`	-	35	30	0.86	314	29	0.09
`clear(opR)`	-	35	10	0.29	260	20	0.08
`clear(opW)`	-	56	4	0.07	93	10	0.11
cycles	110	876	377	0.43	2173	788	0.36

Table 3.1.: Operation and cycle count of normally composed (*comp.*), optimized (*opt.*) and unmasked reference (*unprot.*) PRESENT S-Box. Our optimizations result in significant performance improvements both in absolute count of dummy operations for hardening and memory accesses.

Further, the number of memory accesses can be reduced, yielding additional performance gains. In total, we report a reduction of the cycle count of 57% at 1st, and 64% at 2nd-order. Remarkably, the 2nd-order optimized construction is *faster* than the 1st-order composition of gadgets. Vice-versa, optimization permits increasing the security order for free, *i.e.*, *without* increasing the cycle count.

A similar view is provided in Table 3.2 where we estimate the efficiency after hardening by calculating the ratio between dummy operations for hardening, and operations for implementing the functionality. The comparison shows that the negative impact of hardening is more severe at the 2nd order and that the optimized constructions have less potential for further optimization as only 25 to 28% of the total operations correspond to dummy operations.

We emphasize that the application of the proposed optimizations heavily relies on the speedy verification with scVERIF allowing numerous iterations for testing changes. This is hardly worthwhile in combination with slow physical assessment.

	1st-order		2nd-order	
	Composition	*Optimization*	*Composition*	*Optimization*
$\frac{\#\text{clearings}}{\#\text{operations}}$	0.81	0.38	1.78	0.32

Table 3.2.: The ratio between relevant operations and dummy operations for hardening is significantly reduced in the optimized PRESENT S-Box compared to normal composition.

3.3.3. Physical Resilience of Verified Implementations

We evaluate whether proofs in our fine-grained leakage model connect to resilience in practice.

There are no known methods to evaluate the exhaustive completeness of our model $[\![\mathsf{P}]\!]^{\text{CM0+}}$ directly against the physical leakage $\mathbf{L}^{\text{PHY}}$. We assess the completeness empirically by evaluating all our implementations using leakage detection. In each of our verifiably secure implementations, all modeled leakages $\mathbf{L}^{\text{CM0+}}$ are 1-wise or 2-wise independent of secrets encoded in shares. If our model is complete then all physically observable leakages $\mathbf{L}^{\text{PHY}}$ should be 1-wise or 2-wise independent too (2.14).

TVLA is used to test the hypothesis that the samples measured when executing implementations on fixed secrets X_1 are indistinguishable from invocations on randomly chosen secrets $\mathbf{X}_2 \neq X_1$ [SM15]. We measure two sets of one million traces in randomly interleaved order on the "FRDM-KL82Z" and the "STM32L073RZ" MCUs which integrate the Arm CM0+. Each invocation is provided with i.i.d. randomness and independent encoding of either secret. We use a Welch t-test with an alpha certainty of 0.0001. Significant leakage is detected when the absolute t-statistics are larger than the non-adopted threshold of 4.5. The measurement setup and preparation of the two boards are described in [Bar+21b; Bos+21].

The t-statistic for our 1st-order optimized PRESENT S-Box is shown in Figure 3.6, indicating no significant leakage on both MCUs. We provide in [Bar+21b] more implementations, all verified and all physical leakage resilient at first order. Additional gadgets secure in $[\![\mathsf{P}]\!]^{\text{CM0+}}$ have been measured in co-authored work [Abr+21] on two different setups, one involving EM measurements by a certification laboratory.

To show the applicability of our model at higher order we evaluate our 2nd order PRESENT S-Box in 2nd order multivariate TVLA on the KL82Z by processing the measurements such that every pair of sample points is combined and evaluated [SM15]. The results are shown in Figure 3.7. The combinatorial blow-up requires hundreds of CPU hours to evaluate the S-Box, compared to

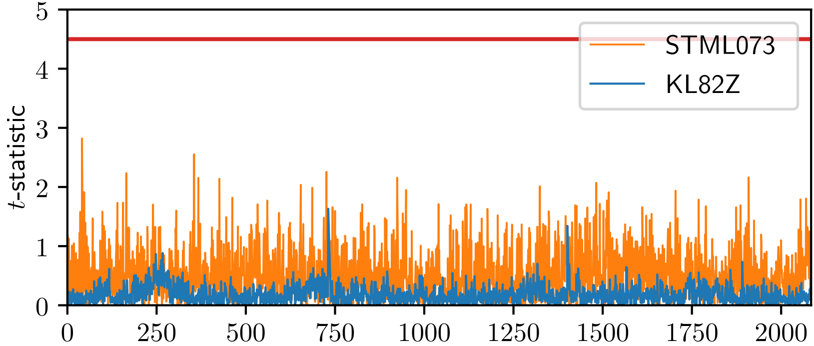


Figure 3.6.: Physical leakage detection t-statistics of optimized 1st order PRESENT S-Box assessment, x axis represents sample points.

a few seconds when using SCVERIF.

Based on the quantity, quality (low ratio of clearings, Table 3.2) and diversity of the evaluated implementations we draw the conclusion that our $[\![\mathsf{P}]\!]^{\text{CM0+}}$ model enjoys empirical completeness for the modeled instructions. This statement holds for first and higher-order security with up to one million traces.

3.4. Discussion

Automated verification in fine-grained leakage models facilitates the design and implementation of efficiently hardened masking with physical resilience.

First, the concept of explicit leakage provides flexible, accurate, and intelligible models of physical characteristics, as well as a separation of concern between modeling and analysis. Second, the abstraction of physical leakage as extended probes allows to capture the data dependency of measurement points rigorously, yet circumvents the effort to model the precise leakage function of physical measurements. Third, the focus on verifying masked implementations in conjunction with fine-grained models improves the ability to reliably ensure the independence of secrets for all subsets of t physical leakage points for $t \geq 1$.

We note that our results are a consequence of a feedback loop. The flexible models paired with speedy and exhaustive verification permit exploring more designs and optimizations, implementing at higher orders, reducing regres-

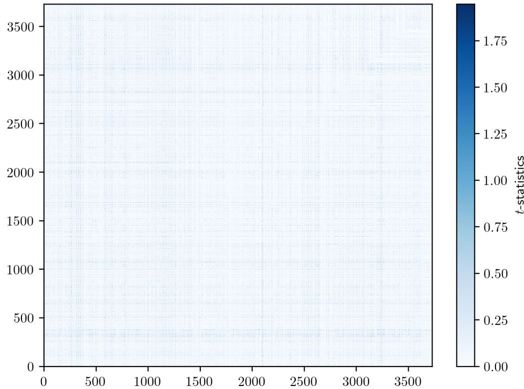


Figure 3.7.: Bivariate TVLA of the optimized 2nd order PRESENT S-Box executed on the KL82Z microcontroller. No significant leakage is detected for every pair of sample points on the x and y axis as the values are far below the threshold of 4.5.

sions, and speeding up development with instrumental debug information in case of flaws. Such a workflow is crucial for the development and application of the presented generic optimization strategies. In return, the construction of accurate models is fostered by the increasingly optimized designs which are more dense due to the reduced number of dummy operations and thereby permit to challenge the accuracy of the model. We report that all verified implementations we developed after reaching the presented model consistently show no detectable leakage in physical evaluation. We conjecture that the aforementioned combination of verification and physical assessment is the central reason for this success.

Leakage models are often studied in the context of leakage resilient cryptography, see e.g., [Cha+99; ISW03; MR04; PR13; DDF19; Blo+18; Fau+18; KR19]. Several works propose strategies, based on the structure of the RPM, to circumvent leakages such as glitches [DBR19; CS21]. Our optimizations are unbiased towards specific leakage behavior and the fine-grained models permit expressing (more) realistic behavior flexibly and developing optimizations on a case-by-case basis. The analysis of composability in leakage models with extended probes is a trend so far focused on fixed models [Cas+21b; Bel+20b]. Our stateful extension of t–(S)NI permits the secure composition in arbitrary

fine-grained leakage models.

Verification of gadgets or implementations at 1st-order is studied in [Mos+12; Bay+13; EWS14; Zha+18; Blo+18]. Others extend to higher order [Bar+15; Cor18; Bar+19; KSM20; Gig+21; Bel+22]. The works assume fixed or configurable models but do not support flexible models. We lower implementations represented in IL models to MaskVerif but remark that the transformation could be adjusted to other verification tools too. Future work may consider lowering IL to SILVER [KSM20] and IronMask [Bel+22] since these are free of false negatives and support verification of output uniformity, respectively random probing security. A work close to ours verifies the security of software executed on a processor's netlist in a gate-level leakage models [Gig+21]. While this permits to *completely* capture high-level leakages caused by the processor architecture it mandates access to netlists, hardware expertise, and increased wait times due to computation delay. In Chapter 5 we extend fine-grained models to contracts that provably capture the (same) gate-level leakage behavior completely but avoid the downsides.

Characterizing and understanding physical leakage behavior is also considered outside verification. The reduction in security order due to leakage is studied in [Cor+12; Bal+14; Gro+16; DDT20; GPM21]. Multiple works model leakage to emulate side-channel measurements [Ves14; PV17; MOW17; CGD18; GOP22; Baz+21]. The reported characterizations and root causes are useful hypotheses in constructing fine-grained models. Notably, [Gao+19] report bit-interactions within operands of instructions of Arm Cortex-M processors and fundamentally question share-slicing schemes [Bar+17]. All emulation tools make assumptions about the leakage behavior and require tool modifications to integrate different behavior. Explicit leakage permits analyses to remain unbiased and the abstraction of leakage functions as extended probes determining the explanatory variables significantly reduces modeling effort. The work of Marshall, Page, and Webb is a systematic study of the leakage behavior on multiple processors [MPW22]. They observe many differences between devices and thereby independently justify device-specific models.

This work focuses on masking and verifiable security. Currently, implementations with randomized control flow, e.g., shuffled masking [Azo+22b], cannot be analyzed. Connecting our fine-grained models to emulation-based tools is an interesting approach filling this gap and performing emulation-based analyses in general. Fine-grained models expressed in IL have already been connected to verification of cache-timing side-channels [Fri20].

Our approach relies on the quality of the used fine-grained model in terms of completeness and minimalism, *i.e.*, least necessary leakage annotation. The quality of leakage models is considered in [DSP16; DSM17; Bro+19]. These works quantify or bound the distance between an estimated leakage function

of an explanatory variable (corresponding to the valuation of $L_i^f \in \mathbf{L}^f$) to the true or ideal leakage function. This does not help in assessing if all explanatory variables L_i^f are completely listed in a model. The recent works of Gao and Oswald compare two leakage models $[\![\mathsf{P}]\!]^{m_1}, [\![\mathsf{P}]\!]^{m_2}$ with an F-test on physical traces [GO22a; GO22b]. Their approach determines whether $[\![\mathsf{P}]\!]^{m_2}$ completely captures the physical leakage modeled by $[\![\mathsf{P}]\!]^{m_1}$ but cannot qualify if either model is exhaustively complete. Our evaluation of empirical completeness (Section 3.3.3) is based on building test cases which (based on verification) are supposed to be free of secret dependent statistical moments $\leq t$. By challenging this hypothesis with TVLA we are able to validate completeness for examined test cases without relying on model assumptions. The approach of [GO22a] remains interesting as it enables iterative minimization of potentially too conservative fine-grained models and, in addition, relies on unmasked test cases, simplifying model validation. In Chapter 5 we approach provable model completeness w.r.t. all gates of the processor executing software. Further, in Chapter 4 we show that missing leakage annotations can be added ad-hoc when verification and physical assessment are combined as in the proposed secure development flow in Figure 1.2

In summary, we answer research question **Q1** affirmative. The described modeling technique and verification flow accurately and reliably predict physical resilience when based on models with empirical completeness. Furthermore, automated verification adds to the ease of use, confidence in security, implementation efficiency, exploration depth, and scales to greater implementation size and higher security order than physical or emulated evaluation. The dependence on model quality is inherent to verification and further addressed in the next chapters.

4. Masking Kyber: First- and Higher-Order Implementations

Public-key cryptography is based on conjectured-to-be-hard mathematical problems. The most widely used examples RSA and ECC are vulnerable to polynomial-time attacks using a quantum computer [Sho94; PZ03]. To defend against this threat, research is directing its attention to post-quantum cryptography (PQC).

One of the PQC algorithms standardized by NIST is Kyber [Bos+18; Sch+20b]; this scheme belongs to the lattice-based KEM family. Kyber's hardness is based on the module learning-with-errors (M-LWE) problem in module lattices [LS15]. Unlike integer factorization and the discrete logarithm problem, the M-LWE problem is conjectured to be hard to solve even by an adversary who has access to a full-scale quantum computer.

Not surprisingly, side-channel attacks also affect PQC [TE15]. Power analysis attacks on lattices have been launched even on masked implementations of lattice-based cryptography by targeting the number-theoretic transform (NTT) [PPM17; PP19; Xu+20], the message encoding [Rav+20a; Ami+20], deoders [Sch+20a], polynomial multiplication [HCY19], error correcting codes [DAn+19], or the IND-CCA2-transform [GJN20; Rav+20b]. NIST specifically asked the scientific community to assist in the evaluation of Kyber from a side-channel perspective [Ala+20], which was done in the co-authored work [Bos+21]. We reproduce this work in the following, with adaptions, and point out & fix an off-by-one-error.

While block ciphers are typically protected just by using Boolean masking, PQC schemes often require a mixture of both arithmetic and Boolean masking in order to be implemented efficiently. Efficient and secure conversions, e.g., [CGV14; Bar+18], between these two masking types play an essential role in protecting such schemes.

Also, IND-CCA2 security is required which is typically achieved with the FO transformation [FO99]. Hence, it is necessary to protect most modulus of the resulting IND-CCA2 scheme and not only the ring learning-with-errors (R-LWE) core. An initial first-order masking scheme of a complete KEM, similar to NewHope [Bos+15; Alk+16] and relying on the concepts of [Rep+15], was presented at CHES'18 [Ode+18]. [Sch+19] and [Bac+20]

propose higher-order efficient masked implementations of a binomial sampler and a polynomial comparison as used in many schemes. However, in [Bha+21] an attack was presented on a masked implementation of R-LWE implementations. The authors show that the first-order masked comparison of [Ode+18] and the higher-order version of [Bac+20] are vulnerable to side-channel attacks, emphasizing the need to formally verify designs.

A challenge when protecting against side-channel attacks is the fact that many popular schemes, such as Kyber, use a prime modulus. Masking schemes with prime modulus results in a significant performance overhead compared to schemes with power-of-two modulus, which allow more efficient bit-operations and conversions [Mig+19; GR19]. Due to the usage of such prime moduli in PQC schemes many prior algorithms needed to be adapted to fit this specific use case. Another PQC algorithm, Saber [DAn+20], does use a power-of-two moduli for its operations, and it has been shown how to turn this into an efficient first-order protected scheme [Bei+20]. An attack on this masked Saber implementation was subsequently presented by Ngo et al. who apply deep learning power analysis in combination with a lattice reductions step to recover the long-term secret key used [Ngo+21]. This attack does not invalidate the first-order masking scheme of [Bei+20], but rather efficiently exploits higher-order leakages due to insufficient noise for this protection level. Therefore, generic solutions to thwart it are increasing the masking order or the noise level of the device executing the implementation.

4.1. Background

We give a brief introduction on Kyber, specifically the functions that process secret-key-dependent material and therefore need to be masked. For the full description of Kyber, we refer to [Sch+20b].

Chapter Specific Notation Vectors are denoted by $\vec{b}$ and matrices by $\hat{a}$. Their counterpart in NTT domain is additionally underlined, *i.e.*, $\underline{\vec{b}}$, respectively $\underline{\hat{a}}$. Given a polynomial $f \in \mathbb{K}_q$, the i-th coefficient of f is denoted as f_i. Given a bitstring $b \in \mathbb{Z}_2^k$, the i-th bit of b is denoted as b_i.

Compression, Decompression and Sampling As an ingredient of Kyber we need to define the centered binomial distribution CBD_η, for a positive integer η. Sampling from this distribution is achieved by sampling 2η elements $(A_i)_{i=[\eta]}, (B_i)_{i=[\eta]}$ uniformly i.i.d. from $\{0,1\}$ and outputting $\sum_{i=[\eta]}(A_i - B_i)$. As a first building block of Kyber, we define a function $\mathsf{Compress}_q(x, d)$ that

takes an element $x \in \mathbb{Z}_q$ and outputs an integer in $\{0, \ldots, 2^d - 1\}$, where $d < \lceil \log_2(q) \rceil$ and $q = 3329$ a parameter of Kyber. We furthermore define a function $\mathsf{Decompress}_q$, such that $x' = \mathsf{Decompress}_q(\mathsf{Compress}_q(x, d), d)$ is an element close to x, more specifically $|x' - x \bmod^{\pm} q| \leq B_q := \lceil \frac{q}{2^{d+1}} \rfloor$. The functions satisfying these requirements are defined as

$$\mathsf{Compress}_q(x, d) = \lceil (2^d/q) \cdot x \rfloor \bmod 2^d \text{ and } \mathsf{Decompress}_q(x, d) = \lceil (q/2^d) \cdot x \rfloor.$$

When $\mathsf{Compress}_q$ or $\mathsf{Decompress}_q$ is used with $x \in \mathbb{K}_q$ or $\vec{x} \in \mathbb{K}_q^k$, the procedure is applied to each coefficient individually.

As a second ingredient, there is the sampling function CBD which converts uniformly random bytes into polynomials whose coefficients are distributed as CBD_η. We refer to the original work for additional details on the algorithm [Bos+21]. Compression, decompression, and sampling will need to be masked.

Kyber PKE Given a parameter k the M-LWE hardness relies on the hardness of distinguishing samples $(\vec{A}_i, B_i) \in \mathbb{K}_q^k \times \mathbb{K}_q$, where all elements are uniformly drawn, from those where the elements of $\vec{A}_i$ are drawn from a uniform distribution and $B_i = \vec{A}_i^T \vec{s} + E_i$ for error E_i sampled from CBD_η i.i.d. for each pair and a fixed secret $\vec{s}$ sampled from CBD_η^k.

The indistinguishability under chosen-plaintext attack (IND-CPA)-secure Kyber public-key encryption (PKE) scheme consists of three algorithms; key generation, encryption (Algorithm B.2) and decryption (Algorithm B.3). Kyber.CPAPKE is parameterized by $n_{\mathrm{K}}, k, q, \eta_1, \eta_2, d_u$ and d_v. The recommended parameter sets are listed in Table B.1. We omit key generation as it only processes a secret key once, and masking is aimed at mitigating multi-trace attacks.

Kyber KEM An IND-CCA2-secure KEM, denoted Kyber.CCAKEM, can be constructed from the Kyber.CPAPKE scheme by applying a version of the FO transform [FO99; HHK17]. The resulting scheme consists of key generation, encapsulation, and decapsulation schemes. The high-level decapsulation, denoted by Kyber.CCAKEM.Dec, is of main interest in the following since this is the only part affected by our masking techniques: its description is given in Algorithm B.1. We refer to [Sch+20b] for details on the key-derivation function KDF and the hash functions $\mathsf{K_G}, \mathsf{K_H}$.

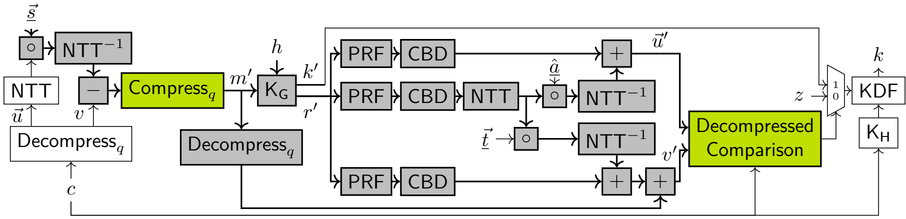


Figure 4.1.: Overview of the components in the KYBER CCA decapsulation. Components that need to be protected/masked are colored gray or green; we present new approaches for the green components.

4.2. Masking Kyber at Arbitrary Order

KYBER first applies an IND-CPA decryption to the ciphertext in order to create a message m. This message is then re-encrypted with the IND-CPA encryption and the resulting ciphertext is compared with the original input. Depending on the Boolean result of this comparison, a session-key k is derived either from the message if the ciphertext and the original input are the same or from a secret fixed value z.

A graphical overview of the various modules in KYBER decapsulation is given in Figure 4.1; the colored components are those that need to be masked. The decapsulation is deterministic and therefore all modules that process sensitive data that is derived from the long-term secret $\vec{s}$ need to be protected against SCA. In this section, we first focus on the two green $\mathsf{Compress}_q$ and $\mathsf{DecompressedComparison}$ modules. We present two new approaches for these modules: masked one-bit compression (Section 4.2.1) and masked comparison (Section 4.2.2). For each, we first provide the basic intuition about their functionality and then prove their t–SNI security in the probing model. We then put the components together to achieve a fully masked KYBER in Section 4.2.3. The proofs of t–SNI are conducted for $n_s = t+1$ in the ISW probing model $[\![\mathsf{G}]\!]^{\text{ISW}}$.

4.2.1. Higher-Order Masked One-Bit Compression

For SABER, where the used modulus is a power-of-two, the compression operation represents a shift of the sensitive value: this can be efficiently masked using LUTs as demonstrated in [Bei+20]. For schemes that use a prime modulus, masking this step is more involved. In [Ode+18] a first-order masked solution based on two mask conversions is proposed: one arithmetic-to-arithmetic

(A2A) and one arithmetic-to-Boolean (A2B) conversion per polynomial coefficient. To improve efficiency and allow extensions to higher orders, we propose a new approach that works for *any modulus* and at *any order* that requires only *one conversion per coefficient.*

Informally, the compression to one bit in KYBER splits the domain of each polynomial coefficient into two disjoint intervals and assigns a bit value depending on which interval the value of the coefficient is contained. In KYBER, this is done with the function $\mathsf{Compress}_q(x, d) = \lceil (2^d/q) \cdot x \rfloor \bmod 2^d$. The compression to one-bit results in the following mapping

$$\mathsf{Compress}_q(x, 1) = \lceil (2/q) \cdot x \rfloor \bmod 2 = \begin{cases} 1 & \text{if } \frac{q}{4} < x < \frac{3q}{4}, \\ 0 & \text{otherwise.} \end{cases}$$

This computation is trivial without masks but poses a challenge when q is prime and masking is required. When the modulus is a power of two, less-than-comparisons can be computed using a B2A conversion [Ode+18]. However, for prime moduli, the value space is not equally divided by specific bits. Masking $\mathsf{Compress}_q$ requires either the use of pre-computed tables or a dedicated masked-compression algorithm.

Let us first recall the first-order based approach from [Ode+18]. Given a masked coefficient $A^{(\cdot)_A}$, they first apply an A2A conversion to produce a masked coefficient $B^{(\cdot)_A}$ with a power-of-two modulus such that

$$\mathsf{ADec}^{n_s}_{q,t}\left(\left(A^{(i)_A}\right)_{i\in[n_s]}\right) = \mathsf{ADec}^{n_s}_{2^k,t}\left(\left(B^{(i)_A}\right)_{i\in[n_s]}\right) = a_i,$$

where $2^k > q$. Next, an appropriate offset is subtracted from $B^{(\cdot)_A}$ such that the most significant bit (MSB) of the sensitive variable denotes the value to which the coefficient should be compressed. This shared bit can be extracted from the Boolean shares after applying an A2B conversion. Hence, this technique requires one A2A and one A2B conversion per coefficient a_i. Given that these conversions are usually quite expensive this introduces a significant overhead. Furthermore, to the best of our knowledge, there are no known results for higher-order A2A conversion for arbitrary moduli. The only other published solution in this direction is presented in [Bei+20] and applies only to power-of-two moduli.

We present a solution that can be applied in first- and higher-order settings, can be applied to the setting where a prime modulus is used, and is faster by omitting one A2A conversion per coefficient compared to the state-of-the-art. In the remainder we introduce the method with a focus on the KYBER application. However, it should be noted that this approach works for any

Algorithm 4.1 Masked version of $\mathsf{Compress}_q(x,1) = \mathsf{Compress}^{\mathsf{s}}_q(x + \lfloor \frac{3q}{4} \rfloor \bmod q)$ as used in KYBER for any order using one A2B conversion per coefficient.

Input: Arithmetic sharings $A_i^{(\cdot)_A}$ of the coefficients of polynomial $a \in \mathbb{Z}_q[X]$.
Output: A Boolean sharing $M'^{(\cdot)_B}$ of message $m' = \mathsf{Compress}_q(a,1) \in \mathbb{Z}_{2^{256}}$.
1: **for** $i = 0$ **to** 255 **do**
2: $\quad A_i^{(0)_A} = A_i^{(0)_A} + \lfloor \frac{3q}{4} \rfloor \bmod q$ ▷ [Bos+21]: $A_i^{(0)_A} + \lfloor \frac{q}{4} \rfloor \bmod q$
3: $\quad A_i^{(\cdot)_B} = \mathsf{A2B}(A_i^{(\cdot)_A})$
4: $X^{(\cdot)_B} = \mathsf{Bitslice}(A^{(\cdot)_B})$
5: $M'^{(\cdot)_B} = \mathsf{SecAND}(\mathsf{SecREF}(\neg X_8^{(\cdot)_B}), X_7^{(\cdot)_B})$
6: $M'^{(\cdot)_B} = \mathsf{SecREF}(\mathsf{SecXOR}(M'^{(\cdot)_B}, X_8^{(\cdot)_B}))$
7: $M'^{(\cdot)_B} = \mathsf{SecAND}(M'^{(\cdot)_B}, X_9^{(\cdot)_B})$
8: $M'^{(\cdot)_B} = \mathsf{SecAND}(M'^{(\cdot)_B}, X_{10}^{(\cdot)_B})$
9: $\tilde{X}_{11}^{(\cdot)_B} = \neg X_{11}^{(\cdot)_B}$ ▷ [Bos+21]: Inlined in next operation.
10: $M'^{(\cdot)_B} = \mathsf{SecAND}(M'^{(\cdot)_B}, \tilde{X}_{11}^{(\cdot)_B})$
11: $M'^{(\cdot)_B} = \mathsf{SecXOR}(M'^{(\cdot)_B}, \tilde{X}_{11}^{(\cdot)_B})$ ▷ [Bos+21]: $\mathsf{SecXOR}(M'^{(\cdot)_B},\ X_{11}^{(\cdot)_B})$
12: **return** $M'^{(\cdot)_B}$

modulus q. We start with adding the offset $\lfloor \frac{3q}{4} \rfloor = 2496$ modulo q from the arithmetic shares with a subsequent A2B conversion to create k-bit Boolean shares of the coefficient, where $k = \lceil \log_2(q) \rceil = 12$. Given these Boolean shares, it then suffices to securely compute whether the masked value is greater than or equal to $\frac{q}{2}$. Let us denote this shifted function as $\mathsf{Compress}^{\mathsf{s}}_q(x)$ such that $\mathsf{Compress}_q(x,1) = \mathsf{Compress}^{\mathsf{s}}_q(x + \lfloor \frac{3q}{4} \rfloor \bmod q)$ where

$$\mathsf{Compress}^{\mathsf{s}}_q(x) := \begin{cases} 1 & \text{if } x < \frac{q}{2}, \\ 0 & \text{otherwise.} \end{cases}$$

To compute $\mathsf{Compress}^{\mathsf{s}}_q$ in a masked fashion, we perform a masked binary search on the Boolean-shared bits of the coefficient starting from the MSB. For example, if the MSB is set to 1, we can ignore the values of all subsequent bits and compress the coefficient to 0 as $2^{\mathsf{MSB}} = 2^{k-1} = 2^{11} > \frac{q}{2}$. If the MSB is set to 0, the remaining bits need to be taken into account. This process is repeated until all possible coefficient values have been mapped to a single-bit value. For the case of KYBER, $\lfloor \frac{q}{2} \rfloor = 1664$, bits 11 to 7 are taken into account.

The boundary case $x = 832$ with $\mathsf{Compress}_q(832,1) = 0$ is incorrectly compressed to 1 in the original publication [Bos+21], which we call an off-by-one error. We thank Markku-Juhani O. Saarinen for pointing this out. Note,

$\mathsf{Compress}^{\mathsf{s}}_q$ maps two input intervals of size $\left\lfloor \frac{q}{2} \right\rfloor = 1664$ and $\left\lceil \frac{q}{2} \right\rceil = 1665$. Hence, the off-by-one error is fixed by swapping the intervals, except for the erroneous value, and negating the output, which we implement with our adjusted offset $\left\lfloor \frac{q}{4} \right\rfloor - 1665 \bmod q = \left\lfloor \frac{q}{4} \right\rfloor - \left\lceil \frac{q}{2} \right\rceil \bmod q = \left\lfloor \frac{3q}{4} \right\rfloor \bmod q = 2496$.

Now, we compute $\mathsf{Compress}^{\mathsf{s}}_q$, following the original approach but with a different offset applied, by checking if $x \geq \frac{q}{w}$ and negating the result. The operation count and complexity estimates remain unchanged as the intermediate $\neg x_{11}$ can be reused instead of negating the result, see Eq. (4.1). Our benchmarks and estimates require no update since the masked implementation of Algorithm 4.1 uses $\neg \tilde{X}_{11}^{(\cdot)_B}$ instead of $X_{11}^{(\cdot)_B}$ in Line 11. Hence, it differs from the fixed design in a single additional superfluous negation such that all original benchmarks and security assessments remain valid.

$$\begin{aligned} \mathsf{Compress}^{\mathsf{s}}_q(x) &= \neg\left(x_{11} \oplus \left(\neg x_{11} \cdot x_{10} \cdot x_9 \cdot \left(x_8 \oplus \left(\neg x_8 \cdot x_7\right)\right)\right)\right) \\ &= \left(\neg x_{11}\right) \oplus \left(\neg x_{11} \cdot x_{10} \cdot x_9 \cdot \left(x_8 \oplus \left(\neg x_8 \cdot x_7\right)\right)\right). \end{aligned} \tag{4.1}$$

In a masked implementation, the $\oplus$ and $\cdot$ operations should be replaced with calls to their secure counterparts (SecXOR and SecAND). Moreover, to improve efficiency, we can first transform the Boolean shares of the polynomial to a bit-sliced representation and compute the compress function for all coefficients in parallel (limited by the word size of the target platform). This complete masked algorithm for $\mathsf{Compress}_q(x, 1)$ is given in Algorithm 4.1. Note that the algorithm is independent of the specific masked algorithms used for the modules A2B, Bitslice, SecAND, SecREF, and SecXOR. Instead, we provide a short description of the computed functionality and the assumed security property for the proof. A2B denotes a t–SNI secure conversion of arithmetic shares with a prime modulus to Boolean shares encoding the same value. In our higher-order implementations, we use Algorithm 3 from [Sch+19]. Bitslice maps a Boolean-masked polynomial to its Boolean-masked bitsliced representation. This is a linear function and can, therefore, be computed on each share separately. The most efficient way to accomplish this strongly depends on the capabilities of the target platform. In our implementation, we realized it as a sequence of bitshift, bitwise OR, and bitwise AND to rearrange the bits share by share. With SecAND and SecREF, we describe t–SNI algorithms to compute the masked bitwise AND and refresh Boolean shares. SecXOR refers to the t–NI computation of the bitwise XOR of Boolean shares. Furthermore, we use $\neg$ to indicate the negation of only the first share of the Boolean-masked input.

Correctness For Kyber, we use $q = 3329$ with the parameters $k = 12$, $\left\lfloor \frac{3q}{4} \right\rfloor = 2496$ and $\left\lfloor \frac{q}{2} \right\rfloor = 1664$. Let us provide the detailed steps to derive

the equation to compute the compression operation $\mathsf{Compress}^{\mathsf{s}}_q(x)$ using only XOR, AND, and negation.

1. $2^{11} > 1664$: If $x_{11} = 1$, then x should be compressed to 0. Otherwise, we need to consider less significant bits, therefore: $\mathsf{Compress}^{\mathsf{s}}_q(x) = \neg(x_{11} \oplus (\neg x_{11} \cdot (\dots)))$.

2. $2^{10} < 1664$: If $x_{10} = 0 \wedge x_{11} = 0$, then x should be compressed to 1. Otherwise, we need to consider less significant bits: $\mathsf{Compress}^{\mathsf{s}}_q(x) = \neg(x_{11} \oplus (\neg x_{11} \cdot x_{10} \cdot (\dots)))$.

3. $2^{10} + 2^9 < 1664$: If $x_9 = 0 \wedge x_{10} = 1 \wedge x_{11} = 0$, then x should be compressed to 1. Otherwise, we need to consider less significant bits: $\mathsf{Compress}^{\mathsf{s}}_q(x) = \neg(x_{11} \oplus (\neg x_{11} \cdot x_{10} \cdot x_9 \cdot (\dots)))$.

4. $2^{10}+2^9+2^8 > 1664$: If $x_8 = 1 \wedge x_9 = 1 \wedge x_{10} = 1 \wedge x_{11} = 0$, then x should be compressed to 0. Otherwise, we need to consider less significant bits: $\mathsf{Compress}^{\mathsf{s}}_q(x) = \neg(x_{11} \oplus (\neg x_{11} \cdot x_{10} \cdot x_9 \cdot (x_8 \oplus (\neg x_8 \cdot (\dots)))))$.

5. $2^{10}+2^9+2^7 = 1664$: If $x_7 = 1 \wedge x_9 = 1 \wedge x_{10} = 1 \wedge x_{11} = 0$, then x should be compressed to 0. All remaining combinations should be compressed to 1 since they are necessarily < 1664, therefore: $\mathsf{Compress}^{\mathsf{s}}_q(x) = \neg(x_{11} \oplus (\neg x_{11} \cdot x_{10} \cdot x_9 \cdot (x_8 \oplus (\neg x_8 \cdot x_7))))$.

Complexity The asymptotic run-time complexity for Algorithm 4.1 is estimated as $\mathcal{T}_{A_{4.1}} = \mathcal{O}\left(n_s^2 \cdot \log_2(k)\right)$ for a constant p and the randomness complexity as $\mathcal{R}_{A_{4.1}} = \mathcal{O}\left(n_s^2 \cdot \log_2(k)\right)$. See [Bos+21] for the derivation.

Security To argue about the higher-order security of Algorithm 4.1, we prove it to be t–SNI with $n_s = t+1$ shares. To this end, we iterate over all possible intermediate variables, starting from the output, and provide formal arguments on how they can be simulated relying on the t–(S)NI properties of the modules. In this step, it is important to ensure that the simulation of t_x probes on one intermediate variable does not require more than t_x shares of another intermediate variable.

Theorem 4.1 (t–SNI of Algorithm 4.1)**.** *Let $A^{(\cdot)_A}$ be the input and let $M'^{(\cdot)_B}$ be the output of Algorithm 4.1. For any set of $t_{A_{4.1}} \in [n_s]$ (the number of observations) observations on intermediate variables and any subset $\mathcal{O} \subset [n_s]$ of output indices with $t_{A_{4.1}} + |\mathcal{O}| \leq t$, there exists a subset $\mathcal{I}$ of input indices with $|\mathcal{I}| \leq t_{A_{4.1}}$ such that the $t_{A_{4.1}}$ observations as well as output shares $\left(M'^{(i)_B}\right)_{i\in\mathcal{O}}$ can be perfectly simulated from $\left(A^{(i)_A}\right)_{i\in\mathcal{I}}$.*

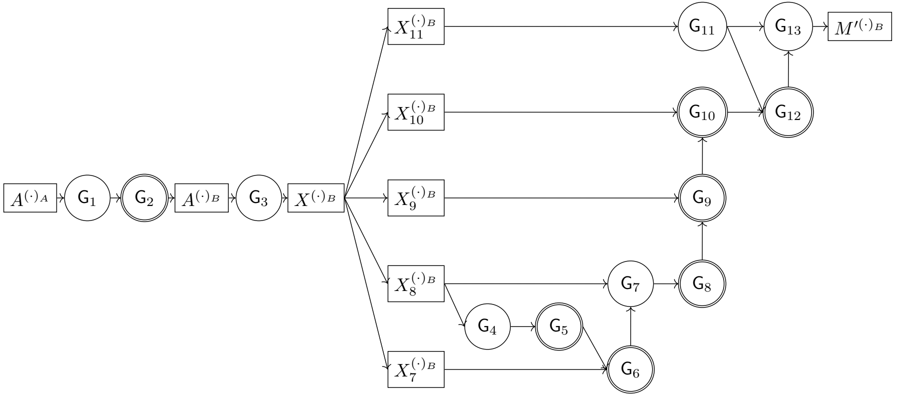


Figure 4.2.: The gadgets considered in the proof of Theorem 4.1. t–NI gadgets are depicted with a single circle and t–SNI gadgets with a double circle.

Proof. We model Algorithm 4.1 as a sequence of t–(S)NI gadgets as depicted in Figure 4.2. The linear operations are modeled as t–NI gadgets since the operations process the inputs share-wise. Furthermore, as the iterations of the initial loop are independent, we consider them to be executed in parallel and summarize them into single gadgets, one for Line 2 and one for Line 3. The exact mapping of gadgets in Figure 4.2 to Algorithm 4.1 is as follows:

- G_1 (NI): Subtraction in Line 2.
- G_2 (SNI): A2B in Line 3.
- G_3 (NI): Bitslice in Line 4.
- G_4 (NI): $\neg$ in Line 5.
- G_5 (SNI): SecREF in Line 5.
- G_6 (SNI): SecAND in Line 5.
- G_7 (NI): SecXOR in Line 6.
- G_8 (SNI): SecREF in Line 6.
- G_9 (SNI): SecAND in Line 7.
- G_{10} (SNI): SecAND in Line 8.
- G_{11} (NI): $\neg$ in Line 9.
- G_{12} (SNI): SecAND in Line 10.
- G_{13} (NI): SecXOR in Line 11.

The number of internal (resp. output) probes for gadget G_i is denoted as t_{G_i} (resp. o_{G_i}) with

$$t_{A_{4.1}} = \sum_{i=1}^{13} t_{\mathsf{G}_i} + \sum_{i=1}^{12} o_{\mathsf{G}_i}, \quad |\mathcal{O}| = o_{\mathsf{G}_{13}}$$

where $t_{A_{4.1}}$ and $|\mathcal{O}|$ refer to the number of probes and output shares of the complete Algorithm 4.1. To prove Theorem 4.1, we show that the internal probes and output shares can be perfectly simulated with $\leq t_{A_{4.1}}$ of the input shares $A^{(\cdot)_A}$. To this end, we argue about the internal probes and output shares of each gadget relying on their t–(S)NI property. In particular, we rely on the characteristic that the simulation of a t–SNI gadget can be performed independent of the number of probed output shares. This allows stopping the propagation of probes from the output shares to the input shares. For example, to simulate the $t_{\mathsf{G}_{13}}$ intermediate and $o_{\mathsf{G}_{13}}$ output probes of the t–NI gadget G_{13}, we require $t_{\mathsf{G}_{13}} + o_{\mathsf{G}_{13}}$ shares of both inputs of G_{13}, *i.e.*, the output of G_{11} and G_{12}. Throughout a larger composition, the shares required for simulation are added up. To avoid an unsound simulation, we use t–SNI gadgets to stop the propagation of probes on the output shares, e.g., the $t_{\mathsf{G}_{12}}$ intermediate and $o_{\mathsf{G}_{12}}$ output probes of the t–SNI gadget G_{12} can be simulated with only $t_{\mathsf{G}_{12}}$ input shares, *i.e.*, without $o_{\mathsf{G}_{12}}$.

In the following, we provide details for the simulation at particular points in the algorithm. The complete explanation for each gadget is provided in Appendix B.2. To simulate the internal probes and output shares of gadgets. The gadgets G_4 to G_{13} operate on individual bits $X_i^{(\cdot)_B}$ of $X^{(\cdot)_B}$. For these gadgets the following number of shares of bits $X_7^{(\cdot)_B}$ to $X_{11}^{(\cdot)_B}$ are needed:

$$
\begin{aligned}
t_{X_7^{(\cdot)_B}} &= t_{\mathsf{G}_6}, & t_{X_8^{(\cdot)_B}} &= t_{\mathsf{G}_4} + o_{\mathsf{G}_4} + t_{\mathsf{G}_5} + t_{\mathsf{G}_7} + o_{\mathsf{G}_7} + t_{\mathsf{G}_8}, \\
t_{X_9^{(\cdot)_B}} &= t_{\mathsf{G}_9}, & t_{X_{10}^{(\cdot)_B}} &= t_{\mathsf{G}_{10}}, \\
t_{X_{11}^{(\cdot)_B}} &= t_{\mathsf{G}_{11}} + o_{\mathsf{G}_{11}} + t_{\mathsf{G}_{12}} + t_{\mathsf{G}_{13}} + o_{\mathsf{G}_{13}}.
\end{aligned}
$$

The sole difference between our fixed algorithm and the original publication is that in our version G_{13} receives the output of G_{11} as input while in [Bos+21] it receives $X_{11}^{(\cdot)_B}$ as input, see Line 11 of Algorithm 4.1. Overall, this does not alter the required shares for $t_{X_{11}^{(\cdot)_B}}$ and $t_{\mathsf{G}_{11}}$ such that the proofs remain identical, with a sole difference for G_{13} pointed out in Appendix B.2.

We calculate the number of probes on the output $X^{(\cdot)_B}$ of G_3 (Bitslice) as the sum of shares needed for the simulation of each individual bit, *i.e.*, $t_{X^{(\cdot)_B}} = \sum_{i=7}^{11} t_{X_i^{(\cdot)_B}}$. The simulation of Bitslice can be performed only if there are no duplicate entries in the sum: without the t–SNI refresh G_5, the simulation would require t_{G_6} shares of both $X_7^{(\cdot)_B}$ and $X_8^{(\cdot)_B}$. In effect, $t_{X^{(\cdot)_B}}$ would be $\geq 2 \cdot t_{\mathsf{G}_6}$, which cannot be simulated for, e.g., $t_{\mathsf{G}_6} = t$. Therefore, it is necessary to refresh[1] the input to G_6, and analogously to G_9. For the other

[1] Refreshing may be avoided by gadgets which conform to stricter security notions, e.g., PINI [CS20].

SecAND gadgets, this issue does not occur and, therefore, we do not need to refresh their inputs.

Given the t–NI property of Bitslice, we can simulate the $t_{X^{(\cdot)_B}}$ shares of $X^{(\cdot)_B}$ with the corresponding number of shares of $A_i^{(\cdot)_B}$. Following the flow through gadgets G_2 and G_1, the simulation of Algorithm 4.1 requires $|\mathcal{I}| = t_{\mathsf{G}_1} + o_{\mathsf{G}_1} + t_{\mathsf{G}_2}$ of the input shares $A^{(\cdot)_A}$. In particular, the t–SNI property of G_2 allows simulating the shares of all $A_i^{(\cdot)_B}$ with only t_{G_2} of its input, *i.e.*, it is independent of the number of the probes on $A_i^{(\cdot)_B}$, which stops the propagation of $t_{X^{(\cdot)_B}}$ to $|\mathcal{I}|$. As $|\mathcal{I}| \leq t_{A_{4.1}}$ and independent of o_{13}, Algorithm 4.1 is t–SNI. □

Extension $\mathsf{Compress}_q(x, d)$ **for** $d > 1$ We use Algorithm 4.1 for compression to $d = 1$ bits, but it can be adapted to create masked compression functions for $d > 1$ bits as well. To this end, it is necessary to derive the Boolean equations for each of the d output bits, analogous to Eq. (4.1). These are then computed using instantiations of SecXOR and SecAND with independent shared inputs. A generic description of Algorithm 4.1 for any d would, therefore, need to refresh the input to any SecAND, which would induce a significant overhead. In this section an optimized version for $d = 1$, as used in KYBER, is provided. The creation of optimized versions for other d is straightforward when using formal verification tools to check which of the refreshes are needed, e.g., SCVERIF. In the following section, we develop a dedicated technique to avoid masking the ciphertext compression of KYBER.CPAPKE.Enc, *i.e.*, extending Algorithm 4.1 to $d > 1$, as this would require to process all input bits.

4.2.2. Higher-Order Masked Comparison

The masked ciphertext comparison requires computing $c \stackrel{?}{=} c'$ in a masked fashion, which assumes prior ciphertext compression in KYBER.CPAPKE.Enc. More explicitly, the comparison verifies for $d_u, d_v \in \{4, 5, 10, 11\}$ according to Table B.1, secret dependent polynomials $\vec{u}' \in \mathbb{Z}_q[X]^k, v' \in \mathbb{Z}_q[X]$ and public ciphertext $c \in \mathbb{Z}_2^{256\cdot(d_u\cdot k+d_v)}$ whether

$$(\mathsf{Compress}_q(\vec{u}', d_u), \mathsf{Compress}_q(v', d_v)) \stackrel{?}{=} c\,. \tag{4.2}$$

For the ciphertext compression, we know of no efficient higher-order solution beyond generic approaches, e.g., masked LUTs, when using prime moduli. In [Ode+18] a hash-based first-order comparison approach is proposed. However, this only checks for equality and is independent of the ciphertext compression. To apply this technique to KYBER, it would be necessary to

perform a masked ciphertext compression as a prior step. In [Bac+20], a higher-order polynomial comparison is proposed which also checks for equality but suffers from similar drawbacks as the techniques from [Ode+18]. Note that this approach was also shown to be flawed in [Bha+21] and the proposed fix significantly reduces the performance.[2] Given that none of the prior-art solutions work without a masked ciphertext compression, we propose a new masked algorithm to perform the comparison between a *masked uncompressed ciphertext* (*i.e.*, output of our masked re-encryption) and a *public compressed ciphertext*.

The core idea is to not perform the costly masked compression of the sensitive values but work the other way around: decompressing the public ciphertext. Since c is public information this can be done efficiently, *i.e.*, without masking. Informally, this changes Eq. (4.2) to

$$(\vec{u}', v') \stackrel{?}{=} \mathsf{Decompress}_q(c). \tag{4.3}$$

Since the compression is lossy, one cannot simply check for equality. Instead, one has to perform a masked range check for each coefficient to verify that the uncompressed sensitive values fall into the decompressed interval. In particular, one has to first derive the interval start- and end-point for a compressed coefficient using public functions S and E. These border values are then subtracted from the compressed masked coefficients separately; which is efficient given that they are arithmetically masked. Then each of these values is transformed to Boolean masking to extract the MSB which contains the result of the coefficient interval check: the MSB can be viewed as something similar to the "sign" bit, for a more detailed explanation see the correctness paragraph below.

If the compressed coefficient is indeed inside the desired interval, the MSB of both range checks should be one. For the comparison, we need to combine the interval checks of all coefficients into one masked output bit. This is achieved using bit-sliced calls to SecAND until one bit remains, which is set to one iff Eq. (4.3) is fulfilled. The complete masked algorithm for the $\mathsf{DecompressedComparison}$ is given in Algorithm 4.2. Again we provide a short description of the new modules and their assumed security property. MSB extracts the Boolean-masked most significant bit of the given input Boolean shares, which is assumed to be t–NI as it can be applied on each share separately. LSR refers to the sharewise logical shift to the right of input Boolean shares by a given offset, which is also assumed to be t–NI.

[2]Note that [Bha+21] also shows a flaw in the implementation of the masked comparison of [Ode+18], but this one can be trivially fixed without impacting the performance.

Algorithm 4.2 Masked $\mathsf{DecompressedComparison}$ as used in Kyber.

Input: 1. Arithmetic sharing $\vec{U}'^{(\cdot)_A}$ of a vector of polynomials $\vec{u}' \in \mathbb{Z}_q[X]^k$,
2. Arithmetic sharing $V'^{(\cdot)_A}$ of a polynomial $v' \in \mathbb{Z}_q[X]$,
3. Ciphertext $c \in \mathbb{Z}_2^{256\cdot(d_u\cdot k+d_v)}$,
4. Two public functions S and E defined by Kyber which specify the start- and end-points of the intervals in compression.

Output: A Boolean sharing $B^{(\cdot)_B}$ of b where $b = 1$ if and only if $(\mathsf{Compress}_q(\vec{u}', d_u), \mathsf{Compress}_q(v', d_v)) = c$, otherwise $b = 0$.

1: **function** DecompressedComparison
2: $(\vec{u}'', v'') = \mathsf{Decode}\,(c)$
3: $T_{\vec{w}}^{(\cdot)_B}, T_{\vec{x}}^{(\cdot)_B} = \mathsf{PolyCompare}(\vec{U}'^{(\cdot)_A}, \vec{u}'')$
4: $T_{y}^{(\cdot)_B}, T_{z}^{(\cdot)_B} = \mathsf{PolyCompare}(V'^{(\cdot)_A}, v'')$
5: $B^{(\cdot)_B} = \mathsf{SecAND}(\mathsf{SecAND}(T_{\vec{w}}^{(\cdot)_B}, T_{\vec{x}}^{(\cdot)_B}), \mathsf{SecAND}(T_{y}^{(\cdot)_B}, T_{z}^{(\cdot)_B}))$
6: **for** $i = \log_2 256 - 1$ **to** 0 **do**
7: $T_b^{(\cdot)_B} = \mathsf{LSR}(B^{(\cdot)_B},\ 2^i)$
8: $B^{(\cdot)_B} = B^{(\cdot)_B} \bmod (2^{2^i} - 1)$
9: $B^{(\cdot)_B} = \mathsf{SecAND}(B^{(\cdot)_B}, T_b^{(\cdot)_B})$
10: **return** $B^{(\cdot)_B}$

11: **function** PolyCompare$(U'^{(\cdot)_A}, u'')$
12: **for** $i = 0$ **to** 255 **do**
13: $s_{u''} = S(u''_i)$
14: $e_{u''} = E(u''_i)$
15: $W_i^{(\cdot)_A} = X_i^{(\cdot)_A} = U_i'^{(\cdot)_A}$
16: $W_i^{(0)_A} = (W_i^{(0)_A} + 2^{\lceil \log_2(q) \rceil - 1} - s_{u''}) \bmod q$
17: $X_i^{(0)_A} = (X_i^{(0)_A} - e_{u''}) \bmod q$
18: $W_i^{(\cdot)_B} = \mathsf{MSB}(\mathsf{A2B}(W_i^{(\cdot)_A}))$
19: $X_i^{(\cdot)_B} = \mathsf{MSB}(\mathsf{A2B}(X_i^{(\cdot)_A}))$
20: **return** $\mathsf{Bitslice}(W^{(\cdot)_B})$, $\mathsf{Bitslice}(X^{(\cdot)_B})$

Correctness To better understand Algorithm 4.2, let us first go through the unmasked decompressed comparison using one coefficient as an example. We move the costly compression step to the public variable as $a \stackrel{?}{=} \mathsf{Decompress}_q(b)$, *i.e.*, we check if the public b would be decompressed to an interval which contains a. As the compression is lossy, there are multiple values for a which can be mapped to b through $\mathsf{Compress}_q$. Therefore, a straightforward check for

this equality would only work for one specific value of a. Instead, we denote the lower bound $\mathsf{S}(b)$ and the upper bound $\mathsf{E}(b)$ such that for $\mathsf{S}(b) \leq a \leq \mathsf{E}(b) - 1$ one has $\mathsf{Compress}_q(a) = b$. Given these pre-computed public values $\mathsf{S}(b)$ and $\mathsf{E}(b)$, we need to decide if a given a is indeed in the interval $[\mathsf{S}(b), \mathsf{E}(b) - 1]$. While this is trivial for unmasked values, a is sensitive and, therefore, this operation needs to be masked.

Performing a generic less-than comparison check is straightforward for power-of-two moduli, but challenging for prime moduli such as used in Kyber. The idea to achieve this is to compute $a - \mathsf{S}(b) \bmod q$ and $a - \mathsf{E}(b)$ and check the "sign" bits. This is done in a masked fashion by performing first an A2B and subsequently extracting the MSB of the masks. If a is indeed in the interval $[\mathsf{S}(b), \mathsf{E}(b) - 1]$ then one expects $a - \mathsf{S}(b)$ to return an MSB with a 0 while $a - \mathsf{E}(b)$ should be 1. These can be combined with a SecAND by first negating the masked bit of the first range check.

In order to combine the output of the two checks without this negation, one can shift the values in the first check appropriately (by adding $2^{\lceil \log_2(q) \rceil - 1}$). Now, both the MSBs of $a - \mathsf{S}(b) + 2^{11}$ and $a - \mathsf{E}(b)$ return 1 if $a \in [\mathsf{S}(b), \mathsf{E}(b) - 1]$ in the setting of Kyber. The resulting MSB need to be put into a SecAND to produce the masked output of the comparison for this coefficient. It should be noted that this approach requires that the size of the largest interval $[\mathsf{S}(b), \mathsf{E}(b) - 1]$ should be smaller or equal to the difference of the used modulus to the next smaller power of two; *i.e.*, $q - 2^{\lceil \log_2(q) \rceil} - 1$. For Kyber this is indeed the case for all parameter sets; for the values $d \in \{4, 5, 10, 11\}$ defined in Kyber and used in $\mathsf{Compress}_q(x, d)$ we have an interval size of at most 209 which is well below $q - 2^{\lceil \log_2(q) \rceil} - 1 = 3329 - 2^{11} = 1281$. Note that the special case where $d = 1$ is handled in detail in Section 4.2.1.

Complexity The asymptotic run-time complexity for Algorithm 4.2 is given by $\mathcal{T}_{A_{4.2}} = \mathcal{O}\left(n_s^2 \cdot \log_2(k)\right)$ for constant p and the randomness complexity is $\mathcal{R}_{A_{4.2}} = \mathcal{O}\left(n_s^2 \cdot \log_2(k)\right)$ [Bos+21].

Security We refer to [Bos+21] for the published proof of Theorem 4.2. It follows the same strategy as for Algorithm 4.1 and represent the operations in Algorithm 4.2 as sequence of gadgets, visualized in Figure 4.3. Linear operations are modeled as t–NI gadgets. Input c and its derived variables $(\vec{u}'', v'')$ and $(s_{\vec{u}''}, s_{\vec{u}''}, s_{v''}, e_{v''})$ are public values and therefore not represented. The iterations of the loops on PolyCompare are independent, hence, we consider them to be executed in parallel and summarize them into single gadgets. For clarity we represent the loop in Line 6 with two iterations, but note that the proof generalizes to any number of iterations due to the t–SNI property of

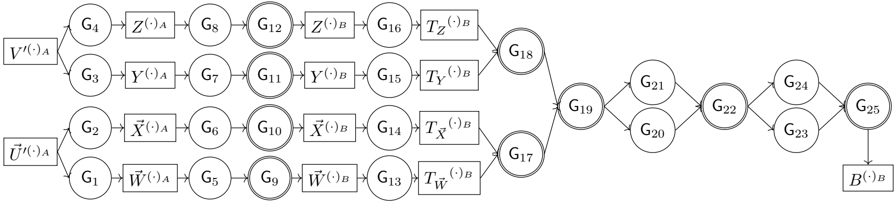


Figure 4.3.: The gadgets considered in the proof of Theorem 4.2. t–NI gadgets are depicted with a single circle and t–SNI gadgets with a double circle.

SecAND. The mapping is as follows:

- G_{1-4} (NI): Assignment in Line 15.
- G_{5-8} (NI): Linear arithmetic in Lines 16 to 17.
- G_{9-12} (SNI): MSB ∘ A2B in Lines 18 to 19.
- G_{13-16} (NI): Bitslice in Lines 20.
- G_{17-19} (SNI): SecAND in Line 5.
- $\mathsf{G}_{20,23}$ (NI): Upper half extraction in Line 7.
- $\mathsf{G}_{21,24}$ (NI): Lower half extraction in Line 8.
- $\mathsf{G}_{22,25}$ (SNI): SecAND in Line 9.

Theorem 4.2 (t–SNI of Algorithm 4.2)**.** *Let $\vec{U}'^{(\cdot)_A}$ and $V'^{(\cdot)_A}$ be the inputs and let $B^{(\cdot)_B}$ be the output of Algorithm 4.2. For any set of $t_{A_{4.2}} \in [n_s]$ (the number of observations) observations on intermediate variables and any subset $\mathcal{O} \subset [n_s]$ of output indices with $t_{A_{4.2}} + |\mathcal{O}| \leq t$, there exists a subset $\mathcal{I} = \mathcal{I}_{\vec{U}} \cup \mathcal{I}_{V'}$ of indices of each input with $|\mathcal{I}| \leq t_{A_{4.2}}$ and such that the $t_{A_{4.2}}$ intermediate variables as well as output shares $\left(B^{(i)_B}\right)_{i \in \mathcal{O}}$ can be perfectly simulated from input shares $\left(\vec{U}'^{(i)_A}\right)_{i \in \mathcal{I}_{\vec{U}}}$ and $\left(V'^{(i)_A}\right)_{i \in \mathcal{I}_{V'}}$.*

4.2.3. Higher-Order Masked Decapsulation

We now return to the Kyber decapsulation given in Figure 4.1 and reason on the SCA security of the complete decapsulation. Note that we omitted the

encode- and decode-operations as well as the generation of the public matrix $\hat{a}$ in the figure, as they are either trivial to mask or process only public values.

All linear polynomial operations (*i.e.*, $\circ$, NTT, $-$, $+$) in the components Kyber.CPAPKE.Dec and Kyber.CPAPKE.Enc are masked as in previous works by applying the operation on each share separately. For $\mathsf{Compress}_q(x, 1)$ of Kyber.CPAPKE.Dec, we rely on our new approach as presented in Section 4.2.1.

To mask the symmetric components $\mathsf{K_G}$ and PRF, we rely on prior art. In particular, we use the masked Keccak approach and implementation of [Bar+16] to instantiate the modules at higher order while we use the more efficient approach from [Ber+10] for the first order. We believe there is room for performance improvement by creating dedicated and more efficient masking schemes of Keccak aiming at a specific masking order (as done for the first-order setting), but this is out of scope for the current work.

For $\mathsf{Decompress}_q$, we first convert each bit of the Boolean-shared message $M'^{(\cdot)_B}$ to arithmetic shares modulo q (e.g., using the efficient one-bit B2A algorithm of [Sch+19] at higher orders) which are then multiplied with a constant.

To mask the sampler CBD of Kyber.CPAPKE.Enc, we adapt the bitsliced approach from [Sch+19] to the parameters of Kyber. For example, for $\eta = 2$ we first sum the input bits using Boolean-masked bit-sliced addition.

As depicted in Figure 4.1, our approach does not explicitly mask the ciphertext compression of Kyber.CPAPKE.Enc. Instead, we instantiate the comparison as presented in Section 4.2.2 which can process masked uncompressed polynomials. Furthermore, in line with [Bha+21], we collapse the result of the comparison to a single masked bit before unmasking it for the selection of the KDF input.

We follow the approach and reasoning of [Bei+20] and do not mask the KDF. Instead, if the comparison outputs true (*i.e.*, the ciphertext is valid), we unmask k' and perform an unmasked KDF. For a valid ciphertext, this leaks only ephemeral secret information and not the long-term secret. Should this short-term secret also be protected, other countermeasures besides masking can be applied to mitigate single-trace attacks. Note that it is important to not unmask k' if the comparison fails because this could be used to attack the long-term secret. If the comparison does fail, we apply an unmasked KDF to the secret value z. This value is independent of the secret key, but leaking it allows an adversary to detect ciphertext rejection explicitly. This does not impact the IND-CCA2 security claims of Kyber [HHK17, Figure 1] as Kyber is γ-spread for sufficiently large γ.

To argue about the probing security of masked Kyber.CCAKEM.Dec, we

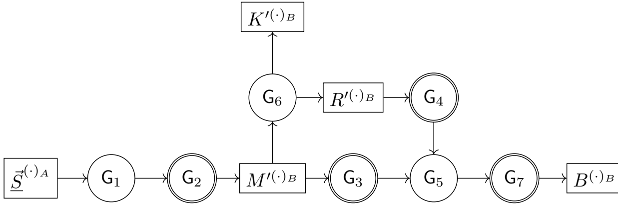


Figure 4.4.: The gadgets considered in the proof of Theorem 4.3. t–NI gadgets are depicted with a single circle and t–SNI gadgets with a double circle.

analyze a reduced composition (denoted as $\mathsf{G}_{\mathsf{Dec}}$) excluding the unmasked components. The structure of $\mathsf{G}_{\mathsf{Dec}}$ is depicted in Figure 4.4.

Theorem 4.3 (t–SNI of Decapsulation $\mathsf{G}_{\mathsf{Dec}}$)**.** *Let $\vec{\underline{S}}^{(\cdot)_A}$ be the input and let $K'^{(\cdot)_B}$ and $B^{(\cdot)_B}$ be the outputs of $\mathsf{G}_{\mathsf{Dec}}$. For any set of $t_{\mathsf{G}_{\mathsf{Dec}}} \in [n_s]$ (the number of observations) observations on intermediate variables and any subset $\mathcal{O} \subset [n_s]$ of output indices with $t_{\mathsf{G}_{\mathsf{Dec}}} + |\mathcal{O}| \leq t$, there exists a subset $\mathcal{I}$ of input indices with $|\mathcal{I}| \leq t_{\mathsf{G}_{\mathsf{Dec}}}$ such that the $t_{\mathsf{G}_{\mathsf{Dec}}}$ observations on intermediate variables as well as the output shares $\left(B^{(i)_B}, K'^{(i)_B}\right)_{i \in \mathcal{O}}$ can be perfectly simulated from the input shares $\left(\vec{\underline{S}}^{(i)_A}\right)_{i \in \mathcal{I}}$.*

Proof. We model the linear operations of the decryption and encryption as t–NI gadgets G_1 and G_5. The new t–SNI $\mathsf{Compress}_q(x, 1)$ and comparison algorithms are included as G_2 and G_7. The symmetric components are modeled as a t–NI gadget G_6. As shown in [Sch+19], the sampling algorithm is t–SNI and their proof is independent of the concrete instantiation parameters. We model it as t–SNI gadget G_4. For $\mathsf{Decompress}_q$, we assume a t–SNI gadget G_3 which relates to the t–SNI B2A conversion. The subsequent linear multiplication is included in G_5.

The number of internal (resp. output) probes G_i is denoted as t_{G_i} (resp. o_{G_i}) with

$$t_{\mathsf{G}_{\mathsf{Dec}}} = \sum_{i=1}^{7} t_{\mathsf{G}_i} + \sum_{i=1}^{5} o_{\mathsf{G}_i}, \quad |\mathcal{O}| = o_{\mathsf{G}_6} + o_{\mathsf{G}_7}$$

where $t_{\mathsf{G}_{\mathsf{Dec}}}$ and $|\mathcal{O}|$ refer to respectively the number of probes and output shares of the complete gadget $\mathsf{G}_{\mathsf{Dec}}$ as used in Theorem 4.3. To prove Theorem 4.3, we show that the internal probes and output shares can be perfectly simulated with at most $t_{\mathsf{G}_{\mathsf{Dec}}}$ of the input shares $\vec{\underline{S}}^{(\cdot)_A}$. Again, we provide

details for the simulation at particular points in the algorithm. The complete explanation for each gadget is provided in Appendix B.3.

To simulate the internal probes and output shares of gadgets G_3 to G_7, we need $t_{\mathsf{G}_3} + t_{\mathsf{G}_6} + o_{\mathsf{G}_6} + t_{\mathsf{G}_4}$ shares of $M'^{(\cdot)_B}$. Following the flow through gadgets $\mathsf{G}_{1,2}$, the simulation of $\mathsf{G}_{\mathsf{Dec}}$ requires $|\mathcal{I}| = t_{\mathsf{G}_1} + o_{\mathsf{G}_1} + t_{\mathsf{G}_2}$ of the input shares $\left(\underline{\vec{S}}^{(i)_A}\right)_{i \in \mathcal{I}}$. As $|\mathcal{I}| \leq t_{\mathsf{G}_{\mathsf{Dec}}}$ and independent of o_{G_6} and o_{G_7}, gadget $\mathsf{G}_{\mathsf{Dec}}$ is t–SNI. □

4.3. Implementation and Evaluation

We present the performance and practical security results of the new masked algorithms from Section 4.2.

We target Kyber768 since this is the parameter set for NIST security level 3. We select two platforms: firstly, benchmarks (using the SysTick timer) and physical measurements are performed on the FRDM-KL82Z [NXP16] already used in Chapter 3. Secondly, although we do not perform measurements or any hardening, we do performance benchmarks on the STM32F407G-DISC1 board that comes equipped with a Cortex-M4F (previously known as STM32F4DISCOVERY). This is the platform used by the embedded crypto benchmark platform pqm4 [Kan+19b] and recent masked implementations of Saber [Bei+20], allowing the comparison to existing work. We make use of the standard measurement framework of pqm4, with minor modifications to measure the run-time of subroutines.

In this section, a component-wise performance comparison for various orders and implementation choices is given. Our masked Kyber implementation is generally written in C and based on the C-reference code from the Round 3 Kyber submission. For the CM4F processor, we included the optimized assembly routines from pqm4, but the used assembly is incompatible with the much simpler Cortex-M0+. On the other hand, for our 1$^{\text{st}}$-order CM0+ implementation we provide low-level formal verification and physical leakage assessments based on power measurements. For this purpose, we target our own components $\mathsf{Compress}_q(.,1)$ and $\mathsf{DecompressedComparison}$. These hardened components (and any components that they rely on) are therefore written in assembly. Although hardening involves adding dummy operations that would decrease efficiency, our hand-written hardened assembly still performs better than the compiler-generated versions from the (unhardened, masked) plain-C implementation. Following the same approach as [Bei+20], we use an already existing masked implementation of Keccak in our masked Kyber implementation. More specifically, we re-use the 1$^{\text{st}}$-order masked implemen-

tation from [Ber+10] and for higher orders use the more generic higher-order secure implementation of Keccak from MaskComp [Bar+16].

Randomness Generation During the execution of decapsulation, fresh randomness is needed for the masked operations. For example, the 1st-order masked implementation on the FRDM-KL82Z uses 11 665 uniformly randomly sampled bytes for the decapsulation operation (see Table 4.2). As we would like the power measurements to be reproducible, the numbers for the FRDM-KL82Z reported in Table 4.2 assume that the random bytes can be readily read off from a table, which is filled before execution of the Kyber functions. Therefore, the cost of randomness generation is not included in our performance numbers for this platform. On the other hand, the STM32F407G-DISC1 board comes equipped with a true random number generator (TRNG). For fair comparison to existing work, we do include the randomness generation in the cycle counts on this platform.

4.3.1. Performance Comparison

The main goal of this section is to demonstrate the feasibility of the new techniques to realize a (higher-order) masked Kyber implementation. For the FRDM-KL82Z we present the results of plain-C implementations and do not optimize on assembly level except for hardening some components. That being said, our 1st-order masked implementation does aim to be efficient from an algorithmic point of view to fairly represent the performance impact. For the STM32F407G, we include the optimized assembly routines from pqm4. All implementations were compiled with `arm-none-eabi-gcc` version 8.3.1 with optimization level O3. The higher-order implementations (*i.e.*, the second and third order results in Tables 4.2 and 4.3) are not as aggressively optimized and therefore have more room left for improvement, in particular, because the existing higher-order masked Keccak implementations are not heavily optimized.

First-Order Masking Recall that we use Algorithm 4.1 for $\mathsf{Compress}_q(., 1)$, Algorithm 4.2 for DecompressedComparison and [Sch+19, Algorithm 3] for the conversion from arithmetic to Boolean shares (A2B). However, for 1st-order masking the algorithm of [Sch+19] is not the most efficient. Instead, we use a LUT based approach; more specifically, the improved [Deb12] version of the Coron–Tchulkine method [CT03]. This algorithm was designed for power-of-two moduli so cannot be used directly for our prime q. To overcome this we simply use larger tables to avoid dealing with any carry propagation.

Listing 4.1: C code of the A2B based on the LUT approach of [Deb12].

```
void A2B(boolean_share_t x, arith_share_t a) {
  uint16_t R, a0;
  rng(&R, KYBER_Q_BITSIZE);
  a0 = csubq(a[0] + KYBER_Q - r_a);
  a0 = csubq(a0 + a[1]);
  x[0] = 𝕃[a0] ^ R;
  x[1] = r_b ^ R;
}
```

Moreover, we also refresh the output with fresh randomness as the input and output masks in our case are from different domains, and to achieve the assumed t–SNI property.

Let us give a concrete example of the approach for the implementation of the 1st-order A2B conversion. The table L satisfies $\mathsf{L}(a) = (a + r_a \bmod q) \oplus r_b$, where a is a secret value in $[0, q-1]$ which is arithmetically masked with randomness $r_a \in [0, q-1]$ on the input side and a Boolean mask with the random value $r_b \in [0, 2^{\lceil \log_2(q) \rceil} - 1]$ is applied on the output side. Then the arithmetic to Boolean conversion is implemented as in Figure 4.1. Here rng(x, y) stores y uniformly random sampled bits in x, and csubq performs a conditional subtraction by the modulus q. More explicitly, the constant-time equivalent of the C-expression

```
c = ((c >= q) ? c-q : c).
```

The A2B of [Deb12] outperforms the method of [Sch+19] at 1st order, and therefore we use it in our 1st-order implementation. It is however not directly clear whether a similar approach can be used for $\mathsf{Compress}_q(., 1)$ and DecompressedComparison. Indeed, a completely analogous masked LUT can be created for these functions. For example, it is possible to replace DecompressedComparison by implementing $\mathsf{Compress}_q(., d)$ with a LUT, followed by a hashed comparison as done in previous work [Ode+18; Bei+20]. We compare the performance of the various options in Table 4.1. Concretely, we use the following four settings in our masked implementations. The first setting (denoted setting 0) uses no LUTs at all; this is the default for the higher-order (> 1) masked implementations. The 1st-order implementations do use the LUT-approach for A2B and the respective possible settings for the FRDM-KL82Z for our modules can be found in Table 4.1. The LUTs are generated fresh for every KYBER invocation and this run-time is included in the overall reported performance results (Init). Since Init, $\mathsf{Compress}_q(., 1)$ and DecompressedComparison are only called once per decapsulation, the total cost

Setting		Approach		#Cycles			
		$\mathsf{Comp}_q(.,1)$	Dec.Comp.	Init	$\mathsf{Comp}_q(.,1)$	Dec.Comp.	**Total**
HO	0	Alg. 4.1	Alg. 4.2	–	–	–	–
FO	1	LUT	LUT	2 032	65	1 407	3 504
	2	LUT	Alg. 4.2	766	66	1 232	2 064
	3	Alg. 4.1	Alg. 4.2	181	145	1 255	1 581

Table 4.1.: The different LUT settings used in our 1st-order (FO) and higher-order (HO) implementations on the FRDM-KL82Z and their corresponding cycle counts rounded up to nearest 10^3 cycles. For 1st order a LUT is used for A2B, for higher orders it is not.

is computed as the sum of the separate functions. It is clear that using the new algorithms introduced in this work is favorable compared to using the LUT-approach even in the 1st-order setting. This is due to the fact that the LUT-initialization takes a non-negligible amount of time, which is significant because the functions are only used once (as opposed to A2B). Therefore, our 1st-order implementations use setting 3.

Performance Discussion We present a complete overview of our performance results on both the FRDM-KL82Z and the STM32F407G in Tables 4.2 and 4.3. As a baseline unmasked implementation for the FRDM-KL82Z, we take the Kyber768 implementation from the PQClean [Kan+19a] software library. This is purely written in C and therefore a comparison to our plain-C implementation is fair. Of course, some of our modules contain assembly modifications to harden them against power analysis, but this leads to only minor differences in cycle counts. For the STM32F407G, we take the best optimized implementation from pqm4.

We see that the overall slowdown factor for `crypto_kem_dec` masked at 1st order on the FRDM-KL82Z is 2.2x. A large part of this can be attributed to the masked encryption step, which uses the masked PRF as part of the binomial sampler while also doubling the cost of the polynomial arithmetic. The $\mathsf{Compress}_q(.,1)$ and DecompressedComparison (denoted `comparison` in the table) also introduce large slowdowns, but this was to be expected as their cost in the unmasked version was almost negligible. It should be noted that the slowdown factor is relatively small due to the lack of assembly optimizations: since the cost of polynomial arithmetic is still significant, while it has a small slowdown factor, the overall slowdown compared to the reference implementation is brought down.

On the other hand, the slowdown factor for 1st-order masked decapsulation on the STM32F407G is 3.5x. Although comparing subroutines directly is difficult due to interleaving in the pqm4 reference, the overall slowdown is larger than on the FRDM-KL82Z as the cost of polynomials is relatively lower due to assembly optimization. More precisely, the cost of masking is dominated by KECCAK operations rather than polynomial arithmetic, which are more expensive to mask. Our slowdown factor is better than the (tentative) factor 4.2x reported recently by Heinz et al. for a 1st-order masked implementation [Dan21] at the PQC Standardization Conference organized by NIST, though their version is hardened for the Cortex-M4 while ours is unhardened. The overall slowdown is larger than the 2.52x factor reported in [Bei+20, Table 5] for masking SABER on the same platform. This is mainly caused by the high cost of the KECCAK-based binomial sampler: constructing the four error polynomials for KYBER requires the use of the binomial sampler which uses rejection sampling modulo 3329 to convert the shares from arithmetic to Boolean (256 per polynomial). A similar operation is also performed in SABER and KYBER for the generation of 3 secret polynomials. However, SABER does not need to generate these error vectors.

Unsurprisingly, the number of random bytes used by KYBER is larger than for SABER. Whereas [Bei+20] makes 1262 calls to a 32-bit TRNG, using 5048 bytes in total, we sample a total of 12 072 random bytes. This is firstly caused by the generation of additional error polynomials, as mentioned above. Secondly, the $\mathsf{Compress}_q(., 1)$ and DecompressedComparison components require more randomness compared to their counterparts in SABER and use 704 and 4396 random bytes respectively.

The performance impact for the higher-order implementations is much larger. In particular, the relative cost of DecompressedComparison compared to the whole decapsulation increases. This is mainly due to the poor performance of the A2B component for higher orders [Sch+19, Algorithm 3], as opposed to the LUT-based version for 1st order, which is both slow and requires most of the random bytes in decapsulation. We expect many optimizations are still possible in the higher-order A2B, some of them addressed in [Azo+22c].

4.3.2. Verification of Side-Channel Resilience

We follow the secure development process proposed in Chapter 1 for developing the implementations of the two proposed algorithms. The disassembled object files of our implementation are verified with SCVERIF to be Stateful t–SNI in an extension of $[\![\cdot]\!]^{\text{CM0+}}$ to ensure resistance against both the CM0+ device-specific leakage behavior and the residual state in concrete execution.

The KYBER implementations make use of instructions that are not covered

Operation	FRDM-KL82Z			
	PQClean[a]	**New**		
	(unmasked)	*1st*	*2nd*	*3rd*
`crypto_kem_dec`	5 530	12 208 (2.2x)	107 352	231 632
`LUT_create`	0	241 (∞x)	0	0
`indcpa_dec`	703	1 096 (1.6x)	6 886	18 166
`hashg`	61	361 (5.9x)	4 457	6 507
`indcpa_enc`	4 160	8 708 (2.1x)	52 132	75 864
`comparison`	5	*1 206 (241.0x)	43 270	130 489
`hashh`	530	535 (1.0x)	540	540
`kdf`	65	65 (1.0x)	66	66
#randombytes	–	11 665	901 880	2 408 880
`indcpa_enc`	4 160	8 708 (2.1x)	52 132	75 864
`decompression`	21	287 (13.7x)	644	994
`gen_at`	1 755	1 723 (1.0x)	1 771	1 748
`poly_getnoise`	494	3 729 (7.5x)	44 227	66 112
`poly_arith`	1 683	2 968 (1.8x)	5 490	7 010
#randombytes	–	6556	277 304	537 684
`indcpa_dec`	703	1 096 (1.6x)	6 886	18 166
`unpack`	53	68 (1.3x)	86	102
`poly_arith`	638	885 (1.4x)	1 388	1 710
`compress`	22	*143 (6.5x)	5 411	16 354
#randombytes	–	704	66 432	201 984

Table 4.2.: Performance benchmarks on the FRDM-KL82Z (Cortex-M0+) platform of the masked implementation of the various parts of Kyber768. The cycle counts are reported in thousands and rounded up to the nearest 10^3 cycles. The FRDM-KL82Z results do not include randomness generation. The slowdown factor of the 1st order implementation compared to PQClean is included in brackets. The ciphertext `comparison` is in compressed form for PQClean, and decompressed form for the masked components. Results marked * are hardened.

[a] `https://github.com/PQClean/PQClean/tree/master/crypto_kem/kyber768/clean` commit `c00cb2d`

Operation	STM32F407G				
	pqm4[a]	New			
	(unmasked)	*1st*		*2nd*	*3rd*
`crypto_kem_dec`	882	3 116	(3.5x)	44 347	115 481
`LUT_create`	–	37		47	47
`indcpa_dec`	–	174		2 916	9 288
`hashg`	–	118		1 543	2 659
`indcpa_enc`	–	2 196		17 743	30 838
`comparison`	–	462		22 017	72 568
`hashh`	–	115		115	115
`kdf`	–	14		14	14
#randombytes	–	12 072		902 126	2 434 170
`indcpa_enc`	676	2 196	(3.3x)	17 743	30 838
`decompression`	–	113		267	490
`gen_at`	–	398		398	398
`poly_getnoise`	–	1 384		16 625	29 347
`poly_arith`	–	301		452	603
#randombytes	–	7 030		277 550	562 974
`indcpa_dec`	64	174	(2.7x)	2 916	9 288
`unpack`	–	23		30	36
`poly_arith`	–	89		119	149
`compress`	–	61		2 767	9 102
#randombytes	–	640		66 432	201 984

Table 4.3.: Performance benchmarks on the STM32F407G (Cortex-M4F) platforms of the masked implementation of the various parts of Kyber768. The cycle counts are reported in thousands and rounded up to the nearest 10^3 cycles. The STM32F407G results include randomness generation. The slowdown factor of the 1st order implementation compared to pqm4 is included in brackets. The high-level pqm4 functions are not subdivided as they are implemented in an interleaved fashion to reduce memory use. The ciphertext `comparison` is in compressed form for pqm4, and decompressed form for the masked components.

[a] `https://github.com/mupq/pqm4/tree/master/crypto_kem/kyber768/m4` commit `157e271`

in $[\![\cdot]\!]^{\text{CM0+}}$ presented in Chapter 3, e.g., arithmetic, branching, and shift. For the mandated extension of this leakage model, we take and evaluate a different approach. We do not start by building an empirically complete model using extensive physical characterization for all the required instructions. Instead, we build hypothetical models for the missing instructions and do not test their accuracy independently. We use this unconfirmed, hypothetical model immediately for hardening the Kyber modules and evaluate the physical resilience once the implementation is verified stateful t–SNI in the hypothetical model. Any detected physical leakage is specifically analyzed using small characterization tests for the concerned instruction as in [PV17], leading to another iteration with a slightly changed model. Fast verification allows us to quickly prove the absence of vulnerabilities arising from newly learned leakage behavior without introducing regression w.r.t. already known behavior. Only a few iterations are necessary as the re-used model covers a majority of leakages and permits to a large extent successful conjectures for the added instructions. We emphasize that this confirms the benefits of verification in fine-grained leakage models for secure development (Figure 1.2) even if such models are only heuristically predicted.

The extended leakage model $[\![\cdot]\!]^{\text{CM0+}}$ is provided in Listing B.1. We omit the semantic model closely following [ARM18] thanks to the purposely designed IL. We give insight into the construction of our initial extension. For instructions with two or three operands, we mainly re-use the `and` model as boilerplate, respectively `load` or `store` for instructions accessing memory. Initially, we follow the same strategy for single operand instructions such as `mov rd, rn` and `negs rd, rn`, which copy register `rn` (resp. its negation) to `rd`. But these differ (as well as shifts involving an immediate) in that they do not alter the modeled `opB` leakage state. Hence, a sequence as in Listing 4.2 caused detectable leakage between `rB` and `rF`. This modeling error is quickly remedied *in the entire implementation* by commenting Line 63 in $[\![\cdot]\!]^{\text{CM0+}}$ (Listing B.1). An explanation for this behavior might be that the register file read port for the second operand (modeled by `opB`) is not active unless required by an instruction to reduce the power consumption of the CM0+. Branching instructions are modeled without leakage. Surprisingly, this is viable for our implementations despite the non-negligible number of branches. We leave it to future investigations to determine whether this is specific to the architecture or indirectly caused by clearing state prior to branching, which is mandated by stateful t–SNI and checked by scVerif.

In the following, we detail the changes to scVerif necessary to use it on the Kyber implementation. Kyber makes use of constants (e.g., the modulus q) which are stored alongside the program code. We extend scVerif to allow program counter (`pc`) relative memory accesses to be partially evaluated for

Listing 4.2: The leakage detected in this sequence of assembly instructions violated our initial model assumptions of the `mov` instruction and required adoption of $[\![\cdot]\!]^{\text{CM0+}}$.

```
1 ands rA, rB
2 mov  rC, rD // initially assumed to clear rB from the
     ↪ microarchitecture
3 ands rE, rF // physical leakage indicates combination of rB,
     ↪ rF
```

the subsequent verification phase. In doing so, we extend the front-end of scVerif to process the assembly `.word` directive which introduces a constant at some fixed address: pairs of addresses and constants are placed in the memory view ρ for the state $\langle \mathsf{P}, \mathfrak{i}, \mu, \rho, \mathsf{P}' \rangle$ of the partial evaluator (Section 3.2.2).

Verification of code containing table lookups, e.g., as used in the table-base A2B conversion, poses problems as it involves secret dependent memory access. Since secrets are represented as abstract symbols ($\mathfrak{v}$) they cannot be partially evaluated to a value $\mathfrak{a}$ such that rule Mem in Figure 3.3 cannot be applied. Hence, the transformation T fails to lower our implementations to the MaskVerif fragment. We resolve this by manually patching the code prior to verification and substituting the respective lookup instruction e.g., `ldr` with a contrived instruction `lut`. The contrived instruction exposes the same leakage behavior but expresses the semantics of the lookup in functional form, *i.e.*, as an expression which falls in the MaskVerif fragment of IL.

Let us give an example of this approach for the implementation of the first-order A2B as explained in Section 4.3.1. We replace the `ldr` instruction with `lut`, for which the model is depicted in Listing 4.3. In assembly the lookup in L is implemented as `ldr rD, [rN, rM]` and hence substituted by `lut rD, [rN, rM]` with `rD` being the destination register. `rN` and `rM` contain the base address of L, respectively the secret dependent offset. During partial evaluation the sum `rN +w32 rM` resolves to a label $\mathfrak{l}$ corresponding to the base address and a share ($\mathfrak{v}$). We remove the label by subtracting the base address and use the resulting expression a in evaluating $\mathsf{L}\,(a) = (a + r_a \bmod q) \oplus r_b$ in Line 8. The global variables $\mathtt{r_a}$ and $\mathtt{r_b}$ are defined in separate annotations to contain uniformly distributed random masks, specific to the masked table. The explicit leaks ensure that the side-channel behavior of the substituted `ldr` is equivalently modeled, whereas the leak-free assignment ensures that the intermediate steps in the semantic operation in Line 8 are not visible. Using this approach we verify our table-based A2B conversion to be Stateful t–SNI.

Listing 4.3: IL model of a virtual `lut` instruction used to model secret dependent table lookups of masked implementations during verification.

```
1  macro lut (w32 rD, w32 rN, w32 rM)
2    w32 val, w32 ra, w32 rb, w32 baseaddr
3  { // In our modified implementation pcptr points to the static
       ↪  table containing the base address for the random masks
       ↪  ra, rb, and the lookup table
4    baseaddr ← [w32 mem (int) pcptr];
5    ra ← [w32 mem (int) (baseaddr +w32 (w32) 0)];
6    rb ← [w32 mem (int) (baseaddr +w32 (w32) 4)];
7
8    val ← (rN +w32 rM -w32 baseaddr +w32 ra) ^w32 rb;
9    leak "lutOperand" (opA, opB, rN, rM);
10   leak "lutMemOperand" (opR, val);
11   leak "lutTransition" (rD, val);
12   opR ← val; opA ← rN; opB ← rD;
13
14   rD ← val;
15 }
```

In principle, our manual approach can be automated by program transformations on the intermediate program representation of scVerif, replacing the occurrence of memory accesses to specific memory regions with a functional representation. Users would then only need to provide annotations defining a memory range of the LUT and corresponding functionality. However, this approach does not extend to higher order masked LUTs which accesses the same table multiple times but in between modify the table entries [Cor14; CRZ18; Cor+23].

In the subsequent verification of our implementations of $\mathsf{Compress}_q(.,1)$ and $\mathsf{DecompressedComparison}$ we replace calls (branches) to $\mathsf{A2B}$ and random number generators by simplified variants implemented in the scVerif intermediate language, exposing a worst-case leakage assumption that leaks a combination of all registers. This allows us to harden our implementation concerning different implementations of random number generators and $\mathsf{A2B}$ implementations. Given these prerequisites, the security of $\mathsf{Compress}_q(.,1)$ and $\mathsf{DecompressedComparison}$ is verified. The large size of $\mathsf{DecompressedComparison}$ forces us to reduce its parameters (*i.e.*, $k = 1$ and $n = 64$) for verification, while $\mathsf{Compress}_q(.,1)$ can be verified for the Kyber768 parameters. Both components take nine minutes each to verify successfully; stating that both implementations are Stateful t–SNI in $[\![\cdot]\!]^{\text{CM0+}}$.

4.3.3. Physical Leakage Assessment

Finally, the practical side-channel resilience of our hardened first-order implementations is evaluated by performing statistical leakage detection on physical side-channel measurements. We use the same setup as in Section 3.3.3 but with the PicoScope 6404C oscilloscope sampling the power consumption at 312.5 MS/s and the micro-controller clocked at 12 MHz. This results in slightly more than 26 samples per clock cycle.

We measure the power consumption of the CM0+ processor executing 50 000 invocations of the algorithms on a fixed secret value which is freshly masked for each execution, and another set of 50 000 invocations on uniformly distributed secret values. In both cases, the implementation is provided with fresh, pre-sampled randomness stored in a table. In line with [Ode+18] and [Bei+20], we instantiate the measured module DecompressedComparison with reduced parameter sets, *i.e.*, $k = 1$ and $n = 64$ to mitigate the large size, while ensuring that the entire function can be assessed. The parameters are chosen such that loops are executed for at least two iterations. We choose the public compressed values in such a way that the invocations with the fixed value compare correctly to all but the last compressed coefficient, whereas the invocations on random (uncompressed) coefficients result in an invalid comparison to the fixed compressed coefficients with a high chance. Only by comparing uncompressed to compressed coefficients that match in the fixed invocation, we can assess the secrecy of all intermediate comparisons and the handling of the resulting flag.

The $\mathsf{Compress}_q(.,1)$ is assessed without reducing parameters (*i.e.*, $n = 256$). Ding et al. shown in [Din+17], that the TVLA threshold needs to be adapted for very long traces, as in our case, to avoid false positives during leakage detection. We adapt their approach to set the threshold for our leakage assessments to avoid erroneous results. For DecompressedComparison, the measurements consist of 1,782,438 sample points for which we set the threshold to 6.89. For $\mathsf{Compress}_q(.,1)$, we need to process 1,726,452 sample points, and therefore set the threshold to 6.88.

For a first-order secure implementation, the assessment is expected to show no significant leakage at first-order, while exceeding the threshold at second order. To validate our setup, we first run the test when the randomness source is turned off. In this case, the thresholds are exceeded for just 1000 traces, as depicted in Figures 4.5a and 4.6a. The visible sawtooth pattern in Figure 4.5a corresponds to the bit-sliced comparison of 32 coefficients in parallel.

In normal operation (*i.e.*, randomness source turned on) our hardened algorithms do not exhibit significant first-order leakage at 100 000 measurements as can be seen in Figures 4.5b and 4.6b. On the other hand, significant univari-

ate second-order leakage is detectable, as depicted in Figures 4.5c and 4.6c, indicating that second-order attacks are likely to succeed. To increase the SCA resilience beyond the first-order resilience which is provided by our first-order masked implementation, additional countermeasures are required, e.g., increasing the masking order. Our presented higher-order masked algorithms enable to implement KYBER at arbitrary orders, allowing protection against higher-order SCA attacks.

4.4. Discussion

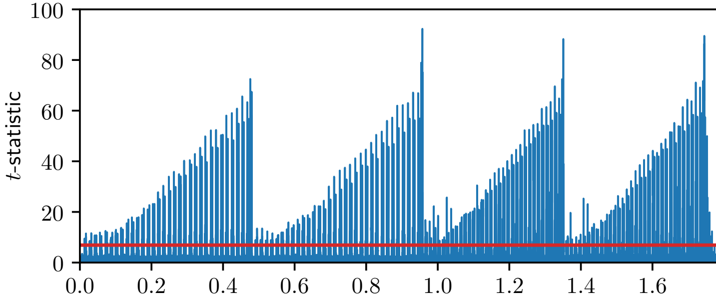
In this chapter, we realize a PQC KEM decapsulation, KYBER.CCAKEM.Dec, masked at arbitrary order and provably t–SNI. Different configurations are evaluated and detailed benchmarks & performance breakdowns are given. Two main components, $\mathsf{Compress}_q$ and DecompressedComparison are implemented, hardened, and evaluated for 1st-order physical resilience.

This is achieved using the secure development flow proposed in Chapter 1, combining formal verification and physical evaluation in iterative development (Figure 1.2). We emphasize the significant speedup in development rendered possible by the fast and exhaustive verification providing qualitative information for fixing bugs in large designs and preventing regressions. Verification scales to large implementations with more than 140 000 cycles (Table 4.2) with reports generated in less than 9 minutes; violations of stateful t–SNI are frequently reported within a minute. We emphasize that exhaustive verification is by orders of magnitude faster than physical evaluation. Also, verification covers all input choices, hence all possible ciphertext values, whereas in our physical evaluation one fixed, artificial ciphertext had to be chosen to avoid false positive leakage detection results. Finally, we demonstrate that verification in fine-grained models is beneficial without sophisticated, *i.e.*, dependable and complete, leakage models. Instead, heuristic extensions are sufficient, leading to a few additional iterations in the secure development flow.

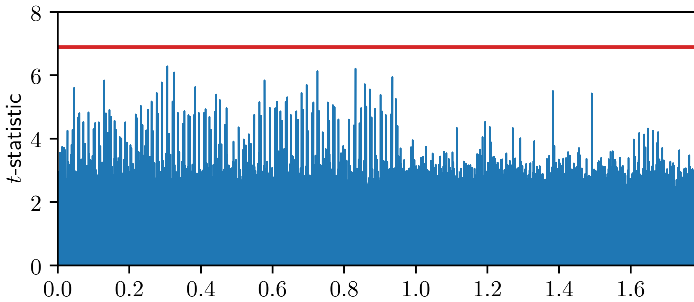
Subsequent works show the curse of the FO transform, coined as the "FO-Calypse". The large and deterministic FO transform causes an inherent susceptibility to horizontal attacks and demands masking at orders 3 to 6 depending on the noise level [Uen+22; Azo+22c; Azo+22a]. Our verification scales to high orders for small components [Bar+19; Bar+16]. However, it appears unlikely that entire components, such as $\mathsf{Compress}_q$ or DecompressedComparison, can be verified at these orders with the current state of the art. This is mainly due to the exponential blow-up of leakage tuples combined with the polynomial increase of implementation size and leakages which is handled so well only thanks to the underlying language-based approach of MASKVERIF. In-

teresting future work is to consider *decomposition* approaches to automatically split large implementations into the smaller gadgets constituting the composition, as for example in Figures 4.2, 4.3 and 4.4. This would allow verifying gadgets properties individually and subsequently the composition, akin to the proofs of Theorem 4.1, 4.2 and 4.3. We emphasize that physical assessment at higher orders is restricted for the same reasons and likely does not scale to the demanded orders. For example, the bivariate 2nd-order TVLA in Section 3.3.3 already required multiple hundred CPU hours and measuring 100,000 traces of the 3rd-order $\mathsf{Compress}_q$ would take roughly 37 hours in our setting. A development flow without verification and without fine-grained leakage models seems very unlikely to be worthwhile.

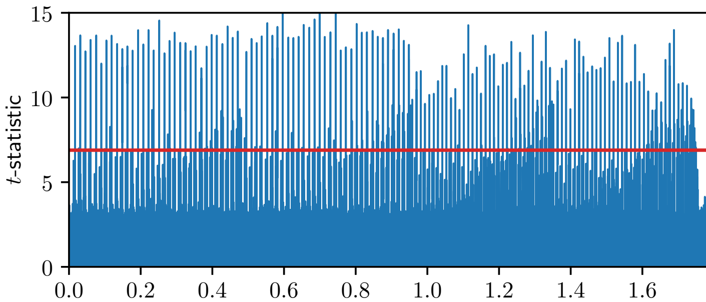
In this respect, we emphasize again the improvement stemming from our combined verification and physical development flow which we perceive to work for small gadget implementations. These gadgets can be used to compose larger components according to hand-written proofs, leaving the automated verification of the implemented composition open. We stress that our Stateful t–$(\mathsf{S})\mathsf{NI}$ notions ensure standard composability in the presence of microarchitectural leakage & residue and hence ease the implementation of a secure composition which achieves security in practice.



(a) 1 000 traces, randomness off

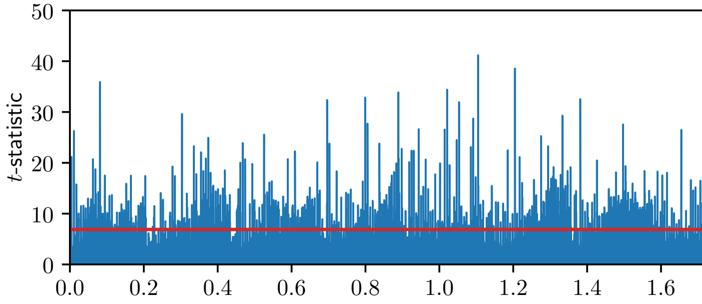


(b) 100 000 traces, randomness on

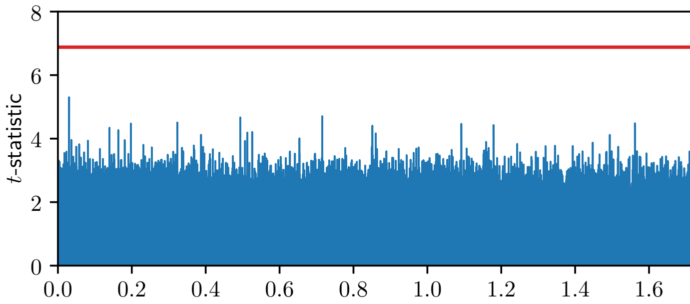


(c) 100 000 traces, 2nd order

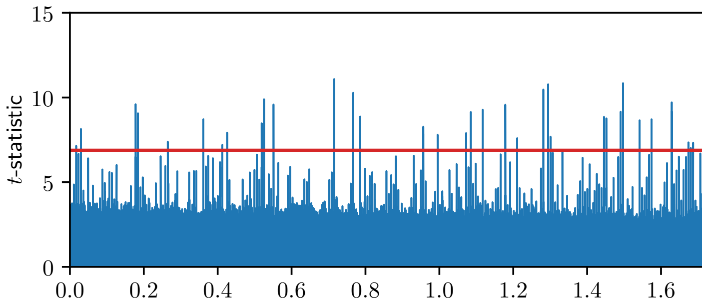
Figure 4.5.: Results of TVLA assessment of decompressed comparison for (a) 1st order without randomness, (b) 1st order with randomness, and (c) 2nd order with randomness. The x axis represents sample point index $\times 10^6$.



(a) 1 000 traces, randomness off



(b) 100 000 traces, randomness on



(c) 100 000 traces, 2nd order

Figure 4.6.: Results of TVLA assessment of 1-bit compression for (a) 1st order without randomness, (b) 1st order with randomness, and (c) 2nd order with randomness. The x axis represents sample point index $\times 10^6$.

5. Verifiable Hardware-Software Contracts

In this chapter, we present our rigorous extension of fine-grained models in the form of contracts, compliance, and the E2E security reduction.

A contract is a user-defined noiseless computation model $[\![\mathsf{P}]\!]^c$ with side-channel leakage $\mathbf{L}^c$

$$[\![\mathsf{P}]\!]^c_{\mathbf{X},\mathbf{R},p} \xmapsto{\mathbf{L}^c} (\mathbf{Y}, \mathbf{O})\,, \tag{5.1}$$

expressed in the GENOA DSL, which we introduce in Section 5.2.

We tackle the problem of the unknown or questionable completeness of fine-grained models by defining a notion of compliance between contracts and hardware computation models $[\![(\mathsf{H},\mathsf{P})]\!]^h$ involving a processor implementation H executing program P. A processor implementation is compliant with a contract, denoted $[\![\cdot]\!]^h \vdash_{\mathcal{V}} [\![\cdot]\!]^c$, if it is functionally equivalent and its leakages can be modeled from the contract with *simulation mapping* $\mathcal{V}$ introduced later.

This notion of compliance is independent of the software implementation and hence requires to show the ability to model leakage once for all possible programs P which can be executed in either model. The property ensured by compliance is close to exhaustive compliance (Definition 2.9) but relates to the gate-level leakage of a hardware computation model instead of the physical computation $[\![\mathsf{P}]\!]^{\text{PHY}}$.

In the sequel, we present methods to verify compliance w.r.t. generic gate-level leakage and instantiate them in a tool focused on the transition leakage defined by the RPM. We verify compliance of a contract for the IBEX processor netlist and use the resulting contract to build hardened 2nd-order t–$(\mathsf{S})\mathsf{NI}$ implementations, using the approach for verification in fine-grained models presented in Chapter 3.

This work significantly reduces the gap in verification of side-channel resilience for software as it for the first time permits to provably include the complete gate-level leakages. The following sections are based on the co-authored work [Blo+22] and reproduce excerpts with adaptions.

5.1. Preliminaries

In this chapter we use $[\![\mathsf{P}]\!]^h$ as a shorthand for the hardware computation model $[\![(\mathsf{H},\mathsf{P})]\!]^h$ specialized to processor implementation H, *i.e.*, with h a specific processor implementation is associated and set to execute P.

The gate-level leakage is defined according to a hardware small-steps semantic $\sigma_j^h \xrightarrow{\mathbf{L}_j^h} \sigma_{j+1}^h$, as in Definition 2.10, where any leakage point in cycle j is defined by a function of the previous and current state

$$\forall L \in \mathbf{L}_j^h : L = \lambda_j^h \left(\sigma_{j-1}^h, \sigma_j^h\right). \tag{5.2}$$

This entails the three leakage effects of the RPM in Definition 2.11. Our tooling and evaluation presented in the following focus on computation and transition leakage $\mathbf{L}_{\mathsf{trans}}^{\mathrm{RPM}}$. We emphasize that our techniques do not only extend to the full RPM but also to any extended or refined gate-level model, e.g., [DBR19; CS21; BM16], as long as the property in Eq. (5.2) is fulfilled.

We formalize a non-probabilistic way of *modeling* the outcome of a computation from a related but different variable given a condition. This corresponds to a (conditionally) stricter notion than perfect simulation (Definition 2.3).

Definition 5.1 (Modeling Function)**.** Let $\lambda_H : \mathcal{H} \to \mathcal{V}$ and $\lambda_C : \mathcal{C} \to \mathcal{U}$ be deterministic functions. We say that a deterministic function $f_S : \mathcal{U} \to \mathcal{V}$ is a *modeling function* which *models* λ_H given λ_C under deterministic relation $\Psi : \mathcal{H} \times \mathcal{C} \to \mathbb{Z}_2$ if

$$\forall h \in \mathcal{H}, c \in \mathcal{C} : \Psi\left(h, c\right) \Rightarrow f_S \circ \lambda_C\left(c\right) = \lambda_H\left(h\right). \tag{5.3}$$

Definition 5.1 is strong: whenever modeling function f_S models λ_H then it also simulates it, captured by Lemma 5.1.

Lemma 5.1 (Modeling Functions Simulate)**.** *Let $\lambda_H : \mathcal{H} \to \mathcal{V}$ and $\lambda_C : \mathcal{C} \to \mathcal{U}$ be deterministic functions and let function $f_S : \mathcal{U} \to \mathcal{V}$ model λ_H given λ_C whenever relation $\Psi : \mathcal{H} \times \mathcal{C} \to \mathbb{Z}_2$ holds. Let (H, C) denote possibly dependent random variables with sample spaces $\mathcal{H}$, respectively $\mathcal{C}$. Let $V = \lambda_H\left(H\right)$ denote a derived dependent random variable. Then, modeling function f_S perfectly simulates V given $\lambda_C\left(C\right)$ for all events $(H = h, C = c)$ where relation $\Psi\left(h, c\right)$ holds.*

Stateful t–$(\mathsf{S})\mathsf{NI}$ requires a probabilistic simulator to simulate observations on leakage or outputs shares independently of secrets and a function modeling public outputs from public inputs. In the following, we develop a verifiable condition to ensure that all gate-level leakage can be modeled from contract leakage and hence any simulator for contract-level t–$(\mathsf{S})\mathsf{NI}$ can simulate the gate-level leakage.

5.2. Hardware-Software Contracts

We introduce an intuitive and industry-grade DSL called GENOA. Then, we turn towards the question of model completeness defining compliance and proving E2E security. Section 5.3 introduces a way to verify compliance.

5.2.1. Expressing Contracts in Genoa

GENOA allows specifying ISA semantic and device-specific leakages in contracts by extending the SAIL language [Arm+19] with explicit leakage to support leakage specifications in the same style as IL. The RISC-V foundation picked SAIL as the official tool to specify the reference RISC-V ISA and all standard extensions [Sew20; Mun+21]. Models for multiple architectures (e.g., ARM) exist, which can be freely adopted and compiled to software emulators [Arm+19]. Modeling leakage in a contract is as easy as adding a few `leak` statements to one of the many existing SAIL models for RISC-V, Arm, *etc.* (GENOA supports all SAIL models), providing an interface to our tool and applying it to check for modeling gaps.

GENOA is an extension of IL, mitigating several limitations, though the two DSLs have no common history. IL mandates to develop ISAs from scratch which is laborious and error-prone. Constructing contracts is as simple as augmenting any of the available SAIL ISA models with `leak` statements since GENOA is a strict superset of SAIL. We stress that the RISC-V SAIL models are the official reference specification and even the Arm models have been extensively validated [Arm+19], hence representing a dependable foundation. Another limitation of IL is the deeply embedded decoding of assembly, which makes it difficult to specify leakage caused by bits of the instruction opcode and speculative leakage as in branches. In GENOA, respectively SAIL, models the decoding of machine code is specified by the user. The framework of SAIL provides a transformation from SAIL code to SMT code which we adopt for verifying compliance. Such a feature does not exist for IL and is a significant engineering effort, comparable to T in Section 3.2.2. On the formal side, GENOA comes with profound and fully formalized operational semantics with interfaces to proof assistants. Parts of the IBEX contract are shown in Listings 5.1 to 5.5 and in C.1.

The SAIL manual [Arm+21] and the work of Armstrong *et al.* [Arm+19] provide in-depth explanations of the syntax. We give a brief overview. In Listing 5.1 we define the architectural state of a processor, consisting of 32-bit registers which are declared as global variables. Additional leakage state is introduced to model leakage which arises from the microarchitectural state in hardware. For example, `rf_pA` is used to remember the value last read from

Listing 5.1: Contract model of state defined in Genoa.

```
// adopted from RISCV Sail Model, see license in Listing C.1
register PC : bits(32)
register nextPC : bits(32)
register x1 : bits(32) ...
// shadow registers
register rf_pA : bits(32) // register file read port A
register rf_pB : bits(32) // register file read port B
register mem_last_addr : bits(32) // address of last access
register mem_last_read : bits(32) // data from last instr.
```

Listing 5.2: Model of instruction-step χ defined in Genoa.

```
// adopted from RISCV Sail Model, see license in Listing C.1
function step_ibex (op : bits(32)) -> bool = {
  nextPC = PC + 4;
  let instruction = encdec(op);
  let ret = execute(instruction);
  tick_pc();
  match ret {RETIRE_SUCCESS => return true,
             RETIRE_FAIL => return false}}
```

the register file but is not used in the specification of instruction semantic, comparable to `opA` in $[\![\mathsf{P}]\!]^{\mathrm{CM0+}}$. Its value is maintained in the model and later on leaked in `leak` statements to model leakage of instructions accessing the register file since such leakage involves the value of the register read last, as observed in [PV17].

Every contract must specify a step function defining how a single instruction is executed. For Ibex, `step_ibex` shown in Listing 5.2 decodes the machine code instruction (`encdec`) provided as parameter `op` and returns whether the instruction executes (`execute`) successfully. Both `encdec` and `execute` are `scattered` into multiple clauses which describe the decoding, respectively execution, for a few instructions loosely belonging to a category. Each category is represented by a datatype `ast`, e.g., `RTYPE` for instructions operating on three ISA registers, represented by three indices for destination and two operands, as well as another datatype `rop` for different operations.

Listing 5.3 shows the model of `RTYPE` instructions; `encdec` maps between instruction bits and `ast` representations using conditional pattern matching. In line 6 `rs1` represents the index bits of the first source register. The instruction semantic and leakage is specified in `execute`, `X(rs1)` returns the value of the

Listing 5.3: Contract model of R-type instructions in GENOA augmented with leakage for IBEX.

```
// adopted from RISCV Sail Model, see license in Listing C.1
type regidx = bits(5) // index of register 0b00001 = x1
enum rop = {RISCV_ADD, RISCV_SUB, RISCV_SLL, RISCV_SLT, RISCV_SLTU,
    ↪ RISCV_XOR, RISCV_SRL, RISCV_SRA, RISCV_OR, RISCV_AND}
union clause ast = RTYPE : (regidx, regidx, regidx, rop),
mapping clause encdec = RTYPE(rs2, rs1, rd, RISCV_ADD)
  <-> 0b0000000 @ rs2 @ rs1 @ 0b000 @ rd @ 0b0110011
  if (rs1[4] == bitzero) & (rs2[4] == bitzero) & (rd[4] == bitzero)
mapping clause encdec = RTYPE(rs2, rs1, rd, RISCV_SLT)
  <-> 0b0000000 @ rs2 @ rs1 @ 0b010 @ rd @ 0b0110011
  if (rs1[4] == bitzero) & (rs2[4] == bitzero) & (rd[4] == bitzero)
...
function clause execute (RTYPE(rs2, rs1, rd, op)) = {
  let rs1_val = X(rs1); let rs2_val = X(rs2);
  common_leakage(rs1_val, rs2_val);
  let result : bits(32) = match op {
    RISCV_ADD => rs1_val + rs2_val,
    RISCV_SLT => EXTZ(bool_to_bits(rs1_val <_s rs2_val)), ... };
  overwrite_leakage(rd, result);
  X(rd) = result;
  return RETIRE_SUCCESS}
```

register addressed by `rs1`. Leakage which is common across multiple instruction categories is exposed with a call to function `common_leakage` (we defer the descriptions to Section 5.4.2, Listing 5.4). The semantic of the different operations (add, signed less than, *etc.*) are defined in the `match` statement. The function `overwrite_leakage` specifies transition leakage emitted while writing the result to the destination register, similar to Definition 3.1.

In summary, GENOA allows designers to quickly construct and adjust contracts, The human-readable specification supports the systematic development of side-channel protected software, and the verification approach outlined in Chapter 3 is applicable.

5.2.2. Contract Formalization

The small-steps semantics of GENOA are defined as a reduction

$$(\delta, \mathfrak{c}, \mathbf{L}) \mapsto \left(\delta', \mathfrak{c}', \mathbf{L}'\right). \tag{5.4}$$

δ is the context containing the definition of functions and the values of local and global variables, $\mathfrak{c}$ is a sequence of statements and tuple $\mathbf{L}$ is the leakage exposed during execution. After the execution of one GENOA statement (not to be confused with an instruction) δ' is the resulting context, $\mathfrak{c}'$ are the statements that remain to be executed next and $\mathbf{L}||\mathbf{L}^{'}$ is the resulting leakage. Leakage cannot be erased. All statements except `leak` do not add leakage and their transformation rules stay as in SAIL, see [Arm+18] for the formal semantics. A `leak` statement appends the valuation of its arguments (expressions) $\mathfrak{e}_1, \ldots, \mathfrak{e}_n$ to the execution leakage. A sequence of statements is denoted $\cdot;\cdot$.

$$(\delta, \texttt{leak}\,(\mathfrak{e}_1, \ldots, \mathfrak{e}_n)\,;\mathfrak{c}, \mathbf{L}) \mapsto (\delta, \mathfrak{c}, \mathbf{L}||\,(\mathfrak{e}_1(\sigma^c), \ldots, \mathfrak{e}_n(\sigma^c)))\,. \tag{5.5}$$

As before, a `leak` statement may expose multiple values, which allows abstracting away from particular assumptions such as HD leakage, as processors are allowed to leak any combination of the values exposed by a `leak`. While GENOA does not feature a language construct to sample random values, sampling can be mimicked by reading from a dedicated state region containing randomness.

The behavior of a program is defined by user-supplied execution semantic which are specified in the contract. The contract specification written in GENOA thus defines the context δ for small-step execution and, as for hardware, the contract state $\sigma^c \in \mathbb{F}_2{}^{|\mathcal{V}^c|}$ denotes the values of variables $v^c \in \mathcal{V}^c$, further on referred to as locations. Based on these definitions, we can now define the semantic for the execution of an entire instruction, denoted by the step function χ, starting in state σ_i^c and returning the state σ_{i+1}^c and a tuple of side-channel leakages $\mathbf{L}_i^c$ of executing the ith instruction:

$$\chi\left(\sigma_i^c\right) = \left(\sigma_{i+1}^c, \mathbf{L}_i^c\right). \tag{5.6}$$

The instruction to be executed is determined by the state σ_i^c itself, e.g., by the value of the program counter. The execution of an instruction corresponds to the many-steps evaluation of the instruction-steps function χ using the small-steps semantics described before. χ is part of the contract (for IBEX `step_ibex`) and supplied by the user; to simplify our tool state σ_i^c is implicitly passed while the instruction to be executed is passed explicitly. A step can either fail or succeed, as indicated by a Boolean flag. The criteria for failing the execution is governed by user-defined assumptions expressed in the contract. For IBEX these prohibit illegal instructions, accesses of non-existent registers, or unaligned memory accesses. In the following, we depict the execution of an entire instruction in the contract with

$$\sigma_i^c \xrightarrow{\mathbf{L}_i^c} \sigma_{i+1}^c. \tag{5.7}$$

Finally, we define the noiseless computation model for a contract. Contract $[\![\cdot]\!]^c$ models the execution of program P starting in initial state σ_0^c and resulting in state σ_n^c while producing the trace $\mathbf{L}^c$, *i.e.*,

$$[\![\mathsf{P}]\!]^c_{\sigma_0^c} \xmapsto{\mathbf{L}^c=\mathbf{L}_0^c||\dots||\mathbf{L}_{n-1}^c} \sigma_n^c \;:=\; \sigma_0^c \xrightarrow{\mathbf{L}_0^c} \sigma_1^c \xrightarrow{\mathbf{L}_1^c} \dots \xrightarrow{\mathbf{L}_{n-1}^c} \sigma_n^c. \tag{5.8}$$

5.2.3. Hardware Compliance With a Contract

We now turn towards the question of model completeness and define *compliance with a contract*, a formal property expressing that the results and leakages from execution on a processor are modeled by a contract according to Definition 5.1.

A program P executed in initial hardware state σ_0^h leads to leakages $\mathbf{L}^h$ and final state σ_m^h when executed on a processor $[\![\cdot]\!]^h$

$$[\![\mathsf{P}]\!]^h_{\sigma_0^h} \xmapsto{\mathbf{L}^h=\mathbf{L}_0^h||\dots||\mathbf{L}_{m-1}^h} \sigma_m^h := \sigma_0^h \xrightarrow{\mathbf{L}_0^h} \sigma_1^h \xrightarrow{\mathbf{L}_1^h} \dots \xrightarrow{\mathbf{L}_{m-1}^h} \sigma_m^h \tag{5.9}$$

As stated at the start of this chapter, $[\![\mathsf{P}]\!]^h$ is a shorthand for the execution of the processor netlist H in a hardware execution model $[\![(\mathsf{H},\mathsf{P})]\!]^h$ where the initial state σ_0^h is set to start execution of P. In our case-study $\mathsf{H} =$ IBEX and $h =$ RPM restricted to $\mathbf{L}_i^{\text{RPM}} = \mathbf{L}_{\mathsf{trans},i}^{\text{RPM}}$ in each cycle $i \in [m]$ in Definition 2.11. In contrast to contracts the execution proceeds in clock cycles instead of instruction steps, *i.e.*, one step in hardware corresponds to one clock cycle as defined in Section 2.5.2.

Compliance expresses the property that all leakage and all outputs of hardware execution $[\![\cdot]\!]^h$ can be modeled (according to Definition 5.1) from execution in a contract $[\![\cdot]\!]^c$ as long as the starting states are similar, *i.e.*, execute the same program under equivalent inputs, depicted in Figure 5.1.

In Definition 5.2 we introduce a Boolean relation between hardware state σ^h and contract state σ^c expressing that the values contained at a specific location v^h in the hardware can be modeled from a location v^c in the contract, e.g., register `x1` models its hardware counterpart. Which contract locations model some hardware location is defined in the simulation mapping $\mathcal{V}$ provided by users alongside every contract and checked by our tool. The mapping specifies for all registers in the hardware (including finite state machines, decode stages, *etc.*) a location in the contract modeling the hardware location. To ease notation assume there are contract locations $v_0^c, v_1^c \in \mathcal{V}^c$ which are constant zero, respectively one, and later on used to express constraints on hardware execution.

$$\begin{array}{lccccccccc}
[\![\mathsf{P}]\!]^c_{\sigma^c_0}: & \sigma^c_0 & \xrightarrow{\mathbf{L}^c_0} & \sigma^c_1 & \xrightarrow{\mathbf{L}^c_1} & \dots & \xrightarrow{\mathbf{L}^c_{n-1}} & \sigma^c_n \\
& \| \mathcal{V} & & & & & & \| \mathcal{V} \\
[\![\mathsf{P}]\!]^h_{\sigma^h_0}: & \sigma^h_0 & \xrightarrow{\mathbf{L}^h_0} & \sigma^h_1 & \xrightarrow{\mathbf{L}^h_1} & \dots & \xrightarrow{\mathbf{L}^h_{m-1}} & \sigma^h_m
\end{array}$$

Figure 5.1.: Compliance for full program execution in hardware and contract. The commutative diagram shows the two executions, where the upper execution happens in the contract and the lower execution happens in the hardware. The starting states shown on the left side are assumed to be similar under mapping $\mathcal{V}$. From both starting states the execution proceeds in instruction steps (contract), respectively clock cycle steps (hardware). Verifying compliance requires showing that the final states are always similar under mapping $\mathcal{V}$ (marked red) and that leaks in hardware (marked red) can be modeled from contract leaks.

Definition 5.2 (Similar States $\sigma^h \simeq_{\mathcal{V}} \sigma^c$)**.** Two states σ^h and σ^c, with respective locations $\mathcal{V}^h$ and $\mathcal{V}^c$ are *similar* under simulation mapping $\mathcal{V} \subseteq \mathcal{V}^h \times \mathcal{V}^c$, written $\sigma^h \simeq_{\mathcal{V}} \sigma^c$, iff

$$\bigwedge_{(v^h, v^c) \in \mathcal{V}} v^h\left(\sigma^h\right) = v^c\left(\sigma^c\right) \tag{5.10}$$

Simulation mapping $\mathcal{V}$ is said to be *complete* if for all hardware locations $\mathcal{V}^h$ a mapping is defined.

We now give the definition of compliance, ensuring that semantic and leakage of execution of hardware is correctly modeled:

Definition 5.3 (Compliance $[\![\cdot]\!]^h \vdash_{\mathcal{V}} [\![\cdot]\!]^c$)**.** A hardware implementation is compliant with a contract under simulation mapping $\mathcal{V}$ if for every program P and starting hardware and contract states σ^h_0 and σ^c_0, the program executions

$$[\![\mathsf{P}]\!]^c_{\sigma^c_0} \xmapsto{\mathbf{L}^c} \sigma^c_n \quad \text{and} \quad [\![\mathsf{P}]\!]^h_{\sigma^h_0} \xmapsto{\mathbf{L}^h} \sigma^h_m$$

fulfill the following conditions:

1. **States remain similar:** Whenever σ^h_0 and σ^c_0 are similar under $\mathcal{V}$, so are resulting states σ^h_m and σ^c_n:

$$\forall \sigma^h_0, \sigma^c_0 : \sigma^h_0 \simeq_{\mathcal{V}} \sigma^c_0 \Rightarrow \sigma^h_m \simeq_{\mathcal{V}} \sigma^c_n. \tag{5.11}$$

2. **Leaks are modeled:** For every leak $\lambda_g\left(\sigma^h_{j-1},\sigma^h_j\right) \in \mathbf{L}^h$ observable in hardware during cycle $j \in [m]$, there exists a leak $\lambda\left(\sigma^c_i\right) \in \mathbf{L}^c$ in the contract during an instruction step $i \in [n]$ and a function $f_\lambda : \mathsf{dom}(\lambda) \to \mathsf{dom}(\lambda_g)$ that models λ_g from λ under relation $\sigma^h_0 \simeq_{\mathcal{V}} \sigma^c_0$ according to Definition 5.1

$$\begin{aligned}\forall j \in [m], \lambda_g\left(\sigma^h_{j-1},\sigma^h_j\right) \in \mathbf{L}^h\ \exists\, i \in [n],\ \lambda\left(\sigma^c_i\right) \in \mathbf{L}^c, f_\lambda \\ \forall \sigma^h_0, \sigma^c_0 : \sigma^h_0 \simeq_{\mathcal{V}} \sigma^c_0 \Rightarrow f_\lambda \circ \lambda\left(\sigma^c_i\right) = \lambda_g\left(\sigma^h_{j-1},\sigma^h_j\right).\end{aligned} \tag{5.12}$$

The notion of similar states allows expressing a key ingredient for the relational definition of compliance: if execution in a contract and hardware start in a similar state, then execution must end in similar states such that the result in hardware execution can be modeled according to the simulation mapping (Clause 1 of Definition 5.3). Further, the second part of compliance expresses that each gate-level leak observable during execution in hardware must be modeled by a single, fixed leak observable during execution in the contract (Clause 2 of Definition 5.3). Combined, this guarantees that software that is Stateful t–(S)NI secure when executed in the contract, is necessarily Stateful t–(S)NI when executed on compliant hardware.

5.2.4. End-to-end security

It remains to prove our E2E security claim: any implementation P of gadget G that is Stateful t–(S)NI w.r.t. the leakages of a contract is Stateful t–(S)NI w.r.t. all gate-level leakages when executed on any compliant hardware and as such its security order cannot be decreased by leakage of the processor.

However, E2E security is claimed for the same software P implementing some gadget G running on a processor and in the contract, *i.e.*, both executions use the same structured inputs and outputs. Since the states in hardware and contract may have different structures we introduce a definition to ensure that the placement of inputs and outputs in hardware $\pi^{\mathrm{I}h}, \pi^{\mathrm{O}h}$ is similar to the ones $\pi^{\mathrm{I}c}, \pi^{\mathrm{O}c}$ for which t–(S)NI was shown in the contract. A hardware policy can be derived from a contract policy by substituting the locations that define where a value resides in the state according to the simulation mapping.

Definition 5.4 (Similar Policies $\pi^h \triangleq_{\mathcal{V}} \pi^c$)**.** Let the policy $\pi^c : (d_1, \ldots, d_n) \leftrightarrow \sigma^c$ of the contract link tuples $d_1, \ldots, d_n$ to contract state σ^c. Hardware policy $\pi^h : (d_1, \ldots, d_n) \leftrightarrow \sigma^h$ is *similar* to π^c, denoted $\pi^h \triangleq_{\mathcal{V}} \pi^c$ if any pair of contract and hardware states constructed from the same sets of values are similar under mapping $\mathcal{V}$

$$\forall \sigma^h = \pi^h\left(d_1, \ldots, d_n\right), \sigma^c = \pi^c\left(d_1, \ldots, d_n\right)\ :\ \sigma^h \simeq_{\mathcal{V}} \sigma^c.$$

Instead of proving the security reduction for t–(S)NI directly, we prove a general *model reduction*: any observations made by an adversary interacting with hardware may be *modeled* with a contract the hardware complies with instead. We emphasize the difference: t–(S)NI requires the existence of a *simulator* whereas compliance guarantees the existence of a (stronger) *modeling function* easing the subsequent security reduction.

Theorem 5.1 (Model Reduction)**.** *Let* P *be a program, with the corresponding executions in the contract*

$$[\![\mathsf{P}]\!]^{c}_{\boldsymbol{\sigma}^c_0} \overset{\mathbf{L}^c}{\longmapsto} \boldsymbol{\sigma}^c_n \quad \textit{respectively} \quad [\![\mathsf{P}]\!]^{h}_{\boldsymbol{\sigma}^h_0} \overset{\mathbf{L}^h}{\longmapsto} \boldsymbol{\sigma}^h_m$$

in hardware, under policies π^{Ih} and π^{Oh}, respectively π^{Ic} and π^{Oc} such that the policies are similar $\pi^{Ih} \triangleq_{\mathcal{V}} \pi^{Ic}$, respectively $\pi^{Oh} \triangleq_{\mathcal{V}} \pi^{Oc}$, under a complete mapping $\mathcal{V}$. Let the input states be similar $\boldsymbol{\sigma}^c_0 = \pi^{Ic}(\mathbf{X}, \mathbf{R}, p)$ and $\boldsymbol{\sigma}^h_0 = \pi^{Ih}(\mathbf{X}, \mathbf{R}, p)$. Let the outputs in the respective execution correspond to $\boldsymbol{\sigma}^c_n = \pi^{Oc}(\mathbf{Y}^c, \mathbf{O}^c)$ and $\boldsymbol{\sigma}^h_m = \pi^{Oh}\left(\mathbf{Y}^h, \mathbf{O}^h\right)$. Further, let the hardware be compliant with the contract $[\![\cdot]\!]^h \vdash_{\mathcal{V}} [\![\cdot]\!]^c$. For every tuple of observations in hardware on $\mathbf{Y}^h$ or $\mathbf{O}^h$ there is an equally sized tuple of observations in the contract on $\mathbf{Y}^c$ or $\mathbf{O}^c$ which allows modeling the observations under the identity function:

$$\forall \mathbf{E}^h_{\mathbf{Y}} \subseteq \mathbf{Y}^h \; \exists \; \mathbf{E}^c_{\mathbf{Y}} \subseteq \mathbf{Y}^c \; : \; \mathbf{E}^h_{\mathbf{Y}} = \mathbf{E}^c_{\mathbf{Y}}, \tag{5.13}$$

$$\forall \mathbf{E}^h_{\mathbf{O}} \subseteq \mathbf{O}^h \; \exists \; \mathbf{E}^c_{\mathbf{O}} \subseteq \mathbf{O}^c \; : \; \mathbf{E}^h_{\mathbf{O}} = \mathbf{E}^c_{\mathbf{O}}. \tag{5.14}$$

In addition, for every tuple of observations in hardware on $\mathbf{L}^h$, a modeling function $T^{\mathbf{L}}$ and a (potentially smaller) tuple of observations in the contract on $\mathbf{L}^c$ allows to model the observations in hardware:

$$\forall \mathbf{E}^h_{\mathbf{L}} \subseteq \mathbf{L}^h \, \exists \, \mathbf{E}^c_{\mathbf{L}} \subseteq \mathbf{L}^c : \left|\mathbf{E}^c_{\mathbf{L}}\right| \leq \left|\mathbf{E}^h_{\mathbf{L}}\right| \wedge \mathbf{E}^h_{\mathbf{L}} = T^{\mathbf{L}}\left(\mathbf{E}^c_{\mathbf{L}}\right). \tag{5.15}$$

Proof. The two program executions operate on equally distributed inputs and the policies for hardware are similar, thus for every initial state σ^h_0 there must be a starting state σ^c_0 under mapping $\mathcal{V}$, *i.e.*, $\sigma^h_0 \simeq_{\mathcal{V}} \sigma^c_0$. Since hardware is compliant with the contract, the resulting states are similar as well, *i.e.*, $\sigma^h_m \simeq_{\mathcal{V}} \sigma^c_n$, and since every observation in $\mathbf{E}^h_{\mathbf{O}}$, respectively $\mathbf{E}^h_{\mathbf{Y}}$, is an observation on the value of a location in σ^h_m it follows directly that there exists a single location in the contract $\mathbf{E}^c_{\mathbf{O}}$, respectively $\mathbf{E}^h_{\mathbf{Y}}$, according to the mapping $\mathcal{V}$ which models the observation, fulfilling (5.13) and (5.14). From Definition 5.1 and Clause 2 of Definition 5.3 it follows that every observation $\lambda_g\left(\boldsymbol{\sigma}^h_{j-1}, \boldsymbol{\sigma}^h_j\right) \in \mathbf{E}^h_{\mathbf{L}}$ can be modeled from some contract leak $\lambda(\boldsymbol{\sigma}^c_i) \in \mathbf{L}^c$ using f_λ as modeling function. Grouping the necessary $\lambda(\boldsymbol{\sigma}^c_i)$ as the random variables $\mathbf{E}^c_{\mathbf{L}}$, results in $\left|\mathbf{E}^c_{\mathbf{L}}\right| \leq \left|\mathbf{E}^h_{\mathbf{L}}\right|$, defining $T^{\mathbf{L}}$ as the collection of respective f_λ implies (5.15), completing the proof. □

The simulatability of mixed observations in Corollary 5.1 follows from Theorem 5.1. Furthermore, the reduction from Stateful t–$(\mathsf{S})\mathsf{NI}$ in hardware to Stateful t–$(\mathsf{S})\mathsf{NI}$ in the contract stated in Corollary 5.2 is a direct consequence of Corollary 5.1 and Lemma 5.1.

Corollary 5.1 (Mixed Observations)**.** *Let the setting be as in Theorem 5.1. Every tuple consisting of a mixture of observations on leakage and shared outputs $\mathbf{E}^h_{\mathbf{L},\mathbf{Y}} \subseteq \mathbf{L}^h \cup \mathbf{Y}^h$, can be modeled from an equally sized tuple $\mathbf{E}^c_{\mathbf{L},\mathbf{Y}} \subseteq \mathbf{L}^c \cup \mathbf{Y}^c$ by some modeling function $T^{\mathbf{L},\mathbf{Y}} : \mathsf{dom}(\mathbf{E}^c_{\mathbf{L},\mathbf{Y}}) \rightarrow \mathsf{dom}(\mathbf{E}^h_{\mathbf{L},\mathbf{Y}})$.*

Corollary 5.2 (End-to-end Security)**.** *Let the setting be as in Theorem 5.1. If program P is t–$(\mathsf{S})\mathsf{NI}$ in the contract then it is also t–$(\mathsf{S})\mathsf{NI}$ in hardware since there exist simulators $T^{\mathbf{L},\mathbf{Y}} \circ S$ and F according to Definition 3.3 which simulate the leakage and outputs of the execution in hardware.*

This proof is valid for higher-order masking, *i.e.*, $t \geq 1$, as *each* of the t hardware observations in $\mathbf{E}^h$ can be simulated from *one* observation in $\mathbf{E}^c$ in the contract conditioned that the executions start in similar states. The presented model reduction can be of help in proving the preservation of other security notions like PINI [CS20], threshold implementations [NRR06] or probing security [ISW03].

5.3. Verifying Hardware Compliance

We present an approach to verify compliance for real processor implementations. One of our main contributions is that we do not search for modeling functions but instead check a necessary and sufficient condition, proven in Theorem 5.2. This permits to represent compliance, despite its relational nature, as *constraints* in 1st order Boolean logic and rely on off-the-shelf SMT solvers for checking compliance.

5.3.1. Concept

Compliance is defined over the execution of entire programs, indirectly mandating to consider an arbitrary number of cycles m, respectively instruction steps n in the verification of compliance. Instead of verifying compliance for big-steps execution, we focus on the execution of one instruction.

We consider a lockstep execution of the same instruction, visualized in Figure 5.2. In the contract, execution completes within one instruction step, and in hardware in exactly k cycles. Both executions start in similar states $\sigma^h_j \simeq_{\mathcal{V}} \sigma^c_i$, emitting gate-level leakage $\mathbf{L}^h_j || \dots || \mathbf{L}^h_{j+k}$, respectively contract leakage $\mathbf{L}^c_i$, and resulting in output states σ^h_{j+k}, σ^c_{i+1}.

We make an inductive argument that if the following two conditions are correctly verified then compliance according to Definition 5.3 holds.

1. *Output states remain similar:* Lockstep execution must result in both output states σ^h_{j+k}, σ^c_{i+1} being similar under $\mathcal{V}$ (marked red).
2. *Gate-level leakage can be modeled:* For every leak $L^h \in \mathbf{L}^h_j||\ldots||\mathbf{L}^h_{j+k}$ emitted by the processor's gates (marked red) there must exist a modeling function f (Definition 5.1) modeling the gate-level leak from a fixed contract leak $\lambda(\sigma^c_i) \in \mathbf{L}^c_i$ emitted during the lockstep execution.

A simple inductive argument allows to conclude big-steps compliance (Definition 5.3) if both conditions are satisfied for every possible pair of related states $\sigma^h_j \simeq_{\mathcal{V}} \sigma^c_i$. For this, observe that the required similarity of $\sigma^h_j \simeq_{\mathcal{V}} \sigma^c_i$ for the starting states $i, j = 0$ follows from the big-steps compliance and inductively extends to all subsequent states.

Users have to specify conditions (Boolean formulas over states) that determine when an instruction becomes in-flight (execution starts) and when it retires (execution completes successfully), as well as a mapping $\mathcal{V}$ and a bound k_{max}. Remark that the mapping ensures that the same instruction is executed in contract and hardware. Since processors have varying-time instruction execution we repeat the (bounded) verification of the conditions for all $k \leq k_{max}$, which restricts the lockstep execution to those instructions which terminate (retire) in exactly k cycles. We emphasize that this includes externally caused stalling but note that some platforms have a long duration, e.g., for division, which might be out of the scope of our simple proof of concept tool.

A limitation of our current implementation is that we do not exhaustively but only heuristically verify that every pair of possible resulting states $\sigma^c_{i+1}, \sigma^h_{j+k}$ falls into the covered pairs of starting states σ^c_i, σ^h_j permitted by the user-supplied mapping $\mathcal{V}$ and conditions. Erroneous configurations hence may invalidate the inductive argument partially and lead to compliance holding only for programs that fall into the considered fragment of instruction execution. However, we heuristically prevent such specification errors in our proof of concept implementation and remark that this problem can be solved exhaustively in future work.

The problem is now sufficiently reduced for automated verification of the first condition. Output states σ^h_{j+k} and σ^c_{i+1} have to remain similar, which corresponds to verifying the functional equivalence of the instruction semantics expressed in the contract and the implementation in hardware. Checking if the relation $\sigma^h_{j+k} \simeq_{\mathcal{V}} \sigma^c_{i+1}$ holds can be done by checking the satisfiability of a violation to Definition 5.2. Hence, whenever (5.16) is unsatisfiable for all $k \leq k_{max}$ in the setting described above then Clause 1 of Definition 5.3 holds

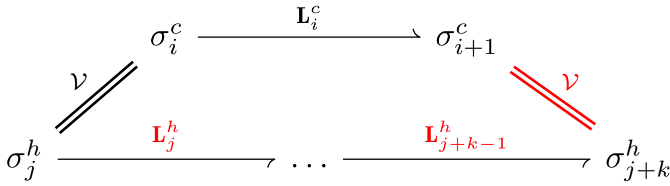


Figure 5.2.: Compliance for single instruction execution. A figure showing two executions, where the first execution happens in the contract and the second execution happens in the hardware. The starting states are connected to indicate similarity. The contract execution shows one step, terminating in its final state. The contract execution shows multiple steps until the final state is reached, indicating multiple clock cycles. The final states are connected with red lines, indicating that similarity should persist.

since the outputs states after the execution of one instruction remain similar. Further detail is given in [Blo+22].

$$\exists \left(v^h, v^c\right) \in \mathcal{V} : v^h \left(\sigma_{j+k}^h\right) \neq v^c \left(\sigma_{i+1}^c\right). \tag{5.16}$$

5.3.2. Verifying Model Completeness

Verifying the second condition is more involved because it requires showing the existence of a modeling function f. Finding function definitions is perceived as a hard problem in general [FKB13].

We avoid this by verifying their existence without finding their definition. The intuition is that for any deterministic function every input value has a single fixed output value, *i.e.*, for every x, x' in the domain of deterministic function f it holds that $x = x' \Rightarrow f(x) = f(x')$. In our setting x is the value of the contract leak which is a function of the state, denoted λ_C. Hence, a modeling function exists if (and only if) the desired output (*i.e.*, the gate-level leakage) does not change unless the input to the function changes (*i.e.*, the contract leak) in all allowed settings (expressed by Ψ). Theorem 5.2 formalizes our approach. The proof can be found in Appendix C.1.

Theorem 5.2 (Existence of Modeling Functions)**.** *There exists a modeling function $f : \mathcal{U} \to \mathcal{V}$ according to Definition 5.1 iff*

$$\begin{gathered}\forall h, h' \in \mathcal{H}, c, c' \in \mathcal{C} : \\ \Psi(h, c) \wedge \Psi(h', c') \wedge \lambda_C(c) = \lambda_C(c') \Rightarrow \lambda_H(h) = \lambda_H(h').\end{gathered} \tag{5.17}$$

Figure 5.3.: The single-instruction lockstep execution in Figure 5.2 is instantiated twice to check if gate-level leakage l^h can be modeled from some contract leak l^c. Such a modeling function exists if and only if the situation that the contract leakages coincide but not the hardware leakages is unsatisfiable ($l^c = l^{\overline{c}} \wedge l^h \neq l^{\overline{h}}$) based on Theorem 5.2.

The application of Theorem 5.2 to our setting requires modeling two parallel lockstep executions. This setting is visualized in Figure 5.3 for $k = 1$ and a single leak. There, $\sigma_j^{\overline{h}}, \sigma_i^{\overline{c}}$ are the starting states of the parallel lockstep execution producing leakages $l^{\overline{h}}, l^{\overline{c}}$. The two parallel executions are related (constrained) by ϕ, corresponding to conditions that ensure normal operation to avoid false negatives. Relation Ψ corresponds to similarity $\simeq_{\mathcal{V}}$ under mapping $\mathcal{V}$.

If there are *two* pairs of lockstep executions that coincide on the contract leakages but disagree on the gate-level leakage then such a function cannot exist. This means that the `leak` (e_1, e_2) statement in the contract which defines l^c does not model the considered gate-level leakage l^h. On the contrary, if the value of e_1, e_2 satisfies the (simplified) condition for all possible pairs of contract states in executions each with similar hardware states, then the contract leak indeed permits to model the gate-level leak.

Our tool repeatedly instantiates one satisfiability constraint for every candidate pair of contract and hardware leak in the k cycle parallel lockstep simulation and repeats this for every $k \leq k_{max}$.

The details of the checking procedures and the algorithms for the involved searches can be found in [Blo+22].

5.3.3. Implementation

We give a brief insight into our implementation as a tool and refer to the main publication [Blo+22].

Our method relies on the synthesized processor netlist to build the constraint formulas. We follow a common procedure: the hardware state σ_j^h is represented symbolically using propositional variables. Each gate g in the processor is a symbolic expression of the variables representing hardware locations $\mathcal{V}^h$. The expressions are generated by topologically iterating through the circuit and building the representation of each gate g from its inputs and type τ. The registers of the first state σ_j^h, respectively σ_{j-1}^h, are variables. In successor states σ_{j+l}^h, the registers are determined by their writebacks from the previous cycle. In this sense, the processor circuit is symbolically unfolded k times.

The translation of a contract to a SMT formula is based on the mentioned SAIL transformation generating SMT formulas for custom predicates. However, the back-end cannot handle `leak` statements. We perform two code-rewriting passes from GENOA to GENOA. The first adds a global state for each value in a `leak` statement and replaces the `leak` by an assignment to the respective global state. This reduces the GENOA DSL to the SAIL subset supported by the SMT back-end. The second pass duplicates the variables representing contract state σ^c and leakages for the parallel lockstep execution, Further, it duplicates the instruction-step function χ by rewriting it to operate on either σ_i^c or $\sigma_i^{c'}$ and resulting in σ_{i+1}^c, respectively $\sigma_{i+1}^{c'}$. Finally, GENOA ensures that the initial and final states are preserved during the transformation to SMT and asserts additional predicates, e.g., for similarity.

Our tool receives the resulting SMT code as input and produces the formulas shown before by referencing the received SMT definitions. Additional configuration files control the generation of the formulas. In particular, all hardware locations need to be declared, and either mapped to contract registers or be constrained by developer assumptions, e.g., input or output gate restrictions, or instruction execution constraints. The IBEX configuration is provided in Listing C.2.

5.4. Case Study: A Compliant Contract for Ibex

We apply the verification method presented to the IBEX processor and detail the process and results. IBEX is an open source RISC-V processor that supports the **I**nteger, **E**mbedded, **M**ultiplication, **C**ompressed and **B**it manipulation ISA extensions [low]. We focus on the **E** extension, although adding

support for the others is possible. The IBEX pipeline consists of two stages, Instruction Fetch (IF) and Instruction Decode/Execute (ID/EX). Computations take place in the ID/EX stage, which consists of a decoder, a controller, and the register file, which forwards the data into the ALU and the LSU. The outputs of the ALU and LSU are routed to the write-back logic (WBL) that decides which data is written into the register file. In the same pipeline stage, and hence in the same clock cycle, the result is written back into the register file.

After providing a configuration with constraints for proper execution and mapping the verification starts. In case of violations of compliance, the verification framework produces a detailed counterexample explaining the problem. The developer must then adjust the configuration, the mapping, the contract, or the processor implementation in order to fix the problem and restart verification. Development and verification form an iterative process incrementally improving the contract (or processor implementation).

5.4.1. Configuration for Verifying Compliance

We align the contract and the hardware by restricting the state of the processor throughout the execution of an instruction. We constrain the values of all registers with regard to the current instruction length and analyzed cycle. For the verification, we look at instructions when they reach the ID/EX stage. At this point, signal `instr_rdata_id` carries the instruction bits and must be set equivalent to the `op` argument of `step_ibex` in the contract.

Additionally, we need to make sure that instructions are only retired in the last cycle $k-1$ of a k-cycle instruction by constraining `instr_id_done` to be $\top$ in the last cycle and $\bot$ otherwise. Similarly, we enforce that the next instruction is fetched exactly in cycle $k-1$ by constraining `fetch_valid` and `id_in_ready`. We assert that there are no outstanding errors caused by the previous instruction by constraining registers `lsu_err_q`, `pmp_err_q`, `branch_set_raw`, and `data_err_i` to be $\bot$. To make sure that the state machines in the LSU and control unit start off in a valid state when the instruction starts executing, we add several further constraints. Finally, we also assert that there is no reset through `rst_ni` and no interrupt signals `irq_*`, `debug_req_i` are triggered to match the behavior expected by the processor.

One of the main challenges in modeling the processor environment is the memory interface. Whenever the processor requests data by setting `data_req_o` to $\top$, the next cycle provides answers by setting `data_rvalid_i` to $\top$ and providing the corresponding read data in `data_rdata_i`. Here, we additionally require memory to only provide acknowledgment through `data_rvalid_i` if there was a request, and not provide any data on the input `data_rdata_i` otherwise.

Listing 5.4: Common leakage occurring in every instruction.

```
1 // see license in Listing C.1
2 function common_leakage(rs1_val, rs2_val) = {
3   leak(rs1_val, rs2_val, rf_pA, rf_pB, mem_last_addr, mem_last_read);
4   rf_pA = rs1_val; rf_pB = rs2_val; /* update read ports */
5   mem_last_read = 0x00000000; /* clear data memory port */ }
```

Listing 5.5: Specialized leakage occurring during loads.

```
6  // see license in Listing C.1
7  function load_leakage(rs1_val : xlenbits, rs2_val : xlenbits,
8                        addr : xlenbits, req_data : xlenbits) = {
9    leak(rf_pA, rf_pB, rs1_val, rs2_val);
10   leak(rf_pA, rf_pB, mem_last_addr, mem_last_read);
11   leak(addr, req_data, mem_last_addr);
12   rf_pA = rs1_val; rf_pB = rs2_val;
13   mem_last_read = req_data; mem_last_addr = addr; }
```

This is due to a small bug in Ibex, which causes the `data_rvalid_i` signal to overrule all other signals in the processor and ultimately issue an erroneous write-back, hence violating similarity, respectively functional correctness.

5.4.2. Complete Power Contract for Ibex

Our tool verified the compliance of Ibex with the contract shown in Listings 5.1 to 5.5 and C.1. We discuss the observed behavior and compare the findings to existing models for other architectures. In contrast to empirical works, we can give insight into the cause of leakage behavior in the microarchitecture reported by our tool.

Most instructions have a common leakage pattern modeled in `common_leakage` ↪ in Listing 5.4. The Ibex processor combines the previous outputs of the register file (modeled in leakage states `rf_pA`, `rf_pB`) with the current outputs `rs1_val` and `rs2_val`, as well as the address and value of the last memory access `mem_last_addr` and `mem_last_read`. This leak statement models all transition leakage and value leakage produced in the ALU and the WBL. None of the operands in the `leak` statement can be removed without breaking compliance since distinct parts of Ibex cause these combinations. The WBL causes additional combinations: ALU or branch instructions after a memory load causes a transition between their results.

This common leakage pattern covers leak effects previously discussed and reported in related work. It models transition leakage produced in the ALU and WB stage, whose source is the two read ports of the register file. Transitions in the first and second operands of instructions are well-known [MPW22; MOW17]. Interestingly, prior empirical analysis of the Arm M0 [MOW17], a processor in the same performance and size class as IBEX, did not report interactions between the data loaded in the previous instruction and the current or past ALU operands.

Furthermore, leakage is even caused by instructions which have no register operands but only work on *immediates*, *i.e.*, constant values, like `LUI` (load unsigned immediate). The root cause of this effect is that the register file always decodes specific instruction bits as register addresses and forwards their contents to the ALU. In the case of `LUI`, these bits are part of an immediate value. This effect was observed by Gigerl *et al.* [Gig+21] but we characterize and describe the behavior accurately to allow fine-grained protection by providing an accurate model of which register content is leaked given a specific immediate. Prior analyses of the Arm M0 did not report similar effects [MOW17], and instead report that instructions with an immediate field behave as if they have only one operand which could be due to the microarchitecture of the processor or gaps in the empirical modeling procedure.

The leakage of load instructions modeled by `load_leakage` (Listing 5.5) differs from all other instructions. Here, the leak statement in `common_leakage` can be broken down into smaller leak statements. First off, because the ALU is always active, the current and previous values of the register file outputs are combined in line 9. Line 10 specifies leakage inherited from the prior instruction's WBL through transition leakage. In contrast to prior work [PV17; Gig+21], consecutive load instructions do not cause transition leakage between the loaded data because the contract disallows misaligned memory accesses. Similarly, IBEX does not expose transition leakage in subsequent memory writes, unlike several Arm architectures [MPW22; Bar+21b]. This is likely because in IBEX there are no additional registers in the memory path as in other processors. However, IBEX does produce transition leakage between memory access addresses of loads and stores separated by an arbitrary amount of other instructions, as shown in line 11. IBEX causes this leakage because it always stores the last address for error-handling purposes. We emphasize the benefit of our approach over empirical approaches as the precise reporting permits finding such cases and mitigates the behavior in the processor implementation.

Verifying that a processor design complies with a contract is computationally intensive, but can be well parallelized. We ran the full verification on an Intel Xeon E5-4669 processor with 88 logical cores running at 2.20GHz. The

most time-consuming verification task is the search for gate modeling functions, which takes about 30.6 hours. This step is done once and then cached, and any changes to leak statements in the contract do not require it to be run again. Verifying the leakage modeling requires additional 4.9 hours.

While we demonstrate our approach on the RISC-V Ibex core we emphasize that it is neither limited to RISC-V processors, nor the Ibex core. Verifying contract compliance for similar architectures and processors requires adapting the tool to their pipeline and properly configuring the verification procedure. Depending on the complexity of the processor pipeline it might be sufficient to adapt a contract and a configuration for proper configuration. The main limitation of our approach is that users need to find a good configuration, corresponding to invariants, which allows for verifying the bounded compliance representative for all instruction pairs. We remark that this effort is required once per processor netlist, and can be made either by the processor designers themselves or by any other person in case the processor netlist is not IP-restricted. Anybody with access to the contract can then verify masked software against it, without the need of having access to the processor netlist itself.

We currently limit the scope of our contracts, and thus also the scope of masking verification, to the processor core itself. There may also be some other components causing side-channel leakage. For example, static random-access memory (SRAM) or data caches are locations where leakage may arise and hence unintentional combinations of shares could occur. Within our framework, the leakage of such components would need to be verified separately and modeled within the processor contract. Additionally, there are also cases when no “good” contract can be written for a processor. One such example would be processors where the register file output is computed with a multiplexer tree and would lead to a `leak` statement containing the whole register file.

So far, we have analyzed instructions starting with the decode stage, which assumes that the fetch stage does not expose leakage depending on the fetched instruction. From what we have seen in Ibex and other open-source processors, no leakage in the fetch stage depends on the bits of the fetched instruction. For a similar reason, speculation or (secret-dependent) branch prediction is not a primary concern for our current analysis since these are usually not present in embedded devices.

Our tool currently focuses on value leakage and transition leakage, while our theoretical framework supports arbitrary gate-level leakage. Extending our verification tool to include further effects such as glitches makes an interesting future research question, and could be achieved by extending the encoding of leakage. Our verification methodology and the contracts them-

Gadget	t	# Instr.	# Clear.	Verification time	
				Contract	Netlist
`AND`	2	62	10	< 1 s ✔	284.63 s ✔
`Refresh`	2	19	0	< 1 s ✔	32.85 s ✔
`XOR`	2	16	1	< 1 s ✔	50.79 s ✔
`NOT`	2	5	0	< 1 s ✔	63.32 s ✔
`PRESENT S-Box`	2	438	18	< 4 s ✔	`timeout`

Table 5.1.: Verifying software implementations of 2nd-order probing secure gadgets using the contract or the netlist of Ibex results in the same confirmation of security (✔) at reduced verification time and validates our approach. E2E security and contracts allow for verifying gate-level security faster than netlist-based verification, yet the complete model allows delivering efficiently masked gadgets (low # of clearings).

selves support bit-sliced and n-sliced masking [Bel+20b], which are among the most popular implementation techniques for masked software. Our analysis, as well as prior work by others, observes joint leakage of bits stored in the same 32-bit register [MOW17; Gao+19; Gig+21], rendering concepts like share-slicing inherently insecure.

5.4.3. Validation of E2E Security

We implement higher-order masked gadgets in software and verify their security against the Ibex contract. We demonstrate the benefit of contracts and validate our methodology by repeating the verification with an independent tool that directly verifies programs against the processor netlist and all of its side effects. Finally, we assess the precision of contracts by confirming that the abstract leakage specification does not demand needless protection.

We port multiple 2nd order masked gadgets presented in [Bar+21b] to RISC-V and check their security using scVerif. For this, we perform a manual translation (which could be automated) of the Genoa contract to the DSL of scVerif and adapt its front-end slightly to accept RISC-V assembly. This contributes an elegant way to verify Stateful t–(S)NI (Definition 3.3), as well as the weaker notion of probing security for assembly implementations against the Ibex contract and due to the compliance immediately prove security against the gate-level leakage. We stress that there are no software

masking verifiers capable of verifying t–(S)NI in gate-level leakage models. Therefore, we compare our results against Coco [Gig+21], a tool that considers the gate-level leakage of processor netlists, but only supports probing security.

All gadgets are hardened by adding the least amount of clearing instructions until they are threshold probing secure, *i.e.*, compliant with the contract under a weaker notion of security yet claimed secure against all gate-level leakage of Ibex. We have checked each gadget with both Coco and scVerif, and the results are shown in Table 5.1. The correctness of our methodology, hardware compliance checking tool, and pen-and-paper model reduction (Theorem 5.1) are confirmed since there is no case where scVerif reports security while Coco rejects it. However, the netlist-based verification fails due to a time-out after multiple hours when applied on a slightly larger design whereas verification based on the contracts completes in less than 4 seconds. We emphasize this benefit in spite of large masked implementations like $\mathsf{Compress}_q$ and DecompressedComparison (Chapter 4), for which verification approaches directly based on netlists appear not to be worthwhile, especially at the demanded higher order for Kyber.

Additionally, we check that, whenever one of the clearing instructions that mitigate contract leakage is removed, Coco also rejects the program due to some gate-level leakage in the processor netlist. If Coco would report that implementation with fewer clearings is still secure, it would mean that the leakage generalization in the contract was too broad and requires needless hardening of the program. In our tests, whenever we remove any of the clearings from Table 5.1, Coco always reports some gate-level leakage violating probing security, except for the Present S-Box which we cannot analyze with Coco. This indicates that our contract does not cause wrong insecurity reports (false negatives) and also does not incur superfluous leakage mitigation.

5.5. Discussion

Contracts serve as a rigorous extension of fine-grained models with provable E2E security. They answer the question of leakage model completeness (research question **Q2**) in multiple ways.

Contracts are provably complete software leakage models based on well-understood gate-level leakage models for hardware circuits and our methods can be extended to cover user-defined gate-level leakage behavior. Verification of probing security notions in contracts implies security at the gate level of any compliant processor, significantly reducing the gap between verified software security and physical resilience. A further contribution is that a single contract

permits to harden an implementation once and immediately port the implementation to multiple compliant processors securely. The flexible modeling further permits the incorporation of additional leakage effects, whether they are based on experience, reported vulnerabilities, or physical characterization.

We show that exhaustively modeling the leakage of the numerous gates in a processor can be achieved with few `leak` statements, resulting in slim and intelligible contracts. This abstraction is key to the fast verification of software in contracts, which outperforms state-of-the-art gate-level verification by orders of magnitude. We conjecture verification in contracts to scale to large implementations and higher orders as evaluated in Chapter 4. This conjecture is based on the similarity of the presented complete Ibex contract with the CM0+ model.

Despite the abstract nature of contracts, no superfluous hardening is caused by an over-approximation of gate leakage in our case study. However, it is up to the designer of a contract to produce small abstractions and there might be processor implementations that result in contracts that are unsuitable for efficient masking. We remark that the full potential of contracts and compliance might be exploited by modifying a processor's implementation to reduce leakage and to choose trade-offs between modeling additional contract leakage impacting software performance, or instead increasing gate cost via leakage-reducing hardware modifications.

Either way, contracts restore the ISA abstraction in the presence of side-channel leakage by incorporating precise & dependable specifications of leakage behavior. Their provably complete and correct modeling of the leakage state and propagation of data into the leakage state permits to systematically derive clearings for hardening of masked implementations. Subsequently, this may serve as a dependable foundation for practically secure composition with the Stateful t–(S)NI notions, at high orders and for complex designs such as Kyber components. Further connections to automated repair of masked implementations as well as secure compilation are within reach.

A limitation of our current approach is that users have to specify processor-specific invariants (part of the configuration) which requires expert knowledge. The erroneous specification can, despite our efforts, lead to partial compliance which does not hold for all programs. We believe that additional heuristic testing is not worthwhile and a systematic, exhaustive approach to verify the assumptions for our inductive approach (Section 5.3.1) or to avoid the specification of invariants at all is promising future work.

Our contract for Ibex bears a lot of similarities with known leakage even though it was created by verifying against a gate-level transition leakage only. This raises the interesting question of whether incorporating further effects such as glitch leakage would make a difference in the model. If indeed more

leakage needs to be specified, is this missing in our fine-grained CM0+ model or is glitch leakage physically irrelevant in software execution on processors?

The gain in practical resilience caused by complete contracts is more difficult to qualify. Remember that the physical adversary $\mathsf{Adv}^{\text{PHY}}$ observes measurement samples $\mathbf{L}^{\text{PHY}}$ often perceived as (weighted) sums of large subsets of gate-level leakage, e.g., in [Cha+99]. Compliance and probing security ensure that each subset of t gate-level leakage $\mathbf{L}^{\text{RPM}}$ is secret independent. For the sums observed by $\mathsf{Adv}^{\text{PHY}}$ security statements can be derived through the reductions to noisy leakage models, relying on noise assumptions. But we emphasize that compliance contributes a first step for model completeness and no other approach is able to provide such an exhaustive basis so far.

Another rather unintended contribution is that compliance provides an approach to solve the chicken-and-egg problem of empirical modeling approaches. Namely, the test patterns used to physically assess a hypothetical leakage model rely on the correctness of the said model, e.g., in [MPW22; PV17]. Our approach may serve as a rigorous basis for systematic or heuristic model extensions, leading to fast model evolution as reported in Chapter 4. Future work may consider the automated generation of test patterns to assess a contract's accuracy heuristically based on physical measurements.

Summarizing, contracts, compliance, and E2E security allow to derive or verify the completeness of fine-grained leakage models w.r.t. well-understood gate-level leakage models and, in addition, allow to incorporate custom leakage behavior at will when combined with physical assessment. They permit the development of large & efficient implementations with higher-order resilience against all gate-level leakage and with fast verification time as required for the secure development flow in Figure 1.2. Finally, note that the flexible yet industry-grade GENOA DSL enables to share dependable leakage models, to restore the ISA abstraction and to foster predictable security & portability in the presence of side-channel, without demanding the disclosure or accessibility of processor implementations.

6. Conclusion and Outlook

This work successfully establishes evidence that formal verification allows for qualitative, reliable, and quick assessment of the physical side-channel security of software.

The concept of explicit leakage is key to the results: First, the explicit representation of leakage facilitates flexible and accurate modeling of diverse physical side-channel behavior in fine-grained leakage models or contracts. Second, the separation of leakage and semantic specification facilitates leakage-preserving program transformations which enable generic verification of security notions and hardware compliance without committing to particular leakage behavior. Third, the abstract and intelligible representation of physical leakage provides developers with accurate information guiding fixes as well as general strategies for systematic hardening and optimization of masked software.

Several conclusions can be drawn from the results of this work:

Predictable physical security The described modeling technique and verification flow accurately and reliably predict physical resilience when based on qualitative models. Fine-grained models allow the representation of diverse leakage behavior and flexible adjustments for covering additional or different leakage.

Secure composition on embedded devices The Stateful t–(S)NI notions systematically prevent composition flaws resulting from the microarchitecture of a processor. The notions permit to securely compose implementations analog to standard t–(S)NI. In turn, this enables the decomposition of the secure development process of large applications into more manageable components.

Scalability Verification in fine-grained models is significantly faster in assessing the security of implementation than physical or emulated evaluation. Further, it scales to greater implementation size and higher security order. Verification in fine-grained models allows verifying and masking larger software implementations than other verification approaches.

Secure development Combining verification and physical assessment is a viable approach to the systematic development of large implementations

attaining security in practice. An important observation is that verification provides accurate information guiding security fixes: verification guides development instead of just proving the outcome of development. The rapid assessment feedback enables exploring the large space of optimization to improve the performance and randomness consumption. In turn, the more efficient implementations have reduced leakage and thereby better resilience to horizontal attacks. Verification provides a rigorous security claim and a clear indication of when to continue with a profound physical assessment. A development flow without verification and fine-grained leakage models seems very unlikely to be worthwhile for large applications like KYBER.

Dependable leakage models The dependence of verification results on the quality of the model is inherent but manageable. First, the model-independent physical assessment in the secure development process systematically detects modeling gaps and permits iterative mitigation of gaps in the model. Second, verifiable compliance allows the creation of complete models that capture the entire known gate-level leakage of processors and thereby yields models without gaps. This yields a sound foundation as it essentially reduces the modeling of device-specific leakage to the well-understood leakage behavior of hardware gates. Analog to fine-grained models any further leakage behavior can be integrated flexibly. Finally, the end-to-end (E2E) security reduction allows making use of existing reductions, e.g., to noisy leakage models. The flexible yet industry-grade GENOA domain specific language permits the sharing of dependable leakage models, to restore the instruction set architecture abstraction and to foster predictable security & portability in the presence of side-channel, without demanding the disclosure or accessibility of processor implementations.

Applicability Our simple proof of concept tool SCVERIF applies to a wide range of implementations, including look-up tables. The underlying concept of explicit leakage can be seamlessly ported to other industry-grade modeling languages and hence permits the use of models in other assessment approaches such as emulation or to perform verification using other software tools than MASKVERIF.

Several questions and problems remain for future work.

Verifying compositions This work focuses on the verification of implementations as a whole unit. Given the need to scale to larger security orders for large implementations like KYBER this approach represents a bottleneck. The problem can be approached from two sides; First, by verifying

the security of a composition of t–(S)NI gadgets relying on the properties of the involved gadget implementations. Second, by automatically decomposing a large implementation into the gadgets composing it. Both demand additional routines which are beyond the scope of this work.

Completeness for physical leakage Verifying compliance proves that a contract models all gate-level leakage behavior modeled by the robust probing model. However, it does not provide direct statements on the completeness of modeling the measurable physical leakage $\mathbf{L}^{\text{PHY}}$. Extending the accuracy of verifying compliance to systematically cover such leakage based on the gates of a processor is a challenging problem that can be approached via the noisy leakage model or random probing model and refined based on additional information such as physical location of gates after place & route. Alternatively, the automated generation of hardened implementations as test patterns for physical evaluation remains an interesting approach for improving the reliability of fine-grained models.

Verifying horizontal resilience The presented verification of security provides resilience against t^{th}-order attacks but connects to resilience against attacks exploiting more leakage via theoretical security reductions only. Incorporating information about the actual noise level of target devices in the verification of implementations or computing bounds on the needed noise level, e.g., via the random probing model, remains an interesting future approach to this problem.

Automating hardening and implementation The implementation and hardening of gadgets is significantly eased but remains a tedious activity. Combining fine-grained models and contracts with existing approaches to automated repair is a promising approach for automating the hardening process but leaves open the problem of producing optimized implementations. A co-advised approach to automated synthesis demonstrates the hardness of this problem [Hei20]. Joint work with Gilles Barthe, Benjamin Grégoire, and Philipp Wolters on approaches towards general secure compilation of gadgets to executable code appears promising but requires more research [Gou19; Wol19]

A. Supporting Material for Section 3

A.1. Listings and Algorithms

Listing A.1: License of scVerif, presented gadgets and code snippets.

```
1  // Modified BSD 3 Clause Clear License
2
3  // Copyright 2019-2020 - VeriSec Consortium, Inria
4  // All rights reserved.
5
6  // Redistribution and use in source and binary forms, with or
     ↪ without modification, are permitted under any copyright
     ↪  provided that the following conditions are met:
7
8  // 1. Redistributions of source code must retain the above
     ↪ copyright notice, this list of conditions and the
     ↪ following disclaimer.
9  // 2. Redistributions in binary form must reproduce the above
     ↪ copyright notice, this list of conditions and the
     ↪ following disclaimer in the documentation and/or other
     ↪ materials provided with the distribution.
10 // 3. Neither the name of the copyright holder nor the names
     ↪ of its contributors may be used to endorse or promote
     ↪ products derived from this software without specific
     ↪ prior written permission.
11
12 // THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
     ↪ CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED
     ↪ WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
     ↪ WARRANTIES OF MERCHANTABILITY, NONINFRINGEMENT, AND
     ↪ FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO
     ↪ EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
     ↪ LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
     ↪ EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
     ↪  LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
```

```
↪ SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
↪ INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
↪ LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR
↪ TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY
↪  WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
↪ OF THE POSSIBILITY OF SUCH DAMAGE.
```

Algorithm A.1 2nd-order Stateful t–SNI maskedAnd [Bar+21b].

Input: Registers r1 pointing to shares $\mathsf{BEnc}^3_{32,2}(a) := \left(A^{(\cdot)_B}\right)$, r2 pointing to shares $\mathsf{BEnc}^3_{32,2}(b) := \left(B^{(\cdot)_B}\right)$ and r3 to randomness R_0, R_1, R_3.

Output: Register r0 pointing to shares $\left(C^{(\cdot)_B}\right) = \mathsf{BEnc}^3_{32,2}(a)$, such that

$$
\begin{aligned}
C^{(0)} &= A^{(0)_B}B^{(0)_B} + R_0 + A^{(0)_B}B^{(1)_B} + R_1 + A^{(1)_B}B^{(0)_B} \\
C^{(1)_B} &= A^{(1)_B}B^{(1)_B} + R_1 + A^{(1)_B}B^{(2)_B} + R_2 + A^{(2)_B}B^{(1)_B} \\
C^{(2)_B} &= A^{(2)_B}B^{(2)_B} + R_2 + A^{(2)_B}B^{(0)_B} + R_0 + A^{(0)_B}B^{(2)_B}.
\end{aligned}
$$

```
1: load(r5, r1, 0);
2: load(r4, r2, 0);
3: and(r4, r5);
4: load(r6, r3, 0);
5: xor(r6, r4);
6: clear(opB);
7: load(r7, r2, 1);
8: and(r7, r5);
9: xor(r6, r7);
10: loadpub(r4);
11: load(r4, r1, 1);
12: load(r5, r2, 0);
13: clear(opB);
14: and(r4, r5);
15: xor(r6, r4);
16: clear(opA);
17: loadpub(r5);
18: load(r5, r3, 1);
19: xor(r6, r5);
20: store(r6, r0, 0);
21: clear(opW);
22: loadpub(r4);
23: loadpub(r5);
24: loadpub(r7);
25: load(r5, r1, 1);
26: load(r4, r2, 1);
27: and(r4, r5);
28: load(r6, r3, 1);
29: xor(r6, r4);
30: clear(opB);
31: load(r7, r2, 2);
32: and(r7, r5);
33: xor(r6, r7);
34: loadpub(r4);
35: load(r4, r1, 2);
36: load(r5, r2, 1);
37: clear(opB);
38: and(r4, r5);
39: xor(r6, r4);
40: clear(opA);
41: loadpub(r5);
42: load(r5, r3, 2);
43: xor(r6, r5);
44: store(r6, r0, 1);
45: clear(opW);
46: loadpub(r4);
47: loadpub(r5);
48: loadpub(r7);
49: load(r5, r1, 2);
50: load(r4, r2, 2);
51: and(r4, r5);
52: load(r6, r3, 2);
53: xor(r6, r4);
54: clear(opB);
55: load(r7, r2, 0);
56: and(r7, r5);
57: xor(r6, r7);
58: loadpub(r4);
59: load(r4, r1, 0);
60: load(r5, r2, 2);
61: clear(opB);
62: and(r4, r5);
63: xor(r6, r4);
64: clear(opA);
65: loadpub(r5);
66: load(r5, r3, 0);
67: xor(r6, r5);
68: store(r6, r0, 2);
69: loadpub(r4);
70: loadpub(r5);
71: loadpub(r6);
72: loadpub(r7);
73: clear(opA);
74: clear(opB);
75: clear(opR);
76: clear(opW);
```

Algorithm A.2 Optimized Stateful t–NI composition of linear function $d = a \oplus b \oplus c$ [Bar+21b].

Input: Registers `r1` pointing to shares $\mathsf{BEnc}^{n_s}_{32,t}(a) := \left(A^{(\cdot)_B}\right)$, `r2` pointing to shares $\mathsf{BEnc}^{n_s}_{32,t}(b) := \left(B^{(\cdot)_B}\right)$ and `r3` pointing to shares $\mathsf{BEnc}^{n_s}_{32,t}(c) := \left(C^{(\cdot)_B}\right)$.

Output: Register `r0` pointing to shares $\left(D^{(\cdot)_B}\right)$ such that $D^{(i)_B} = A^{(i)_B} \oplus B^{(i)_B} \oplus C^{(i)_B}$ for $i \in [n_s]$.

```
1:  for (i = 0 to n_s) do
2:      load(r5, r2, i);
3:      load(r4, r1, i);
4:      xor(r4, r5);
5:      load(r5, r3, i);
6:      xor(r4, r5);
7:      store(r4, r0, i);
8:      clear(opW);
9:      loadpub(r4);
10: loadpub(r5);
11: clear(opA);
12: clear(opB);
13: clear(opR);
```

B. Supporting Material for Section 4

B.1. Parameters and Algorithms of Kyber

The recommended parameters for KYBER and the most relevant reference algorithms are listed in the following. The correspond to the round three version submitted to the NIST PQC standardization process [Nat17]. For the full description of KYBER, refer to [Sch+20b]. The reference Algorithms B.1 to B.3 use byte arrays $\mathbb{F}_{256}^{k}$ of length k with $k = *$ for arbitrary length. For a byte array a notation $a + k$ indicates the byte array starting at byte k of a, with indexing starting at zero.

	n_{K}	k	q	η_1	η_2	(d_u, d_v)	δ
KYBER512	256	2	3329	3	2	(10,4)	2^{-139}
KYBER768	256	3	3329	2	2	(10,4)	2^{-164}
KYBER1024	256	4	3329	2	2	(11,5)	2^{-174}

Table B.1.: Recommended parameter sets for KYBER where δ is the failure probability of the decryption.

Algorithm B.1 $\textsc{Kyber}.\mathsf{CCAKEM.Dec}(c, sk)$: decapsulation.

Input: Ciphertext $c \in \mathbb{F}_{256}^{d_u \cdot k \cdot n_{\text{K}}/8 + d_v \cdot n_{\text{K}}/8}$
Input: Secret key $sk \in \mathbb{F}_{256}^{24 \cdot k \cdot n_{\text{K}}/8 + 96}$
Output: Shared key $k \in \mathbb{F}_{256}^{*}$

1: $pk \coloneqq sk + 12 \cdot k \cdot n_{\text{K}}/8$
2: $h \coloneqq sk + 24 \cdot k \cdot n_{\text{K}}/8 + 32 \in \mathbb{F}_{256}^{32}$
3: $z \coloneqq sk + 24 \cdot k \cdot n_{\text{K}}/8 + 64$
4: $m' \coloneqq \textsc{Kyber}.\mathsf{CPAPKE.Dec}(\vec{s}, (\vec{u}, v))$
5: $(k', r') \coloneqq \mathsf{K_G}(m' \| h)$
6: $c' \coloneqq \textsc{Kyber}.\mathsf{CPAPKE.Enc}(pk, m', r')$
7: **if** $c = c'$ **then**
8:     **return** $k \coloneqq \mathsf{KDF}(k' \| \mathsf{K_H}(c))$
9: **else**
10:     **return** $k \coloneqq \mathsf{KDF}(z \| \mathsf{K_H}(c))$
11: **return** k

Algorithm B.2 KYBER.$\mathsf{CPAPKE.Enc}(pk, m, r)$: encryption.

Input: Public key $pk \in \mathbb{F}_{256}^{12 \cdot k \cdot n_K/8+32}$
Input: Message $m \in \mathbb{F}_{256}^{32}$
Input: Random coins $r \in \mathbb{F}_{256}^{32}$
Output: Ciphertext $c \in \mathbb{F}_{256}^{d_u \cdot k \cdot n_K/8 + d_v \cdot n_K/8}$

1: $N \coloneqq 0$
2: $\underline{\vec{t}} \coloneqq \mathsf{Decode}_{12}(pk)$
3: $\rho \coloneqq pk + 12 \cdot k \cdot n_K/8$
4: **for** i from 0 to $k-1$ **do** ▷ Generate matrix $\hat{a} \in \mathbb{K}_q^{k \times k}$ in NTT domain
5: **for** j from 0 to $k-1$ **do**
6: $\underline{\hat{a}}^T[i][j] \coloneqq \mathsf{Parse}(\mathsf{XOF}(\rho, i, j))$
7: **for** i from 0 to $k-1$ **do** ▷ Sample $\vec{r} \in \mathbb{K}_q^k$ from CBD_{η_1}
8: $\vec{r}[i] \coloneqq \mathsf{CBD}_{\eta_1}(\mathsf{PRF}(r, N))$
9: $N \coloneqq N + 1$
10: **for** i from 0 to $k-1$ **do** ▷ Sample $\vec{e}_1 \in \mathbb{K}_q^k$ from CBD_{η_2}
11: $\vec{e}_1[i] \coloneqq \mathsf{CBD}_{\eta_2}(\mathsf{PRF}(r, N))$
12: $N \coloneqq N + 1$
13: $e_2 \coloneqq \mathsf{CBD}_{\eta_2}(\mathsf{PRF}(r, N))$ ▷ Sample $e_2 \in \mathbb{K}_q$ from CBD_{η_2}
14: $\underline{\vec{r}} \coloneqq \mathsf{NTT}(\vec{r})$
15: $\vec{u} \coloneqq \mathsf{NTT}^{-1}(\underline{\hat{a}}^T \circ \underline{\vec{r}}) + \vec{e}_1$ ▷ $\vec{u} \coloneqq \hat{a}^T \vec{r} + \vec{e}_1$
16: $v \coloneqq \mathsf{NTT}^{-1}(\underline{\vec{t}}^T \circ \underline{\vec{r}}) + e_2 + \mathsf{Decompress}_q(\mathsf{Decode}_1(m), 1)$ ▷ $v \coloneqq \vec{t}^T \vec{r} + e_2 + \mathsf{Decompress}_q(m, 1)$
17: $c_1 \coloneqq \mathsf{Encode}_{d_u}(\mathsf{Compress}_q(\vec{u}, d_u))$
18: $c_2 \coloneqq \mathsf{Encode}_{d_v}(\mathsf{Compress}_q(v, d_v))$
19: **return** $c = (c_1 \| c_2)$ ▷ $c \coloneqq (\mathsf{Compress}_q(\vec{u}, d_u), \mathsf{Compress}_q(v, d_v))$

Algorithm B.3 KYBER.$\mathsf{CPAPKE.Dec}(sk, c)$: decryption.

Input: Secret key $sk \in \mathbb{F}_{256}^{12 \cdot k \cdot n_K/8}$
Input: Ciphertext $c \in \mathbb{F}_{256}^{d_u \cdot k \cdot n_K/8 + d_v \cdot n_K/8}$
Output: Message $m \in \mathbb{F}_{256}^{32}$

1: $\vec{u} \coloneqq \mathsf{Decompress}_q(\mathsf{Decode}_{d_u}(c), d_u)$
2: $v \coloneqq \mathsf{Decompress}_q(\mathsf{Decode}_{d_v}(c + d_u \cdot k \cdot n_K/8), d_v)$
3: $\underline{\vec{s}} \coloneqq \mathsf{Decode}_{12}(sk)$
4: $m \coloneqq \mathsf{Encode}_1(\mathsf{Compress}_q(v - \mathsf{NTT}^{-1}(\underline{\vec{s}}^T \circ \mathsf{NTT}(\vec{u})), 1))$ ▷ $m \coloneqq \mathsf{Compress}_q(v - \vec{s}^T \vec{u}, 1))$
5: **return** m

B.2. Proof of Theorem 4.1

- G_{13} (NI): The $t_{\mathsf{G}_{13}}$ internal probes and $o_{\mathsf{G}_{13}}$ output shares of the gadget can be simulated with $t_{\mathsf{G}_{13}} + o_{\mathsf{G}_{13}}$ shares of the output G_{11} and G_{12}.
- [Bos+21]: G_{13} (NI): The $t_{\mathsf{G}_{13}}$ internal probes and $o_{\mathsf{G}_{13}}$ output shares of the gadget can be simulated with $t_{\mathsf{G}_{13}} + o_{\mathsf{G}_{13}}$ shares of $X_{11}^{(\cdot)_B}$ and of the output of G_{12}.
- G_{12} (SNI): The $t_{\mathsf{G}_{12}}$ internal probes and $o_{\mathsf{G}_{12}}$ output shares of the gadget can be simulated with $t_{\mathsf{G}_{12}}$ shares of the output G_{10} and G_{11}.
- G_{11} (NI): The $t_{\mathsf{G}_{11}}$ internal probes and $o_{\mathsf{G}_{11}}$ output shares of the gadget can be simulated with $t_{\mathsf{G}_{11}} + o_{\mathsf{G}_{11}}$ shares of $X_{11}^{(\cdot)_B}$.
- G_{10} (SNI): The $t_{\mathsf{G}_{10}}$ internal probes and $o_{\mathsf{G}_{10}}$ output shares of the gadget can be simulated with $t_{\mathsf{G}_{10}}$ shares of $X_{10}^{(\cdot)_B}$ and of the output of G_9.
- G_9 (SNI): The t_{G_9} internal probes and o_{G_9} output shares of the gadget can be simulated with t_{G_9} shares of $X_9^{(\cdot)_B}$ and of the output of G_8.
- G_8 (SNI): The t_{G_8} internal probes and o_{G_8} output shares of the gadget can be simulated with t_{G_8} shares of the output of G_7.
- G_7 (NI): The t_{G_7} internal probes and o_{G_7} output shares of the gadget can be simulated with $t_{\mathsf{G}_7} + o_{\mathsf{G}_7}$ shares of $X_8^{(\cdot)_B}$ and of the output of G_6.
- G_6 (SNI): The t_{G_6} internal probes and o_{G_6} output shares of the gadget can be simulated with t_{G_6} shares of $X_7^{(\cdot)_B}$ and of the output of G_5.
- G_5 (SNI): The t_{G_5} internal probes and o_{G_5} output shares of the gadget can be simulated with t_{G_5} shares of the output of G_4.
- G_4 (NI): The t_{G_4} internal probes and o_{G_4} output shares of the gadget can be simulated with $t_{\mathsf{G}_4} + o_{\mathsf{G}_4}$ shares of $X_8^{(\cdot)_B}$.
- G_3 (NI): The t_{G_3} internal probes and o_{G_3} output shares of the gadget can be simulated with $t_{\mathsf{G}_3} + o_{\mathsf{G}_3}$ shares of $X^{(\cdot)_B}$.
- G_2 (SNI): The t_{G_2} internal probes and o_{G_2} output shares of the gadget can be simulated with t_{G_2} shares of the output of G_1.
- G_1 (NI): The t_{G_1} internal probes and o_{G_1} output shares of the gadget can be simulated with $t_{\mathsf{G}_1} + o_{\mathsf{G}_1}$ shares of the input $X^{(\cdot)_A}$.

B.3. Proof of Theorem 4.3

- G_7 (SNI): The t_{G_7} internal probes and o_{G_7} output shares of the gadget can be simulated with t_{G_7} shares of the output of G_5.
- G_6 (NI): The t_{G_6} internal probes and o_{G_6} output shares of the gadget can be simulated with $t_{\mathsf{G}_6} + o_{\mathsf{G}_6}$ shares of the output of G_2.
- G_5 (NI): The t_{G_5} internal probes and o_{G_5} output shares of the gadget can be simulated with $t_{\mathsf{G}_5} + o_{\mathsf{G}_5}$ shares of the output of G_3 and G_4.
- G_4 (SNI): The t_{G_4} internal probes and o_{G_4} output shares of the gadget can be simulated with t_{G_4} shares of the output of G_6.
- G_3 (SNI): The t_{G_3} internal probes and o_{G_3} output shares of the gadget can be simulated with t_{G_3} shares of the output of G_2.
- G_2 (SNI): The t_{G_2} internal probes and o_{G_2} output shares of the gadget can be simulated with t_{G_2} shares of the output of G_1.
- G_1 (NI): The t_{G_1} internal probes and o_{G_1} output shares of the gadget can be simulated with $t_{\mathsf{G}_1} + o_{\mathsf{G}_1}$ shares of the input $\underline{\vec{s}}^{(\cdot)_A}$.

B.4. Fine-Grained Leakage Model for Arm Cortex M0+

We provide the formal leakage model which was used to verify the security of our masked assembly implementations of Algorithms 4.1 and 4.2 for the CM0+ processor. It is based on [Bar+21b] as presented in Chapter 3 but covers additional instructions as detailed in Section 4.3.2.

Listing B.1: Fine-grained side-channel leakage model used during verification of concrete assembly implementations.

```
1 w32 opA; // global leakage state to model leakage behavior
    ↪ which depends on past instructions
2 w32 opB;
3 w32 opR;
4 w32 opW;
5
6 macro ands2_leak (w32 op1, w32 op2) {
7   leak andsCompResult (op1 &w32 op2);
8   leak andsTransition (op1, op1 &w32 op2);
9   leak andsOperand (opA, op1, opB, op2);
```

```
10    leak andsOperandA (opA, op1);
11    leak andsOperandB (opB, op2);
12
13    opA ← op1;
14    opB ← op2;
15  }
16
17  macro eors2_leak (w32 op1, w32 op2) {
18    leak eorsCompResult (op1 ^w32 op2);
19    leak eorsTransition (op1, op1 ^w32 op2);
20    leak eorsOperand (opA, op1, opB, op2);
21    leak eorsOperandA (opA, op1);
22    leak eorsOperandB (opB, op2);
23
24    opA ← op1;
25    opB ← op2;
26  }
27
28  macro ldr3_leak (w32 dst, w32 adr, w32 ofs)
29    w32 val
30  {
31    val ← [w32 mem (int) (adr +w32 ofs)];
32
33    leak ldrOperand1 (opA, adr, opB, ofs);
34
35    leak ldrOperand2A (opA, adr);
36    leak ldrOperand2B (opB, dst);
37    leak ldrMemOperand (opR, val);
38    leak ldrTransition (dst, val);
39
40    opA ← adr;
41    opB ← dst;
42    opR ← val;
43  }
44
45  macro str3_leak (w32 val, w32 adr, w32 ofs) {
46    leak strOperand1 (opA, adr, opB, ofs);
47
48    leak strOperand2A (opA, adr);
49    leak strOperand2B (opB, val);
50    leak strMemOperand (opW, val);
51
52    opA ← adr;
53    opB ← val;
```

```
54     opW ← val;
55   }
56
57   macro mov2_leak (w32 dst, w32 src) {
58     leak movCompResult (src);
59     leak movOperand (opA, dst, opB, src);
60     leak movTransition (dst, src);
61
62     opA ← src; // assumption here was "opA ← dst" which is wrong
63   //  opB ← src; // assumption here was wrong, opB is not
         ↪ cleared but propagated
64   }
65
66   macro adds3_leak (w32 dst, w32 op1, w32 op2) {
67     leak addsCompResult (op1 +w32 op2);
68     leak addsTransition (dst, op1 +w32 op2);
69     leak addsOperand (opA, op1, opB, op2);
70
71     opA ← opA &w32 opB &w32 op1; // worst case assumption
72     opB ← opA &w32 opB &w32 op2; // worst case assumption
73   }
74
75   macro add3_leak (w32 dst, w32 op1, w32 op2) {
76     leak addCompResult (op1 +w32 op2);
77     leak addTransition (dst, op1 +w32 op2);
78     leak addOperand (opA, op1, opB, op2);
79
80     opA ← opA &w32 opB &w32 op1; // worst case assumption
81     opB ← opA &w32 opB &w32 op2; // worst case assumption
82   }
83
84   macro mvns2_leak (w32 dst, w32 src) {
85     leak movCompResult (!w32 src);
86     leak movOperand (opA, dst, opB, src);
87     leak movTransition (dst, !w32 src);
88
89     opA ← opA &w32 src;
90   }
91
92   macro adcs2_leak (w32 dst, w32 op) {
93     leak sbcsCompResult (op -w32 dst);
94     leak sbcsOperand (opA, dst, opB, op);
95     leak sbcsTransition (dst, op -w32 dst);
96
```

```
97    opA ← dst; // assumption
98    opB ← op;
99  }
100
101 macro sbcs2_leak (w32 dst, w32 op) {
102   leak sbcsCompResult (op -w32 dst);
103   leak sbcsOperand (opA, dst, opB, op);
104   leak sbcsTransition (dst, op -w32 dst);
105
106   opA ← opA &w32 op;
107 }
108
109 macro subs3_leak (w32 dst, w32 op1, w32 op2) {
110   leak subsCompResult (op1 +w32 op2);
111   leak subsTransition (dst, op1 +w32 op2);
112   leak subsOperand (opA, op1, opB, op2);
113
114   opA ← op1;
115   opB ← op2;
116 }
117
118 macro sub3_leak (w32 dst, w32 op1, w32 op2) {
119   leak subCompResult (op1 +w32 op2);
120   leak subTransition (dst, op1 +w32 op2);
121   leak subOperand (opA, op1, opB, op2);
122
123   opA ← op1;
124   opB ← op2;
125 }
126
127 macro ldrb3_leak (w32 dst, w32 adr, w32 ofs) {
128   ldr3_leak(dst, adr, ofs);
129 }
130
131 macro ldrh3_leak (w32 dst, w32 adr, w32 ofs) {
132   ldr3_leak(dst, adr, ofs);
133 }
134
135 macro strb3_leak (w32 op, w32 adr, w32 ofs) {
136   str3_leak(op, adr, ofs);
137 }
138
139 macro strh3_leak (w32 op, w32 adr, w32 ofs) {
140   str3_leak(op, adr, ofs);
```

```
141 }
142
143 macro sxtb2_leak (w32 dst, w32 src) {
144   leak sxtbCompResult (src);
145   leak sxtbOperand (opA, dst, opB, src);
146   leak sxtbTransition (dst, src);
147
148   opA ← opA &w32 src;
149 }
150
151 macro sxth2_leak (w32 dst, w32 src) {
152   leak sxthCompResult (src);
153   leak sxthOperand (opA, dst, opB, src);
154   leak sxthTransition (dst, src);
155
156   opA ← opA &w32 src;
157 }
158
159 macro lsls3_leak (w32 dst, w32 op1, w32 shift) {
160   leak lslsCompResult (op1);
161   leak lslsOperand (opA, dst, opB, op1);
162   leak lslsTransition (dst, op1);
163
164   opA ← op1; // assumption
165 //  opB ← op1;
166 }
167
168 macro lsls2_leak (w32 op1, w32 op2) {
169   lsls3_leak(op1, op1, op2);
170 }
171
172 macro lsrs3_leak (w32 dst, w32 op1, w32 shift) {
173   leak lsrsCompResult (op1);
174   leak lsrsOperand (opA, dst, opB, op1);
175   leak lsrsTransition (dst, op1);
176
177   opA ← op1; // assumption
178 //  opB ← op1;
179 }
180
181 macro lsrs2_leak (w32 op1, w32 op2) {
182   lsrs3_leak(op1, op1, op2);
183 }
184
```

```
185 macro asrs3_leak (w32 dst, w32 op1, w32 shift) {
186   leak asrsCompResult (op1);
187   leak asrsOperand (opA, dst, opB, op1);
188   leak asrsTransition (dst, op1);
189
190   opA ← op1; // assumption
191 //   opB ← op1;
192 }
193
194 macro asrs2_leak (w32 dst, w32 op1) {
195   asrs3_leak(dst, dst, op1);
196 }
197
198 macro muls3_leak (w32 dst, w32 op1, w32 op2) {
199   leak mulsCompResult (op1 *w32 op2);
200   leak mulsTransition (dst, op1 *w32 op2);
201   leak mulsOperand (opA, op1, opB, op2);
202
203   opA ← op1;
204   opB ← op2;
205 }
206
207 macro muls2_leak (w32 op1, w32 op2) {
208   muls3_leak(op1, op1, op2);
209 }
210
211 macro orrs2_leak (w32 op1, w32 op2) {
212   leak orrsCompResult (op1 |w32 op2);
213   leak orrsTransition (op1, op1 |w32 op2);
214   leak orrsOperand (opA, op1, opB, op2);
215   leak orrsOperandA (opA, op1);
216   leak orrsOperandB (opB, op2);
217
218   opA ← op1;
219   opB ← op2;
220 }
221
222 macro rsbs2_leak (w32 op1, w32 op2) {
223   leak orrsCompResult (op2 -w32 op1);
224   leak orrsTransition (op1, op2 -w32 op1);
225   leak orrsOperand (opA, op1, opB, op2);
226
227   opA ← opA &w32 op2;
228 }
```

```
229
230 macro negs2_leak (w32 op1, w32 op2) {
231   rsbs2_leak(op1, op2);
232 }
233
234 macro uxtb2_leak (w32 dst, w32 src) {
235   leak uxtbCompResult (src);
236   leak uxtbOperand (opA, dst, opB, src);
237   leak uxtbTransition (dst, src);
238
239   opA ← opA &w32 src;
240 }
241
242 macro uxth2_leak (w32 dst, w32 src) {
243   uxtb2_leak(dst, src);
244 }
245
246 macro cmp2_leak (w32 op1, w32 op2) {
247   subs3_leak(op1, op1, op2);
248 }
249
250 macro tst2_leak (w32 op1, w32 op2) {
251   leak tstCompResult (op1 &w32 op2);
252   leak tstOperand (opA, op1, opB, op2);
253   leak tstOperandA (opA, op1);
254   leak tstOperandB (opB, op2);
255
256   opA ← opA &w32 op1;
257   opB ← opB &w32 op2;
258 }
259
260 // the semantic of pop instructions is build from the load,
      ↪ therefore the values on the stack leak as well
261 macro pop1_leak (w32 op1) {
262   leak pop (op1, opR);
263 }
264
265 macro pop2_leak (w32 op1, w32 op2) {
266   leak pop (op1, op2, opR);
267 }
268
269 macro pop3_leak (w32 op1, w32 op2, w32 op3) {
270   leak pop (op1, op2, op3, opR);
271 }
```

```
272
273 macro pop4_leak (w32 op1, w32 op2, w32 op3, w32 op4) {
274   leak pop (op1, op2, op3, op4, opR);
275 }
276
277 macro pop5_leak (w32 op1, w32 op2, w32 op3, w32 op4, w32 op5)
      ↪ {
278   leak pop (op1, op2, op3, op4, op5, opR);
279 }
280
281 macro pop6_leak (w32 op1, w32 op2, w32 op3, w32 op4, w32 op5,
      ↪ w32 op6) {
282   leak pop (op1, op2, op3, op4, op5, op6, opR);
283 }
284
285 macro pop7_leak (w32 op1, w32 op2, w32 op3, w32 op4, w32 op5,
      ↪ w32 op6, w32 op7) {
286   leak pop (op1, op2, op3, op4, op5, op6, op7, opR);
287 }
288
289 macro pop8_leak (w32 op1, w32 op2, w32 op3, w32 op4, w32 op5,
      ↪ w32 op6, w32 op7, w32 op8) {
290   leak pop (op1, op2, op3, op4, op5, op6, op7, op8, opR);
291 }
292
293 macro pop9_leak (w32 op1, w32 op2, w32 op3, w32 op4, w32 op5,
      ↪ w32 op6, w32 op7, w32 op8, w32 op9) {
294   leak pop (op1, op2, op3, op4, op5, op6, op7, op8, op9, opR);
295 }
296
297 macro push1_leak (w32 op1) {
298   leak push (op1, opW);
299 }
300
301 macro push2_leak (w32 op1, w32 op2) {
302   leak push (op1, op2, opW);
303 }
304
305 macro push3_leak (w32 op1, w32 op2, w32 op3) {
306   leak push (op1, op2, op3, opW);
307 }
308
309 macro push4_leak (w32 op1, w32 op2, w32 op3, w32 op4) {
310   leak push (op1, op2, op3, op4, opW);
```

```
311 }
312
313 macro push5_leak (w32 op1, w32 op2, w32 op3, w32 op4, w32
      ↪ op5) {
314   leak push (op1, op2, op3, op4, op5, opW);
315 }
316
317 macro push6_leak (w32 op1, w32 op2, w32 op3, w32 op4, w32
      ↪ op5, w32 op6) {
318   leak push (op1, op2, op3, op4, op5, op6, opW);
319 }
320
321 macro push7_leak (w32 op1, w32 op2, w32 op3, w32 op4, w32
      ↪ op5, w32 op6, w32 op7) {
322   leak push (op1, op2, op3, op4, op5, op6, op7, opW);
323 }
324
325 macro push8_leak (w32 op1, w32 op2, w32 op3, w32 op4, w32
      ↪ op5, w32 op6, w32 op7, w32 op8) {
326   leak push (op1, op2, op3, op4, op5, op6, op7, op8, opW);
327 }
328
329 macro push9_leak (w32 op1, w32 op2, w32 op3, w32 op4, w32
      ↪ op5, w32 op6, w32 op7, w32 op8, w32 op9) {
330   leak push (op1, op2, op3, op4, op5, op6, op7, op8, op9,
        ↪ opW);
331 }
```

C. Supporting Material for Section 5

C.1. Proof of Theorem 5.2

We provide the proof of Theorem 5.2 published in [Blo+22].

Proof. We prove the equality of the two statements by showing an implication in both directions. First, we prove that (5.17) follows from (5.3). From the functional congruence of f, we have:

$$\forall c, c' \in \mathcal{C} : (\lambda_C(c) = \lambda_C(c')) \Rightarrow (f \circ \lambda_C(c) = f \circ \lambda_C(c')) .$$

After instantiating the statement (5.3) separately for the primed and non-primed versions of $h \in \mathcal{H}$ and $c \in \mathcal{C}$, we get:

$$\forall h \in \mathcal{H}, c \in \mathcal{C} : \Psi(h, c) \Rightarrow f \circ \lambda_C(c) = \lambda_H(h) ,$$
$$\forall h' \in \mathcal{H}, c' \in \mathcal{C} : \Psi(h', c') \Rightarrow f \circ \lambda_C(c') = \lambda_H(h') .$$

If all three premises are fulfilled simultaneously, then also all consequences of the implication must be fulfilled simultaneously. Therefore, we consolidate the left- and right-hand sides. Afterwards, we simplify the right-hand side by substituting $f \circ \lambda_C(c)$ with $\lambda_H(h)$, respectively $f \circ \lambda_C(c')$ with $\lambda_H(h')$, to get (5.17).

For the second direction of the proof, we assume (5.17) and construct f so that it fulfills (5.3) and is well defined for all $u \in \mathcal{U}$. First, we define the subset $\widehat{\mathcal{U}} \subseteq \mathcal{U}$ of function inputs as

$$\widehat{\mathcal{U}} := \{u \mid \exists h \in \mathcal{H}, c \in \mathcal{C} : u = \lambda_C(c) \wedge \Psi(h, c)\} . \tag{C.1}$$

For inputs $u \in \mathcal{U} \setminus \widehat{\mathcal{U}}$, we define $f(u)$ as an arbitrary result $v \in \mathcal{V}$. This partial definition trivially fulfills (5.3). For all other $u \in \widehat{\mathcal{U}}$, we define $f(u) := \lambda_H(h)$, for an arbitrary qualified h and c as in (C.1). We now argue that this portion of f is well defined, because $\lambda_H(h)$ is fixed for u. Consider the case where we

can pick two such pairs:

$$\exists h, h' \in \mathcal{H}, c, c' \in \mathcal{C} : u = \lambda_C(c) \wedge \Psi(h, c) \wedge \\ u = \lambda_C(c') \wedge \Psi(h', c').$$

Because $\lambda_C(c) = \lambda_C(c') = u$, assumption (5.17) implies that $\lambda_H(h)$ is unique since we always get $\lambda_H(h') = \lambda_H(h)$. □

C.2. Compliant Contract for the Ibex RISC-V Processor

The missing parts of the contract for IBEX discussed in Section 5.4 are depicted in Listing C.1.

Listing C.1: Contract model of remaining instructions for IBEX.

```
1  /*================================*/
2  /* RISCV Sail Model */
3  /* This Sail
4  iscv architecture model, comprising all files and directories except
     ↪ for the snapshots of the Lem and Sail libraries in the
     ↪ prover_snapshots directory (which include copies of their
     ↪ licences), is subject to the BSD two-clause licence below. */
5  /* Copyright (c) 2017-2021 Prashanth Mundkur, Rishiyur S. Nikhil and
     ↪ Bluespec Inc., Jon French, Brian Campbell, Robert Norton-Wright,
     ↪  Alasdair Armstrong, Thomas Bauereiss, Shaked Flur, Christopher
     ↪ Pulte, Peter Sewell, Alexander Richardson, Hesham Almatary,
     ↪ Jessica Clarke, Microsoft, for contributions by Robert Norton-
     ↪ Wright and Nathaniel Wesley Filardo, Peter Rugg and Aril
     ↪ Computer Corp., for contributions by Scott Johnson */
6  /* Copyright 2020-2022 - TUHH, TU Graz */
7  /* All rights reserved. */
8  /* This software was developed by the above within the Rigorous
     ↪ Engineering of Mainstream Systems (REMS) project, partly funded
     ↪ by EPSRC grant EP/K008528/1, at the Universities of Cambridge
     ↪ and Edinburgh. */
9  /* This software was developed by SRI International and the University
     ↪ of Cambridge Computer Laboratory (Department of Computer Science
     ↪  and Technology) under DARPA/AFRL contract FA8650-18-C-7809 ("
     ↪ CIFV"), and under DARPA contract HR0011-18-C-0016 ("ECATS") as
     ↪ part of the DARPA SSITH research programme. */
10 /* This project has received funding from the European Research Council
     ↪  (ERC) under the European Union's Horizon 2020 research and
     ↪ innovation programme (grant agreement 789108, ELVER). */
```

```
11 /* This software has received funding from the Federal Ministry of
     ↪ Education and Research (BMBF) as part of the VE-Jupiter project
     ↪ grant 16ME0231K. */
12 /* This work was supported by the Austrian Research Promotion Agency (
     ↪ FFG) through the FERMION project (grant number 867542). */
13 /* Redistribution and use in source and binary forms, with or without
     ↪ modification, are permitted provided that the following
     ↪ conditions are met: */
14 /* 1. Redistributions of source code must retain the above copyright
     ↪ notice, this list of conditions and the following disclaimer. */
15 /* 2. Redistributions in binary form must reproduce the above copyright
     ↪  notice, this list of conditions and the following disclaimer in
     ↪  the documentation and/or other materials provided with the
     ↪ distribution. */
16 /* THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS''
     ↪ AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
     ↪ LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND
     ↪ FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT
     ↪ SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT,
     ↪ INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
     ↪ DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
     ↪ SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
     ↪ BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
     ↪ LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (
     ↪ INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE
     ↪  USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
     ↪ SUCH DAMAGE. */
17 /*==============================*/
18
19 val common_leakage : (xlenbits, xlenbits) -> unit effect {rreg, wreg,
     ↪ leakage}
20 function common_leakage(rs1_val : xlenbits, rs2_val : xlenbits) =
21 {
22   leak(rs1_val, rs2_val, rf_pA, rf_pB, mem_last_addr, mem_last_read);
23   rf_pA = rs1_val;
24   rf_pB = rs2_val;
25   mem_last_read = 0x00000000;
26 }
27
28 val overwrite_leakage : (regidx, xlenbits) -> unit effect {rreg,
     ↪ leakage}
29 function overwrite_leakage(dest_idx : regidx, res : xlenbits) = {
30   let x1_n = if (dest_idx == 0b00001)
31     then {res} else {x1} in leak(x1 , x1_n );
```

```
32   let x2_n = if (dest_idx == 0b00010)
33     then {res} else {x2} in leak(x2 , x2_n );
34   let x3_n = if (dest_idx == 0b00011)
35     then {res} else {x3} in leak(x3 , x3_n );
36   let x4_n = if (dest_idx == 0b00100)
37     then {res} else {x4} in leak(x4 , x4_n );
38   let x5_n = if (dest_idx == 0b00101)
39     then {res} else {x5} in leak(x5 , x5_n );
40   let x6_n = if (dest_idx == 0b00110)
41     then {res} else {x6} in leak(x6 , x6_n );
42   let x7_n = if (dest_idx == 0b00111)
43     then {res} else {x7} in leak(x7 , x7_n );
44   let x8_n = if (dest_idx == 0b01000)
45     then {res} else {x8} in leak(x8 , x8_n );
46   let x9_n = if (dest_idx == 0b01001)
47     then {res} else {x9} in leak(x9 , x9_n );
48   let x10_n = if (dest_idx == 0b01010)
49     then {res} else {x10} in leak(x10, x10_n);
50   let x11_n = if (dest_idx == 0b01011)
51     then {res} else {x11} in leak(x11, x11_n);
52   let x12_n = if (dest_idx == 0b01100)
53     then {res} else {x12} in leak(x12, x12_n);
54   let x13_n = if (dest_idx == 0b01101)
55     then {res} else {x13} in leak(x13, x13_n);
56   let x14_n = if (dest_idx == 0b01110)
57     then {res} else {x14} in leak(x14, x14_n);
58   let x15_n = if (dest_idx == 0b01111)
59     then {res} else {x15} in leak(x15, x15_n);
60 }
61
62 /* **************************** */
63
64 enum uop = {RISCV_LUI, RISCV_AUIPC}
65 union clause ast = UTYPE : (bits(20), regidx, uop)
66
67 mapping encdec_uop : uop <-> bits(7) = {
68   RISCV_LUI <-> 0b0110111,
69   RISCV_AUIPC <-> 0b0010111
70 }
71
72 mapping clause encdec = UTYPE(imm, rd, op)
73   <-> imm @ rd @ encdec_uop(op)
74   if (rd[4] == bitzero)
75
```

```
76  function clause execute UTYPE(imm, rd, op) = {
77    let rs1_val = X(0b0 @ imm[6 .. 3]);
78    let rs2_val = X(0b0 @ imm[11 .. 8]);
79    common_leakage(rs1_val, rs2_val);
80
81    let off : xlenbits = EXTS(imm @ 0x000);
82    let ret : xlenbits = match op {
83      RISCV_LUI => off,
84      RISCV_AUIPC => get_arch_pc() + off
85    };
86
87    X(rd) = ret;
88    RETIRE_SUCCESS
89  }
90
91  /* ****************************** */
92
93  union clause ast = RISCV_JAL : (bits(21), regidx)
94
95  mapping clause encdec =
96    RISCV_JAL(imm_19 @ imm_7_0 @ imm_8 @ imm_18_13 @ imm_12_9 @ 0b0, rd)
97    <-> imm_19 : bits(1) @ imm_18_13 : bits(6) @ imm_12_9 : bits(4) @
        ↪ imm_8 : bits(1) @ imm_7_0 : bits(8) @ rd @ 0b1101111
98    if (rd[4] == bitzero)
99
100 function clause execute (RISCV_JAL(imm, rd)) = {
101   let rs1_val = X(0b0 @ imm[18 .. 15]);
102   let rs2_val = X(0b0 @ imm[3 .. 1]
103                 @ subrange_bits(imm, 11, 11));
104   common_leakage(rs1_val, rs2_val);
105
106   let t : xlenbits = PC + EXTS(imm);
107   let rd_next = get_next_pc();
108
109   overwrite_leakage(rd, rd_next);
110   X(rd) = rd_next;
111   if t[1 .. 0] == 0b00 then {
112     set_next_pc(t);
113     RETIRE_SUCCESS
114   } else RETIRE_FAIL
115 }
116
117 /* ****************************** */
118
```

```
119 union clause ast =
120   RISCV_JALR : (bits(12), regidx, regidx)
121
122 mapping clause encdec = RISCV_JALR(imm, rs1, rd)
123   <-> imm @ rs1 @ 0b000 @ rd @ 0b1100111
124   if (rs1[4] == bitzero & rd[4] == bitzero)
125
126 function clause execute (RISCV_JALR(imm, rs1, rd)) = {
127   let rs1_val = X(rs1);
128   let rs2_val = X(0b0 @ imm[3..0]);
129   common_leakage(rs1_val, rs2_val);
130
131   let t : xlenbits =
132     [(rs1_val + EXTS(imm)) with 0 = bitzero];
133   if t[1 .. 0] == 0b00 then {
134     overwrite_leakage(rd, get_next_pc());
135     X(rd) = get_next_pc();
136     set_next_pc(t);
137     RETIRE_SUCCESS
138   } else RETIRE_FAIL
139 }
140
141 /* **************************** */
142
143 enum bop = {RISCV_BEQ, RISCV_BNE, RISCV_BLT, RISCV_BGE, RISCV_BLTU,
      ↪ RISCV_BGEU}
144 union clause ast =
145   BTYPE : (bits(13), regidx, regidx, bop)
146
147 mapping encdec_bop : bop <-> bits(3) = {
148   RISCV_BEQ <-> 0b000,
149   RISCV_BNE <-> 0b001,
150   RISCV_BLT <-> 0b100,
151   RISCV_BGE <-> 0b101,
152   RISCV_BLTU <-> 0b110,
153   RISCV_BGEU <-> 0b111
154 }
155
156 mapping clause encdec = BTYPE(imm7_6 @ imm5_0 @ imm7_5_0 @ imm5_4_1 @ 0
      ↪ b0, rs2, rs1, op)
157   <-> imm7_6 : bits(1) @ imm7_5_0 : bits(6) @ rs2 @ rs1 @ encdec_bop(op
      ↪ ) @ imm5_4_1 : bits(4) @ imm5_0 : bits(1) @ 0b1100011
158   if (rs1[4] == bitzero & rs2[4] == bitzero)
159
```

```
160 function clause execute (BTYPE(imm, rs2, rs1, op)) = {
161   let rs1_val = X(rs1);
162   let rs2_val = X(rs2);
163   common_leakage(rs1_val, rs2_val);
164   let taken : bool = match op {
165     RISCV_BEQ => rs1_val == rs2_val,
166     RISCV_BNE => rs1_val != rs2_val,
167     RISCV_BLT => rs1_val <_s rs2_val,
168     RISCV_BGE => rs1_val >=_s rs2_val,
169     RISCV_BLTU => rs1_val <_u rs2_val,
170     RISCV_BGEU => rs1_val >=_u rs2_val
171   };
172
173   overwrite_leakage(0b00000, 0b00000000000000000000000000000000);
174
175   let t : xlenbits = PC + EXTS(imm);
176   if (t[1 .. 0] != 0b00) then
177     return RETIRE_FAIL;
178   if taken then { set_next_pc(t); };
179   return RETIRE_SUCCESS
180 }
181
182 /* **************************** */
183
184 enum iop = {RISCV_ADDI, RISCV_SLTI, RISCV_SLTIU, RISCV_XORI, RISCV_ORI,
       ↪ RISCV_ANDI}
185 union clause ast =
186   ITYPE : (bits(12), regidx, regidx, iop)
187
188 mapping encdec_iop : iop <-> bits(3) = {
189   RISCV_ADDI <-> 0b000,
190   RISCV_SLTI <-> 0b010,
191   RISCV_SLTIU <-> 0b011,
192   RISCV_ANDI <-> 0b111,
193   RISCV_ORI <-> 0b110,
194   RISCV_XORI <-> 0b100
195 }
196
197 mapping clause encdec = ITYPE(imm, rs1, rd, op)
198   <-> imm @ rs1 @ encdec_iop(op) @ rd @ 0b0010011
199   if (rs1[4] == bitzero) & (rd[4] == bitzero)
200
201 function clause execute (ITYPE (imm, rs1, rd, op)) = {
202   let rs1_val = X(rs1);
```

```
203   let rs2_val = X(0b0 @ imm[3 .. 0]);
204   common_leakage(rs1_val, rs2_val);
205   let immext : xlenbits = EXTS(imm);
206   let result : xlenbits = match op {
207     RISCV_ADDI => rs1_val + immext,
208     RISCV_SLTI =>
209       EXTZ(bool_to_bits(rs1_val <_s immext)),
210     RISCV_SLTIU =>
211       EXTZ(bool_to_bits(rs1_val <_u immext)),
212     RISCV_ANDI => rs1_val & immext,
213     RISCV_ORI => rs1_val | immext,
214     RISCV_XORI => rs1_val ^ immext
215   };
216   overwrite_leakage(rd, result);
217   X(rd) = result;
218   RETIRE_SUCCESS
219 }
220
221 /* **************************** */
222
223 enum sop = {RISCV_SLLI, RISCV_SRLI, RISCV_SRAI}
224 union clause ast =
225   SHIFTIOP : (bits(6), regidx, regidx, sop)
226
227 mapping encdec_sop : sop <-> bits(3) = {
228   RISCV_SLLI <-> 0b001,
229   RISCV_SRLI <-> 0b101,
230   RISCV_SRAI <-> 0b101
231 }
232
233 mapping clause encdec = SHIFTIOP(shamt, rs1, rd, RISCV_SLLI)
234   <-> 0b000000 @ shamt @ rs1 @ 0b001 @ rd @ 0b0010011
235   if (shamt[5] == bitzero) &(rs1[4] == bitzero) & (rd[4] == bitzero)
236 mapping clause encdec = SHIFTIOP(shamt, rs1, rd, RISCV_SRLI)
237   <-> 0b000000 @ shamt @ rs1 @ 0b101 @ rd @ 0b0010011
238   if (shamt[5] == bitzero) &(rs1[4] == bitzero) & (rd[4] == bitzero)
239 mapping clause encdec = SHIFTIOP(shamt, rs1, rd, RISCV_SRAI)
240   <-> 0b010000 @ shamt @ rs1 @ 0b101 @ rd @ 0b0010011
241   if (shamt[5] == bitzero) &(rs1[4] == bitzero) & (rd[4] == bitzero)
242
243 function clause execute (SHIFTIOP(shamt, rs1, rd, op)) = {
244   let rs1_val = X(rs1);
245   let rs2_val = X(0b0 @ shamt[3..0]);
246   common_leakage(rs1_val, rs2_val);
```

```
247    /* the decoder guard ensures that shamt[5] = 0 for RV32E */
248    let result : xlenbits = match op {
249      RISCV_SLLI => if sizeof(xlen) == 32
250                    then rs1_val << shamt[4..0]
251                    else rs1_val << shamt,
252      RISCV_SRLI => if sizeof(xlen) == 32
253                    then rs1_val >> shamt[4..0]
254                    else rs1_val >> shamt,
255      RISCV_SRAI => if sizeof(xlen) == 32
256                    then shift_right_arith32(rs1_val, shamt[4..0])
257                    else shift_right_arith64(rs1_val, shamt)};
258    overwrite_leakage(rd, result);
259    X(rd) = result;
260    RETIRE_SUCCESS
261  }
262
263  /* *************************** */
264
265  enum rop = {RISCV_ADD, RISCV_SUB, RISCV_SLL, RISCV_SLT,
266              RISCV_SLTU, RISCV_XOR, RISCV_SRL, RISCV_SRA,
267              RISCV_OR, RISCV_AND}
268  union clause ast = RTYPE : (regidx, regidx, regidx, rop)
269
270  mapping clause encdec = RTYPE(rs2, rs1, rd, RISCV_ADD)
271    <-> 0b0000000 @ rs2 @ rs1 @ 0b000 @ rd @ 0b0110011
272    if (rs1[4] == bitzero) & (rs2[4] == bitzero) & (rd[4] == bitzero)
273
274  mapping clause encdec = RTYPE(rs2, rs1, rd, RISCV_SLT)
275    <-> 0b0000000 @ rs2 @ rs1 @ 0b010 @ rd @ 0b0110011
276    if (rs1[4] == bitzero) & (rs2[4] == bitzero) & (rd[4] == bitzero)
277  mapping clause encdec = RTYPE(rs2, rs1, rd, RISCV_SLTU)
278    <-> 0b0000000 @ rs2 @ rs1 @ 0b011 @ rd @ 0b0110011
279    if (rs1[4] == bitzero) & (rs2[4] == bitzero) & (rd[4] == bitzero)
280  mapping clause encdec = RTYPE(rs2, rs1, rd, RISCV_AND)
281    <-> 0b0000000 @ rs2 @ rs1 @ 0b111 @ rd @ 0b0110011
282    if (rs1[4] == bitzero) & (rs2[4] == bitzero) & (rd[4] == bitzero)
283  mapping clause encdec = RTYPE(rs2, rs1, rd, RISCV_OR)
284    <-> 0b0000000 @ rs2 @ rs1 @ 0b110 @ rd @ 0b0110011
285    if (rs1[4] == bitzero) & (rs2[4] == bitzero) & (rd[4] == bitzero)
286  mapping clause encdec = RTYPE(rs2, rs1, rd, RISCV_XOR)
287    <-> 0b0000000 @ rs2 @ rs1 @ 0b100 @ rd @ 0b0110011
288    if (rs1[4] == bitzero) & (rs2[4] == bitzero) & (rd[4] == bitzero)
289  mapping clause encdec = RTYPE(rs2, rs1, rd, RISCV_SLL)
290    <-> 0b0000000 @ rs2 @ rs1 @ 0b001 @ rd @ 0b0110011
```

```
291   if (rs1[4] == bitzero) & (rs2[4] == bitzero) & (rd[4] == bitzero)
292 mapping clause encdec = RTYPE(rs2, rs1, rd, RISCV_SRL)
293   <-> 0b0000000 @ rs2 @ rs1 @ 0b101 @ rd @ 0b0110011
294   if (rs1[4] == bitzero) & (rs2[4] == bitzero) & (rd[4] == bitzero)
295 mapping clause encdec = RTYPE(rs2, rs1, rd, RISCV_SUB)
296   <-> 0b0100000 @ rs2 @ rs1 @ 0b000 @ rd @ 0b0110011
297   if (rs1[4] == bitzero) & (rs2[4] == bitzero) & (rd[4] == bitzero)
298 mapping clause encdec = RTYPE(rs2, rs1, rd, RISCV_SRA)
299   <-> 0b0100000 @ rs2 @ rs1 @ 0b101 @ rd @ 0b0110011
300   if (rs1[4] == bitzero) & (rs2[4] == bitzero) & (rd[4] == bitzero)
301
302 function clause execute (RTYPE(rs2, rs1, rd, op)) = {
303   let rs1_val = X(rs1);
304   let rs2_val = X(rs2);
305   common_leakage(rs1_val, rs2_val);
306
307   let result : xlenbits = match op {
308     RISCV_ADD => rs1_val + rs2_val,
309     RISCV_SLT => EXTZ(bool_to_bits(rs1_val <_s rs2_val)),
310     RISCV_SLTU => EXTZ(bool_to_bits(rs1_val <_u rs2_val)),
311     RISCV_AND => rs1_val & rs2_val,
312     RISCV_OR => rs1_val | rs2_val,
313     RISCV_XOR => rs1_val ^ rs2_val,
314     RISCV_SLL => if sizeof(xlen) == 32
315                  then rs1_val << (rs2_val[4..0])
316                  else rs1_val << (rs2_val[5..0]),
317     RISCV_SRL => if sizeof(xlen) == 32
318                  then rs1_val >> (rs2_val[4..0])
319                  else rs1_val >> (rs2_val[5..0]),
320     RISCV_SUB => rs1_val - rs2_val,
321     RISCV_SRA => if sizeof(xlen) == 32
322                  then shift_right_arith32(rs1_val, rs2_val[4..0])
323                  else shift_right_arith64(rs1_val, rs2_val[5..0])
324   };
325   // leak(X(rd), result);
326   overwrite_leakage(rd, result);
327
328   X(rd) = result;
329   RETIRE_SUCCESS
330 }
331
332 /* **************************** */
333
334 enum word_width = {BYTE, HALF, WORD, DOUBLE}
```

```
335 union clause ast = LOAD :
336   (bits(12), regidx, regidx, bool, word_width, bool, bool)
337
338 mapping clause encdec = LOAD(imm, rs1, rd, is_unsigned, size, false,
      ↪ false)
339   if ((word_width_bytes(size) < sizeof(xlen_bytes)) | (not_bool(
        ↪ is_unsigned) & word_width_bytes(size) <= sizeof(xlen_bytes)))
        ↪ & (rs1[4] == bitzero) & (rd[4] == bitzero)
340   <-> imm @ rs1 @ bool_bits(is_unsigned) @ size_bits(size) @ rd @ 0
        ↪ b0000011
341   if ((word_width_bytes(size) < sizeof(xlen_bytes)) | (not_bool(
        ↪ is_unsigned) & word_width_bytes(size) <= sizeof(xlen_bytes)))
        ↪ & (rs1[4] == bitzero) & (rd[4] == bitzero)
342
343 function aligned(vaddr : xlenbits, width : word_width) -> bool =
344   { width == BYTE | (width == HALF & vaddr[0] == bitzero) | (width ==
        ↪ WORD & vaddr[1 .. 0] == 0b00) }
345
346 val load_leakage : (xlenbits, xlenbits, xlenbits, xlenbits)
347   -> unit effect {rreg, wreg, leakage}
348 function load_leakage(rs1_val : xlenbits, rs2_val : xlenbits, addr:
      ↪ xlenbits, req_data: xlenbits) = {
349   leak(rf_pA, rf_pB, rs1_val, rs2_val);
350   leak(rf_pA, rf_pB, mem_last_addr, mem_last_read);
351   leak(addr, req_data, mem_last_addr);
352   rf_pA = rs1_val;
353   rf_pB = rs2_val;
354   mem_last_read = req_data;
355   mem_last_addr = addr;
356 }
357
358 function clause execute(LOAD(imm, rs1, rd, is_unsigned, width, aq, rl))
      ↪  = {
359   let offset : xlenbits = EXTS(imm);
360   let rs1_val = X(rs1);
361   let rs2_val = X(0b0 @ imm[3 .. 0]);
362   let addr = rs1_val + offset;
363   let req_addr = addr[(sizeof(xlen) - 1) .. 2] @ 0b00;
364   let req_data = read_mem(Read_plain, sizeof(xlen), req_addr, 4);
365   load_leakage(rs1_val, rs2_val, addr, req_data);
366   let req_byte : bits(8) = match (addr[1 .. 0]) {
367     0b00 => req_data[ 7 .. 0],
368     0b01 => req_data[15 .. 8],
369     0b10 => req_data[23 .. 16],
```

```
370      0b11 => req_data[31 .. 24]};
371    let req_half : bits(16) = match (addr[1]) {
372      bitzero => req_data[15 .. 0],
373      bitone => req_data[31 .. 16]};
374    match (width, addr[1 .. 0]) {
375      (BYTE, _) => process_load(rd, addr, req_byte, is_unsigned),
376      (HALF, 0b00) => process_load(rd, addr, req_half, is_unsigned),
377      (HALF, 0b10) => process_load(rd, addr, req_half, is_unsigned),
378      (WORD, 0b00) => process_load(rd, addr, req_data, is_unsigned),
379      (_, _) => RETIRE_FAIL // takes care of misaligned}
380  }
381
382  /* ************************** */
383
384  union clause ast = STORE :
385    (bits(12), regidx, regidx, word_width, bool, bool)
386
387  mapping clause encdec = STORE(imm7 @ imm5, rs2, rs1, size, false, false
         ↪ )
388    if (word_width_bytes(size) <= sizeof(xlen_bytes)) & (rs1[4] ==
         ↪ bitzero) & (rs2[4] == bitzero)
389    <-> imm7 : bits(7) @ rs2 @ rs1 @ 0b0 @ size_bits(size) @ imm5 : bits
         ↪ (5) @ 0b0100011
390    if (word_width_bytes(size) <= sizeof(xlen_bytes)) & (rs1[4] ==
         ↪ bitzero) & (rs2[4] == bitzero)
391
392  function clause execute (STORE(imm, rs2, rs1, width, aq, rl)) = {
393    let offset : xlenbits = EXTS(imm);
394    let rs1_val = X(rs1);
395    let rs2_val = X(rs2);
396    common_leakage(rs1_val, rs2_val);
397    let addr = rs1_val + offset;
398    // address comptation and register file access leakage
399    leak(mem_last_addr, addr);
400    mem_last_addr = addr;
401    if aligned(addr, width) then {
402      let result = rs2_val;
403      overwrite_leakage(0b00000, result);
404      let success : bool = match(width) {
405        BYTE => write_mem(Write_plain, sizeof(xlen), addr, 1, result
             ↪ [7..0]),
406        HALF => write_mem(Write_plain, sizeof(xlen), addr, 2, result
             ↪ [15..0]),
407        WORD => write_mem(Write_plain, sizeof(xlen), addr, 4, result),
```

```
408         _ => false};
409       if success then {RETIRE_SUCCESS} else {RETIRE_FAIL}
410     } else { RETIRE_FAIL }
411   }
412
413   /* ************************** */
414
415   mapping clause encdec = ILLEGAL(s) <-> s
416   function clause execute (ILLEGAL(s)) =
417     { return RETIRE_FAIL }
```

C.3. Ibex Configuration

In the following, we give the configuration file that specifies the mapping and normal operating conditions simultaneously.

Listing C.2: IBEX configuration file

```
1   // Power Contract for IBEX
2   //
3   // Copyright (c) 2020-2022 - TUHH, TU Graz
4   //
5   // All rights reserved.
6   //
7   // This software has received funding from the Federal Ministry of
        ↪ Education and Research (BMBF) as part of the VE-Jupiter project
        ↪ grant 16ME0231K.
8   //
9   // This work was supported by the Austrian Research Promotion Agency (
        ↪ FFG) through the FERMION project (grant number 867542).
10  //
11  // Redistribution and use in source and binary forms,
12  // with or without modification, are permitted provided
13  // that the following conditions are met:
14  // 1. Redistributions of source code must retain the
15  // above copyright notice, this list of conditions
16  // and the following disclaimer.
17  // 2. Redistributions in binary form must reproduce the
18  // above copyright notice, this list of conditions
19  // and the following disclaimer in the documentation
20  // and/or other materials provided with the
21  // distribution.
22  //
23  // THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND
```

```
24 // CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR
25 // IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
26 // TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY
27 // AND FITNESS FOR A PARTICULAR PURPOSE ARE
28 // DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR
29 // CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
30 // INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
31 // DAMAGES (INCLUDING, BUT NOT LIMITED TO,
32 // PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS
33 // OF USE, DATA, OR PROFITS; OR BUSINESS
34 // INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
35 // LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
36 // OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
37 // ARISING IN ANY WAY OUT OF THE USE OF THIS
38 // SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
39 // SUCH DAMAGE.
40
41 //////////////////////////////////////////////////////
42 // This file contains the configuration of our tool.
43 // It specifies
44 // - the state modeled in the contract (registers, memory, leakage
       ↪ state)
45 // - the registers of the IBEX processor (registers and memory)
46 // - a mapping between the states
47 // - which states may contain sensitive data
48 // - conditions which have to hold before/during execution of an
       ↪ instruction
49 // - which HW and contract state is printed in counterexamples
50 //////////////////////////////////////////////////////
51
52 //////////////////////////////////////////////////////
53 // specification of architectural registers and HW/CT mapping
54 //////////////////////////////////////////////////////
55 // PC
56 contract register PC BitVec 32
57 hardware public u_ibex_core.pc_id
58 mapping register PC u_ibex_core.pc_id
59
60 // next PC
61 hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.
       ↪ prefetch_buffer_i.fifo_i.instr_addr_q
62 contract register nextPC BitVec 32
63 mapping register nextPC u_ibex_core.if_stage_i.gen_prefetch_buffer.
       ↪ prefetch_buffer_i.fifo_i.instr_addr_q
```

```
64
65  // REGISTERS
66  contract register x1 BitVec 32
67  contract register x2 BitVec 32
68  contract register x3 BitVec 32
69  contract register x4 BitVec 32
70  contract register x5 BitVec 32
71  contract register x6 BitVec 32
72  contract register x7 BitVec 32
73  contract register x8 BitVec 32
74  contract register x9 BitVec 32
75  contract register x10 BitVec 32
76  contract register x11 BitVec 32
77  contract register x12 BitVec 32
78  contract register x13 BitVec 32
79  contract register x14 BitVec 32
80  contract register x15 BitVec 32
81  hardware variable register_file_i.rf_reg_q[1]
82  hardware variable register_file_i.rf_reg_q[2]
83  hardware variable register_file_i.rf_reg_q[3]
84  hardware variable register_file_i.rf_reg_q[4]
85  hardware variable register_file_i.rf_reg_q[5]
86  hardware variable register_file_i.rf_reg_q[6]
87  hardware variable register_file_i.rf_reg_q[7]
88  hardware variable register_file_i.rf_reg_q[8]
89  hardware variable register_file_i.rf_reg_q[9]
90  hardware variable register_file_i.rf_reg_q[10]
91  hardware variable register_file_i.rf_reg_q[11]
92  hardware variable register_file_i.rf_reg_q[12]
93  hardware variable register_file_i.rf_reg_q[13]
94  hardware variable register_file_i.rf_reg_q[14]
95  hardware variable register_file_i.rf_reg_q[15]
96  mapping register x1 register_file_i.rf_reg_q[1]
97  mapping register x2 register_file_i.rf_reg_q[2]
98  mapping register x3 register_file_i.rf_reg_q[3]
99  mapping register x4 register_file_i.rf_reg_q[4]
100 mapping register x5 register_file_i.rf_reg_q[5]
101 mapping register x6 register_file_i.rf_reg_q[6]
102 mapping register x7 register_file_i.rf_reg_q[7]
103 mapping register x8 register_file_i.rf_reg_q[8]
104 mapping register x9 register_file_i.rf_reg_q[9]
105 mapping register x10 register_file_i.rf_reg_q[10]
106 mapping register x11 register_file_i.rf_reg_q[11]
107 mapping register x12 register_file_i.rf_reg_q[12]
```

```
108 mapping register x13 register_file_i.rf_reg_q[13]
109 mapping register x14 register_file_i.rf_reg_q[14]
110 mapping register x15 register_file_i.rf_reg_q[15]
111
112 contract opcode op BitVec 32
113 // instruction bits for the instruction whose last execution cycle is
       ↪ this cycle
114 // only true if the assertion for the valid_d is present
115 hardware opcode u_ibex_core.instr_rdata_id
116
117 // memory request name_contract name_hardware
118 contract register read_val_1 BitVec 32
119 contract register read_addr_1 BitVec 32
120 hardware variable data_rdata_i
121
122 memory raddr u_ibex_core.load_store_unit_i.adder_result_ex_i
       ↪ read_addr_1
123 memory rdata data_rdata_i read_val_1
124 memory req data_req_o
125 memory gnt data_gnt_i
126 memory ack data_rvalid_i
127 memory we data_we_o
128
129 ////////////////////////////////////////////////////
130 // Non-regport signals that must be constrained
131 ////////////////////////////////////////////////////
132 // fetching next instruction was successful, ready to continue
       ↪ execution
133 hardware const@end-1 u_ibex_core.if_stage_i.fetch_valid 0b1
134 hardware const@pre u_ibex_core.if_stage_i.fetch_valid 0b1
135 // do not load a new instruction until last cycle
136 hardware const@start:end-1 u_ibex_core.id_stage_i.id_in_ready_o 0b0
137 // make sure that nothing retires before the end of the last cycle
138 hardware const@start:end-1 u_ibex_core.instr_id_done 0b0
139 // make sure that an instruction has its last cycle in our last cycle
140 hardware const@end-1 u_ibex_core.instr_id_done 0b1
141 hardware const@pre u_ibex_core.instr_id_done 0b1
142 hardware const@start u_ibex_core.id_stage_i.decoder_i.illegal_insn 0b0
143
144 ////////////////////////////////////////////////////
145 // important signals that must be constrained
146 ////////////////////////////////////////////////////
147 // never trigger a reset of the core
148 hardware public rst_ni
```

```
149 hardware const@pre: rst_ni 0b1
150 // make sure that initially, the ID FSM is in state instr_first_cycle_i
151 // this means that we look at the case where we started executing in 0
        ↪ th cycle
152 hardware public u_ibex_core.id_stage_i.id_fsm_q
153 hardware const@start u_ibex_core.id_stage_i.id_fsm_q 0b0
154 // no compressed (valid or invalid) instructions at the output of
        ↪ instruction fetch stage
155 hardware public u_ibex_core.if_stage_i.instr_new_id_q
156 hardware public u_ibex_core.if_stage_i.instr_is_compressed_id_o
157 hardware const@start u_ibex_core.if_stage_i.instr_is_compressed_id_o 0
        ↪ b0
158 hardware public u_ibex_core.if_stage_i.illegal_c_insn_id_o
159 hardware const@start u_ibex_core.if_stage_i.illegal_c_insn_id_o 0b0
160 // this is a hidden assumption made by IBEX developers
161 hardware public u_ibex_core.if_stage_i.instr_rdata_alu_id_o
162 hardware public u_ibex_core.if_stage_i.instr_rdata_id_o
163 // this encodes pre cycle and first cycle assumptions
164 hardware equiv@pre:start+1 u_ibex_core.if_stage_i.instr_rdata_id_o
        ↪ u_ibex_core.if_stage_i.instr_rdata_alu_id_o
165
166 ////////////////////////////////////////////////////
167 // annotation of input ports of ibex_top
168 ////////////////////////////////////////////////////
169 hardware public clk_i
170 hardware public ram_cfg_i
171 hardware public test_en_i
172 hardware public hart_id_i
173 hardware public boot_addr_i
174 // Instruction memory interface
175 hardware public instr_gnt_i
176 hardware public instr_rvalid_i
177 hardware public instr_rdata_i
178 hardware public instr_err_i
179 hardware public data_gnt_i
180 hardware public data_rvalid_i
181 hardware public data_err_i
182 hardware const@pre: data_err_i 0b0
183 // interrupts
184 hardware public irq_software_i
185 hardware public irq_timer_i
186 hardware public irq_external_i
187 hardware public irq_fast_i
188 hardware public irq_nm_i
```

```
189 // disabling interrupts
190 hardware const@pre: irq_software_i 0b0
191 hardware const@pre: irq_timer_i 0b0
192 hardware const@pre: irq_external_i 0b0
193 hardware const@pre: irq_fast_i 0b000000000000000
194 hardware const@pre: irq_nm_i 0b0
195 // core debug
196 hardware public debug_req_i
197 hardware const@pre: debug_req_i 0b0
198 hardware public fetch_enable_i
199 hardware public scan_rst_ni
200
201 //////////////////////////////////////
202 // annotate internal state of ibex
203 //////////////////////////////////////
204 hardware public core_busy_q
205 hardware public u_ibex_core.instr_fetch_err
206 hardware public u_ibex_core.instr_fetch_err_plus2
207 // instructions in the prefetch fifo
208 hardware public u_ibex_core.if_stage_i.instr_valid_id_q
209 hardware public u_ibex_core.if_stage_i.instr_rdata_c_id_o
210 hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.
      ↪ prefetch_buffer_i.fifo_i.rdata_q0
211 hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.
      ↪ prefetch_buffer_i.fifo_i.rdata_q1
212 hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.
      ↪ prefetch_buffer_i.fifo_i.rdata_q2
213 hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.
      ↪ prefetch_buffer_i.fifo_i.err_q
214 hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.
      ↪ prefetch_buffer_i.fifo_i.valid_q
215 hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.
      ↪ prefetch_buffer_i.rdata_pmp_err_q
216 hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.
      ↪ prefetch_buffer_i.discard_req_q
217 hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.
      ↪ prefetch_buffer_i.branch_discard_q
218 hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.
      ↪ prefetch_buffer_i.rdata_outstanding_q
219 hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.
      ↪ prefetch_buffer_i.fetch_addr_q
220 hardware public u_ibex_core.if_stage_i.gen_prefetch_buffer.
      ↪ prefetch_buffer_i.stored_addr_q
221
```

```
222 // instruction decode
223 hardware public u_ibex_core.id_stage_i.controller_i.ctrl_fsm_cs
224 hardware public u_ibex_core.id_stage_i.controller_i.load_err_q
225 hardware public u_ibex_core.id_stage_i.controller_i.store_err_q
226 hardware public u_ibex_core.id_stage_i.controller_i.exc_req_q
227 hardware public u_ibex_core.id_stage_i.branch_set_raw
228 hardware const@start u_ibex_core.id_stage_i.branch_set_raw 0b0
229 hardware public u_ibex_core.id_stage_i.branch_jump_set_done_q
230 hardware public u_ibex_core.load_store_unit_i.data_we_q
231
232 // since data_pmp_err_i is 0, this should not be 1
233 hardware public u_ibex_core.load_store_unit_i.pmp_err_q
234 hardware const@start u_ibex_core.load_store_unit_i.pmp_err_q 0b0
235 // since data_err_i is never 1 and pmp_err_q is also not 1, this must
      ↪ be 0
236 hardware public u_ibex_core.load_store_unit_i.lsu_err_q
237 hardware const@start u_ibex_core.load_store_unit_i.lsu_err_q 0b0
238 hardware public u_ibex_core.load_store_unit_i.handle_misaligned_q
239 hardware public u_ibex_core.id_stage_i.controller_i.illegal_insn_q
240 hardware public u_ibex_core.id_stage_i.controller_i.do_single_step_q
241 hardware public u_ibex_core.id_stage_i.controller_i.
      ↪ enter_debug_mode_prio_q
242 // LSU register handling misaligned memory accesses which are not
      ↪ allowed by the contract
243 hardware public u_ibex_core.load_store_unit_i.rdata_q
244 contract leakagestate mem_last_read BitVec 32
245 // Must be idle when new instruction reaches ID/EX
246 hardware public u_ibex_core.load_store_unit_i.ls_fsm_cs
247 hardware const@start u_ibex_core.load_store_unit_i.ls_fsm_cs 0b000
248 hardware variable u_ibex_core.load_store_unit_i.addr_last_q
249 hardware variable u_ibex_core.load_store_unit_i.rdata_offset_q
250 contract leakagestate mem_last_addr BitVec 32
251 mapping leakagestate mem_last_addr u_ibex_core.load_store_unit_i.
      ↪ addr_last_q
252 mapping leakagestate mem_last_addr u_ibex_core.load_store_unit_i.
      ↪ rdata_offset_q
253 hardware public u_ibex_core.load_store_unit_i.data_type_q
254 hardware public u_ibex_core.load_store_unit_i.data_sign_ext_q
255
256 // system registers
257 hardware public u_ibex_core.cs_registers_i.mie_q
258 hardware public u_ibex_core.cs_registers_i.mtval_q
259 hardware public u_ibex_core.cs_registers_i.mcause_q
260 hardware public u_ibex_core.cs_registers_i.mscratch_q
```

```
261 hardware public u_ibex_core.cs_registers_i.dscratch0_q
262 hardware public u_ibex_core.cs_registers_i.dscratch1_q
263 hardware public u_ibex_core.cs_registers_i.mstack_q
264 hardware public u_ibex_core.cs_registers_i.mstack_cause_q
265 hardware public u_ibex_core.cs_registers_i.mstack_epc_q
266 hardware public u_ibex_core.cs_registers_i.mstatus_q
267 hardware public u_ibex_core.cs_registers_i.dcsr_q
268 hardware const@start u_ibex_core.cs_registers_i.dcsr_q 0
        ↪ b00000000000000000000000000000000
269 hardware public u_ibex_core.cs_registers_i.mhpmcounter[0]
270 hardware public u_ibex_core.cs_registers_i.mhpmcounter[1]
271 hardware public u_ibex_core.cs_registers_i.mhpmcounter[2]
272 hardware public u_ibex_core.cs_registers_i.mcountinhibit
273 hardware public u_ibex_core.csr_depc
274 hardware public u_ibex_core.csr_mtvec
275 hardware public u_ibex_core.csr_mepc
276 hardware public u_ibex_core.dummy_instr_en
277 hardware public u_ibex_core.dummy_instr_mask
278 hardware public u_ibex_core.data_ind_timing
279 hardware public u_ibex_core.icache_enable
280 hardware public u_ibex_core.debug_mode
281 hardware public u_ibex_core.priv_mode_id
282 hardware public u_ibex_core.nmi_mode
283
284 contract leakagestate rf_pA BitVec 32
285 contract leakagestate rf_pB BitVec 32
```

Bibliography

[AA04] Dmitri Asonov and Rakesh Agrawal. “Keyboard Acoustic Emanations”. In: *2004 IEEE Symposium on Security and Privacy*. Berkeley, CA, USA: IEEE Computer Society Press, May 2004, pp. 3–11. DOI: 10.1109/SECPRI.2004.1301311.

[Abr+21] Arnold Abromeit, Florian Bache, Leon A. Becker, Marc Gourjon, Tim Güneysu, Sabrina Jorn, Amir Moradi, Maximilian Orlt, and Falk Schellenberg. “Automated Masking of Software Implementations on Industrial Microcontrollers”. In: *Design, Automation & Test in Europe Conference & Exhibition, DATE 2021, Grenoble, France, February 1-5, 2021*. IEEE, 2021, pp. 1006–1011. DOI: 10.23919/DATE51398.2021.9474183. URL: https://doi.org/10.23919/DATE51398.2021.9474183.

[Agr+03] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. “The EM Side-Channel(s)”. In: *Cryptographic Hardware and Embedded Systems – CHES 2002*. Ed. by Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar. Vol. 2523. Lecture Notes in Computer Science. Redwood Shores, CA, USA: Springer, Berlin, Heidelberg, Germany, Aug. 2003, pp. 29–45. DOI: 10.1007/3-540-36400-5_4.

[Ala+20] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. *Status report on the second round of the NIST post-quantum cryptography standardization process*. Tech. rep. NISTIR 8309. https://doi.org/10.6028/NIST.IR.8309. National Institute of Standards and Technology, 2020.

[Alk+16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. “Post-quantum key exchange – a new hope”. In: *Proceedings of the 25th USENIX Security Symposium*. USENIX Association, 2016.

[Ami+20] Dorian Amiet, Andreas Curiger, Lukas Leuenberger, and Paul Zbinden. "Defeating NewHope with a Single Trace". In: *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020*. Ed. by Jintai Ding and Jean-Pierre Tillich. Paris, France: Springer, Cham, Switzerland, Apr. 2020, pp. 189–205. DOI: 10.1007/978-3-030-44223-1_11.

[Arm+18] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. *MiniSail: A core calculus for Sail*. https://www.cl.cam.ac.uk/~mpew2/papers/minisail_anf.pdf (accessed January 12, 2022). 2018.

[Arm+19] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. "ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 71:1–71:31. DOI: 10.1145/3290384. URL: https://doi.org/10.1145/3290384.

[Arm+21] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Kathryn E. Gray, Robert Norton-Wright, Christopher Pulte, Shaked Flur, and Peter Sewell. *The Sail instruction-set semantics specification language*. https://raw.githubusercontent.com/rems-project/sail/sail2/manual.pdf (accessed January 12, 2022). 2021.

[ARM18] ARM Limited. *ARM v6-M Architecture Reference Manual*. Tech. rep. ARM DDI 0419E (ID070218). ARM Limited, 2018.

[Azo+20] Melissa Azouaoui, Davide Bellizia, Ileana Buhan, Nicolas Debande, Sébastien Duval, Christophe Giraud, Éliane Jaulmes, François Koeune, Elisabeth Oswald, François-Xavier Standaert, and Carolyn Whitnall. "A Systematic Appraisal of Side Channel Evaluation Strategies". In: *Security Standardisation Research - 6th International Conference, SSR 2020, London, UK, November 30 - December 1, 2020, Proceedings*. Ed. by Thyla van der Merwe, Chris J. Mitchell, and Maryam Mehrnezhad. Vol. 12529. Lecture Notes in Computer Science. Springer, 2020, pp. 46–66. DOI: 10.1007/978-3-030-64357-7_3. URL: https://doi.org/10.1007/978-3-030-64357-7%5C_3.

[Azo+22a] Melissa Azouaoui, Joppe W. Bos, Björn Fay, Marc Gourjon, Yulia Kuzovkova, Joost Renes, Tobias Schneider, and Christine van Vredendaal. "Surviving the FO-Calypse: Securing PQC Implementations in Practice". In: *RWC 2022: Real World Crypto Symposium*. `https://iacr.org/submit/files/slides/2022/rwc/rwc2022/48/slides.pdf` (Acessed November 11, 2023). Amsterdam, Netherlands, Apr. 2022.

[Azo+22b] Melissa Azouaoui, Olivier Bronchain, Vincent Grosso, Kostas Papagiannopoulos, and François-Xavier Standaert. "Bitslice Masking and Improved Shuffling: How and When to Mix Them in Software?" In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2022.2 (2022), pp. 140–165. DOI: `10.46586/tches.v2022.i2.140-165`.

[Azo+22c] Melissa Azouaoui, Olivier Bronchain, Clément Hoffmann, Yulia Kuzovkova, Tobias Schneider, and François-Xavier Standaert. "Systematic Study of Decryption and Re-encryption Leakage: The Case of Kyber". In: *Constructive Side-Channel Analysis and Secure Design - 13th International Workshop, COSADE 2022, Leuven, Belgium, April 11-12, 2022, Proceedings*. Ed. by Josep Balasch and Colin O'Flynn. Vol. 13211. Lecture Notes in Computer Science. Springer, 2022, pp. 236–256. DOI: `10.1007/978-3-030-99766-3_11`. URL: `https://doi.org/10.1007/978-3-030-99766-3%5C_11`.

[Bac+20] Florian Bache, Clara Paglialonga, Tobias Oder, Tobias Schneider, and Tim Güneysu. "High-Speed Masking for Polynomial Comparison in Lattice-based KEMs". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020.3 (2020), pp. 483–507. ISSN: 2569-2925. DOI: `10.13154/tches.v2020.i3.483-507`. URL: `https://tches.iacr.org/index.php/TCHES/article/view/8598`.

[Bal+14] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. "On the Cost of Lazy Engineering for Masked Software Implementations". In: *CARDIS*. Vol. 8968. LNCS. Springer, 2014, pp. 64–81.

[Bar+15] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. "Verified Proofs of Higher-Order Masking". In: *Advances in Cryptology – EUROCRYPT 2015, Part I*. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9056. Lecture Notes in Computer Science. Sofia,

Bulgaria: Springer, Berlin, Heidelberg, Germany, Apr. 2015, pp. 457–485. DOI: 10.1007/978-3-662-46800-5_18.

[Bar+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. "Strong Non-Interference and Type-Directed Higher-Order Masking". In: *ACM CCS 2016: 23rd Conference on Computer and Communications Security*. Ed. by Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi. Vienna, Austria: ACM Press, Oct. 2016, pp. 116–129. DOI: 10.1145/2976749.2978427.

[Bar+17] Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. "Parallel Implementations of Masking Schemes and the Bounded Moment Leakage Model". In: *Advances in Cryptology – EUROCRYPT 2017, Part I*. Ed. by Jean-Sébastien Coron and Jesper Buus Nielsen. Vol. 10210. Lecture Notes in Computer Science. Paris, France: Springer, Cham, Switzerland, Apr. 2017, pp. 535–566. DOI: 10.1007/978-3-319-56620-7_19.

[Bar+18] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Benjamin Grégoire, Mélissa Rossi, and Mehdi Tibouchi. "Masking the GLP Lattice-Based Signature Scheme at Any Order". In: *Advances in Cryptology – EUROCRYPT 2018, Part II*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10821. Lecture Notes in Computer Science. Tel Aviv, Israel: Springer, Cham, Switzerland, Apr. 2018, pp. 354–384. DOI: 10.1007/978-3-319-78375-8_12.

[Bar+19] 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: *ESORICS 2019: 24th European Symposium on Research in Computer Security, Part I*. Ed. by Kazue Sako, Steve Schneider, and Peter Y. A. Ryan. Vol. 11735. Lecture Notes in Computer Science. Luxembourg: Springer, Cham, Switzerland, Sept. 2019, pp. 300–318. DOI: 10.1007/978-3-030-29959-0_15.

[Bar+20] Gilles Barthe, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, and Lars Porth. *Open Source Publication of Complete Leakage Model, Implementations and the Veri-*

fication Tool. GitHub. https://github.com/scverif/scverif, https://github.com/scverif/gadgets. 2020.

[Bar+21a] Gilles Barthe, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, and Lars Porth. "Artifact of Masking in Fine-Grained Leakage Models: Construction, Implementation and Verification". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.2 (2021). Artifact available at https://artifacts.iacr.org/tches/2021/a8.

[Bar+21b] Gilles Barthe, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, and Lars Porth. "Masking in Fine-Grained Leakage Models: Construction, Implementation and Verification". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.2 (2021), pp. 189–228. ISSN: 2569-2925. DOI: 10.46586/tches.v2021.i2.189-228. URL: https://tches.iacr.org/index.php/TCHES/article/view/8792.

[Bar+22] R. Barnes, K. Bhargavan, B. Lipp, and C. Wood. *Hybrid Public Key Encryption*. Request for Comments 9180. Stream: RFC, Category: Informational, Published: February 2022, ISSN: 2070-1721. Internet Engineering Task Force (IETF), Feb. 2022. URL: %7Bhttps://www.rfc-editor.org/rfc/rfc9180.txt%7D.

[Bat+16] Alberto Battistello, Jean-Sébastien Coron, Emmanuel Prouff, and Rina Zeitoun. "Horizontal Side-Channel Attacks and Countermeasures on the ISW Masking Scheme". In: *Cryptographic Hardware and Embedded Systems – CHES 2016*. Ed. by Benedikt Gierlichs and Axel Y. Poschmann. Vol. 9813. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Berlin, Heidelberg, Germany, Aug. 2016, pp. 23–39. DOI: 10.1007/978-3-662-53140-2_2.

[Bay+13] Ali Galip Bayrak, Francesco Regazzoni, David Novo, and Paolo Ienne. "Sleuth: Automated Verification of Software Power Analysis Countermeasures". In: *Cryptographic Hardware and Embedded Systems – CHES 2013*. Ed. by Guido Bertoni and Jean-Sébastien Coron. Vol. 8086. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Berlin, Heidelberg, Germany, Aug. 2013, pp. 293–310. DOI: 10.1007/978-3-642-40349-1_17.

[Baz+21] Omid Bazangani, Alexandre Iooss, Ileana Buhan, and Lejla Batina. *ABBY: Automating the creation of fine-grained leakage models*. Cryptology ePrint Archive, Report 2021/1569. 2021. URL: https://eprint.iacr.org/2021/1569.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. “Correlation Power Analysis with a Leakage Model”. In: *Cryptographic Hardware and Embedded Systems – CHES 2004*. Ed. by Marc Joye and Jean-Jacques Quisquater. Vol. 3156. Lecture Notes in Computer Science. Cambridge, Massachusetts, USA: Springer, Berlin, Heidelberg, Germany, Aug. 2004, pp. 16–29. DOI: 10.1007/978-3-540-28632-5_2.

[BCS21] Olivier Bronchain, Gaëtan Cassiers, and François-Xavier Standaert. *Give Me 5 Minutes: Attacking ASCAD with a Single Side-Channel Trace*. Cryptology ePrint Archive, Report 2021/817. 2021. URL: https://eprint.iacr.org/2021/817.

[BCZ18] Luk Bettale, Jean-Sébastien Coron, and Rina Zeitoun. “Improved High-Order Conversion From Boolean to Arithmetic Masking”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.2 (2018), pp. 22–45. ISSN: 2569-2925. DOI: 10.13154/tches.v2018.i2.22-45. URL: https://tches.iacr.org/index.php/TCHES/article/view/873.

[Bec+22] Arthur Beckers, Lennert Wouters, Benedikt Gierlichs, Bart Preneel, and Ingrid Verbauwhede. “Provable Secure Software Masking in the Real-World”. In: *Constructive Side-Channel Analysis and Secure Design - 13th International Workshop, COSADE 2022, Leuven, Belgium, April 11-12, 2022, Proceedings*. Ed. by Josep Balasch and Colin O’Flynn. Vol. 13211. Lecture Notes in Computer Science. Springer, 2022, pp. 215–235. DOI: 10.1007/978-3-030-99766-3_10. URL: https://doi.org/10.1007/978-3-030-99766-3%5C_10.

[Bei+20] Michiel Van Beirendonck, Jan-Pieter D’Anvers, Angshuman Karmakar, Josep Balasch, and Ingrid Verbauwhede. *A Side-Channel Resistant Implementation of SABER*. Cryptology ePrint Archive, Report 2020/733. 2020. URL: https://eprint.iacr.org/2020/733.

[Bel+16] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. “Randomness Complexity of Private Circuits for Multiplication”. In: *Advances in Cryptology – EUROCRYPT 2016, Part II*. Ed. by Marc Fischlin and Jean-Sébastien Coron. Vol. 9666. Lecture Notes in Computer Science. Vienna, Austria: Springer, Berlin, Heidelberg, Germany, May 2016, pp. 616–648. DOI: 10.1007/978-3-662-49896-5_22.

[Bel+20a] Sonia Belaïd, Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Abdul Rahman Taleb. “Random Probing Security: Verification, Composition, Expansion and New Constructions”. In: *Advances in Cryptology – CRYPTO 2020, Part I*. Ed. by Daniele Micciancio and Thomas Ristenpart. Vol. 12170. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Cham, Switzerland, Aug. 2020, pp. 339–368. DOI: 10.1007/978-3-030-56784-2_12.

[Bel+20b] Sonia Belaïd, Pierre-Évariste Dagand, Darius Mercadier, Matthieu Rivain, and Raphaël Wintersdorff. “Tornado: Automatic Generation of Probing-Secure Masked Bitsliced Implementations”. In: *Advances in Cryptology – EUROCRYPT 2020, Part III*. Ed. by Anne Canteaut and Yuval Ishai. Vol. 12107. Lecture Notes in Computer Science. Zagreb, Croatia: Springer, Cham, Switzerland, May 2020, pp. 311–341. DOI: 10.1007/978-3-030-45727-3_11.

[Bel+22] Sonia Belaïd, Darius Mercadier, Matthieu Rivain, and Abdul Rahman Taleb. “IronMask: Versatile Verification of Masking Security”. In: *2022 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2022, pp. 142–160. DOI: 10.1109/SP46214.2022.9833600.

[Bel+23a] Sonia Belaïd, Gaëtan Cassiers, Matthieu Rivain, and Abdul Rahman Taleb. “Unifying Freedom and Separation for Tight Probing-Secure Composition”. In: *Advances in Cryptology – CRYPTO 2023, Part III*. Ed. by Helena Handschuh and Anna Lysyanskaya. Vol. 14083. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Cham, Switzerland, Aug. 2023, pp. 440–472. DOI: 10.1007/978-3-031-38548-3_15.

[Bel+23b] Sonia Belaïd, Gaëtan Cassiers, Camille Mutschler, Matthieu Rivain, Thomas Roche, François-Xavier Standaert, and Abdul Rahman Taleb. “Towards Achieving Provable Side-Channel Security in Practice”. In: *IACR Cryptol. ePrint Arch.* (2023), p. 1198. URL: https://eprint.iacr.org/2023/1198.

[Ber+10] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. “Building power analysis resistant implementations of Keccak”. In: *Second SHA-3 candidate conference*. Vol. 142. Citeseer. 2010.

[Ber+23] Sebastian Berndt, Thomas Eisenbarth, Sebastian Faust, Marc Gourjon, Maximilian Orlt, and Okan Seker. “Combined Fault and Leakage Resilience: Composability, Constructions and Compiler”. In: *Advances in Cryptology – CRYPTO 2023, Part III.* Ed. by Helena Handschuh and Anna Lysyanskaya. Vol. 14083. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Cham, Switzerland, Aug. 2023, pp. 377–409. DOI: 10.1007/978-3-031-38548-3_13.

[BGR18] Sonia Belaïd, Dahmun Goudarzi, and Matthieu Rivain. “Tight Private Circuits: Achieving Probing Security with the Least Refreshing”. In: *Advances in Cryptology – ASIACRYPT 2018, Part II.* Ed. by Thomas Peyrin and Steven Galbraith. Vol. 11273. Lecture Notes in Computer Science. Brisbane, Queensland, Australia: Springer, Cham, Switzerland, Dec. 2018, pp. 343–372. DOI: 10.1007/978-3-030-03329-3_12.

[Bha+21] Shivam Bhasin, Jan-Pieter D’Anvers, Daniel Heinz, Thomas Pöppelmann, and Michiel Van Beirendonck. *Attacking and Defending Masked Polynomial Comparison for Lattice-Based Cryptography.* Cryptology ePrint Archive, Report 2021/104. https://eprint.iacr.org/2021/104. 2021.

[Bla79] G. R. Blakley. “Safeguarding Cryptographic Keys”. In: *Proceedings of AFIPS 1979 National Computer Conference* 48 (1979), pp. 313–317.

[Blo+18] Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. “Formal Verification of Masked Hardware Implementations in the Presence of Glitches”. In: *Advances in Cryptology – EUROCRYPT 2018, Part II.* Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10821. Lecture Notes in Computer Science. Tel Aviv, Israel: Springer, Cham, Switzerland, Apr. 2018, pp. 321–353. DOI: 10.1007/978-3-319-78375-8_11.

[Blo+22] Roderick Bloem, Barbara Gigerl, Marc Gourjon, Vedad Hadzic, Stefan Mangard, and Robert Primas. “Power Contracts: Provably Complete Power Leakage Models for Processors”. In: *ACM CCS 2022: 29th Conference on Computer and Communications Security.* Ed. by Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi. Los Angeles, CA, USA: ACM Press, Nov. 2022, pp. 381–395. DOI: 10.1145/3548606.3560600.

[BM16] Guido Bertoni and Marco Martinoli. *A Methodology for the Characterisation of Leakages in Combinatorial Logic.* Cryptology ePrint Archive, Report 2016/841. 2016. URL: https://eprint.iacr.org/2016/841.

[Bos+15] Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. "Post-Quantum Key Exchange for the TLS Protocol from the Ring Learning with Errors Problem". In: *2015 IEEE Symposium on Security and Privacy – SP*. IEEE Computer Society, 2015, pp. 553–570. DOI: 10.1109/SP.2015.40.

[Bos+18] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. "CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM". In: *2018 IEEE European Symposium on Security and Privacy – Euro S&P*. IEEE, 2018, pp. 353–367. DOI: 10.1109/EuroSP.2018.00032.

[Bos+21] Joppe W. Bos, Marc Gourjon, Joost Renes, Tobias Schneider, and Christine van Vredendaal. "Masking Kyber: First- and Higher-Order Implementations". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.4 (2021), pp. 173–214. ISSN: 2569-2925. DOI: 10.46586/tches.v2021.i4.173-214. URL: https://tches.iacr.org/index.php/TCHES/article/view/9064.

[Bro+19] Olivier Bronchain, Julien M. Hendrickx, Clément Massart, Alex Olshevsky, and François-Xavier Standaert. "Leakage Certification Revisited: Bounding Model Errors in Side-Channel Security Evaluations". In: *Advances in Cryptology – CRYPTO 2019, Part I*. Ed. by Alexandra Boldyreva and Daniele Micciancio. Vol. 11692. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Cham, Switzerland, Aug. 2019, pp. 713–737. DOI: 10.1007/978-3-030-26948-7_25.

[Cam+18] Giovanni Camurati, Sebastian Poeplau, Marius Muench, Tom Hayes, and Aurélien Francillon. "Screaming Channels: When Electromagnetic Side Channels Meet Radio Transceivers". In: *ACM CCS 2018: 25th Conference on Computer and Communications Security*. Ed. by David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang. Toronto, ON, Canada: ACM Press, Oct. 2018, pp. 163–177. DOI: 10.1145/3243734.3243802.

[Cas+21a] Gaëtan Cassiers, Sebastian Faust, Maximilian Orlt, and François-Xavier Standaert. “Towards Tight Random Probing Security”. In: *Advances in Cryptology – CRYPTO 2021, Part III*. Ed. by Tal Malkin and Chris Peikert. Vol. 12827. Lecture Notes in Computer Science. Virtual Event: Springer, Cham, Switzerland, Aug. 2021, pp. 185–214. DOI: 10.1007/978-3-030-84252-9_7.

[Cas+21b] Gaëtan Cassiers, Benjamin Grégoire, Itamar Levi, and François-Xavier Standaert. “Hardware Private Circuits: From Trivial Composition to Full Verification”. In: *IEEE Trans. Computers* 70.10 (2021), pp. 1677–1690. DOI: 10.1109/TC.2020.3022979. URL: https://doi.org/10.1109/TC.2020.3022979.

[CFE16] Cong Chen, Mohammad Farmani, and Thomas Eisenbarth. “A Tale of Two Shares: Why Two-Share Threshold Implementation Seems Worthwhile - and Why It Is Not”. In: *Advances in Cryptology – ASIACRYPT 2016, Part I*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Vol. 10031. Lecture Notes in Computer Science. Hanoi, Vietnam: Springer, Berlin, Heidelberg, Germany, Dec. 2016, pp. 819–843. DOI: 10.1007/978-3-662-53887-6_30.

[CGD18] Yann Le Corre, Johann Großschädl, and Daniel Dinu. “Micro-architectural Power Simulator for Leakage Assessment of Cryptographic Software on ARM Cortex-M3 Processors”. In: *COSADE 2018: 9th International Workshop on Constructive Side-Channel Analysis and Secure Design*. Ed. by Junfeng Fan and Benedikt Gierlichs. Vol. 10815. Lecture Notes in Computer Science. Singapore: Springer, Cham, Switzerland, Apr. 2018, pp. 82–98. DOI: 10.1007/978-3-319-89641-0_5.

[CGV14] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. “Secure Conversion between Boolean and Arithmetic Masking of Any Order”. In: *Cryptographic Hardware and Embedded Systems – CHES 2014*. Ed. by Lejla Batina and Matthew Robshaw. Vol. 8731. Lecture Notes in Computer Science. Busan, South Korea: Springer, Berlin, Heidelberg, Germany, Sept. 2014, pp. 188–205. DOI: 10.1007/978-3-662-44709-3_11.

[Cha+99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. “Towards Sound Approaches to Counteract Power-Analysis Attacks”. In: *Advances in Cryptology – CRYPTO'99*. Ed. by Michael J. Wiener. Vol. 1666. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Berlin, Heidelberg,

Germany, Aug. 1999, pp. 398–412. DOI: 10.1007/3-540-48405-1_26.

[Cor+12] Jean-Sébastien Coron, Christophe Giraud, Emmanuel Prouff, Soline Renner, Matthieu Rivain, and Praveen Kumar Vadnala. "Conversion of Security Proofs from One Leakage Model to Another: A New Issue". In: *COSADE 2012: 3rd International Workshop on Constructive Side-Channel Analysis and Secure Design*. Ed. by Werner Schindler and Sorin A. Huss. Vol. 7275. Lecture Notes in Computer Science. Darmstadt, Germany: Springer, Berlin, Heidelberg, Germany, May 2012, pp. 69–81. DOI: 10.1007/978-3-642-29912-4_6.

[Cor+14] Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Thomas Roche. "Higher-Order Side Channel Security and Mask Refreshing". In: *Fast Software Encryption – FSE 2013*. Ed. by Shiho Moriai. Vol. 8424. Lecture Notes in Computer Science. Singapore: Springer, Berlin, Heidelberg, Germany, Mar. 2014, pp. 410–424. DOI: 10.1007/978-3-662-43933-3_21.

[Cor+21] Jean-Sébastien Coron, François Gérard, Simon Montoya, and Rina Zeitoun. "High-order Table-based Conversion Algorithms and Masking Lattice-based Encryption". In: *IACR Cryptol. ePrint Arch.* (2021), p. 1314. URL: https://eprint.iacr.org/2021/1314.

[Cor+23] Jean-Sébastien Coron, François Gérard, Simon Montoya, and Rina Zeitoun. "High-order Polynomial Comparison and Masking Lattice-based Encryption". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2023.1 (2023), pp. 153–192. DOI: 10.46586/tches.v2023.i1.153-192.

[Cor14] Jean-Sébastien Coron. "Higher Order Masking of Look-Up Tables". In: *Advances in Cryptology – EUROCRYPT 2014*. Ed. by Phong Q. Nguyen and Elisabeth Oswald. Vol. 8441. Lecture Notes in Computer Science. Copenhagen, Denmark: Springer, Berlin, Heidelberg, Germany, May 2014, pp. 441–458. DOI: 10.1007/978-3-642-55220-5_25.

[Cor17] Jean-Sébastien Coron. "High-Order Conversion from Boolean to Arithmetic Masking". In: *Cryptographic Hardware and Embedded Systems – CHES 2017*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. Lecture Notes in Computer Science. Taipei, Taiwan: Springer, Cham, Switzerland, Sept. 2017, pp. 93–114. DOI: 10.1007/978-3-319-66787-4_5.

[Cor18] Jean-Sébastien Coron. "Formal Verification of Side-Channel Countermeasures via Elementary Circuit Transformations". In: *ACNS 18: 16th International Conference on Applied Cryptography and Network Security*. Ed. by Bart Preneel and Frederik Vercauteren. Vol. 10892. Lecture Notes in Computer Science. Leuven, Belgium: Springer, Cham, Switzerland, July 2018, pp. 65–82. DOI: 10.1007/978-3-319-93387-0_4.

[CRR03] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. "Template Attacks". In: *Cryptographic Hardware and Embedded Systems – CHES 2002*. Ed. by Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar. Vol. 2523. Lecture Notes in Computer Science. Redwood Shores, CA, USA: Springer, Berlin, Heidelberg, Germany, Aug. 2003, pp. 13–28. DOI: 10.1007/3-540-36400-5_3.

[CRZ18] Jean-Sébastien Coron, Franck Rondepierre, and Rina Zeitoun. "High Order Masking of Look-up Tables with Common Shares". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.1 (2018), pp. 40–72. ISSN: 2569-2925. DOI: 10.13154/tches.v2018.i1.40-72. URL: https://tches.iacr.org/index.php/TCHES/article/view/832.

[CS18] Gaëtan Cassiers and François-Xavier Standaert. *Improved Bitslice Masking: from Optimized Non-Interference to Probe Isolation*. Cryptology ePrint Archive, Report 2018/438. 2018. URL: https://eprint.iacr.org/2018/438.

[CS19] Gaëtan Cassiers and François-Xavier Standaert. "Towards Globally Optimized Masking: From Low Randomness to Low Noise Rate". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2019.2 (2019), pp. 162–198. ISSN: 2569-2925. DOI: 10.13154/tches.v2019.i2.162-198. URL: https://tches.iacr.org/index.php/TCHES/article/view/7389.

[CS20] Gaëtan Cassiers and François-Xavier Standaert. "Trivially and Efficiently Composing Masked Gadgets With Probe Isolating Non-Interference". In: *IEEE Trans. Inf. Forensics Secur.* 15 (2020), pp. 2542–2555. DOI: 10.1109/TIFS.2020.2971153. URL: https://doi.org/10.1109/TIFS.2020.2971153.

[CS21] Gaëtan Cassiers and François-Xavier Standaert. "Provably Secure Hardware Masking in the Transition- and Glitch-Robust Probing Model: Better Safe than Sorry". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.2 (2021), pp. 136–158. ISSN: 2569-2925. DOI: 10.46586/tches.v2021.i2.

136–158. URL: https://tches.iacr.org/index.php/TCHES/article/view/8790.

[CT03] Jean-Sébastien Coron and Alexei Tchulkine. "A New Algorithm for Switching from Arithmetic to Boolean Masking". In: *Cryptographic Hardware and Embedded Systems – CHES 2003*. Ed. by Colin D. Walter, Çetin Kaya Koç, and Christof Paar. Vol. 2779. Lecture Notes in Computer Science. Cologne, Germany: Springer, Berlin, Heidelberg, Germany, Sept. 2003, pp. 89–97. DOI: 10.1007/978-3-540-45238-6_8.

[DAn+19] Jan-Pieter D'Anvers, Marcel Tiepelt, Frederik Vercauteren, and Ingrid Verbauwhede. "Timing Attacks on Error Correcting Codes in Post-Quantum Schemes". In: *Proceedings of ACM Workshop on Theory of Implementation Security Workshop, TIS@CCS 2019, London, UK, November 11, 2019*. Ed. by Begül Bilgin, Svetla Petkova-Nikova, and Vincent Rijmen. ACM, 2019, pp. 2–9. DOI: 10.1145/3338467.3358948. URL: https://doi.org/10.1145/3338467.3358948.

[DAn+20] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, Frederik Vercauteren, Jose Maria Bermudo Mera, Michiel Van Beirendonck, and Andrea Basso. *SABER*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions. National Institute of Standards and Technology, 2020.

[Dan21] Daniel Heinz and Matthias J. Kannwischer and Georg Land and Thomas Pöppelmann and Peter Schwabe and Daan Sprenkels. *First-Order Masked Kyber on ARM Cortex-M4*. https://csrc.nist.gov/CSRC/media/Presentations/first-order-masked-kyber-on-arm-cortex-m4/images-media/session-4-heinz-first-order-masked-kyber.pdf. 2021.

[DBR19] Lauren De Meyer, Begül Bilgin, and Oscar Reparaz. "Consolidating Security Notions in Hardware Masking". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2019.3 (2019), pp. 119–147. ISSN: 2569-2925. DOI: 10.13154/tches.v2019.i3.119-147. URL: https://tches.iacr.org/index.php/TCHES/article/view/8291.

[DDF19] Alexandre Duc, Stefan Dziembowski, and Sebastian Faust. "Unifying Leakage Models: From Probing Attacks to Noisy Leakage".

In: *Journal of Cryptology* 32.1 (Jan. 2019), pp. 151–177. DOI: 10.1007/s00145-018-9284-1.

[DDT20] Lauren De Meyer, Elke De Mulder, and Michael Tunstall. *On the Effect of the (Micro)Architecture on the Development of Side-Channel Resistant Software*. Cryptology ePrint Archive, Report 2020/1297. 2020. URL: https://eprint.iacr.org/2020/1297.

[De +17] Thomas De Cnudde, Begül Bilgin, Benedikt Gierlichs, Ventzislav Nikov, Svetla Nikova, and Vincent Rijmen. "Does Coupling Affect the Security of Masked Implementations?" In: *COSADE 2017: 8th International Workshop on Constructive Side-Channel Analysis and Secure Design*. Ed. by Sylvain Guilley. Vol. 10348. Lecture Notes in Computer Science. Paris, France: Springer, Cham, Switzerland, Apr. 2017, pp. 1–18. DOI: 10.1007/978-3-319-64647-3_1.

[Deb12] Blandine Debraize. "Efficient and Provably Secure Methods for Switching from Arithmetic to Boolean Masking". In: *Cryptographic Hardware and Embedded Systems – CHES 2012*. Ed. by Emmanuel Prouff and Patrick Schaumont. Vol. 7428. Lecture Notes in Computer Science. Leuven, Belgium: Springer, Berlin, Heidelberg, Germany, Sept. 2012, pp. 107–121. DOI: 10.1007/978-3-642-33027-8_7.

[DFS15] Alexandre Duc, Sebastian Faust, and François-Xavier Standaert. "Making Masking Security Proofs Concrete - Or How to Evaluate the Security of Any Leaking Device". In: *Advances in Cryptology – EUROCRYPT 2015, Part I*. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9056. Lecture Notes in Computer Science. Sofia, Bulgaria: Springer, Berlin, Heidelberg, Germany, Apr. 2015, pp. 401–429. DOI: 10.1007/978-3-662-46800-5_16.

[Din+17] A. Adam Ding, Liwei Zhang, François Durvaux, François-Xavier Standaert, and Yunsi Fei. "Towards Sound and Optimal Leakage Detection Procedure". In: *Smart Card Research and Advanced Applications - 16th International Conference, CARDIS 2017, Lugano, Switzerland, November 13-15, 2017, Revised Selected Papers*. Ed. by Thomas Eisenbarth and Yannick Teglia. Vol. 10728. Lecture Notes in Computer Science. Springer, 2017, pp. 105–122. DOI: 10.1007/978-3-319-75208-2_7. URL: https://doi.org/10.1007/978-3-319-75208-2%5C_7.

[DP08] Stefan Dziembowski and Krzysztof Pietrzak. “Leakage-Resilient Cryptography”. In: *49th Annual Symposium on Foundations of Computer Science*. Philadelphia, PA, USA: IEEE Computer Society Press, Oct. 2008, pp. 293–302. DOI: 10.1109/FOCS.2008.56.

[DSM17] François Durvaux, François-Xavier Standaert, and Santos Merino Del Pozo. “Towards easy leakage certification: extended version”. In: *Journal of Cryptographic Engineering* 7.2 (June 2017), pp. 129–147. DOI: 10.1007/s13389-017-0150-0.

[DSP16] François Durvaux, François-Xavier Standaert, and Santos Merino Del Pozo. “Towards Easy Leakage Certification”. In: *Cryptographic Hardware and Embedded Systems – CHES 2016*. Ed. by Benedikt Gierlichs and Axel Y. Poschmann. Vol. 9813. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Berlin, Heidelberg, Germany, Aug. 2016, pp. 40–60. DOI: 10.1007/978-3-662-53140-2_3.

[DSV14] François Durvaux, François-Xavier Standaert, and Nicolas Veyrat-Charvillon. “How to Certify the Leakage of a Chip?” In: *Advances in Cryptology – EUROCRYPT 2014*. Ed. by Phong Q. Nguyen and Elisabeth Oswald. Vol. 8441. Lecture Notes in Computer Science. Copenhagen, Denmark: Springer, Berlin, Heidelberg, Germany, May 2014, pp. 459–476. DOI: 10.1007/978-3-642-55220-5_26.

[EWS14] Hassan Eldib, Chao Wang, and Patrick Schaumont. “Formal Verification of Software Countermeasures against Side-Channel Attacks”. In: *ACM Trans. Softw. Eng. Methodol.* 24.2 (2014), 11:1–11:24. DOI: 10.1145/2685616. URL: https://doi.org/10.1145/2685616.

[Fau+18] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. “Composable Masking Schemes in the Presence of Physical Defaults & the Robust Probing Model”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.3 (2018), pp. 89–120. ISSN: 2569-2925. DOI: 10.13154/tches.v2018.i3.89-120. URL: https://tches.iacr.org/index.php/TCHES/article/view/7270.

[Fed13] Federal Office for Information Security (BSI). *Minimum Requirements for Evaluating Side-Channel Attack Resistanceof RSA, DSA and Diffie-Hellman Key ExchangeImplementations*. Tech.

rep. Version 1.0. Federal Office for Information Security (BSI), Jan. 2013. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Interpretationen/AIS_46_BSI_guidelines_SCA_RSA_V1_0_e_pdf.pdf.

[Fed16] Federal Office for Information Security (BSI). *Minimum Requirements for Evaluating Side-Channel Attack Resistance of Elliptic Curve Implementations.* Tech. rep. Version 2.0. Federal Office for Information Security (BSI), Nov. 2016. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Zertifizierung/Interpretationen/AIS_46_ECCGuide_e_pdf.pdf.

[Fed23] Federal Office for Information Security (BSI). *BSI–Technical Guideline Designation: Cryptographic Mechanisms: Recommendations and Key Lengths.* Tech. rep. BSI-TR-02102-1. Version 2023-01. Federal Office for Information Security (BSI), Jan. 2023. URL: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.pdf.

[FKB13] Andreas Fröhlich, Gergely Kovásznai, and Armin Biere. "More on the Complexity of Quantifier-Free Fixed-Size Bit-Vector Logics with Binary Encoding". In: *Computer Science - Theory and Applications - 8th International Computer Science Symposium in Russia, CSR 2013, Ekaterinburg, Russia, June 25-29, 2013. Proceedings.* Ed. by Andrei A. Bulatov and Arseny M. Shur. Vol. 7913. Lecture Notes in Computer Science. Springer, 2013, pp. 378–390. DOI: 10.1007/978-3-642-38536-0_33. URL: https://doi.org/10.1007/978-3-642-38536-0%5C_33.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. "Secure Integration of Asymmetric and Symmetric Encryption Schemes". In: *Advances in Cryptology – CRYPTO'99.* Ed. by Michael J. Wiener. Vol. 1666. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Berlin, Heidelberg, Germany, Aug. 1999, pp. 537–554. DOI: 10.1007/3-540-48405-1_34.

[FPS17] Sebastian Faust, Clara Paglialonga, and Tobias Schneider. "Amortizing Randomness Complexity in Private Circuits". In: *Advances in Cryptology – ASIACRYPT 2017, Part I.* Ed. by Tsuyoshi Takagi and Thomas Peyrin. Vol. 10624. Lecture Notes in Computer Science. Hong Kong, China: Springer, Cham, Switzerland, Dec. 2017, pp. 781–810. DOI: 10.1007/978-3-319-70694-8_27.

[Fri+21] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. “Masked Accelerators and Instruction Set Extensions for Post-Quantum Cryptography”. In: *IACR Cryptol. ePrint Arch.* (2021), p. 479. URL: https://eprint.iacr.org/2021/479.

[Fri+22] Tim Fritzmann, Michiel Van Beirendonck, Debapriya Basu Roy, Patrick Karl, Thomas Schamberger, Ingrid Verbauwhede, and Georg Sigl. “Masked Accelerators and Instruction Set Extensions for Post-Quantum Cryptography”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2022.1 (2022), pp. 414–460. DOI: 10.46586/tches.v2022.i1.414-460.

[Fri20] Jannik Friemann. “Detecting timing side-channels in executables”. Bachelor Thesis. Hamburg: Technische Universität Hamburg, June 26, 2020. DOI: 10.15480/882.2852. URL: http://hdl.handle.net/11420/6908.

[Gao+19] Si Gao, Ben Marshall, Dan Page, and Elisabeth Oswald. “Share-slicing: Friend or Foe?” In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020.1 (2019), pp. 152–174. ISSN: 2569-2925. DOI: 10.13154/tches.v2020.i1.152-174. URL: https://tches.iacr.org/index.php/TCHES/article/view/8396.

[GGS18] Qian Guo, Vincent Grosso, and François-Xavier Standaert. *Modeling Soft Analytical Side-Channel Attacks from a Coding Theory Viewpoint.* Cryptology ePrint Archive, Report 2018/498. 2018. URL: https://eprint.iacr.org/2018/498.

[Gie+08] Benedikt Gierlichs, Lejla Batina, Pim Tuyls, and Bart Preneel. “Mutual Information Analysis”. In: *Cryptographic Hardware and Embedded Systems – CHES 2008.* Ed. by Elisabeth Oswald and Pankaj Rohatgi. Vol. 5154. Lecture Notes in Computer Science. Washington, DC, USA: Springer, Berlin, Heidelberg, Germany, Aug. 2008, pp. 426–442. DOI: 10.1007/978-3-540-85053-3_27.

[Gig+21] Barbara Gigerl, Vedad Hadzic, Robert Primas, Stefan Mangard, and Roderick Bloem. “Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs”. In: *USENIX Security 2021: 30th USENIX Security Symposium.* Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, Aug. 2021, pp. 1469–1468.

[GJN20] Qian Guo, Thomas Johansson, and Alexander Nilsson. "A Key-Recovery Timing Attack on Post-quantum Primitives Using the Fujisaki-Okamoto Transformation and Its Application on FrodoKEM". In: *Advances in Cryptology – CRYPTO 2020, Part II*. Ed. by Daniele Micciancio and Thomas Ristenpart. Vol. 12171. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Cham, Switzerland, Aug. 2020, pp. 359–386. DOI: 10.1007/978-3-030-56880-1_13.

[GM82] Joseph A. Goguen and José Meseguer. "Security Policies and Security Models". In: *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*. IEEE Computer Society, 1982, pp. 11–20. DOI: 10.1109/SP.1982.10014. URL: https://doi.org/10.1109/SP.1982.10014.

[GMO01] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. "Electromagnetic Analysis: Concrete Results". In: *Cryptographic Hardware and Embedded Systems – CHES 2001*. Ed. by Çetin Kaya Koç, David Naccache, and Christof Paar. Vol. 2162. Lecture Notes in Computer Science. Paris, France: Springer, Berlin, Heidelberg, Germany, May 2001, pp. 251–261. DOI: 10.1007/3-540-44709-1_21.

[GO22a] Si Gao and Elisabeth Oswald. "A Novel Completeness Test for Leakage Models and Its Application to Side Channel Attacks and Responsibly Engineered Simulators". In: *Advances in Cryptology – EUROCRYPT 2022, Part III*. Ed. by Orr Dunkelman and Stefan Dziembowski. Vol. 13277. Lecture Notes in Computer Science. Trondheim, Norway: Springer, Cham, Switzerland, May 2022, pp. 254–283. DOI: 10.1007/978-3-031-07082-2_10.

[GO22b] Si Gao and Elisabeth Oswald. *A Novel Framework for Explainable Leakage Assessment*. Cryptology ePrint Archive, Report 2022/182. 2022. URL: https://eprint.iacr.org/2022/182.

[GOP22] Si Gao, Elisabeth Oswald, and Dan Page. "Towards Micro-architectural Leakage Simulators: Reverse Engineering Micro-architectural Leakage Features Is Practical". In: *Advances in Cryptology – EUROCRYPT 2022, Part III*. Ed. by Orr Dunkelman and Stefan Dziembowski. Vol. 13277. Lecture Notes in Computer Science. Trondheim, Norway: Springer, Cham, Switzerland, May 2022, pp. 284–311. DOI: 10.1007/978-3-031-07082-2_11.

[Gou19] Marc Gourjon. "Towards Secure Compilation of Power Side-Channel Countermeasures". In: *PriSC 2019: 3rd Workshop on Principles of Secure Compilation*. https://popl19.sigplan.org/details/prisc-2019/10/Towards-Secure-Compilation-of-Power-Side-Channel-Countermeasures (Accessed: November 11, 2023). Cascais, Portugal, Jan. 2019.

[Gou21] Marc Gourjon. "Explicit Leakage: Handling Side-Channel Behavior in Program Rewriting and Analysis". In: *PriSC 2021: 5th Workshop on Principles of Secure Compilation*. https://popl21.sigplan.org/details/prisc-2021-papers/1/Explicit-Leakage-Handling-Side-Channel-Behavior-in-Program-Rewriting-and-Analysis (Accessed November 11, 2023). Virtual, Jan. 2021.

[Gou22a] Marc Gourjon. "Fine-Grained Power Leakage Models and Verification of Software Masking". In: *VeriSICC Seminar 2022: Verification and Generation of Side-Channel Countermeasures*. https://www.cryptoexperts.com/verisicc/slides/slides_Marc.pdf (Accessed: November 11, 2023). Paris, France, Sept. 2022.

[Gou22b] Marc Gourjon. "Power Contracts: Provably Complete Power Leakage Models for Secure Execution of Masked Software on Processors". In: *TASER Workshop 2022: Topics in hArdware SEcurity and RISC-V*. https://ches.iacr.org/2022/taser/TASER22-gourjon.pdf (Accessed November 11, 2023). Leuven, Belgium, Sept. 2022.

[GP99] Louis Goubin and Jacques Patarin. "DES and Differential Power Analysis (The "Duplication" Method)". In: *Cryptographic Hardware and Embedded Systems – CHES'99*. Ed. by Çetin Kaya Koç and Christof Paar. Vol. 1717. Lecture Notes in Computer Science. Worcester, Massachusetts, USA: Springer, Berlin, Heidelberg, Germany, Aug. 1999, pp. 158–172. DOI: 10.1007/3-540-48059-5_15.

[GPM21] Barbara Gigerl, Robert Primas, and Stefan Mangard. "Secure and Efficient Software Masking on Superscalar Pipelined Processors". In: *Advances in Cryptology – ASIACRYPT 2021, Part II*. Ed. by Mehdi Tibouchi and Huaxiong Wang. Vol. 13091. Lecture Notes in Computer Science. Singapore: Springer, Cham, Switzerland, Dec. 2021, pp. 3–32. DOI: 10.1007/978-3-030-92075-3_1.

[GR19] François Gérard and Mélissa Rossi. “An Efficient and Provable Masked Implementation of qTESLA”. In: *Smart Card Research and Advanced Applications - 18th International Conference, CARDIS 2019, Prague, Czech Republic, November 11-13, 2019, Revised Selected Papers*. Ed. by Sonia Belaïd and Tim Güneysu. Vol. 11833. Lecture Notes in Computer Science. Springer, 2019, pp. 74–91. DOI: 10.1007/978-3-030-42068-0_5. URL: https://doi.org/10.1007/978-3-030-42068-0%5C_5.

[Gro+16] Wouter de Groot, Kostas Papagiannopoulos, Antonio de la Piedra, Erik Schneider, and Lejla Batina. “Bitsliced Masking and ARM: Friends or Foes?” In: *Lightweight Cryptography for Security and Privacy - 5th International Workshop, LightSec 2016, Aksaray, Turkey, September 21-22, 2016, Revised Selected Papers*. Ed. by Andrey Bogdanov. Vol. 10098. Lecture Notes in Computer Science. Springer, 2016, pp. 91–109. DOI: 10.1007/978-3-319-55714-4_7. URL: https://doi.org/10.1007/978-3-319-55714-4%5C_7.

[HCY19] Wei-Lun Huang, Jiun-Peng Chen, and Bo-Yin Yang. “Power Analysis on NTRU Prime”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020.1 (2019), pp. 123–151. ISSN: 2569-2925. DOI: 10.13154/tches.v2020.i1.123-151. URL: https://tches.iacr.org/index.php/TCHES/article/view/8395.

[Hei20] Rainer Heinelt. “Automated Generation of Higher-Order Side-Channel Resilient Code”. Master Thesis. Bochum: Ruhr Universität Bochum, Jan. 2020.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. “A Modular Analysis of the Fujisaki-Okamoto Transformation”. In: *TCC 2017: 15th Theory of Cryptography Conference, Part I*. Ed. by Yael Kalai and Leonid Reyzin. Vol. 10677. Lecture Notes in Computer Science. Baltimore, MD, USA: Springer, Cham, Switzerland, Nov. 2017, pp. 341–371. DOI: 10.1007/978-3-319-70500-2_12.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. “Private Circuits: Securing Hardware against Probing Attacks”. In: *Advances in Cryptology – CRYPTO 2003*. Ed. by Dan Boneh. Vol. 2729. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Berlin, Heidelberg, Germany, Aug. 2003, pp. 463–481. DOI: 10.1007/978-3-540-45146-4_27.

[Kan+19a] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, Douglas Stebila, and Thom Wiggers. *The PQClean project.* `https://github.com/PQClean/PQClean`. 2019.

[Kan+19b] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. *pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4.* Workshop Record of the Second PQC Standardization Conference. `https://cryptojedi.org/papers/#pqm4`. 2019.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *Advances in Cryptology – CRYPTO'99.* Ed. by Michael J. Wiener. Vol. 1666. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Berlin, Heidelberg, Germany, Aug. 1999, pp. 388–397. DOI: `10.1007/3-540-48405-1_25`.

[Koc96] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Advances in Cryptology – CRYPTO'96.* Ed. by Neal Koblitz. Vol. 1109. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Berlin, Heidelberg, Germany, Aug. 1996, pp. 104–113. DOI: `10.1007/3-540-68697-5_9`.

[KR19] Yael Tauman Kalai and Leonid Reyzin. *A Survey of Leakage-Resilient Cryptography.* Cryptology ePrint Archive, Report 2019/302. 2019. URL: `https://eprint.iacr.org/2019/302`.

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. "SILVER - Statistical Independence and Leakage Verification". In: *Advances in Cryptology – ASIACRYPT 2020, Part I.* Ed. by Shiho Moriai and Huaxiong Wang. Vol. 12491. Lecture Notes in Computer Science. Daejeon, South Korea: Springer, Cham, Switzerland, Dec. 2020, pp. 787–816. DOI: `10.1007/978-3-030-64837-4_26`.

[Kuh02] Markus G. Kuhn. "Optical Time-Domain Eavesdropping Risks of CRT Displays". In: *2002 IEEE Symposium on Security and Privacy.* Berkeley, CA, USA: IEEE Computer Society Press, May 2002, pp. 3–18. DOI: `10.1109/SECPRI.2002.1004358`.

[Lav+21] Corentin Lavaud, Robin Gerzaguet, Matthieu Gautier, Olivier Berder, Erwan Nogues, and Stéphane Molton. "Whispering Devices: A Survey on How Side-channels Lead to Compromised Information". In: *J. Hardw. Syst. Secur.* 5.2 (2021), pp. 143–168. DOI: `10.1007/s41635-021-00112-6`. URL: `https://doi.org/10.1007/s41635-021-00112-6`.

[Lip+21] 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: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2021.

[low] lowRISC. *Ibex RISC-V Core*. `https://github.com/lowRISC/ibex`.

[LS15] Adeline Langlois and Damien Stehlé. “Worst-case to average-case reductions for module lattices”. In: *Designs, Codes and Cryptography* 75.3 (2015), pp. 565–599.

[Mes00] Thomas S. Messerges. “Using Second-Order Power Analysis to Attack DPA Resistant Software”. In: *Cryptographic Hardware and Embedded Systems – CHES 2000*. Ed. by Çetin Kaya Koç and Christof Paar. Vol. 1965. Lecture Notes in Computer Science. Worcester, Massachusetts, USA: Springer, Berlin, Heidelberg, Germany, Aug. 2000, pp. 238–251. DOI: `10.1007/3-540-44499-8_19`.

[Mig+19] Vincent Migliore, Benoît Gérard, Mehdi Tibouchi, and Pierre-Alain Fouque. “Masking Dilithium - Efficient Implementation and Side-Channel Evaluation”. In: *ACNS 19: 17th International Conference on Applied Cryptography and Network Security*. Ed. by Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung. Vol. 11464. Lecture Notes in Computer Science. Bogota, Colombia: Springer, Cham, Switzerland, June 2019, pp. 344–362. DOI: `10.1007/978-3-030-21568-2_17`.

[Mor14] Amir Moradi. “Side-Channel Leakage through Static Power - Should We Care about in Practice?” In: *Cryptographic Hardware and Embedded Systems – CHES 2014*. Ed. by Lejla Batina and Matthew Robshaw. Vol. 8731. Lecture Notes in Computer Science. Busan, South Korea: Springer, Berlin, Heidelberg, Germany, Sept. 2014, pp. 562–579. DOI: `10.1007/978-3-662-44709-3_31`.

[Mos+12] Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. “Compiler Assisted Masking”. In: *Cryptographic Hardware and Embedded Systems – CHES 2012*. Ed. by Emmanuel Prouff and Patrick Schaumont. Vol. 7428. Lecture Notes in Computer Science. Leuven, Belgium: Springer, Berlin, Heidelberg, Germany, Sept. 2012, pp. 58–75. DOI: `10.1007/978-3-642-33027-8_4`.

[MOW17] David McCann, Elisabeth Oswald, and Carolyn Whitnall. "Towards Practical Tools for Side Channel Aware Software Engineering: 'Grey Box' Modelling for Instruction Leakages". In: *USENIX Security 2017: 26th USENIX Security Symposium*. Ed. by Engin Kirda and Thomas Ristenpart. Vancouver, BC, Canada: USENIX Association, Aug. 2017, pp. 199–216.

[MPG05] Stefan Mangard, Thomas Popp, and Berndt M. Gammel. "Side-Channel Leakage of Masked CMOS Gates". In: *Topics in Cryptology – CT-RSA 2005*. Ed. by Alfred Menezes. Vol. 3376. Lecture Notes in Computer Science. San Francisco, CA, USA: Springer, Berlin, Heidelberg, Germany, Feb. 2005, pp. 351–365. DOI: 10.1007/978-3-540-30574-3_24.

[MPO05] Stefan Mangard, Norbert Pramstaller, and Elisabeth Oswald. "Successfully Attacking Masked AES Hardware Implementations". In: *Cryptographic Hardware and Embedded Systems – CHES 2005*. Ed. by Josyula R. Rao and Berk Sunar. Vol. 3659. Lecture Notes in Computer Science. Edinburgh, UK: Springer, Berlin, Heidelberg, Germany, Aug. 2005, pp. 157–171. DOI: 10.1007/11545262_12.

[MPW22] Ben Marshall, Dan Page, and James Webb. "MIRACLE: MIcRoArChitectural Leakage Evaluation A study of micro-architectural power leakage across many devices". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2022.1 (2022), pp. 175–220. DOI: 10.46586/tches.v2022.i1.175-220.

[MR03] Silvio Micali and Leonid Reyzin. *Physically Observable Cryptography*. Cryptology ePrint Archive, Report 2003/120. 2003. URL: https://eprint.iacr.org/2003/120.

[MR04] Silvio Micali and Leonid Reyzin. "Physically Observable Cryptography (Extended Abstract)". In: *TCC 2004: 1st Theory of Cryptography Conference*. Ed. by Moni Naor. Vol. 2951. Lecture Notes in Computer Science. Cambridge, MA, USA: Springer, Berlin, Heidelberg, Germany, Feb. 2004, pp. 278–296. DOI: 10.1007/978-3-540-24638-1_16.

[Mun+21] Prashanth Mundkur, Rishiyur S. Nikhil, Bluespec Inc, Jon French, Brian Campbell, Robert Norton-Wright, Alasdair Armstrong, Thomas Bauereiss, Shaked Flur, Christopher Pulte, Peter Sewell, Alexander Richardson, Hesham Almatary, Jessica Clarke, Microsoft, Nathaniel Wesley Filardo, Peter Rugg, and Aril Com-

puter Corp. *RISCV Sail Model*. https://github.com/riscv/sail-riscv (accessed January 17, 2022). 2021.

[Nat17] National Institute of Standards and Technology. *Post-Quantum Cryptography Standardization*. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization (Accessed: November 11, 2023). Jan. 2017.

[Ngo+21] Kalle Ngo, Elena Dubrova, Qian Guo, and Thomas Johansson. “A Side-Channel Attack on a Masked IND-CCA Secure Saber KEM”. In: *IACR Cryptol. ePrint Arch.* 2021 (2021), p. 79. URL: https://eprint.iacr.org/2021/079.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. “Threshold Implementations Against Side-Channel Attacks and Glitches”. In: *ICICS 06: 8th International Conference on Information and Communication Security*. Ed. by Peng Ning, Sihan Qing, and Ninghui Li. Vol. 4307. Lecture Notes in Computer Science. Raleigh, NC, USA: Springer, Berlin, Heidelberg, Germany, Dec. 2006, pp. 529–545. DOI: 10.1007/11935308_38.

[NRS11] Svetla Nikova, Vincent Rijmen, and Martin Schläffer. “Secure Hardware Implementation of Nonlinear Functions in the Presence of Glitches”. In: *Journal of Cryptology* 24.2 (Apr. 2011), pp. 292–321. DOI: 10.1007/s00145-010-9085-7.

[NXP16] NXP Semiconductors. *FRDM-KL82Z User's Guide*. https://www.nxp.com/docs/en/user-guide/FRDMKL82ZUG.pdf. 2016.

[Ode+18] Tobias Oder, Tobias Schneider, Thomas Pöppelmann, and Tim Güneysu. “Practical CCA2-Secure Masked Ring-LWE Implementations”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.1 (2018), pp. 142–174. ISSN: 2569-2925. DOI: 10.13154/tches.v2018.i1.142-174. URL: https://tches.iacr.org/index.php/TCHES/article/view/836.

[PP19] Peter Pessl and Robert Primas. “More Practical Single-Trace Attacks on the Number Theoretic Transform”. In: *Progress in Cryptology - LATINCRYPT 2019: 6th International Conference on Cryptology and Information Security in Latin America*. Ed. by Peter Schwabe and Nicolas Thériault. Vol. 11774. Lecture Notes in Computer Science. Santiago, Chile: Springer, Cham, Switzerland, Oct. 2019, pp. 130–149. DOI: 10.1007/978-3-030-30530-7_7.

[PPM17] Robert Primas, Peter Pessl, and Stefan Mangard. “Single-Trace Side-Channel Attacks on Masked Lattice-Based Encryption”. In: *Cryptographic Hardware and Embedded Systems – CHES 2017*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. Lecture Notes in Computer Science. Taipei, Taiwan: Springer, Cham, Switzerland, Sept. 2017, pp. 513–533. DOI: 10.1007/978-3-319-66787-4_25.

[PR13] Emmanuel Prouff and Matthieu Rivain. “Masking against Side-Channel Attacks: A Formal Security Proof”. In: *Advances in Cryptology – EUROCRYPT 2013*. Ed. by Thomas Johansson and Phong Q. Nguyen. Vol. 7881. Lecture Notes in Computer Science. Athens, Greece: Springer, Berlin, Heidelberg, Germany, May 2013, pp. 142–159. DOI: 10.1007/978-3-642-38348-9_9.

[PV17] Kostas Papagiannopoulos and Nikita Veshchikov. “Mind the Gap: Towards Secure 1st-Order Masking in Software”. In: *COSADE 2017: 8th International Workshop on Constructive Side-Channel Analysis and Secure Design*. Ed. by Sylvain Guilley. Vol. 10348. Lecture Notes in Computer Science. Paris, France: Springer, Cham, Switzerland, Apr. 2017, pp. 282–297. DOI: 10.1007/978-3-319-64647-3_17.

[PZ03] J. Proos and C. Zalka. “Shor’s discrete logarithm quantum algorithm for elliptic curves”. In: *Quantum Inf. Comput.* 3 (2003), pp. 317–344. URL: https://cds.cern.ch/record/602816.

[QS01] Jean-Jacques Quisquater and David Samyde. “ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards”. In: *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001, Cannes, France, September 19-21, 2001, Proceedings*. Ed. by Isabelle Attali and Thomas P. Jensen. Vol. 2140. Lecture Notes in Computer Science. Springer, 2001, pp. 200–210. DOI: 10.1007/3-540-45418-7_17. URL: https://doi.org/10.1007/3-540-45418-7%5C_17.

[Rav+20a] Prasanna Ravi, Shivam Bhasin, Sujoy Sinha Roy, and Anupam Chattopadhyay. *Drop by Drop you break the rock - Exploiting generic vulnerabilities in Lattice-based PKE/KEMs using EM-based Physical Attacks*. Cryptology ePrint Archive, Report 2020/549. 2020. URL: https://eprint.iacr.org/2020/549.

[Rav+20b] Prasanna Ravi, Sujoy Sinha Roy, Anupam Chattopadhyay, and Shivam Bhasin. "Generic Side-channel attacks on CCA-secure lattice-based PKE and KEMs". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020.3 (2020), pp. 307–335. ISSN: 2569-2925. DOI: 10.13154/tches.v2020.i3.307-335. URL: https://tches.iacr.org/index.php/TCHES/article/view/8592.

[Rep+15] Oscar Reparaz, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. "A Masked Ring-LWE Implementation". In: *Cryptographic Hardware and Embedded Systems – CHES 2015*. Ed. by Tim Güneysu and Helena Handschuh. Vol. 9293. Lecture Notes in Computer Science. Saint-Malo, France: Springer, Berlin, Heidelberg, Germany, Sept. 2015, pp. 683–702. DOI: 10.1007/978-3-662-48324-4_34.

[RP10] Matthieu Rivain and Emmanuel Prouff. "Provably Secure Higher-Order Masking of AES". In: *Cryptographic Hardware and Embedded Systems – CHES 2010*. Ed. by Stefan Mangard and François-Xavier Standaert. Vol. 6225. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Berlin, Heidelberg, Germany, Aug. 2010, pp. 413–427. DOI: 10.1007/978-3-642-15031-9_28.

[Sch+19] Tobias Schneider, Clara Paglialonga, Tobias Oder, and Tim Güneysu. "Efficiently Masking Binomial Sampling at Arbitrary Orders for Lattice-Based Crypto". In: *PKC 2019: 22nd International Conference on Theory and Practice of Public Key Cryptography, Part II*. Ed. by Dongdai Lin and Kazue Sako. Vol. 11443. Lecture Notes in Computer Science. Beijing, China: Springer, Cham, Switzerland, Apr. 2019, pp. 534–564. DOI: 10.1007/978-3-030-17259-6_18.

[Sch+20a] Thomas Schamberger, Julian Renner, Georg Sigl, and Antonia Wachter-Zeh. "A Power Side-Channel Attack on the CCA2-Secure HQC KEM". In: *Smart Card Research and Advanced Applications - 19th International Conference, CARDIS 2020, Virtual Event, November 18-19, 2020, Revised Selected Papers*. Ed. by Pierre-Yvan Liardet and Nele Mentens. Vol. 12609. Lecture Notes in Computer Science. Springer, 2020, pp. 119–134. DOI: 10.1007/978-3-030-68487-7_8. URL: https://doi.org/10.1007/978-3-030-68487-7%5C_8.

[Sch+20b] Peter Schwabe, Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Gregor Seiler, and Damien Stehlé. *CRYSTALS-KYBER*. Tech. rep. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions. National Institute of Standards and Technology, 2020.

[Sek+18] Okan Seker, Abraham Fernandez-Rubio, Thomas Eisenbarth, and Rainer Steinwandt. "Extending Glitch-Free Multiparty Protocols to Resist Fault Injection Attacks". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.3 (2018), pp. 394–430. ISSN: 2569-2925. DOI: 10.13154/tches.v2018.i3.394-430. URL: https://tches.iacr.org/index.php/TCHES/article/view/7281.

[Sew20] Peter Sewell. *ISA Formal Spec Public Review*. https://github.com/riscvarchive/ISA_Formal_Spec_Public_Review (accessed January 14, 2022). 2020.

[Sha49] Claude E. Shannon. "Communication theory of secrecy systems". In: *Bell Systems Technical Journal* 28.4 (1949), pp. 656–715.

[Sha79] Adi Shamir. "How to Share a Secret". In: *Communications of the Association for Computing Machinery* 22.11 (Nov. 1979), pp. 612–613. DOI: 10.1145/359168.359176.

[She+21a] Madura A. Shelton, Lukasz Chmielewski, Niels Samwel, Markus Wagner, Lejla Batina, and Yuval Yarom. "Rosita++: Automatic Higher-Order Leakage Elimination from Cryptographic Code". In: *ACM CCS 2021: 28th Conference on Computer and Communications Security*. Ed. by Giovanni Vigna and Elaine Shi. Virtual Event, Republic of Korea: ACM Press, Nov. 2021, pp. 685–699. DOI: 10.1145/3460120.3485380.

[She+21b] Madura A. Shelton, Niels Samwel, Lejla Batina, Francesco Regazzoni, Markus Wagner, and Yuval Yarom. "Rosita: Towards Automatic Elimination of Power-Analysis Leakage in Ciphers". In: *ISOC Network and Distributed System Security Symposium – NDSS 2021*. Virtual: The Internet Society, Feb. 2021. DOI: 10.14722/ndss.2021.23137.

[Sho94] Peter W. Shor. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring". In: *35th Annual Symposium on Foundations of Computer Science*. Santa Fe, NM, USA: IEEE Com-

puter Society Press, Nov. 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700.

[SLP05] Werner Schindler, Kerstin Lemke, and Christof Paar. “A Stochastic Model for Differential Side Channel Cryptanalysis”. In: *Cryptographic Hardware and Embedded Systems – CHES 2005*. Ed. by Josyula R. Rao and Berk Sunar. Vol. 3659. Lecture Notes in Computer Science. Edinburgh, UK: Springer, Berlin, Heidelberg, Germany, Aug. 2005, pp. 30–46. DOI: 10.1007/11545262_3.

[SM15] Tobias Schneider and Amir Moradi. “Leakage Assessment Methodology - A Clear Roadmap for Side-Channel Evaluations”. In: *Cryptographic Hardware and Embedded Systems – CHES 2015*. Ed. by Tim Güneysu and Helena Handschuh. Vol. 9293. Lecture Notes in Computer Science. Saint-Malo, France: Springer, Berlin, Heidelberg, Germany, Sept. 2015, pp. 495–513. DOI: 10.1007/978-3-662-48324-4_25.

[TE15] Mostafa Taha and Thomas Eisenbarth. *Implementation Attacks on Post-Quantum Cryptographic Schemes*. Cryptology ePrint Archive, Report 2015/1083. 2015. URL: https://eprint.iacr.org/2015/1083.

[Uen+22] Rei Ueno, Keita Xagawa, Yutaro Tanaka, Akira Ito, Junko Takahashi, and Naofumi Homma. “Curse of Re-encryption: A Generic Power/EM Analysis on Post-Quantum KEMs”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2022.1 (2022), pp. 296–322. DOI: 10.46586/tches.v2022.i1.296-322.

[Ves14] Nikita Veshchikov. “SILK: high level of abstraction leakage simulator for side channel analysis”. In: *Proceedings of the 4th Program Protection and Reverse Engineering Workshop, PPREW@ACSAC 2014, New Orleans, LA, USA, December 9, 2014*. Ed. by Mila Dalla Preda and Jeffrey Todd McDonald. ACM, 2014, 3:1–3:11. DOI: 10.1145/2689702.2689706. URL: https://doi.org/10.1145/2689702.2689706.

[VGS14] Nicolas Veyrat-Charvillon, Benoît Gérard, and François-Xavier Standaert. “Soft Analytical Side-Channel Attacks”. In: *Advances in Cryptology – ASIACRYPT 2014, Part I*. Ed. by Palash Sarkar and Tetsu Iwata. Vol. 8873. Lecture Notes in Computer Science. Kaoshiung, Taiwan, R.O.C.: Springer, Berlin, Heidelberg, Germany, Dec. 2014, pp. 282–296. DOI: 10.1007/978-3-662-45611-8_15.

[Wal01] Colin D. Walter. “Sliding Windows Succumbs to Big Mac Attack”. In: *Cryptographic Hardware and Embedded Systems – CHES 2001*. Ed. by Çetin Kaya Koç, David Naccache, and Christof Paar. Vol. 2162. Lecture Notes in Computer Science. Paris, France: Springer, Berlin, Heidelberg, Germany, May 2001, pp. 286–299. DOI: 10.1007/3-540-44709-1_24.

[Wol19] Philipp Wolters. “Towards Property-Preserving Compilation of Masked Implementations”. Master Thesis. Hamburg: Technische Universität Hamburg, Aug. 26, 2019.

[Xu+20] Zhuang Xu, Owen Pemberton, Sujoy Sinha Roy, and David Oswald. *Magnifying Side-Channel Leakage of Lattice-Based Cryptosystems with Chosen Ciphertexts: The Case Study of Kyber*. Cryptology ePrint Archive, Report 2020/912. 2020. URL: https://eprint.iacr.org/2020/912.

[Zha+18] Jun Zhang, Pengfei Gao, Fu Song, and Chao Wang. “SCInfer: Refinement-Based Verification of Software Countermeasures Against Side-Channel Attacks”. In: *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*. Ed. by Hana Chockler and Georg Weissenbacher. Vol. 10982. Lecture Notes in Computer Science. Springer, 2018, pp. 157–177. DOI: 10.1007/978-3-319-96142-2_12. URL: https://doi.org/10.1007/978-3-319-96142-2%5C_12.

[ZZT05] Li Zhuang, Feng Zhou, and J. D. Tygar. “Keyboard Acoustic Emanations Revisited”. In: *ACM CCS 2005: 12th Conference on Computer and Communications Security*. Ed. by Vijayalakshmi Atluri, Catherine Meadows, and Ari Juels. Alexandria, Virginia, USA: ACM Press, Nov. 2005, pp. 373–382. DOI: 10.1145/1102120.1102169.